Laboratoire Pierre Aigrain

PERL
pour les physiciens

Nicolas Regnault

Avant propos

*"Les trois principales vertus du programmeur sont
la Paresse, l'Impatience et l'Orgueil"*

Larry Wall.

Ce cours de PERL est né sous l'impulsion de demandes répétées de divers collègues concernant des problèmes informatiques que l'utilisation d'un langage de scripts puissant aurait permis de résoudre rapidement. Bien que s'adressant à la base à un public composé de physiciens, ces notes ne nécessitent aucune connaissance en Physique (hormis un ou deux exercices et une section du dernier chapitre) et peuvent constituer une introduction à PERL pour d'autres personnes.

Une des des forces de PERL est sa très grande flexibilité concernant sa syntaxe. Le revers de la médaille est qu'une présentation de ce langage ne permet pas forcément d'en voir toutes les facettes. Ces notes ne dérogent pas à la règle et sont donc une vision (honteusement) biaisée de PERL. Nous irons même jusqu'à prodiguer des conseils en programmation qui pourraient nous faire passer pour des hérétiques aux yeux de certains informaticiens.

Le temps d'un physiciens étant précieux, vous devez vous poser la question de l'investissement d'une dizaine d'heures pour apprendre un nouveau langage. Si votre utilisation de l'outils informatique se limite à consulter votre courier, à surfer sur Internet et à taper du laTeX, passez votre chemin. Par contre si vous êtes amenés à mamipuler et traiter des garndes quantités de fichiers de données, de textes ou autres, alors vous ne regréterez pas l'apprentissage de PERL.

Table des matières

Chapitre 1

Introduction

1.1 Qu'est-ce que PERL

PERL est l'acronyme de "Practical Extraction and Report Language" ou "Pathologically Exclectic Rubbish Lister", son inventeur Larry Wall n'ayant pas encore tranché entre ces deux possibilités. Ce langage a été conçu à l'origine pour écrire des programmes (si possible réutilisables) allant au-delà de certaines limites des scripts shell UNIX. Cette image de super langage shell est une assez bonne définition de ce qu'est PERL. Il permet d'écrire rapidement de petits programmes qui officient là où les manipulations manuelles seraient inenvisageables parce que trop répétitives ou ardues.

Une des forces majeures de PERL est sa capacité à pouvoir manipuler fichiers et chaînes de caractères, comme nous le verrons au fil de ce cours. Ceux qui se sont risqués à renommer huit cents fichiers à la main via un explorateur quelconque savent à quel point un tel langage est salvateur.

Pourquoi PERL ? Il existe bien sûr d'autres langages pouvant remplir un office comparable à PERL à commencer par le shell UNIX. Nous citerons aussi PYTHON ou RUBY. Alors pourquoi PERL plutôt qu'un de ces autres langages ? Contrairement aux scripts UNIX, PERL est portable et fonctionne sur de nombreuses plateformes. Il n'est pas à la base orienté objet (si vous ne savez pas ce que cela signifie, vous pouvez le voir comme une conception avancée d'un langage de programmation) simplifiant ainsi son apprentissage. Avouons aussi que le fait que le rédacteur de ces notes soit un adepte de ce langage n'est pas sans rapport avec ce choix.

1.2 Comment installer PERL

Si vous travaillez sur un système UNIX, BSD ou MAC OS X, il y a de fortes chances que PERL soit déjà présent. Ouvrez un terminal et tapez la commande *perl -v*. Si *perl* est présent sur votre système, vous devriez obtenir un message du style

> *This is perl, v5.8.4 built for x86_64-linux*
>
> *Copyright 1987-2004, Larry Wall*
>
> *Perl may be copied only under the terms of either the Artistic License or the GNU General Public License, which may be found in the Perl 5 source kit.*

> *Complete documentation for Perl, including FAQ lists, should be found on this system using 'man perl' or 'perldoc perl'. If you have access to the Internet, point your browser at http ://www.perl.com/, the Perl Home Page.*

Dans le cas contraire (ce qui signifie que ce n'est pas votre jour de chance, vous êtes tombés sur le seul système UNIX-BSD où PERL n'est pas fourni en standard), je vous invite à consulter l'aide à l'installation de logiciels de votre système.

Pour toutes les autres plates-formes, il existe un site http ://www.cpan.org/ports indiquant où obtenir PERL pour chacune d'entre elles.

Comme quelques irréductibles continuent d'utiliser Windows, nous allons détailler l'installation de PERL sur ce système. Il vous faudra tout d'abord récupérer la dernière version en date sur le site d'http ://www.activestate.com/Products/Download/Download.plex ?id=-ActivePerl. La version dite MSI possède un asssistant d'installation, les options par défaut suffisent. Profitez de l'occasion pour télécharger http ://open-perl-ide.sourceforge.net, un environnement de développement PERL, qui consiste en un fichier zip contenant le fichier exécutable qui est le programme lui-même (décompressez l'archive là où bon vous semble).

Si vous ne souhaitez pas installer PERL, sachez qu'il est disponible en démarrant sur un CD de http ://www.knoppix.org. Vous pourrez ainsi apprendre à utiliser ce langage sans avoir à l'installer.

1.3 La documentation

Il existe deux livres incontournables pour PERL. Le premier est *Introduction à PERL* de R. Schwartz et T. Christiansen aux éditions O'Reilly. Cet ouvrage est ce qui se fait de mieux dans ce domaine et sa qualité d'écriture en fait sûrement le meilleur ouvrage informatique (voire littéraire pour tout bon geek qui se respecte). Il est aussi suffisant pour la majorité des utilisateurs de base. Le deuxième livre est le livre de référence de Larry Wall, *Programmation en PERL*, chez O'Reilly. Cet ouvrage est la Bible de PERL, mais je déconseille son usage à tout débutant sous peine d'allergie définitive à ce langage (pour les initiés, ce serait comme commencer la mécanique du point avec le Landau ou la théorie des champs avec le Zinn-Justin).

Sous UNIX et BSD, les pages man sont d'un grand secours, en particulier *man perl* et *man perlfunc*. Dans la même veine, il existe *man faq* disponible aussi sur le http ://faq.perl.org qui répondra à une majorité des questions simples. Sous Windows, Open-PERL-IDE contient une aide en ligne très complète (issue des pages man).

Le web regorge de documentations et autres introductions à PERL. En faire une liste même non exhaustive serait trop long (une petite recherche sur votre Google favori vous en convaincra). Je vous renvoie en particulier à http ://www.perl.com/pub/q/documentation qui contient l'équivalent des pages man, ce qui est un atout important lorsque vous cherchez une fonction ou une syntaxe précise.

1.4 Hello World !

Vous devez disposer maintenant de PERL sur votre ordinateur. Vous devez sentir comme un besoin impétueux d'écrire votre premier programme. Pour taper un programme en PERL,

vous pouvez prendre n'importe quel éditeur de texte ASCII comme (x)emacs, vi(m), nedit, jed, nano, pico, kwriter... pour le monde UNIX, BSD, MAC OS X. Sachez cependant que certains éditeurs permettent de colorer la syntaxe du code, le rendant de ce fait plus lisible. Sous Windows, Notepad, winedt ou write feront dans un premier temps l'affaire. Si vous le souhaitez, vous pouvez utiliser le logiciel libre Open-PERL-IDE qui offre un véritable environnement de développement PERL sous Windows.

Nous commencerons par le traditionnel et indémodable "Hello World" :

```
1. #!/usr/bin/perl
2.
3. # premier exemple : Hello World !
4. print ("Hello World !\n") ;
```

Les numéros de lignes ne sont pas à recopier, ils ne servent que de repère pour les explications qui suivent. Prenez garde à la casse en recopiant ce programme. Majuscules et minuscules ne sont pas équivalentes en PERL. Sauvegardez ce fichier sous **helloworld.pl** (l'extension .pl désigne en général un script PERL). Pour exécuter ce script, vous avez plusieurs possibiltés :

- Sous UNIX-BSD, ouvrez un terminal et placez vous dans le répertoire contenant le fichier **helloworld.pl** et tapez la commande *perl helloworld.pl*. Une autre technique consiste à rendre ce script exécutable en tapant *chmod u+x helloworld.pl*. Pour exécuter ce script, il vous suffira alors d'un simple *./helloworld.pl*. Il se peut que vous obteniez un message d'erreur du type *sh : ./helloworld.pl : /usr/bin/perl : bad interpreter : No such file or directory*. Dans ce cas, vérifiez le chemin pour accéder à la commande *perl*. Ceci peut être fait grâce à la commande *whereis perl*, le premier chemin indiqué par cette commande devant être utilisé en lieu et place de */usr/bin* indiqué à la première ligne du script.
- Sous Windows, si vous avez tapez ce programme à l'aide de OpenPERL-IDE, il vous suffit de cliquer sur run. Pour les intrépides, vous pouvez aussi double cliquer sur le fichier. Mais en général, Windows ouvre et ferme immédiatement la fenêtre dans laquelle il a exécuté votre script. il est aussi possible de passer par la ligne de commande. Placez vous alors dans répertoire contenant votre script et tapez *perl helloworld.pl*.

Vous venez d'afficher un joli Hello World à l'écran. Essayons de comprendre le fonctionnement de ce programme très simple. Tout ce qui se situe après un dièse est considéré par PERL comme un commentaire. Les lignes 1 et 3 ne sont donc pas traitées. De même PERL ne tient pas compte des lignes vides et des tabulations, des espaces (qui ne sont pas inclus dans une chaîne de caractères bien sûr). Vous pouvez en user et abuser pour rendre votre code plus lisible. Pour être tout à fait rigoureux, la première ligne est utilisée dans le monde UNIX/BSD si vous souhaitez rendre votre script exécutable. Elle correspond à l'invocation de PERL pour interpréter le reste du script.

La quatrième ligne est la seule ligne non triviale. La commande permet d'afficher la chaîne de caractères entre guillemets. Notez que la ligne se termine par un point virgule comme dans d'autres langages (comme le C et le Java, ..). Ce caractère est impératif, c'est lui qui indique véritablement la fin de le ligne de commande et non le saut de ligne. Vous pouvez d'ailleurs concaténer plusieurs lignes de commandes sur la même ligne de texte, du moment que chacune d'entre elle est séparée par un point virgule (*print ("toto") ; print*

("tata");). Les parenthèses qui entourent la chaîne de caractères ne sont pas obligatoires, PERL est relativement souple sur sa syntaxe. Pour ceux qui viennent du monde C, Java, ... ou qui souhaitent y passer un jour, mieux vaut ajouter ces parenthèses (elles sont obligatoires dans ces autres langages).

Concernant le texte lui-même, il ne présente rien de particulier hormis le caractère \n qui signifie le saut de ligne. PERL utilise la même convention que le C concernant ce type de symbole. En particulier, pour obtenir un caractère \dans une chaîne de caractères, il faut écrire \\.

A partir de maintenant, vous devez être en mesure de recopier et d'exécuter n'importe quel programme écrit en PERL. Il ne vous reste plus qu'à étudier le reste de ces notes pour maîtriser ce langage.

Chapitre 2

Les variables

2.1 Généralités

Toute variable en PERL doit avoir un nom qui commence par le symbole $. Ainsi *$a=1* affecte la valeur 1 à la variable *$a*. Contrairement à d'autres langages, PERL n'est pas un langage typé. Ceci signifie que nous n'avons pas à déclarer au préalable quel type (entier, flottant, chaîne de caractères, ...) est associé à une variable. De même, le type de variable peut changer en cours d'exécution. Ainsi pour le morceau de programme suivant

```
1.  $a = 1 ;
2.  $a = "Newton" ;
```

Après la ligne 1, *$a* est un nombre valant 1, alors qu'après la ligne 2, *$a* est une chaîne de caractères contenant Newton.

Toute variable peut être affichée en la passant comme argument à l'instruction . Ainsi le programme suivant affiche le contenu de la variable *$a* à l'écran :

```
1.  #!/usr/bin/perl
2.
3.  $a = "Newton" ;
4.  print ($a) ;
```

2.2 Opérations de base avec les chaînes de caractères

L'opération la plus simple que nous puissions imaginer entre deux chaînes de caractères est l'opération de concaténation, c'est à dire l'ajout d'une chaîne à la suite d'une autre. L'opération est notée . en PERL et s'utilise ainsi :

```
1.  #!/usr/bin/perl
2.
3.  $a = "Hello" ;
4.  $b = "World" ;
5.  $c = $a.$b ;
6.  print ($c) ;
```

Ce progamme affiche HelloWorld qui est bien la concaténation de *$a* et de *$b*. Il n'est pas obligé de passer par la variable (temporaire) *$c* pour stocker le résultat de l'opération.

```
1.  #!/usr/bin/perl
2.
3.  $a = "Hello" ;
4.  $b = "World" ;
5.  print ($a.$b) ;
```

Le Hello World que nous venons d'afficher souffre de deux problèmes. Les deux mots qui le composent ne sont pas séparés par un espace. Un retour à la ligne serait aussi judicieux. Réparons de suite ces quelques défauts de jeunesse

```
1.  #!/usr/bin/perl
2.
3.  $a = "Hello" ;
4.  $b = "World" ;
5.  print ($a." ".$b."\n") ;
```

Comme vous le constatez, il est possible de chaîner plusieurs opérations de concaténation (comme nous ferions pour des additions) et directement utiliser des chaînes de caractères sans passer par des variables.

Un langage de programmation n'en serait pas un s'il n'était pas possible de faire des tests conditionnels et des branchements. Il est grand temps d'introduire le couple infernal . Il s'utilise de la façon suivante

```
1.  if (condition)
2.  {
3.  bloc d'instruction à exécuter si la condition est vérifiée
4.    }
5.  else
```

```
6.    {
7.  bloc d'instruction à exécuter si la condition n'est pas vérifiée
8.  }
```

L'indentation du code n'est pas obligatoire mais le rend beaucoup plus lisible. Comme dans les langages proche du C, tout bloc d'instruction doit être défini entre accolades. PERL est même plus strict que le C de ce point de vue puisque les accolades sont obligatoires même lorsque le bloc n'est composé que d'une seule ligne de commande. Le bloc d'instructions à exécuter si la condition n'est pas vérifiée n'est pas obligatoire et une syntaxe du type

```
1.  if (condition)
2.  {
3.  bloc d'instruction à exécuter si la condition est vérifiée
4.    }
```

est parfaitement légale. Il est aussi possible de considérer la négation d'une condition en plaçant le symbole ! devant celle-ci

```
1.  if (!(condition))
2.  {
3.  bloc d'instruction à exécuter si la condition n'est pas vérifiée
4.    }
```

Pour ceux qui se demanderaient s'il n'y a pas un problème de mise en page dans ces notes de cours, ne voyant pas le rapport entre ce qui précède et les chaînes de caractères, nous allons nous intéresser à la comparaison de ces dernières. Pour comparer deux chaînes, il y a six opérateurs

- teste si deux chaînes sont identiques.
- teste si deux chaînes sont différentes.
- (resp.) teste si la chaîne à gauche de l'opérateur est plus petite (resp. strictement plus petite) que celle à droite de l'opérateur.
- (resp.) teste si la chaîne à gauche de l'opérateur est plus grande (resp. strictement plus grande) que celle à droite de l'opérateur.

Ces opérateurs s'utilisent ainsi :

```
1.  #!/usr/bin/perl
2.
3.  $a = "toto" ;
4.  if ($a eq "toto") ;
```

```
5.   {
6.     print ("la chaîne de caractère vaut toto\n");
7.   }
8.  else
9.   {
10.    print ("la chaîne de caractère est différente de toto\n");
11.  }
```

Plusieurs remarques sont à faire concernant ces comparaisons. Elles sont sensibles à la casse : si dans notre programme précédent, *$a* contenait Toto, le test aurait échoué. Pour savoir si une chaîne est plus petite ou plus grande qu'un autre, PERL compare le code ASCII de chacun des caractères et s'arrête dès que la condition n'est plus vérifiée (ce n'est pas a priori un test sur la longueur de la chaîne).

Pour terminer ce rapide (et premier) tour des chaînes de caractères. Nous terminerons par deux fonctions utiles. La première s'appelle et permet d'enlever le caractère retour à la ligne si ce dernier est présent en fin de chaîne. Ainsi pour le code

```
1.  $a = "toto\n";
2.  chomp($a);
```

La variable *$a* contient après la ligne 2 la chaîne toto.

Dernier point, et non des moindres, est la technique pour récupérer le résultat d'une ligne tapée au clavier lors de l'exécution du script

```
1.  #!/usr/bin/perl
2.
3.  $a = <STDIN>;
4.  print ("vous venez de taper ".$a);
```

Affecter *<STDIN>* (STDIN désignant l'entrée standard) à *$a* permet de récupérer ce qui est tapé au clavier dans cette variable, jusqu'à ce que la touche entrée soit enfoncée. Le caractère de saut de ligne est inclus, il est donc souvent utile d'appliquer un *chomp* à la variable à laquelle on a effecté *<STDIN>*, en particulier si vous avez demandé un nombre.

2.3 Opérations de base avec les nombres

Bien que PERL n'ait pas vocation à être un langage permettant la création de logiciels scientifiques performants, il gère aussi bien que le C les nombres entiers et à virgule flottante. La précision des flottants est équivalente à celle des doubles en C (une mantisse avec 14 chiffres significatifs et un exposant pouvant aller de -308 à +308). Pour être totalement

rigoureux concernant les entiers (et ceci a son importance pour nous, les physiciens), PERL n'a pas vraiment de type entier en interne et travaille avec des double. Les entiers sont donc compris entre -10^{14} et 10^{14}. PERL possède tous les opérateurs binaires de base

- l'addition (+).
- la soustraction (−).
- la multiplication (∗).
- la division (/).
- le reste de la division entière ou modulo (%).
- la puissance (∗∗).

Les règles de priorité des ces opérateurs sont les même que celles dont nous avons l'habitude en mathématiques ou en C.

PERL possède aussi un certain nombre de fonctions numériques de base dont la liste exhaustive est donnée ci-dessous

- : valeur absolue.
- : partie entière.
- : racine carrée.
- : arctangente du rapport de deux nombres. atan2($y, $x) retourne l'arctangente de $y/$x (entre $-\pi$ et π).
- : cosinus.
- : sinus.
- : exponentielle.
- : logarithme en base néperienne.
- : retourne un nombre entre 0 et 1 si aucun argument n'est passé, ou entre 0 et $x si $x est l'argument passé.
- : fixe la graine du générateur de nombres aléatoires.

Pour ceux qui auraient de gros besoin (calcul formel, precision infinie, FFT, diagonalisation et même calcul quantique !), il existe de nombreuses extensions de ce type pour http ://www.cpan.org/modules/01modules.index.html (environ 150). Mais ne nous emportons pas, l'utilisation des modules n'est pas encore à l'ordre du jour.

La comparaison entre deux nombres se fait via des opérateurs différents de ceux utilisés dans le cas des chaînes de caractères :

- == teste si deux nombres sont égaux.
- ! = teste si deux nombres sont différents.
- <= (<) teste si le nombre à gauche de l'opérateur est plus petit ou égal (resp. strictement plus petit) que celui à droite de l'opérateur.
- >= (>) teste si le nombre à gauche de l'opérateur est plus grand ou égal (resp. strictement plus grand) que celui à droite de l'opérateur.

Pour ceux qui se poseraient la question de l'utilité de différencier les opérateurs de comparaison des chaînes et ceux des nombres, regardez le petit programme suivant

```
1.  #!/usr/bin/perl
2.
3.  $a = 4;
4.  $b = 23;
5.  if ($a < $b )
6.    {
7.      print ($a." est plus petit que ".$b."\n");
8.    }
```

```
 9.  else
10.    {
11.      print ($a." est plus grand que ou egal a ".$b."\n");
12.    }
13.  if ($a lt $b )
14.    {
15.      print ($a." est plus petit que ".$b."\n");
16.    }
17.  else
18.    {
19.      print ($a." est plus grand que ou egal a ".$b."\n");
20.    }
```

Voici le résultat que vous devez observer :

```
4 est plus petit que 23
4 est plus grand que ou egal a 23
```

Le résultat est surprenant pour un physicien mais pas pour un informaticien. La ligne 5 correspond à un test entre deux nombres. Le résultat de la comparaison est bien celui escompté. Par contre à la ligne 13, il s'agit d'une comparaison entre chaînes de caractères. PERL va donc commencer par convertir *$a* et *$b* en chaîne de caractères. De ce point de vue, le premier caractère de *$a* (4) est plus grand que celui de *$b* (2). Le test s'arrête à ce point et échoue. D'où le résultat que nous obtenons.

Parmi les structures de base faisant appel aux nombres, il y a les boucles. PERL possède l'équivalent des instructions et du C et consorts. Le petit programme suivant affiche dix fois Hello World :

```
1.  #!/usr/bin/perl
2.
3.  $a = 0;
4.  while ($a < 10)
5.    {
6.      print ("Hello World\n");
7.      $a =$a + 1;
8.    }
```

Le programme exécute le bloc d'instructions qui suit le tant que la condition entre parenthèses est vérifiée. Notez qu'il n'y a pas de point virgule après la condition.

Une remarque concernant la ligne 7. Ce type de commande qui consiste à incrémenter une variable peut se réécrire de plusieurs manières. Tout d'abord, il est possible d'utiliser l'opérateur += (soit *$a += 1*) qui ajoute la variable ou la constante située à droite de l'opérateur à la variable située gauche. Signalons que l'équivalent de cet opérateur existe pour les opérations mathématiques -, *, et / et l'opération de concaténation . . L'autre technique est d'utiliser la syntaxe *$a++* qui incrémente la variable *$a* de 1 (il existe aussi *$a-* - qui décrémente *$a* de 1).

Le même type de programme s'écrit de la façon suivante à l'aide de l'instruction *for*

```
1.  #!/usr/bin/perl
2.
3.  for ($a = 0; $a < 10; $a++)
4.    {
5.      print ("Hello World\n");
6.    }
```

La syntaxe est identique à celle du C. L'instruction est suivie par une expression entre parenthèses qui contient trois parties : l'initialisation (*$a=0*), la condition à vérifier pour continuer la boucle et l'instruction à exécuter à la fin de chaque itération (en général, un incrément). Chacune de ces parties est séparée de l'autre par un point-virgule. Après les parenthèses, nous retrouvons le bloc d'instuctions à exécuter à chaque itération.

2.4 Variables particulières

(N.B : cette section peut être passée en première lecture).
PERL possède quelques variables dont le nom est réservé. Le but ici n'est pas de les décrire toutes mais au moins dans donner la liste :

$ARG, $_, $1, $2, ..., $MATCH, $&, $PREMATCH, $‘, $POSTMATCH, $’, $LAST_PAREN_MATCH, $+, $^N, $, $INPUT_LINE_NUMBER, $NR, $., $LIST_SEPARATOR, $”, $SUBSCRIPT_SEPARATOR $SUBSEP, $;, $#, $ACCUMULATOR, $^A, $CHILD_ERROR, $?, ${^ENCODING}, $OS_ERROR, $ERRNO, $!, $EXTENDED_OS_ERROR, $^E, $EVAL_ERROR, $, $PROCESS_ID, $PID, $$, $REAL_USER_ID, $UID, $<, $EFFECTIVE_USER_ID, $EUID, $>,$REAL_GROUP_ID, $GID, $(,$EFFECTIVE_GROUP_ID, $EGID, $), $PROGRAM_NAME, $0,$[, $], $COMPILING, $^C,$DEBUGGING, $^D, $SYSTEM_FD_MAX, $^F, $^H, $INPLACE_EDIT, $^I, $^M, $OSNAME, $^O, ${^OPEN},$PERLDB, $^P, $LAST_REGEXP_CODE_RESULT, $^R, $EXCEPTIONS_BEING_CAUGHT, $^S, $BASETIME, $^T, ${^TAINT}, ${^UNICODE}, $PERL_VERSION, $^V, $WARNING, $^W, ${^WARNING_BITS}, $EXECUTABLE_NAME, $^X, $ARGV.*

Pour une description complète, vous pouvez consulter les pages man perlvar. Nous nous contenterons d'étudier la variable *$_* ou variable par défaut. Si vous utilisez une fonction (comme ou) qui nécessite a priori un argument et que vous l'omettez, PERL va utiliser la variable *$_* comme argument. Regardons le petit programme suivant :

```
1.  #!/usr/bin/perl
2.
3.  $_ = "essai\n" ;
4.  print ;
```

Le contenu de la variable par défaut est bien affiché. Quel est l'intérêt de cette variable ? Une des devises de PERL est "There's more than one way to do it", comprendre il y a plus d'une façon de le faire. Et il n'est pas rare de voir certains écrire leur code PERL de la façon la plus compacte possible. Certains font même des miracles en une seule ligne (les one-liners). Ce genre d'exploits montrera à quel point vous êtes un virtuose du PERL, mais à tendance à rendre le code complètement illisible et peut provoquer un isolement vis-à-vis de vos proches (y compris de vos collègues théoriciens).

2.5 Exercices

Calulez π par deux techniques

- Monté Carlo. Vous demanderez le nombre d'itérations à effectuer et vérifierez que le nombre obtenu est positif. Nous rappelons que cet algorithme consiste à tirer deux nombres aléatoires x et y et à incrementer une variable notée N si $x^2 + y^2 <= 1$. Le rapport entre N et le nombre d'itérations tend vers $\pi/4$.
- intégrale de $\int_0^{+\infty} dx \quad exp(-x^2/2)$ par la méthode des trapèzes. Vous demanderez le nombre de subdivisions et la borne supérieure. Vous vérifierez que les nombre obtenu sont valides.

Chapitre 3

Les tableaux

3.1 Généralités

Les tableaux en PERL sont identifiés par le symbole @ à l'image du $ pour les variables. Comme ces dernières, les tableaux ne sont pas typés et un même tableau peut mélanger des éléments de différents types. Comme en C, le plus petit indice d'un tableau est l'indice 0. Il n'est par contre pas nécessaire de déclarer la taille d'un tableau avant son utilisation.

Le petit programme suivant va servir à illustrer notre propos.

```
1.  #!/usr/bin/perl
2.
3.  @particule = ("photon", "boson", 0, 0, 1);
4.  print ("nom : ".$particule[0]."\n");
5.  print ("statistique : ".$particule[1]."\n");
6.  print ("masse : ".$particule[2]." MeV\n");
7.  print ("charge : ".$particule[3]."\n");
8.  print ("spin :".$particule[4]."\n");
```

La ligne 3 crée un tableau appelé particule. Pour initialiser ce dernier nous avons choisi une des techniques possibles qui consiste à lui affecter une liste d'éléments. La liste est définie entre parenthèses et chaque élément est séparé du suivant par une virgule.

Les choses se compliquent dans les lignes suivantes. En effet, pour accéder à l'un des éléments, la syntaxe n'est pas évidente. L'indice de l'élément auquel nous souhaitons accéder est ajouté entre crochets à la fin du nom du tableau (comme dans les langages ayant une syntaxe proche du C). En revanche le symbole @ est devenu $. La logique de ce changement se comprend par le fait que nous ne manipulons plus le tableau mais seulement un élément, soit une variable. Cette subtilité est la seule difficulté liée à la manipulation des tableaux.

Nous avons vu comment remplir un tableau en lui affectant une liste. Nous citerons deux autres méthodes. La plus évidente est l'affectation élément par élément :

```
1.  $particule[0] = "photon" ;
2.  $particule[1] = "bosons" ;
```

L'autre technique consiste à ajouter un élément à la fin du tableau grâce à l'instruction :

```
1.  push (@particule, "photon");
2.  push (@particule, "boson");
```

3.2 Parcourir un tableau

Si nous vous posions la question de la technique à employer pour parcourir successivement tous les éléments d'un tableau (du premier au dernier élément sans saut), l'idée qui vous viendrait immédiatement à l'esprit est d'avoir une variable allant de 0 au dernier indice du tableau. C'est effectivement une des techniques possibles :

```
1.  #!/usr/bin/perl
2.
3.  @energies = (1, 4, 9, 16, 25, 36, 49, 64);
4.  $i = 0;
5.  while ($i < 8)
6.    {
7.      print ("energie ".$i." : ".$energies[$i]."\n");
8.      $i++;
9.    }
```

A priori, le nombre d'éléments dans le tableau n'est pas connu au moment où nous écrivons le programme. La ligne 5 est particulière au cas que nous traitons. En toute généralité, nous pouvons connaître à tout instant le nombre d'éléments dans un tableau grâce à la syntaxe *$#nom_du_tableau*. Ainsi il serait judicieux de réécrire la ligne 5 comme *while ($i <= $#energies)*.

Il existe une autre technique, plus simple, pour parcourir un tableau. Ceci se fait grâce à l'instruction

```
1.  #!/usr/bin/perl
2.
3.  @energies = (1, 4, 9, 16, 25, 36, 49, 64);
4.  foreach $valeur (@energies)
5.    {
6.      print ($valeur."\n");
7.    }
```

A chaque itération, affecte à la variable (ici *$valeur*) la valeur suivante du tableau dont le nom est indiqué entre parenthèses. possède un seul défaut. Il n'est pas possible d'imbriquer un dans un d'un même tableau. PERL n'indiquera aucune erreur de syntaxe mais le programme n'aura pas le résultat escompté.

3.3 Trier les tableaux

Les tableaux peuvent être triés grâce à la commande . Cette commande est extrêmement puissante et nous ne donnerons ici que son utilisation de base, renvoyant le lecteur au chapitre 15 de [1] ou aux pages man (man perlfunc ou man perlfaq4).

Pour un tableau nommé *@tableau*, *sort (@tableau)* renvoie un tableau trié par ordre croissant des éléments de *@tableau*, les éléments étant comparés entre eux par les techniques de comparaison entre chaînes de caractères. Regardons le programme suivant :

```
1.  #!/usr/bin/perl
2.
3.  @particules = ("photon","electron","proton","neutron","muon","pion");
4.  @particulestriees = sort (@particules);
5.  foreach $tmp (@particulestriees)
6.    {
7.      print ($tmp." ");
8.    }
9.  print ("\n");
```

Cette technique est certes intéressante, mais en physique, nous aurons plus souvent affaire à des nombres et nous avons vu que trier des nombres comme des chaînes de caractères n'est pas une très bonne idée. Pour trier un tableau de nombre, il faut procéder comme suit

```
1.  sort {$a <=> $b} (@tableau);
```

Ce qu'il y a entre accolades et qui est situé entre l'instruction et le tableau à trier n'est ni plus ni moins qu'une petite fonction qui décrit comment comparer deux éléments *$a* et *$b* (ces noms de variables sont imposés par PERL pour une fonction de comparaison). La puissance de la fonction réside dans la possibilité de définir cette fonction. Pour ceux qui se poseraient la question, le symbole <=> est en fait un raccourci désignant la fonction qui compare deux éléments et qui retourne -1 si le premier est plus petit que le deuxième, 0 s'ils sont égaux et 1 sinon.

L'intérêt de cette technique va vous apparaître immédiatement si vous devez trier le tableau dans l'ordre décroissant. Il vous suffit pour cela d'inverser codeline$a et *$b* dans la fonction de tri. Mais vous pouvez faire bien plus !

(N.B. : le reste de cette section peut être réservée à une seconde lecture).

Si vous devez trier les éléments d'un tableau par rapport à un autre, il y a plusieurs méthodes possibles. Nous en décrirons une qui n'est certes pas la plus rapide mais qui a le mérite de laisser entrevoir les possibilités offertes par le fait que la fonction de comparaison soit redéfinissable.

```
1.  #!/usr/bin/perl
2.
3.  @masses = (938.3, 939.6, 0.511);
4.  @noms = ("proton", "neutron", "electron");
5.  @indices = (0, 1, 2);
6.  @indicestries = sort {$masses[$a] <=> $masses[$b]} (@indices);
7.  foreach $tmp (@indicestries)
8.    {
9.      print ($noms[$tmp]." (".$masses[$tmp]." MeV)\n");
10.   }
```

L'idée qui est employée dans ce programme est de créer un tableau contenant un indice associé à chaque particule (ligne 5). A la ligne 6, nous trions ce tableau non pas en comparant les éléments mais les masses associées à chacun de ces indices. Le tri s'effectue donc en comparant les masses, mais c'est bien le tableau d'indices qui est trié. Une fois le tableau d'indices trié, nous nous en servons pour afficher dans l'ordre croissant des masses les noms des particules.

3.4 Les tableaux associatifs : les tables de hash

(N.B. : cette section peut être réservée à une seconde lecture)

En règle générale, nous avons l'habitude de manipuler des tableaux à l'aide d'indices entiers positifs (strictement ou non suivant les langages). Pourtant, il peut s'avérer qu'une indexation par des chaînes de caractères pourait être beaucoup plus pratique. Si nous reprenons l'exemple des particules et de leur masse, il serait pratique d'indexer par rapport au nom. Les tables de hash sont là pour combler cette lacune. Regardons l'exemple suivant :

```
1.  #!/usr/bin/perl
2.
3.  $particules{"proton"} = 938.3;
4.  $particules{"neutron"} = 939.6;
5.  $particules{"electron"} = 0.511;
6.  while (($nom, $masse) = each(%particules))
7.    {
8.      print ($nom." (".$masse." MeV)\n");
9.    }
```

Le symbole qui définit la table de hash est le pourcent %. Pour accéder aux éléments

d'une table de hash, ce symbole devient un *$* comme c'est le cas pour un tableau à la seule différence que l'indice est indiqué entre accolades et non entre crochets. Ainsi, ligne 3, 4 et 5, nous affectons à la table de hash *%particules* les différentes masses en se servant des noms des particules comme indices. Pour parcourir la table de hash, nous utilisons la fonction . Cette dernière prend comme argument la table de hash et retourne au fur et à mesure l'indice (appelé généralement clé pour une table de hash) et la valeur correspondante dans un tableau à deux éléments. C'est ce que nous faisons ligne 6. Le tableau retourné par est récupéré dans *($nom, $masse)* qui permet d'affecter la clé à *$nom* et la valeur à *$masse*. Notez qu'en incluant cette affectation dans un comme nous l'avons fait, cela permet de faire une boucle jusqu'à ce que tous les éléments aient été itérés.

Si vous exécutez le programme, vous allez obtenir un affichage du type :

```
neutron (939.6 MeV)
proton (938.3 MeV)
electron (0.511 MeV)
```

Comme vous pouvez le constater, l'ordre dans lequel nous avons rentré les éléments n'est pas celui dans lequel ils apparaissent. PERL ordonne les éléments de la table de hash de la façon optimale pour lui d'effectuer une recherche. Ne présupposez donc jamais de l'ordre des éléments d'une table de hash.

Pour terminer ce rapide tour d'horizon des tables de hash, signalons deux commandes utiles et qui permettent de récupérer dans un tableau respectivement les clés et les valeurs d'une table de hash.

3.5 Tableaux réservés

(N.B. : cette section peut être réservée à une seconde lecture)

A l'image des variables réservées, il existe un certain nombre de tableaux et tables de hash réservés en PERL. En voici la liste exhaustive :

@_, @ARGV, @LAST_MATCH_END, @+, @LAST_MATCH_START, @-, @F, @INC, %INC, %ENV, %SIG

Pour le détail de ces tableaux, nous vous invitons à consulter la page man perlvar.

Nous regarderons plus particulièrement deux de ces tableaux. Il y a tout d'abord le tableau par défaut @_. Il s'agit du pendant de la variable par défaut pour les tableaux et s'utilise de la même manière. Le petit programme suivant vous montre l'utilisation du tableau par défaut :

```
1.  #!/usr/bin/perl
2.
3.  @_ = ("proton","neutron","electron");
4.  foreach ()
5.    {
6.      print ($_."\n");
7.    }
```

Le deuxième tableau, beaucoup plus intéressant, est *@ARGV*. Lorsque vous tapez une ligne de commande, vous pouvez ajouter des arguments généralement séparés par un espace. Chacun de ces arguments est passé au programme via le tableau *@ARGV*. Le petit programme suivant va vous montrer le fonctionnement de *@ARGV* :

```
1.  #!/usr/bin/perl
2.
3.  foreach $argument (@ARGV)
4.    {
5.      print ($argument."\n");
6.    }
```

Maintenant rendez exécutable votre script (que nous appelerons **argument.pl**) et exécutez le avec une liste d'arguments comme *./argument.pl essai 1 3.14*. Le programme affiche à la suite chaque argument.

Cette technique est très utile et vous permet d'éviter d'écrire en dur dans le script certains paramètres susceptibles d'être changés entre deux exécutions sans utiliser *<STDIN>*, la sortie standard n'étant pas forcément la méthode la plus adaptée pour scripter vos scripts, c'est-á-dire appeler vos scripts depuis un autre (ce qui pourrait passer pour du vice mais qui se révèle parfois pratique).

3.6 Exercices

- Créez un tableau de nombres aléatoires dont la taille sera passée comme premier argument. Puis cherchez le minimum et le maximum par deux méthodes : une rapide à exécuter et une autre plus lente mais s'écrivant de la manière la plus compacte possible.
- Reprendre l'exemple du tri croisé mais utilisez cette fois une table de hash pour réduire la taille du code et ne pas passer par un tableau d'indices.

Chapitre 4

Les fichiers

4.1 Tests de bases sur les fichiers

Si nous regardons notre connaissance actuel de PERL, nous avons déjà acquis une certaine compétence en programmation mais nous restons très limités dès qu'il s'agit d'échanges avec l'extérieur : nous savons récupérer des chaînes de caractères entrées au clavier et en afficher à l'écran. Ce chapitre va essayer de nous apprendre à effectuer des entrées/sorties dans les fichiers et à manipuler ces derniers.

Avant de lire ou d'écrire dans un fichier, il est en général intéressant de savoir si ce fichier existe, s'il est accessible en écriture... Pour cela, PERL offre plusieurs tests :

```
 1.  #!/usr/bin/perl
 2.
 3.  $fichier = "toto" ;
 4.  if (-e $fichier)
 5.    {
 6.      print ("le fichier ".$fichier." existe\n") ;
 7.      if (-d $fichier)
 8.        {
 9.          print ($fichier." est un repertoire\n") ;
10.        }
11.      else
12.        {
13.          print ($fichier." est un fichier\n") ;
14.        }
15.      if (-r $fichier)
16.        {
17.          print ($fichier." est accessible en lecture\n") ;
18.        }
19.      if (-w $fichier)
20.        {
21.          print ($fichier." est accessible en ecriture\n") ;
22.        }
23.      if ((-f $fichier) && (-x $fichier))
24.        {
25.          print ($fichier." est fichier executable\n") ;
```

```
26.         }
27.     }
28.   else
29.     {
30.       print ("le fichier ".$fichier." n'existe pas\n");
31.     }
```

Ce petit programme montre le panel des tests possibles sur un fichier dont le nom est ici donné par la variable *$fichier*. La ligne 4 teste si le fichier existe : *-e nom_du_fichier* retourne la valeur vrai si le fichier existe et faux dans le cas contraire. Le reste du programme montre d'autres tests possibles :

- *-d* teste si le fichier est un répertoire (ligne 7).
- *-f* teste si le fichier est bien un fichier et non un répertoire (ligne 23).
- *-r* teste si le fichier est accessible en lecture (ligne 15).
- *-w* teste si le fichier est accessible en écriture (ligne 19).
- *-x* teste si le fichier est exécutable (ligne 23).

Vous remarquerez que le de la ligne 23 contient deux conditions. C'est l'occasion de faire un petit retour sur l'écriture des conditions. Le symbole *&&* désigne le et logique entre deux conditions. Le ou logique s'écrit ||. Les règles de distributivité sont les même qu'en mathématiques. La seule chose à laquelle vous devez prêter attention lorsque vous combinez des conditions, est que ce sont de tests fainéants : ils sont a priori effectués de la gauche vers la droite et dès qu'une des conditions permet à PERL de savoir si le test n'est pas réalisé, il s'arrête alors et n'effectue pas le reste des tests. Dans l'exemple de la ligne 23, si notre programme est un répertoire, PERL s'arrêtera d'évaluer les conditions suivant *(-f $fichier)*. Si vous vous posez la question de savoir pourquoi nous effectuons un test si compliqué pour savoir si le fichier est exécutable, c'est qu'un répertoire est toujours considéré comme exécutable, il est donc préférable de les exclure de ce type de test.

Si vous exécutez ce programme, la question qui vient naturellement à l'esprit est de savoir où PERL va chercher le fichier. Si le nom de fichier ne contient pas le chemin absolu (c'est à dire depuis la racine / ou depuis le nom d'un lecteur sous Windows comme *C :*), le programme cherche alors uniquement à partir du répertoire courant, c'est à dire celui où vous avez exécuté le programme. Pour se placer dans un répertoire donné dans un programme, il suffit d'utiliser la commande qui prend comme argument le nom du répertoire. Une remarque si vous utlisez PERL depuis Windows, le séparateur entre nom de répertoire est le slash / et non le backslash \ afin de rendre vos scripts compatibles avec le reste du monde.

4.2 Lecture et écriture de fichiers

Maintenant que nous sommes en mesure de vérifier qu'un fichier est bien présent et accessible en lecture, nous pouvons le lire en toute tranquilité. Voici un exemple de lecture de fichier :

```
1.  #!/usr/bin/perl
2.
3.  open (INFILE, "test.txt");
4.  while ($ligne = <INFILE>)
```

```
5.    {
6.      print ($ligne) ;
7.    }
8.  close (INFILE) ;
```

La ligne 3 ouvre en lecture le fichier **test.txt**. Ceci se fait via la commande qui prend deux arguments. Le premier est le descripteur de fichier, il servira par la suite à identifier le fichier que nous allons ouvrir. Le second argument est le nom du fichier à ouvrir. Il est tout à fait possible d'utiliser une variable ou une combinaison de chaînes et de variables.

Le descripteur de fichier est en général indiqué en majuscule (bien que cela n'ait aucun caractère obligatoire). N'importe quel nom peut convenir (hormis s'il s'agit d'un nom réservé PERL), mais *INFILE* est très usité. Le descripteur de fichier est à rapprocher des descripteurs standards comme STDIN (entrée standard), STDOUT (sortie standard) ou encore STDERR (sortie standard d'erreurs).

L'opérateur diamant <> permet d'obtenir successivement toutes les lignes (c'est à dire les chaînes de caractères se terminant par un retour par un saut de ligne). Tel le Monsieur Jourdain de l'informatique, vous faisiez de l'opérateur diamant sans même le savoir avec *<STDIN>*. Ainsi la ligne 4 permet de récupérer ligne par ligne le contenu du fichier dans la variable *$ligne*. Notez que certaines versions de PERL peuvent vous demander une synatxe un peu plus sécurisée de la forme *while (defined($ligne = <INFILE>))*, la fonction permettant de vérifier qu'une variable (ici la valeur de retour de l'affectation) est bien définie. Cette fonction peut aussi s'avérer utile dans d'autres occasions comme la recherche dans une table de hash pour savoir si une valeur est associée à une clé, ou si un élément d'un tableau est défini.

Une fois que nous en avons terminé avec le fichier, il est nécessaire de le fermer, ce qui est fait ligne 8 grâce à l'instruction .

Notre programme souffre d'un léger défaut. S'il ne parvient pas à ouvrir le fichier, rien ne se produit, même pas un message d'erreur. Remédions à cela en remplaçant la ligne 3 par :

```
1.  unless (open (INFILE, "test.txt"))
2.    {
3.      die ("impossible de lire le fichier test.txt\n") ;
4.    }
```

L'instuction applique le bloc d'instruction qui la suit si la condition indiquée n'est pas vérifiée. Elle est donc équivalente à un *if (!(condition))*. L'instruction retourne la valeur faux si une erreur est survenue lors de l'ouverture du fichier. L'instruction permet d'arrêter net l'exécution du script en affichant un message (optionel) passé en paramètre. Afin de comprendre pourquoi un programme ne marche pas (et cela vous arrivera nécessairement), il est fortement conseillé d'au moins faire ce test basique lors de l'ouverture (il est possible de raffiner ce type de test mais nous n'aborderons pas ce point ici).

L'écriture dans un fichier est tout aussi simple. Voici un exemple qui va écrire les premiers factoriels dans un fichier

```
 1.  #!/usr/bin/perl
 2.
 3.  unless (open (OUTFILE, ">factoriel"))
 4.    {
 5.      die ("impossible de creer le fichier factoriel\n");
 6.    }
 7.  $fact = 1;
 8.  for ($iter = 1; $iter <= 10; $iter++)
 9.    {
10.      $fact *= $iter;
11.      print OUTFILE ($fact."\n");
12.    }
13.  close (OUTFILE);
```

Par rapport à l'exemple précédent, la seule différence, hormis le choix d'un autre nom tout aussi arbitraire que *INFILE* pour le descripteur de fichier, est la présence du signe > devant le nom du fichier. C'est ce symbole qui indique que le fichier doit être ouvert en écriture. Si un fichier portant le même nom est déjà présent, il sera alors écrasé (si vous en avez les droits). Si vous souhaitez écrire à la suite d'un fichier déjà existant, il suffit de remplacer le symbole > par >>. L'autre différence, si vous mettez de côté le calcul du factoriel) est la ligne 11. Pour écrire dans un fichier ouvert, il suffit d'intercaler entre un et la chaîne de caractères que vous souhaitez enregistrer, le descripteur de fichier. En réalité, lorsque vous n'indiquez aucun descripteur pour une fontion *print*, PERL suppose implicitement qu'il s'agit du descripteur de la sortie standard (*STDOUT*), c'est à dire la sortie écran. Ainsi un *print ("toto");* est équivalent à *print STDOUT ("toto");*.

4.3 Manipuler les fichiers

Nous avons maintes fois insisté sur l'une des forces des scripts tel que PERL face aux applications graphiques lorsqu'il s'agit de manipuler des nombreux fichiers ou d'effectuer dessus des tâches répétitives. Il est désormais temps d'en faire la preuve.

Tout d'abord, regardons comment obtenir la liste des fichiers contenu dans un répertoire. Voici comment se faire une petite commande *ls* (ou *dir* pour les amateurs de DOS) :

```
1.  #!/usr/bin/perl
2.
3.  foreach $fichier (<*>)
4.    {
5.      print ($fichier."\n");
6.    }
```

Si vous placez le symbole * dans l'opérateur diamant, vous obtenez un itérateur sur les fichiers du répertoire courant. Notez que nous avons utilisé pour cet itérateur, mais nous au-

rions aussi pu utiliser une boucle comme pour la lecture d'un fichier. L'étoile (ou opérateur joker/wildcard) est le même que celui que vous utilisez lorsque vous faites un *ls *.pl* pour filtrer uniquement les fichiers ayant l'extension pl. Il est d'ailleurs possible de filtrer les fichiers avec une extension donnée en utilisant la même technique. L'équivalent d'un *ls *.pl* s'écrit :

```
1.  #!/usr/bin/perl
2.
3.  foreach $fichier (<*.pl>)
4.    {
5.      print ($fichier."\n");
6.    }
```

Quatre autres méthodes utiles lors de la manipulation des fichiers/répertoires sont :
– permet de renommer un fichier. Elle s'utilise comme suit *rename(ancien_nom, nouveau_nom)*.
– permet d'effacer le fichier dont le nom est passé comme argument.
– permet de créer un répertoire dont le nom est passé comme argument.
– permet d'effacer un répertoire dont le nom est passé comme argument.

Toutes ces fonctions, comme ou , retournent la valeur faux si jamais une erreur est survenue au cours de l'opération. Notez que vous pouvez directement utiliser les outils de votre système pour effectuer ce genre de manipulation, mais vous perdez le côté portable d'un système à un autre de votre code. Ceci sera décrit au chapitre 8.

4.4 Exercices

1 Ecrivez un programme qui crée cent fichiers **testxx** (avec **xx** allant de 0 à 99), chaque fichier devant contenir son propre nom. Ensuite créez un deuxième programme qui va renommer chacun de ces fichiers de **testxx** en **totoyy** avec **yy = xx+100**.

2 Ecrivez un programme qui crée un fichier **donnees1.dat** dont chaque ligne est une valeur de l'échantillonage des nombres entre 0 et 1 avec un pas de 0.01. Faire un deuxième programme qui va lire le fichier précédent. Pour chaque ligne lue, le programme devra extraire la valeur numérique x contenue, calculer $cos(2\pi x)$ et stocker dans un fichier **donnees2.dat** une ligne composée de x et de la valeur de la fonction précédente évaluée en ce point (les deux nombres seront séparés par un espace).

Chapitre 5

Les expressions régulières

5.1 Reconnaissance de formes

Autant mettre tout de suite en garde le lecteur, voici la partie la plus ardue de PERL. Les dieux du "sed,awk,grep" n' ont rien à craindre. Pour les autres, soit ils périront en chemin, soit ils passeront maître dans l'art de l'expression régulière, devenant ainsi aux yeux du commun des mortels, des druides écrivant des formules invoquant on ne sait quel démon. Nous vous prévenons aussi qu'avec ce cours, vous n'aurez que le rang de druide et non celui de demi-dieu, il nous faudrait un chapitre aussi important en volume que l'intégralité de ces notes pour arriver à ce niveau.

Pourquoi dans ce cas s'infliger les expressions régulières ? Prenons un exemple concret. Comment savoir si une chaîne de caractères contenue dans une variable *$chaine* est composée d'un certain nombre de caractères suivie de quatre chiffres puis de l'extension .dat ? La réponse tient en une ligne :

```
1.  if ($chaine =~ /\d\d\d\d\.dat$/)
```

Ceux qui connaissent le C ou un langage assimilé, peuvent s'amuser à faire la même chose. Nous leur souhaitons bien du plaisir...

Une expression régulière permet de définir un motif, une description, que nous pourrons comparer à une chaîne donnée. Dans l'exemple que nous avons écrit, ce qui est entre les caractères divisé constitue l'expression régulière proprement dite. L'opérateur =~ compare la chaîne de caractères à gauche de l'opérateur avec l'expression régulière située à droite de l'opérateur.

Quelles sont les règles pour écrire une expression régulière ? Le symbole le plus simple dans une expression régulière est le point (.) qui signifie n'importe quel caractère hormis le retour chariot (\n). Vous pouvez aussi indiquer n'importe quel caractère alphabétique (en majuscule ou minuscule) ou chiffre. Ainsi */a.2/* est une expression régulière qui cherche un motif avec la lettre a en minuscule, suivie de n'importe quel caractère ASCII (hormis \n) puis du chiffre 2, Pour désigner n'importe quel chiffre, vous pouvez utiliser le code $\backslash d$, pour n'importe quel caractère alphabétique ou chiffre le code $\backslash w$ et pour n'importe quel type d'espace le code $\backslash s$. Si vous souhaitez rechercher le caratère point dans une expression régulière, vous vous retrouvez devant une certaine difficulté puisque ce symbole a une autre signification que nous avons vue précédemment. Pour désigner le point il faut utiliser \.. En

fait il en va de même de tous les caractères spéciaux qui ont presque tous une signifiation particulière dans une expression régulière. Il suffit de faire précéder le symbole souhaité par un backslash \ pour qu'il soit interprété comme un symbole et non comme une commande. En particulier, il faut utiliser \\ pour le symbole \.

Nous sommes maintenant en mesure de comprendre en partie notre exemple. La première partie de l'expression régulière *\d\d\d\d* recherche un nombre composé de quatre chiffres. Après ces derniers, nous demandons à ce qu'il y ait l'extension *.dat*.

Nous pouvons demander à ce qu'un caractère appartienne à un groupe donné. Pour cela, il suffit de désigner ce groupe entre crochets. Par exemple *[abcde]* désigne l'un des cinq premiers caractères de l'alphabet en minuscule. Une autre façon de l'écrire aurait été *[a-e]*, le signe moins permettant d'indiquer un intervalle de caractères. Si nous souhaitons uniquement savoir s'il s'agit d'un des cinq premiers caractères de l'alphabet sans se soucier de la casse, il suffit d'indiquer *[a-eA-E]*. Il est possible d'écrire la négation d'un groupe, en indiquant le signe ^ juste après le crochet ouvrant. Ainsi *[^a-eA-E]* correspond à n'importe quel caractère hormis les cinq premiers caractères de l'alphabet.

Il s'avère souvent utile d'indiquer si le motif que nous souhaitons traiter doit se trouver en début ou en fin de ligne. Ceci est particulièrement vrai si vous cherchez des fichiers avec une extension donnée. Le symbole ^ placé au début de l'expression régulière indique que la suite du motif doit impérativement se trouver en début de chaîne. Ainsi */^[aA]/* permet de vérifier que la chaîne commence par la lettre a en majuscule ou en minuscule. Si l'expression régulière se termine par un *$*, le motif qui précède ce signe permet de rechercher uniquement en fin de chaîne. Par exemple, */\.dat$/* permet de tester que la chaîne se termine par *.dat*.

Il est possible de rechercher plusieurs apparitions successive d'un caratère ou pseudo-caractère (comme *\d*, *[a-e]*, ...), il suffit pour cela de le faire suivre par l'un des symboles suivants :

- * zéro, une ou plusieurs apparitions du caractère
- + le caractère doit apparaître au moins une fois.
- ? le caractère doit apparaître au plus une fois.
- {*x*} le caractère doit apparaître x fois.
- {*x,y*} le caractère peut apparaître au moins x fois et au plus y fois. x ou y peuvent être omis si ils sont égaux respectivement à 0 ou l'infini.

Pour terminer notre (premier) périple au pays des expressions régulières, indiquons que le contenu d'une expression régulière peut être dynamique, c'est à dire donnée par une variable. Regardons un petit exemple :

```
1.  #!/usr/bin/perl
2.
3.  $chaine = "qhe001.dat" ;
4.  $expression = "\d*\.dat\$" ;
5.  if ($chaine =~ /\b$expression\b/)
6.    {
7.      print ($chaine." verifie l'expression reguliere ".$expression."\n") ;
8.    }
```

Il suffit d'indiquer la variable entre *\b* dans l'expression régulière pour que le contenu de la variable soit interprété comme un motif. Notez que dans la variable *$expression*, lorsque

nous voulons écrire le symbole $\d$ ou \., il faut doubler le backslash pour qu'au final, la chaîne contienne un seul backslash comme c'est le cas dans un argument de la fonction (rappelez vous qu'il ne s'agit, tant que nous sommes à la ligne 4, que d'une chaîne de caractères). De même, pour que le $ ne soit pas interprété comme le début d'un nom de variable, il est nécessaire de mettre un backslash devant.

5.2 Substituer un motif

Reconnaître un motif est un premier pas. Le deuxième consiste à remplacer la partie d'une chaîne vérifiant une expression régulière par une autre. L'exemple le plus utile est lorsque vous souhaitez renommer des fichiers de la forme *toto\d*\.dat* en *tata\d*\.dat* (vous devez maintenant suffisament connaître les expressions régulières pour comprendre cette notation) tout en en gardant le même numéro. Pour cela, PERL vous offre une fonction de substitution très puissante

```
1. #!/usr/bin/perl
2.
3. $fichier = "toto1984.dat" ;
4. $fichier =~ s/toto/tata/ ;
5. print ($fichier."\n") ;
```

La ligne 4 ressemble fortement à un test d'expression régulière si ce n'est le *s* avant l'expression et la présence d'un deuxième morceau indiquant par quoi remplacer le motif.

Si vous essayez maintenant le programme suivant

```
1. #!/usr/bin/perl
2.
3. $fichier = "toto1984toto.dat" ;
4. $fichier =~ s/toto/tata/ ;
5. print ($fichier."\n") ;
```

vous remarquerez que seul le premier toto a été remplacé. En fait PERL s'arrête dès que la première occurence du motif a été trouvé. Si vous souhaitez changer toutes les occurences d'un motif donné, il suffit d'ajouter un *g* après le dernier slash

```
1. $fichier =~ s/toto/tata/g ;
```

PERL arrête aussi la substitution s'il rencontre un saut de ligne. Si vous ne souhaitez pas que cela arrive vous pouvez ajouter un *m* après le dernier slash (qui peut venir compléter le *g* précédent mais peut aussi être seul).

Autre problème concret, vous souhaitez incrémenter le nombre que contient le nom du fichier. Il existe plusieurs techniques et voici la plus rudimentaire :

```
1. #!/usr/bin/perl
2.
3. $fichier = "toto1984.dat" ;
4. $fichier =~ s/toto//;
5. $fichier =~ s/\.dat//;
6. $fichier += 100;
7. $fichier = "toto".$fichier."dat" ;
8. print ($fichier."\n");
```

Vous pouvez très bien substituer le motif recherché par rien, ce qui revient à effacer son occurence dans la chaîne. C'est l'astuce que nous utilisons ligne 4 et 5 pour ne garder que le nombre inclus dans le nom du fichier. Pour la suite, vous pouvez remercier PERL de savoir traiter une variable contenant un nombre soit comme une chaîne de caractères soit comme un vrai nombre. Ceci nous permet d'effectuer une addition avec la variable *$fichier* puis concaténer cette dernière avec des chaînes pour obtenir le nouveau nom.

Nous avons annoncé qu'il y avait d'autres techniques, en voici encore une

```
1. #!/usr/bin/perl
2.
3. $fichier = "toto1984.dat" ;
4. $fichier =~ s/toto(\d*)\.dat/$1/;
5. $fichier += 100;
6. $fichier = "toto".$fichier."dat" ;
7. print ($fichier."\n");
```

Il est possible, lors de l'analyse par une expression régulière, de récupérer une partie contenue dans la chaîne qui vérifie le motif. C'est ce qui est fait ligne 4 (qui remplace les lignes 4 et 5 du précédent programme). Dans l'expression régulière, ce qui est vérifié par l'expression située entre parenthèses est stockée dans la variable *$1* (soit ici, le nombre inclus dans le nom de fichier). Vous pouvez multiplier le nombre de morceaux à récupérer. Vous obtiendrez ainsi des variables *$2, $3, ...* dont la valeur sera celle du morceau correspondant dans l'ordre d'apparition dans l'expression régulière.

Voici un exemple simple d'un programme qui pourra vous être très utile lorsqu'un programme vous demande un séparateur particulier (ici nous supposerons qu'il s'agit du symbole |) pour les fichiers de nombres alors que votre fichier d'origine se sert du caractère espace.

```
1. #!/usr/bin/perl
2.
3. $nomfichier = $ARGV[0];
```

```
 4.  $contenu = "" ;
 5.  unless (open (INFILE, $nomfichier))
 6.    {
 7.      die ("impossible d'ouvrir ".$nomfichier." en lecture\n") ;
 8.    }
 9.  while ($ligne = <INFILE>)
10.    {
11.      $ligne =~ s/ /\|/g ;
12.      $contenu .= $ligne ;
13.    }
14.  close (INFILE) ;
15.  unless (open (OUTFILE, ">".$nomfichier))
16.    {
17.      die ("impossible d'ouvrir ".$nomfichier." en ecriture\n") ;
18.    }
19.  print OUTFILE $contenu ;
20.  close (OUTFILE) ;
```

Pour chaque ligne du fichier d'entrée (lignes 8 à 12), le programme remplace les espaces par le symbole | (ligne 10) puis concatène la ligne à la fin de la variable *$contenu* qui contient donc le fichier avec les bons séparateurs. Le problème qui pourrait survenir à la première itération si la ligne 4 n'était pas présente, serait que la variable *$contenu* ne soit pas initialisée. Il serait donc impossible pour PERL d'effectuer une concaténation puisque le type ne serait pas encore défini. C'est pour cela que nous initialisons cette variable avec une chaîne vide ligne 4. Le reste du code doit être évident pour vous compte tenu de votre niveau en PERL.

5.3 Le duo comique split et join

Supposons que vous souhaitiez récupérer dans un tableau les différents éléments qui composent une ligne, chacun de ces éléments étant séparé des autres par un ou plusieur caractères spécifiques. Avec vos connaissances actuelles et une bonne dose d'aspirine, vous devriez pouvoir arriver tant bien que mal à vos fins. Heureusement PERL pense à votre bien-être et vous fournit une fonction dédiée à cet effet. La fonction remplit cette tâche de la façon suivante :

```
1.  @tableau=split(/ /, $ligne) ;
```

Le premier argument de est l'expression régulière qui permet de reconnaître le séparateur entre les éléments. Le deuxième argument est la ligne à découper. retourne le tableau correspondant.

Le compère de est la fonction qui fait exactement le travail inverse. Elle permet de réunir en une seule ligne, les éléments d'un tableau

```
1.  $ligne=join(" ", @tableau) ;
```

Le premier argument est la chaîne de caractères qui va servir de séparateur entre les éléments du tableau à concaténer.

Il faut prendre certaines précautions en utilisant . Une erreur classique est d'utiliser une ligne composée de nombres séparés par un caractère donné et qui possède un saut de ligne à la fin . Dans ce cas, le dernier élément du tableau retourné par sera affublé lui aussi d'un saut de ligne et ne pourra donc être traité comme un nombre. En général, un avant le est souvent salvateur.

Voici un exemple complet mettant en oeuvre qui a pour but de transformer un fichier de nombres composés de deux lignes en un fichier composé de deux colonnes :

```
 1.  #!/usr/bin/perl
 2.
 3.  $nomfichier = $ARGV[0] ;
 4.  unless (open (INFILE, $nomfichier))
 5.    {
 6.      die ("impossible d'ouvrir ".$nomfichier." en lecture\n") ;
 7.    }
 8.  $ligne = <INFILE> ;
 9.  chomp ($ligne) ;
10.  @valeurx = split (/ /, $ligne) ;
11.  $ligne = <INFILE> ;
12.  chomp ($ligne) ;
13.  @valeury = split (/ /, $ligne) ;
14.  close (INFILE) ;
15.  unless (open (OUTFILE, ">".$nomfichier))
16.    {
17.      die ("impossible d'ouvrir ".$nomfichier." en ecriture\n") ;
18.    }
19.  $iter = 0 ;
20.  foreach $ligne (@valeurx)
21.    {
22.      print OUTFILE ($ligne." ".$valeury[$iter]."\n") ;
23.      $iter++ ;
24.    }
25.  close (OUTFILE) ;
```

Ce programme ressemble beaucoup à celui présenté à la fin de la section précédente. Nous avons supprimé la boucle sur les lignes du fichier d'entrée pour n'extraire que les deux première lignes. Ceci est effectué à partir de la ligne 7 jusqu'à la ligne 12. Nous supposons, bien que cela soit mal, que le fichier possède bien deux lignes (aucun test d'erreur n'est effectué). Vous remarquerez l'utilisation du qui permet d'éviter quelques surprises. Vous

pouvez essayer de les enlever pour voir l'ampleur des dégâts.

5.4 Exercices

1 Ecrivez un programme qui recherche dans tous les fichiers ayant l'extension **txt** (vous vous assurerez qu'il s'agit bien d'un fichier), ceux qui contiennent une chaîne de caractères simple (uniquement des chiffres et des lettres sans espaces ni caractères spéciaux). Cette chaîne sera donnée par le premier argument du programme.

2 Ecrivez un premier programme qui crée un certain nombre de fichiers donné par le premier argument du programme, et ayant pour nom **qhexxx.dat** (**xxx** allant de 0 au nombre donné moins un). Chaque fichier contiendra son propre nom. Ecrivez un second programme qui recherche dans un répertoire tous les fichiers de la forme **qhe\d*\.dat** et qui les renomme en **fqhe\d*\.dat**.

3 Ecrivez un premier programme qui crée un fichier de nombres aléatoires organisés en x colonnes et y lignes (x et y seront passés en paramètres au programme). Ecrivez un deuxième programme qui cherche le maximum de la première colonne, puis divise tous les nombres par ce maximum et finalement remplace l'ancien contenu du fichier par les données renormalisées.

4 Dans le monde DOS/Windows, les fichiers textes utilisent deux caractères ASCII pour le saut de ligne ($\backslash r$ retour chariot et $\backslash n$ saut de ligne) alors que sous UNIX, il ne faut qu'un seul caractère ($\backslash n$). Le résultat de cette différence est, lorsque vous ouvrez un fichier texte DOS/Windows sous UNIX, l'apparition de symbole ^M en fin de ligne. Dans l'autre sens, votre fichier apparaît comme composé d'une seule ligne. Ecrivez un programme **dos2unix.pl** qui convertit un fichier dont le nom est donné par le premier argument de la commande, du standard DOS/Windows au format UNIX. Ecrivez un programme **unix2dos.pl** qui fait l'opération inverse.

Chapitre 6

Les fonctions

6.1 Généralités

Dès que votre programme dépasse la cinquantaine de lignes, il y a de fortes chances que vous puissiez mettre certaines partie du code dans des fonctions. Les fonctions sont impératives lorsque vous voulez faire des appels récursifs par exemple. Mais elles évitent surtout d'avoir du code redondant. N'oubliez jamais que si vous répétez plusieurs fois un ensemble d'instructions et qu'une erreur s'y est glissée, ce sera autant de corrections à faire (rappelez vous, jeunes apprentis, des trois qualités de l'informaticien).

Comment écrire une fonction en PERL ? A la fin de votre programme principal, il vous suffit de déclarer vos fonctions de la sorte :

```
1.  sub mafonction
2.    {
3.      instrutions;
4.      ...
5.    }
```

Vous pouvez mettre à la suite autant de fonctions que vous le souhaitez. Pour utiliser cette fonction, il vous suffit de l'appeler par son nom précédé du signe & (non obligatoire mais recommandé). Voyons ceci sur un petit exemple :

```
 1.  #!/usr/bin/perl
 2.
 3.  print ("debut de la recherche...\n");
 4.  &cherchefichier;
 5.  print ("fin de la recherche.\n");
 6.
 7.  sub cherchefichier
 8.    {
 9.      foreach $fichier(<*.tex>)
10.        {
11.          print ($fichier."\n");
```

```
12.       }
13.   }
```

Ce petit programme affiche tous les fichiers ayant l'extension **.tex** contenus dans le répertoire courant. La fonction *cherchefichier* se charge d'imprimer tous les fichiers ayant la bonne extension. Si vous souhaitez chercher de manière récursive dans tous les sous-répertoires à partir du répertoire courant, voici comment modifier la fonction *cherchefichier* :

```
 1.  #!/usr/bin/perl
 2.
 3.  print ("debut de la recherche...\n");
 4.  &cherchefichier;
 5.  print ("fin de la recherche.\n");
 6.
 7.  sub cherchefichier
 8.    {
 9.      foreach $fichier(<*>)
10.        {
11.          if (-d $fichier)
12.            {
13.              chdir ($fichier);
14.              &cherchefichier;
15.              chdir ("..");
16.            }
17.          else
18.            {
19.              if ($fichier =~ /\.tex$/)
20.                {
21.                  print ($fichier."\n");
22.                }
23.            }
24.        }
25.   }
```

A la ligne 11, dans la boucle qui regarde le contenu du répertoire courant, la fonction teste si le fichier en cours d'analyse est un répertoire. Si c'est le cas, la fonction se place dans ce répertoire (ligne 12) et s'appelle elle-même pour rechercher dans ce qui est maintenant le nouveau répertoire courant. Au retour, la fonction se replace dans le répertoire et l'exploration continue. Si le fichier analysé n'est pas un répertoire, la fonction regarde si l'extension est celle recherchée (ligne 18) et affiche le nom du fichier dans l'affirmative.

Comme dans la plupart des langages, il est possible de passer des paramètres à la fonction. Au moment de l'appel, il suffit d'indiquer ces paramètres entre parenthèses après le nom de la fonction, séparés par des virgules. Ces paramètres sont ensuite récupérés dans la fonction par l'intermédiaire du tableau par défaut, dans l'ordre dans lequel ils ont été passés.

De même, il est possible de retourner une valeur à la fin de l'exécution de la fonction. Le programme suivant va servir à illustrer notre propos :

```
1.  #!/usr/bin/perl
2.
3.  $ligne = "3.14 133 1.6" ;
4.  $ligne .= " ".&divise($ligne, 0, 2) ;
5.  print ($ligne."\n") ;
6.
7.  sub divise
8.    {
9.      @tableau = split (/ /, $_[0]) ;
10.     return ($tableau[$_[1]] / $tableau[$_[2]]) ;
11.   }
```

Ce petit script nous permet, à partir d'une ligne de texte contenant des nombres séparés par un espace, d'effectuer l'opération de division de l'un de ces nombres par un autre. A la ligne 3, nous définissons la ligne à traiter avec juste trois nombres (mais nous aurions pu en avoir bien plus). La ligne 4 fait appel à la fonction *divise* et concatène le résultat retourné à un espace puis à la variable *$ligne* rajoutant le résultat à la liste des nombres décrits par cette variable. Regardons de plus près la fonction divise. Le premier argument est la chaîne de caractères que nous souhaitons traiter. Elle est contenue dans la variable *$_[0]*. Grâce à la fonction , nous récupérons chacun des nombres dans le tableau *@tableau*. La commande retourne la valeur, variable ou tableau, qui la suit.Vous ne pouvez retourner qu'une seule valeur. Si vous voulez retourner plusieurs variables, il est nécessaire de les stocker au préalable dans un tableau puis de retourner ce dernier. Dans notre exemple, nous retournons le résultat de l'opération de la division du nombre dont la position est repérée par le contenu de la variable passée en second argument et qui est donc stockée dans *$_[1]*, par le nombre dont la position est repérée par le contenu de la variable passée en troisième argument (stocké dans *$_[2]*).

Pour ceux qui se risqueraient à passer un tableau, ils auront la désagréable surprise de voir les éléments de leur tableau inclus un par un dans le tableau par défaut : PERL insère le tableau lors du passage à une fonction. Pour passer un tableau en tant qu'entité, il est nécessaire de passer par les références que nous décrirons à la fin de ce chapitre.

6.2 Portée des variables

Pour ceux qui connaissent déjà d'autres langages, la question qui vient naturellement à l'esprit lorsque nous parlons des fonctions est celle concernant la portée des variables (dans cette section le terme variable couvrira aussi les tableaux, les tables de hash,...). Regardons le programme qui suit :

```
1.  #!/usr/bin/perl
2.
3.  $variable = 1.0;
4.  print ($variable."\n");
5.  &incremente();
6.  print ($variable."\n");
7.
8.  sub incremente
9.    {
10.     $variable++;
11.   }
```

Qu'affiche le programme ?

```
1
2
```

Nous observons que la variable *$variable* a été modifiée par la fonction. Par défaut, la portée des variables en PERL est globale : une variable définie à un endroit du programme peut être utilisée et modifiée à n'importe quel autre endroit, y compris dans une fonction. A l'opposée des variables globales, nous trouvons les variables locales dont l'utilisation ou la modification n'est possible que dans le bloc d'instruction où elle est définie et les sous-blocs de ce bloc. Par exemple, une variable locale définie dans la partie principale du programme n'est pas accessible depuis une fonction, la réciproque étant aussi valable. Cela permet d'utiliser à deux endroits différents le même nom de variable sans pour autant se rapporter au même contenu. Si cela ne vous parait pas limpide, regardez la version modifiée du programme précédent :

```
1.  #!/usr/bin/perl
2.
3.  my $variable = 1.0;
4.  print ($variable."\n");
5.  &incremente();
6.  print ($variable."\n");
7.
8.  sub incremente
9.    {
10.     my $variable = 1.0;
11.     $variable++;
12.   }
```

Le programme affiche maintenant

```
1
1
```

Vous pouvez constater que même si nous avons défini deux variables avec des noms identiques ligne 3 (dans la partie principale du programme) et 10 (dans la fonction *incremente*), il s'agit bien de variables distinctes. Pour définir une variable locale, il suffit, avant de l'utiliser, de la déclarer (comprendre indiquer son nom) avec l'instruction . Dans la déclaration, vous pouvez aussi effectuer une affectation comme nous l'avons fait dans notre exemple mais cela n'a pas de caractère obligatoire.

Pourquoi ne pas travailler qu'avec des variables globales ? La réponse sera détaillée au prochain chapitre et concerne la difficulté de gérer des variables globales dès que le programme atteint une certaine taille.

6.3 Passage par référence

(N.B. : cette section peut être réservée à une seconde lecture)

Si vous travaillez uniquement avec des variables locales, vous serez vite confronté au problème suivant : demander à une fonction de modifier directement le contenu d'une variable. La solution fait appel au passage par référence. Cette technique n'est pas évidente et rappellera pour certains le cauchemar des pointeurs en C. L'idée est de passer non plus la valeur, mais l'endroit où elle se situe. Regardons sur l'exemple suivant comment cela fonctionne :

```
 1.  #!/usr/bin/perl
 2.
 3.  my $variable = 1.0;
 4.  print ($variable."\n");
 5.  &incremente(\$variable);
 6.  print ($variable."\n");
 7.
 8.  sub incremente
 9.    {
10.      my $parametre = $_[0];
11.      $$parametre++;
12.    }
```

Si vous exécutez ce code, il affichera 1 puis 2. A la ligne 5, lorsque nous appelons la fonction *incremente*, nous avons mis un backslash devant le nom de la variable que nous voulons passer. Par cette syntaxe, nous passons une référence sur la variable et non la variable elle-même. Dans la fonction, nous commençons par récupérer cette référence dans la variable *$parametre*. Après cette affectation, la variable contient une référence et non le contenu

de la variable passée par référence. Ceci signifie que vous ne pouvez utiliser directement *$parametre*. De façon imagée, vous pouvez supposer que *$parametre* contient l'adresse ou le numéro de casier où se situe la variable *$variable*.

A la ligne 11, nous souhaitons incrémenter la variable référencée par *$parametre*. Pour pouvoir accéder à la variable elle-même, c'est à dire pour pouvoir agir sur le contenu du casier, il faut indiquer à PERL de quel type de variable, il s'agit : variable simple, tableau, table de hash,... Pour cela, il suffit de rajouter le symbole correspondant devant la référence, ce qui explique le double signe $ devant le nom de la référence (un pour la référence et un pour indiquer le type). Ainsi *$$parametre* devient strictement équivalent à *$variable*.

Dans le cas d'un tableau, voici comment se passe le passage par référence :

```
 1.  #!/usr/bin/perl
 2.
 3.  my @tableau = (1, 4, 9, 16);
 4.  print (join(" ", @tableau)."\n");
 5.  &ajoute(\@tableau);
 6.  print (join(" ", @tableau)."\n");
 7.
 8.  sub ajoute
 9.    {
10.      my $parametre = $_[0];
11.      push (@$parametre, 25);
12.      $$parametre[5] = 36;
13.    }
```

Les références ont d'autres utilités que celle de permettre de donner accès à des variables locales externes à des fonctions. Vous pouvez tout à fait stocker des références dans un tableau (ou une table de hash). En particulier, vous pouvez faire en sorte que chaque élément d'un tableau soit une référence vers un autre tableau. Si vous avez compris les références, cette dernière phrase doit sonner dans votre tête comme une révélation : vous venez de créer un tableau multidimensionnel !

```
 1.  #!/usr/bin/perl
 2.
 3.  my @matrice;
 4.  for ($indexi = 0; $indexi < 10; $indexi++)
 5.    {
 6.      my @colonne;
 7.      for ($indexj = 0; $indexj < 10; $indexj++)
 8.        {
 9.          push (@colonne, $indexi + $indexj);
10.        }
11.      push (@matrice, colonne);
12.    }
13.
```

```
14.  for ($indexi = 0; $indexi < 10; $indexi++)
15.    {
16.      my $reference = $matrice[$indexi];
17.      for ($indexj = 0; $indexj < 10; $indexj++)
18.        {
19.          print ($$reference[$indexj];
20.        }
21.      print ("\n");
22.    }
```

Ce programme est composé de deux séquences. Des lignes 3 à 12, nous créons le tableau à deux dimensions (i.e. une matrice). *@matrice* va contenir les réferences des différentes colonnes de notre matrice. A chaque itération sur les colonnes (*$indexi*), nous commençons par créer un tableau *@colonne* que nous remplissons. Nous ajoutons ensuite la référence sur ce tableau à *@matrice*. Pour parcourir la matrice (lignes 14 à 22), nous commençons par récupérer la référence (ligne 16) puis nous pratiquons de la même manière que celle indiquée dans le programme précédent pour récupérer les éléments du tableau.

Bien que l'envie soit grande d'user et d'abuser des tableaux multidimensionnels, nous indiquons au lecteur que nous ne pourrions être tenus responsables des migraines provoquées par l'utilisation excessive de ces derniers. Bien que puissante la notion de référence n'est pas la plus aisée à comprendre et à manipuler.

6.4 Exercices

1 Reprendre l'exercice 2 du chapitre 5 qui consistait à renommer les fichiers en effectuant aussi l'opération dans tous les sous-répertoires.

2 Faire une fonction de recherche de minimum et maximum dans un tableau. Les valeurs seront retournées en utilisant soit les référence soit un tableau. Vous pourrez utiliser tester cette fonction en l'insérant dans le programme de l'exercice 3 du chapitre5.

Chapitre 7

Hygiène et programmation : les presque 10 commandements

7.1 Remarque préliminaire

Comme nous l'avons déjà signalé, ces notes offrent volontairement une vision tronquée de PERL. Mais ce chapitre est sûrement celui qui est le plus polémique. Nous allons vous présenter ce qui constitue, **de notre point de vue**, les règles aidant à la programmation. Vous n'êtes en aucun cas obligé de les suivre et si vous discutez avec d'autres programmeurs en PERL, vous pourriez même obtenir des avis allant à l'encontre de ceux donnés ici. A vous de faire votre choix et de garder ce qui vous convient. Note aux thésards et autres intermittents de la science : pour obtenir un poste permanent, veillez à ne surtout pas respecter ces commandements. votre code ne pourra être utilisable et maintenu que par vous. Ainsi vous vous rendrez ainsi indispensable, ce qui garantira votre embauche.

7.2 1er commandement : les warnings tu utiliseras

Lorsque vous appelez un programme PERL mieux vaut demander d'afficher les erreurs non critiques ou avertissements (warnings). En général, ces avertissements vous indiqueront ce qui est à coup sûr un problème de votre programme, sans pour autant entraver son exécution. Regardons ce petit exemple :

```
1.  #!/usr/bin/perl
2.
3.  $a = 1.2;
4.  $b = "test" ;
5.  $b += $a;
6.  print ($b."\n");
```

A priori, ce programme a un petit problème d'arithmétique. Pourtant, aucune erreur n'est indiquée (il affiche 1.2). Si maintenant vous ajoutez un -w après la commande *perl* soit à la première ligne du programme pour ceux qui on l'habitude de rendre leur script exécutable, soit au moment de l'exécution du script (*perl -w warning.pl*) soit encore via les options de

votre IDE (Open-PERL-IDE, KDevelop,...), vous obtiendrez le message suivant

```
Argument "test" isn't numeric in addition (+) at warning.pl line 5.
1.2
```

Il est donc plus que fortement conseillé d'être en mode warning pour éviter de chercher les erreurs triviales de votre programme.

7.3 2ème commandement : des noms compréhensibles à tes variables et fonctions tu donneras

PERL ne possède que très peu de limitation concernant le nombre et le type de caractères qui composent les noms de variables ou de fonctions. Alors n'hésitez pas à choisir ces derniers de la façon la plus explicite possible, surtout s'il s'agit de variables non triviales ou cruciales. Certes, certains pourront se moquer de vos noms à rallonge. Mais si vous devez vous replonger dans un code plusieurs mois après l'avoir écrit, ces railleries vous sembleront dérisoires comparées à l'avantage que cela vous procure.

En particulier, un nom peut très bien être composé de plusieurs mots. Il existe plusieurs techniques pour séparer ces mots : vous pouvez utiliser le caractère souligné _ ou la notation hongroise (chaque mot commençant par une majuscule). Eviter aussi les abréviations : si elles sont évidentes pour vous, elles ne le seront pas forcément pour celui qui reprendra votre programme dans trois ans.

7.4 3ème commandement : les valeurs en dur tu éviteras

Un programme PERL a souvent besoin d'information concernant des noms de fichiers, des tailles, des chaînes de caractères ... La tendance naturelle est d'écrire ces informations directement dans le script. Cela constitue la pire des techniques. Il n'est pas rare de vouloir changer un paramètre et il faut dans ce cas là éditer le script, repérer tous les endroits où ce paramètre apparait ... L'idéal est donc de fournir ces informations au moment de l'exécution, en utilisant *STDIN* pour demander à l'utilisateur, *@ARGV*, voire dans certains cas un fichier de configuration qui sera analysé au début de l'exécution de votre programme (après tout, l'analyse de texte est un domaine dans lequel PERL excelle).

Au pire, veillez à regrouper ces quelques paramètres au début du programme où vous les stockerez dans des variables adéquates. La meilleure technique est en fait de tout mettre dans une table de hash, ce qui permet de manipuler un seul objet contenant toute la configuration.

7.5 4ème commandement : des fonctions tu abuseras

Comme votre principale activité (hormis pour certains acharnés) est la physique, moins vous passerez de temps à programmer, plus la science avancera. Les fonctions ont plusieurs avantages. Elles forment des morceaux de code testés que vous pourrez très facilement réutiliser. Elles permettent d'éviter la redondance de code toujours sujette à des bugs plus importants (corrections ou modifications que vous oubliez de répercuter dans les autres blocs). Elles facilitent aussi la lecture du code évitant d'être distrait par des parties non

sensibles de l'algorithme. En bref les fonctions sont vos alliés les plus puissantes, servez-vous en.

7.6 5ème commandement : tes fonctions tu documenteras

Vouloir réutiliser des fonctions est une idée louable qui peut malheureusement tourner au cauchemar si vous ne les avez pas documentées. Les commentaires à l'intérieur de la fonction elle-même ne sont pas en général les plus utiles : si vous devez vous plonger dans le corps de la fonction proprement dit pour pouvoir l'utiliser, ce qui nécessite beaucoup de temps, vous ne gagnerez rien. De plus, un programme bien codé ne nécessite que peu de commentaires pour être compréhensible.

Il est important de documenter la fonction de telle sorte que vous puissiez savoir immédiatement comment l'utiliser, c'est à dire ce qu'elle fait, quels sont les paramètres à passer et que retourne votre fonction. Voici une façon parmi d'autres de rédiger une documentation minimaliste :

```
 1.  # fonction qui additionne deux nombres
 2.  #
 3.  # $_[0] = premier nombre
 4.  # $_[1] = deuxieme nombre
 5.  # valeur de retour = somme des deux nombres
 6.
 7.  sub Addition
 8.    {
 9.      return ($_[0] + $_[1]);
10.    }
```

Certes, cet exemple est un peu pathologique (vous vous doutez bien de ce qu'il faut passer et ce que la fonction va retourner) mais en règle générale cela vous évitera une énorme perte de temps. Même sur une fonction aussi petite, ayez le réflexe de documenter : c'est une bonne habitude à prendre. N'hésitez pas à indiquer d'éventuelles contraintes sur les paramètres d'entrée et la valeur de retour afin d'éviter tout comportement anormal avec un mauvais jeu d'arguments.

7.7 6ème commandement : les variables par défaut tu banniras

Même si il est possible de trouver pratique les variables par défaut, elles peuvent être source d'erreur et dans tous les cas, rendent votre code moins lisible (surtout pour les non-initiés). En particulier, dans une fonction, nous conseillons vivement dans les premières lignes de celles-ci, de récupérer les arguments qui ont été placés dans le tableau par défaut et de les placer dans des variables avec des noms explicites. Cela facilitera grandement la compréhension du code de votre fonction, surtout si celle-ci dépasse la vingtaine de lignes.

7.8 7ème commandement : les variables globales tu éviteras

Les variables globales ont tendance à rendre votre programme plus difficile à maintenir dès qu'il atteint une certaine taille. Pour un bon usage de PERL, mieux vaut ajouter la ligne *use strict 'vars';* en début de programme (après la ligne d'invocation de PERL pour les unixiens/bsdiens). Ceci permet d'interdire l'usage des variables globales et nécessite donc de déclarer toutes les variables comme locales. L'effet de bord de l'utilisation de *strict* est que vous évitez le type d'erreur suivant :

```
1.  #!/usr/bin/perl
2.
3.  $variable = "test\n" ;
4.  print ($varable) ;
5.  print ("erreur\n") ;
```

Nous avons juste fait une faute de frappe ligne 4. Le programme n'affiche que le message *erreur*. Si maintenant vous ajoutez *use strict 'vars';* :

```
1.  #!/usr/bin/perl
2.
3.  use strict 'vars' ;
4.
5.  my $variable = "test\n" ;
6.  print ($varable) ;
7.  print ("erreur\n") ;
```

Vous obtenez le message suivant :

```
Global symbol "$varable" requires explicit package name at strict.pl line 6.
Execution of strict.pl aborted due to compilation errors.
```

Notez qu'un warning vous aurait aussi permis de détecter cette erreur mais aurait tout de même exécuté le reste du programme, ce qui peut être dramatique comme dans l'exemple qui suit :

```
1.  #!/usr/bin/perl -w
2.
3.  $document = "" ;
```

```
 4.  open (INFILE, "test");
 5.  while ($ligne = <INFILE>)
 6.    {
 7.      $ligne =~ s/toto/tata/g;
 8.      $document .= $ligne;
 9.    }
10.  close (INFILE);
11.  open (OUTFILE, ">test");
12.  print OUTFILE $docment;
13.  close (OUFILE);
```

Ce petit programme remplace tous les mots *toto* par *tata*. Le contenu du fichier est temporairement stocké dans *$document*. Le problème est qu'au moment de sauvegarder les modifications, nous enregistrons le contenu de *$docment* qui n'est pas défini. Au moment de l'exécution, vous obtenez un warning mais le contenu de votre fichier est tout de même remplacé par du vide. Vous venez ainsi d'effacer votre fichier !

7.9 8ème commandement : ton programme tu aèreras

Comme nous l'avons déjà souligné, PERL se moque des espaces, tabulations et autres sauts de ligne dans l'écriture du programme. Il faut vraiment utiliser cet avantage pour donner un aspect visuel qui permet en un clin d'oeil de voir la structure du code. Si vous avez un bon éditeur, l'indentation des blocs d'instructions est automatique. Cela permet en particulier de deviner directement à quel sous-bloc appartient une ligne d'instruction. Séparez nettement les fonctions les unes des autres pour bien en voir les limites. Une autre technique qui devient très utile lorsque vous utilisez des opérateurs est d'avoir des espaces entre les opérateurs et les données. Cela vous permettra de repérer plus rapidement ce que fait votre ligne de code.

7.10 9ème commandement : des commentaires tu ajouteras ?

Vous entendrez souvent dire qu'il faut mettre des commentaires dans votre programme. Il faut distinguer ceux qui permettent de documenter le code (ceux qui apparaissent au début des fonctions) de ceux qui sont véritablement dans les blocs d'instruction. En règle générale, si vous avez suivi les précentes recommandations (commandement 2, 5 et 8) cela n'est vraiment pas utile sauf si certaines parties de votre code deviennent trop longues (de l'ordre de la centaine de lignes). Un commentaire peut alors permettre d'aider à comprendre l'articulation. Préférez dans ce cas un redécoupage en fonction.

7.11 Où est passé le 10ème commandement ?

Lorsque le Tout Puissant Tux déclama les dix commandements du programmeur en PERL, le peuple trouva le dernier commandement tellement ennuyeux à respecter qu'il refusa de le graver dans le silicone. Les historiens s'accordent pour dire que ce commandement concernait la rédaction de documentation. Après tout, à quoi bon faire un programme si

personne, ni même son concepteur après six mois de non utilisation, ne peut l'utiliser sans se plonger dans le code ?

Malheureusement, c'est une tâche particulièrement ingrate, surtout quand l'informatique n'est pas votre activité principale. Voilà pourquoi ce qui était le plus important des commandements est tombé dans l'oubli.

Chapitre 8

PERL maître du monde

8.1 Utilisation de PERL depuis d'autres programmes et des modules

A priori, PERL est un programme comme un autre. Il est donc possible de l'appeler depuis un autre logiciel si celui-ci le permet. Vous pourrez ainsi étendre les possibilités de ce dernier en utilisant PERL surtout si vous profitez des nombreux modules de notre langage de script préféré.

Dans la suite de cette section, nous allons décrire la solution à un petit problème d'expérimentateurs. Un frigo à dilution est piloté par un PC sous Windows avec LabView. Pour ne pas gâcher tous leurs week-ends, ils souhaitaient savoir à distance si le frigo fonctionne correctement. Le logiciel sous LabView qui pilote l'expérience permet d'enregistrer deux images **image1.jpg** et **image2.jpg** qui donnent toutes les informations nécessaires. Nous allons donc demander à ce dernier d'enregistrer ces deux images à intervalle régulier puis d'appeler un petit programme PERL qui va se charger :

- de créer une page web **index.html** avec l'heure et la date et qui affiche deux images enregistrées par LabView.
- d'envoyer **index.html**, **image1.jpg** et **image2.jpg** sur un serveur web par FTP.

Commençons par la fonction qui va se charger de créer la page HTML :

```
 1.  # cree la page web index.html qui doit etre envoyee par FTP
 2.  #
 3.
 4.  sub CreationPageWeb
 5.    {
 6.      my @Date = gmtime(time);
 7.      $Date[5] += 1900;
 8.      $Date[4] += 1
 9.      my $Jour = $Date[3]."/".$Date[4]."/".$Date[5];
10.      my $Heure = $Date[2]."h".$Date[1];
11.      unless (open (OUTFILE, ">index.html"))
12.        {
13.          die ("ne peut creer le fichier index.html\n");
14.        }
15.
```

```
16.     print OUTFILE ("<html>
17.  <body>
18.  page cr&eacute;&eacute;e le ".$Jour." &agrave;".$Heure."<br />
19.  <img src=\"image1.jpg\">image 1</img><br />
20.  <img src=\"image2.jpg\">image 2</img><br />
21.  </body>
22.  </html>\n");
23.     close(OUTFILE);
24.   }
```

Comme vous pouvez le constater, cette fonction n'a besoin d'aucun paramètre et ne retourne aucune valeur. Ligne 6 nous commençons par récupérer la date. Pour cela nous faisons appel aux fonctions et . retourne le temps écoulé en seconde depuis le 1er janvier 1970 à 0h0min0s. Ceci est loin d'être la solution la plus commode pour lire la date. C'est pour cela que nous passons le résultat de à . permet d'obtenir à partir de la date au format fournie par un tableau qui contient :

- élément 0 : les secondes
- élément 1 : les minutes
- élément 2 : les heures
- élément 3 : le jour du mois
- élément 4 : le mois (0 pour janvier, ...)
- élément 5 : le nombre d'année depuis 1900

retourne d'autres éléments que nous ne détaillerons pas ici. Notez que fonctionne dans le fuseau GMT. En général, il est préférable d'utiliser qui fait exactement la même tâche mais dans le fuseau local (à condition que votre ordinateur soit bien configuré).

Ligne 9 et 10 nous formons les chaînes de caractères associées à l'heure et au jour. Ensuite nous passons à la création du fichier HTML proprement dit. Le contient une longue chaîne qui décrit le fichier. Si vous ne connaissez pas le HTML, ce n'est pas grave. En revanche si vous connaissez trop bien le HTML, ne nous en voulez pas de ne pas être respectueux de la norme HTML 4.01 strict.

Et voici maintenant la partie principale du programme :

```
 1.  #!/usr/bin/perl -w
 2.
 3.  use strict 'vars';
 4.  use Net : :FTP;
 5.
 6.  my $Serveur = "webserver.world.fr";
 7.  my $Repertoire = "public_html/test";
 8.  my $Utilisateur = "einstein";
 9.  my $MotDePasse = "*******";
10.
11.  &CreationPageWeb();
12.  my $Ftp = Net : :FTP->new($Serveur);
13.  $Ftp->login($Utilisateur, $MotDePasse);
14.  $Ftp->cwd($Repertoire);
```

```
15. $Ftp->put("index.html");
16. $Ftp->put("image1.jpg");
17. $Ftp->put("image2.jpg");
18. $Ftp->quit();
```

Pour utiliser le protocole FTP, nous allons nous servir d'un module PERL nommé Net : :FTP. Vous trouverez toute la documentation concernant ce module sur le site http ://search.cpan.org/~gbarr/libnet-1.19/Net/FTP.pm. Pour utiliser un module, il faut bien sûr que celui-ci soit installé. Pour demander à PERL d'utiliser le module, il suffit d'indiquer au début du programme la commande suivie du nom du module, comme cela est fait ligne 4. Si jamais vous obtenez un message d'erreur du type :

```
Can't locate Net/FTP.pm in @INC (@INC contains : /etc/perl
/usr/lib/perl5/site_perl/5.8.4/x86_64-linux /usr/lib/perl5/site_perl/5.8.4
/usr/lib/perl5/site_perl/5.8.2/x86_64-linux /usr/lib/perl5/site_perl/5.8.2
/usr/lib/perl5/site_perl /usr/lib/perl5/vendor_perl/5.8.4/x86_64-linux
/usr/lib/perl5/vendor_perl/5.8.4 /usr/lib/perl5/vendor_perl/5.8.2/x86_64-linux .)
at ./ftp.pl line 4.
BEGIN failed–compilation aborted at ./ftp.pl line 4.
```

C'est que Net : :FTP n'est pas installé sur votre système. Vous pouvez consulter les http ://www.cpan.org/misc/cpan-faq.html pour obtenir une aide à l'installation des modules quelque soit votre OS.

Comme nous sommes bien éduqués, toute la configuration se trouve au début du fichier dans les quatres variables définies de la ligne 6 à la ligne 9. *$Serveur* contient le nom du serveur, *$Repertoire* le nom du répertoire sur le serveur où seront stockés les fichiers, et enfin *$Utilisateur* et *$MotDePasse* le nom et mot de passe de l'utilisateur qui possède le compte associé au site web sur le serveur.

Ligne 11, nous créons la page web. Puis ligne 12, nous créons une connexion FTP avec le serveur. Le reste du programme doit être famillier pour ceux qui utilisent FTP en ligne de commande. La ligne 13 permet de s'identifier auprès du serveur (commande *login* de FTP), la ligne 14 permet de se placer dans le bon réperoire (*cwd* de FTP). Puis nous plaçons les trois fichiers sur le serveur grâce à *put*.

Ce programme est la version la plus simple que nous puissions écrire. Charge à vous de l'améliorer pour ajouter un historique par exemple.

La force de ce programme réside dans l'utilisation du module Net : :FTP. Comme nous l'avons souligné précédemment, PERL possède des centaines de modules disponibles. La principale difficulté est de savoir le nom du module qui effectue la tâche que vous souhaitez voir remplir. Pour cela, votre tâche suivie de PERL et de module dans votre moteur de recherche devrait assez rapidement vous founir le nom du module. Il suffit d'aller ensuite sur le site du http ://search.cpan.org pour trouver la documentation associée et des exemples.

8.2 Etendre PERL avec d'autres programmes

Bien que PERL possède une vaste bibliothèque de fonctions et de modules, il s'avère souvent utile de faire appel à un programme externe. En effet, un programme peut donner accès à des capacités qui ne sont pas couvertes par les modules. Vous pouvez aussi ne pas vouloir apprendre à utiliser une nouvelle fonctionnalité et préférer utiliser un logiciel que vous maîtrisez. Voici la technique la plus simple pour appeler un programme depuis PERL :

```
1.  #!/usr/bin/perl -w
2.
3.  use strict 'vars';
4.
5.  my $Sortie = `pwd`;
6.  print ($Sortie);
```

A la ligne 5, nous faisons appel à la commande *pwd*. Pour exécuter un programme, il suffit d'indiquer la ligne de commande entre guillemets inversés (comprendre accent grave dans la langue de Molière). Il est possibe de récupérer ce que le programme affiche normalement via la sortie standard (i.e. ce qui apparaît dans le terminal) dans une chaîne de caractères. Ainsi la variable *$Sortie* va contenir le nom du répertoire courant (avec le caractère saut de ligne, puisque c'est ce que vous donne la commande *pwd*).

Nous sommes arrivés au dernier exemple de ce cours. Il s'agit du Saint Graal du physicien paresseux : le générateur de PRL. L'idée est de créer une PRL au format PDF à partir d'un fichier de données issues d'une expérience (ici un cycle d'hysteresis). La PRL doit contenir à la fin le graphe et un tableau associés à ces données. Pour arriver à notre objectif, nous devons :

- extraire et traiter les données.
- créer le fichier eps du graphe via *gnuplot*.
- créer le tableau de valeurs en RevTex.
- créer le fichier RevTex puis compiler le fichier et obtenir le PDF correspondant.

Le http ://www.phys.ens.fr/~regnault/perl/chapitre8/cycle.dat a le format suivant :

```
magneto optical hysteresis loop

sample : aucoau
start field : -1500.000000 Oe
end field : 1500.000000 Oe
field step : 50.000000 Oe
wait time : 0.000000 ms
no of averages : 1.000000
wait time start : 1.000000 s

measured loop parameters :
minimal MO signal : 0.2
```

```
MO signal change : 5.2
relative MO change : 2208.4 percent
negative Hc : 401.4 Oe
positive Hc : -403.6 Oe
loop width : -805.1 Oe
loop shift : -1.1 Oe

measured field (Oe) nominal field (Oe) MO signal (a.u.)
9.8582318E+2 -1.5000000E+3 2.6789543E-1
9.9476666E+2 -1.4500000E+3 2.6994109E-1
9.9608563E+2 -1.4000000E+3 3.1937671E-1
9.9637488E+2 -1.3500000E+3 3.2768381E-1
9.6587244E+2 -1.3000000E+3 3.1222937E-1
9.3337543E+2 -1.2500000E+3 2.8826058E-1
9.0064948E+2 -1.2000000E+3 2.6366332E-1
8.6678162E+2 -1.1500000E+3 2.7219626E-1
... ... ...
```

Nous avons besoin de la première colonne de ce fichier qui représente le champ magnétique et de la troisième qui est une quantité proportionnelle à l'aimantation (avec un décalage et multipliée par -1). Regardons la fonction qui va extraire les données du fichier :

```
1.  # recupere les deux colonnes correspondant au champ magnetique et a l'aimantation
2.  #
3.  # $_[0] = nom du fichier contenant les donnees
4.  # $_[1] = reference sur le tableau qui va contenir les valeurs du champ magnetique
5.  # $_[2] = reference sur le tableau qui va contenir les valeurs de l'aimantation
6.
7.  sub ExtraitDonnees
8.    {
9.      my $NomFichier = $_[0];
10.     my $Champ = $_[1];
11.     my $Aimantation = $_[2];
12.     unless (open(INFILE, $NomFichier))
13.       {
14.         die ("impossible d'ouvrir ".$NomFichier."\n");
15.       }
16.     my $Ligne;
17.     while ((defined($Ligne = <INFILE>)) && (!($Ligne =~ /^measured field (\Oe\)/)))
18.       {
19.       }
20.     while (defined($Ligne = <INFILE>))
21.       {
22.         chomp ($Ligne);
23.         my @TmpValeurs = split (/\t/, $Ligne);
24.         push (@$Champ, $TmpValeurs[0]);
```

```
25.          push (@$Aimantation, $TmpValeurs[2]) ;
26.        }
27.      close (INFILE) ;
28.    }
```

Nous commençons par ouvrir le fichier dont le nom est passé en paramètre et à extraire les lignes jusqu'à celle qui contient la dernière ligne de texte (qui est identifiée par l'expression régulière */ˆmeasured field \(Oe\)/*). Ceci est fait par la boucle située ligne 17. A partir de la ligne 19, nous savons que les prochaines lignes qui vont être lues sont celles qui contiennent les données. Le séparateur entre les colonnes est la tabulation, c'est donc $\backslash t$ que nous indiquons dans l'expression régulière de ligne 23. Les données sont récupérées dans deux tableaux qui ont été passés par référence.

Une fois que nous disposons de nos données stockées dans des tableaux, nous pouvons effectuer tout les traitements que nous voulons. Nous allons ici soustraire la valeur moyenne de la grandeur représentant l'aimantation (pou enlever le décalage sur cette valeur) et nous multiplierons l'aimantation par -1 pour retrouver le bon signe. Ceci est pris en charge par la fonction suivante.

```
 1.  # calcul la valeur moyenne d'un tableau puis la soustrait et multiplie par -1
 2.  #
 3.  # $_[0] = reference sur le tableau a modifier
 4.
 5.  sub SoutraitMoyenneEtInverse
 6.    {
 7.      my $Tableau = $_[0] ;
 8.      my $ValeurMoyenne = 0.0 ;
 9.      my $Valeur ;
10.      foreach $Valeur (@$Tableau)
11.        {
12.          $ValeurMoyenne += $Valeur ;
13.        }
14.      $Valeur = 0 ;
15.      $ValeurMoyenne /= ($#$Tableau + 1) ;
16.      while ($Valeur <= $#$Tableau)
17.        {
18.          $$Tableau[$Valeur] = - $ValeurMoyenne + $$Tableau[$Valeur] ;
19.          $Valeur++ ;
20.        }
21.    }
```

Cette fonction ne doit vous poser aucun problème hormis le fait que pour effectuer la moyenne, il nous faut diviser par le nombre d'éléments. Or la syntaxe *$#NomDuTableau* retourne l'indice du dernier élément. Il faut donc ajouter un pour obtenir le nombre d'éléments.

Nous sommes maintenant arrivés au moment de tracer le cycle d'hysteresis. Nous allons pour cela faire appel à *gnuplot* qui va permettre d'obtenir ce graphe au format EPS. Pour ceux

qui ne connaissent pas *gnuplot*, Il s'agit d'un programme qui permet de tracer différents types de graphes et de les exporter dans différents formats. Il peut s'utiliser en mode interactif ou en mode fichier de commandes. C'est ce dernier que nous utiliserons ici. Voici un exemple simple de fichiers de commandes qui permet de tracer un graphe à partir d'un fichier **donnees.dat** composé de deux colonnes de nombres et de l'enregistrer dans **graphe.eps** au format EPS :

```
set nokey
set size 1.0, 1.0
set xlabel "abscisse"
set ylabel "ordonnee"
set terminal postscript eps enhanced "Helvetica" 16
set output "graphe.eps"
plot "donnees.dat" using 1 :2 notitle with lines 1
```

Pour utiliser *gnuplot*, il nous faut donc mettre nos données dans un fichier et créer un fichier de commande. Il nous suffira ensuite d'exécuter *gnuplot* pour obtenir notre graphe. Cette tâche est effectuée par la fonction suivante :

```
1.  # trace la courbe dans un fichier eps
2.  #
3.  # $_[0] = nom du fichier eps a creer
4.  # $_[1] = reference sur le tableau qui va contenir les valeurs a mettre en abscisse
5.  # $_[2] = reference sur le tableau qui va contenir les valeurs a mettre en ordonnee
6.  # $_[3] = legende a porter sur l'axe des abscisses
7.  # $_[4] = legende a porter sur l'axe des ordonnees
8.
9.  sub TraceEPS
10.   {
11.     my $FichierEPS = $_[0];
12.     my $ValeurX = $_[1];
13.     my $ValeurY = $_[2];
14.     my $LengendeX = $_[3];
15.     my $LengendeY = $_[4];
16.     my $Index = 0;
17.     my $Valeur;
18.     my $FichierTempDonnees = &ObtenirNomFichierTemporaire().".dat";
19.     my $FichierTempGnuplot = &ObtenirNomFichierTemporaire().".p";
20.
21.     unless (open (OUTFILE, ">".$FichierTempDonnees))
22.       {
23.         die ("impossible de creer le fichier temporaire ".$FichierTempDonnees."\n");
24.       }
25.     foreach $Valeur (@$ValeurX)
26.       {
27.         print OUTFILE ($Valeur." ".$$ValeurY[$Index]."\n");
```

```
28.         $Index++;
29.       }
30.     close (OUTFILE);
31.
32.     unless (open (OUTFILE, ">".$FichierTempGnuplot))
33.       {
34.         die ("impossible de creer le fichier temporaire ".$FichierTempGnuplot."\n");
35.       }
36.     print OUTFILE ("set size 1.0, 1.0\n");
37.     if (defined($LengendeX))
38.       {
39.         print OUTFILE ("set xlabel \"".$LengendeX."\"\n");
40.       }
41.     if (defined($LengendeX))
42.       {
43.         print OUTFILE ("set ylabel \"".$LengendeY."\"\n");
44.       }
45.     print OUTFILE ("set nokey
46. set terminal postscript eps enhanced \"Helvetica\" 16
47. set output \"".$FichierEPS."\"
48. plot \"".$FichierTempDonnees."\" using 1 :2 notitle with lines 1
49. ");
50.     close (OUTFILE);
51.
52.     `gnuplot $FichierTempGnuplot`;
53.     unlink($FichierTempGnuplot);
54.     unlink($FichierTempDonnees);
55.   }
```

Comme à l'accoutumée, nous débutons notre fonction en récupérant les paramètres passés. Nous avons vu que pour utiliser *gnuplot*, nous aurons besoin de deux fichiers. Ces fichiers ne sont utiles que lors de la création de la figure, c'est à dire qu'il s'agit de fichiers temporaires. Pour cela, nous créons deux noms de fichiers temporaires lignes 18 et 19. Nous faisons appel à la fonction *ObtenirNomFichierTemporaire* qui est définie par :

```
1.  # donne un nom pour un fichier temporaire (sans extension)
2.  #
3.  # valeur de retour = nom de fichier
4.
5.  sub ObtenirNomFichierTemporaire
6.    {
7.      return "tmp".time();
8.    }
```

Le nom temporaire est obtenu en concaténant la chaîne tmp avec la date en seconde. En toute rigueur, cela peux poser problème (plusieurs fichiers pouvant être créés à la même seconde) mais c'est sans importance ici.

Dans un premier temps nous créons de la ligne 21 à la ligne 30 le fichier contenant les deux colonnes de nombres. Puis de la ligne 32 à la ligne 50 nous créons le fichier de commandes *gnuplot*. Il s'agit d'une copie conforme de celui décrit précédement, où nous avons inséré les bonnes légendes et les bons noms de fichiers. A noter que, puisque *gnuplot* demande à ce que les chaînes de caractères soient entre guilllemets, il faut inclure celles-ci dans le ce qui réclame d'utiliser le symbole \".

Une fois ces deux fichiers enregistrés, nous pouvons invoquer *gnuplot* ligne 52. Remarquez que nous passons comme argument à *gnuplot* le nom du fichier de commandes contenu dans la variable *$FichierTempGnuplot*. Une fois la commande exécutée (nous ne recupérons pas les éventuels messages de *gnuplot*), nous effaçons les deux fichiers temporaires.

Il nous faut maintenant construire le tableau RevTeX contenant les données. Comme nous avons encore un peu de temps, profitons-en pour généraliser cette fonction et faire en sorte qu'elle puisse traiter un nombre quelconque de colonnes au lieu de deux.

```
 1. # construit un tableau en revtex (en colonne) a partir de tableaux de valeurs
 2. #
 3. # $_[0] = reference sur le tableau contenant les titres de chaque colonne
 4. # $_[1] = reference sur le tableau bidimensionnel contenant les donnees en colonne
 5. # $_[2] = legende a associer au tableau
 6. # valeur de retour = chaine de caractere donnant l'equivalent revtex du tableau
 7.
 8. sub TableauVersRevTex
 9.   {
10.     my $Titres = $_[0] ;
11.     my $Donnees = $_[1] ;
12.     my $Legende = $_[2] ;
13.     my $RevTex = "\\begin{table*}
14. \\begin{ruledtabular}
15. \\begin{tabular}{" ;
16.     my $Index1 = 0 ;
17.     my $NbrColonnes = $#$Titres + 1 ;
18.     while ($Index1 < $NbrColonnes)
19.       {
20.         $RevTex .= "c" ;
21.         $Index1++ ;
22.       }
23.     $RevTex .= "}\n" ;
24.
25.     $RevTex .= join (" & ", @$Titres) ;
26.     $RevTex .= "\\\\ \\hline\n" ;
27.
28.     my $ReferenceTableau = $$Donnees[0] ;
29.     my $NbrLignes = $#$ReferenceTableau + 1 ;
30.     my $Index2 = 0 ;
31.     while ($Index2 < $NbrLignes)
32.       {
```

```
33.         $ReferenceTableau $$Donnees[0];
34.         $RevTex .= $$ReferenceTableau[$Index2];
35.         $Index1 = 1;
36.         while ($Index1 < $NbrColonnes)
37.           {
38.             $ReferenceTableau = $$Donnees[$Index1];
39.             $RevTex .= " & ".$$ReferenceTableau[$Index2];
40.             $Index1++;
41.           }
42.         $RevTex .= "\\\\ \n";
43.         $Index2++;
44.       }
45.
46.     $RevTex .= "\\end{tabular}
47.  \\end{ruledtabular}
48.  \\caption{".$Legende."}
49.  \\end{table*}\n";
50.     return $RevTex;
51.   }
```

Voici ce que contient typiquement la variable *$RevTex* à la fin de la fonction :

```
\begin{table*}
\begin{ruledtabular}
\begin{tabular}{cc}
magnetic field (Oe) & magnetization (arb. units.)\\ \hline
9.8582318E+2 & 0.87121134734411\\
9.9476666E+2 & 0.904473855050131\\
\end{tabular}
\end{ruletabular}
\caption{a nice array}
\end{table*}
```

Après avoir récupéré les paramètres (lignes 10 à 12), nous commençons par écrire le préambule du tableau RevTeX (ligne 13 à 15). Il nous faut ensuite indiquer le nombre de colonnes et la justification du texte dans chacune d'elles. Nous choisirons ici un texte centré (cacratère *c*). Pour obtenir le nombre de colonnes, il suffit de compter le nombre d'éléments du tableau contenant les titres, tableau dont la référence est *$Titres*. Ceci est fait ligne 17. Le code situé entre les lignes 18 et 23 nous permet d'écrire à la suite de *$RevTex* autant de caractères *c* qu'il y a de colonnes.

Nous allons ensuite indiquer les titres des colonnes. Nous allons utiliser pour cela la commande qui va permettre de concaténer les éléments du tableau des titres en insérant le séparateur *&* qui désigne le séparateur de colonne en RevTeX (ligne 25). Il nous faut maintenant insérer les données dans le tableau. Ceci est effectué par la partie du code situé

entre les lignes 28 et 44. Nous commençons par récupérer le nombre de lignes que nous devons écrire grâce au nombre d'éléments de la première colonne (ligne 29). S'en suit une boucle sur le nombre de lignes. Nous récupérons les différents éléments d'une ligne en insérant le symbole & en faisant attention que ce symbole ne doit apparaître qu'entre deux éléments. C'est pour cela que l'écriture du premier élément (lignes 33 et 34) est découplée de celle des autres (lignes 36 à 41).

Nous disposons maintenant de toutes les briques de base nécessaires à l'accomplissement de notre objectif. Le programme principal qui va s'acquitter de la tâche peut s'écrire ainsi :

```
 1. #!/usr/bin/perl -w
 2.
 3. use strict 'vars';
 4.
 5. if (!(defined($ARGV[1])))
 6.   {
 7.     die ("syntaxe : prl nom_de_l_article donnees.dat\n");
 8.   }
 9. my $NomArticle = $ARGV[0];
10. my $DebutRevTex = "debut.tex";
11. my $FinRevTex = "fin.tex";
12. my $FichierDonnees = $ARGV[1];
13. my $FichierEPS = $FichierDonnees.".eps";
14. my @Champ;
15. my @Aimantation;
16.
17. &ExtraitDonnees($FichierDonnees, Champ, Aimantation);
18. &SoutraitMoyenneEtInverse(Aimantation);
19. &CorrigeDerive(Champ, Aimantation);
20. &TraceEPS($FichierEPS, Champ, Aimantation, "magnetic field (Oe)",
21. "magnetization (arb. units.)");
22.
23. unless (open (OUTFILE, ">".$NomArticle.".tex"))
24.   {
25.     die ("impossible de creer le fichier ".$NomArticle.".tex\n");
26.   }
27.
28. my $Ligne;
29. unless (open (INFILE, $DebutRevTex))
30.   {
31.     die ("impossible d'ouvrir le fichier ".$DebutRevTex."\n");
32.   }
33. while (defined($Ligne = <INFILE>))
34.   {
35.     print OUTFILE ($Ligne);
36.   }
37. close (INFILE);
38.
39. print OUTFILE ("\\begin{figure}[!htbp]
```

```
40.  \\begin{center}
41.  \\includegraphics[ width=3.25in, keepaspectratio]{".$FichierEPS."}
42.  \\end{center}
43.  \\caption{it deserves a Nobel prize}
44.  \\end{figure}\n") ;
45.
46.  my @Titres = ("magnetic field (Oe)", "magnetization (arb. units.)") ;
47.  my @Tableau = (Champ, Aimantation) ;
48.  print OUTFILE (&TableauVersRevTex(Titres, Tableau, "a nice array")) ;
49.
50.  unless (open (INFILE, $FinRevTex))
51.    {
52.      die ("impossible d'ouvrir le fichier ".$FinRevTex."\n") ;
53.    }
54.  while (defined($Ligne = <INFILE>))
55.    {
56.      print OUTFILE ($Ligne) ;
57.    }
58.  close (INFILE) ;
59.
60.  close (OUTFILE) ;
61.
62.  my $Commande = "latex ".$NomArticle.".tex" ;
63.  `$Commande` ;
64.  `$Commande` ;
65.  $Commande = "dvipdf ".$NomArticle.".dvi ".$NomArticle.".pdf" ;
66.  `$Commande` ;
```

Comme nous avons l'intention de produire plusieurs PRLs, il est utile de pouvoir définir le nom du fichier de données et de l'article sans avoir à éditer le programme. Le début du programme récupère les informations passées par la ligne de commande et vérifie en particulier que deux paramètres sont passés grâce au de la ligne 5 qui vérifie que le second élément du tableau *@ARGV* est bien présent.

Au lieu d'écrire directement dans le script le début et la fin du fichier RevTeX, nous stockons ces derniers dans deux fichiers dont les noms sont **debut.tex** et **fin.tex**. Lignes 17, 18 et 19, nous procédons à l'extraction et au traitement des données, puis à la création du fichier EPS. De la ligne 23 à 60, nous créons le fichier RevTeX. Nous commençons par recopier les lignes contenues dans le fichier **debut.tex** (lignes 27 à 36). Nous insérons ensuite les commandes pour inclure le fichier EPS (ligne 38 à 43). L'écriture du tableau est un peu plus délicate, notre fonction *TableauVersRevTex* a besoin d'un tableau bidimensionnel. Ce dernier est défini ligne 46. La ligne 47 se charge d'inclure le tableau. Il ne reste plus qu'à compléter le fichier RevTeX en ajoutant la fin (lignes 49 à 57).

Le fichier RevTeX est maintenant sur le disque. Nous souhaitons obtenir le PDF de ce fichier. Nous allons commencer par compiler le fichier. La commande qui va se charger de cette opération est définie ligne 61 puis exécutée ligne 62. Notez que ce qui est indiqué entre guillemets inversés est passé au système. Nous pouvons indiquer une variable dont le contenu est la commande que PERL se chargera d'exécuter. Dans notre cas, la commande exécutée

sera *latex* suivie du nom de fichier à compiler. Comme *latex* nécessite deux passes en général, nous exécutons la commande deux fois (lignes 62 et 63). Les lignes 65 et 66 sont du même type et permettent d'obtenir le PDF à partir du fichier DVI précédemment créé.

Bien des améliorations de ce programme sont envisageables, comme un traitement plus fin des données, limiter la taille du tableau dans le fichier PDF... Mais vous disposez maintenant de suffisament de connaissances en PERL pour ne plus avoir peur de sortir votre éditeur de texte préféré et affronter les tâches informatiques répétitives. Comme le disait le vieux sage "Remember, PERL will be with you... Always.".

8.3 Exercices

1 Ecrivez un programme qui analyse un fichier **fichier.log** et si ce dernier contient le mot alert, envoyez-vous un mail en utilisant MIME : :Lite avec le contenu du fichier dans le corps du mail et la date et heure dans le titre.

2 Reprenez la fonction *ObtenirNomFichierTemporaire* et faites en sorte que tous les noms créés soient uniques.

Annexe A

Solution des exercices

A.1 chapitre2

```
 1.  #!/usr/bin/perl
 2.
 3.  print ("nombre d'iterations : ");
 4.  $NbrIterations = <STDIN>;
 5.  chomp ($NbrIterations);
 6.  if ($NbrIterations > 1)
 7.    {
 8.      $NbrTouches = 0;
 9.      for ($Iteration = 0; $Iteration < $NbrIterations; $Iteration++)
10.        {
11.          $PositionX = rand;
12.          $PositionY = rand;
13.          if (($PositionX * $PositionX + $PositionY * $PositionY) <= 1)
14.            {
15.              $NbrTouches++;
16.            }
17.        }
18.    }
19.  print ("valeur approchee de pi : ".($NbrTouches / $NbrIterations * 4)."\n");
```

```
 1.  #!/usr/bin/perl
 2.
 3.  print ("borne superieure : ");
 4.  $Max = <STDIN>;
 5.  chomp ($Max);
 6.  if ($Max > 0)
 7.    {
 8.      print ("nombre de subdivision : ");
 9.      $NbrSubdivisions = <STDIN>;
10.      chomp ($NbrSubdivisions);
11.      if ($NbrSubdivisions > 1)
```

```
12.       {
13.         $Increment = $Max / $NbrSubdivisions;
14.         $Position = 0.0;
15.         $Integrale = 0.0;
16.         for ($Iteration = 0; $Iteration < $NbrSubdivisions; $Iteration++)
17.           {
18.             $Integrale += exp(-0.5 * ($Position * $Position));
19.             $Position += $Increment;
20.           }
21.         $Integrale *= 2.0 * $Integrale * $Increment * $Increment;
22.         print ("valeur approchee de pi : ".$Integrale."\n");
23.       }
24.   }
```

A.2 chapitre3

A.2.1 exercice 1

```
 1.  #!/usr/bin/perl
 2.
 3.  $NombreElements = $ARGV[0];
 4.  for ($Index = 0; $Index < $NombreElements; $Index++)
 5.    {
 6.      $Tableau[$Index] = int(rand(100));
 7.      print ($Tableau[$Index]."\n");
 8.    }
 9.
10.  $Minimum = $Tableau[0];
11.  $Maximum = $Minimum;
12.  for ($Index = 1; $Index < $NombreElements; $Index++)
13.    {
14.      if ($Tableau[$Index] > $Maximum)
15.        {
16.          $Maximum = $Tableau[$Index];
17.        }
18.      else
19.        {
20.          if ($Tableau[$Index] < $Minimum)
21.            {
22.              $Minimum = $Tableau[$Index];
23.            }
24.        }
25.    }
26.  print ("minimum = ".$Minimum." maximum = ".$Maximum."\n");
```

```
1.  #!/usr/bin/perl
2.
3.  $NombreElements = $ARGV[0];
4.  for ($Index = 0; $Index < $NombreElements; $Index++)
5.    {
6.      $Tableau[$Index] = int(rand(100));
7.      print ($Tableau[$Index]."\n");
8.    }
9.
10. @Tableau = sort {$a <=> $b} (@Tableau);
11. print ("minimum = ".$Tableau[0]." maximum = ".$Tableau[$NombreElements - 1]."\n");
```

Comme vous pouvez le constater, le code est beaucoup plus compact avec le tri mais la complexité est différente ($O(N)$ contre $O(N \log N)$. La deuxième technique doit donc être utilisée uniquement si la vitesse d'exécution n'est pas un problème.

A.2.2 exercice 2

```
1.  #!/usr/bin/perl
2.
3.  $particules{938.3} = "proton";
4.  $particules{939.6} = "neutron";
5.  $particules{0.511} = "electron";
6.  foreach $cle (sort {$a <=> $b} (keys(%particules)))
7.    {
8.      print ($particules{$cle}." (".$cle." MeV)\n");
9.    }
```

La seule difficulté de ce code réside dans la ligne 6 qui est très compacte. En effet, le tableau passé à *foreach* est obtenu après le tri numérique du tableau de clés de la table de hash *%particules*. Les plus attentifs d'entre vous remarqueront que ce tri n'est pas strictement équivalent à l'exemple du tri croisé. En effet, vous ne pouvez avoir deux masses identiques, la clé (ici la masse) devant être unique par définition.

A.3 chapitre4

A.3.1 exercice 1

```
1.  #!/usr/bin/perl
2.
3.  for ($index = 0; $index < 100; $index++)
4.    {
5.      unless (open (OUTFILE, ">test".$index))
```

```
 6.       {
 7.         die ("impossible de creer le fichier test".$index."\n");
 8.       }
 9.     print OUTFILE ("test".$index);
10.     close (OUTFILE);
11.   }
```

```
1.  #!/usr/bin/perl
2.
3.  for ($index = 0; $index < 100; $index++)
4.    {
5.      if (-e "test".$index)
6.        {
7.          rename ("test".$index, "toto".($index + 100));
8.        }
9.    }
```

A.3.2 exercice 2

```
 1.  #!/usr/bin/perl
 2.
 3.  unless (open (OUTFILE, ">donnees1.dat"))
 4.    {
 5.      die ("impossible de creer le fichier donnees1.dat\n");
 6.    }
 7.  $valeur = 0.0;
 8.  $nbrpas = 100;
 9.  $increment = 1.0 / $nbrpas;
10.  for ($index = 0; $index <= $nbrpas; $index++)
11.    {
12.      print OUTFILE ($valeur."\n");
13.      $valeur += $increment;
14.    }
15.  close (OUTFILE);
```

```
1.  #!/usr/bin/perl
2.
3.  unless (open (INFILE, "donnees1.dat"))
4.    {
5.      die ("impossible d'ouvrir le fichier donnees1.dat\n");
6.    }
```

```
 7.  unless (open (OUTFILE, ">donnees2.dat"))
 8.    {
 9.      die ("impossible de creer le fichier donnees2.dat\n");
10.    }
11.  $pi = 2.0 * atan2 (1.0, 0.0);
12.  while (defined($ligne = <INFILE>))
13.    {
14.      chomp ($ligne);
15.      print OUTFILE ($ligne." ".cos(2.0 * $pi * $ligne)."\n");
16.    }
17.  close (OUTFILE);
18.  close (INFILE);
```

La ligne 11 est une façon élégante d'obtenir une valeur approchée de π en évitant d'avoir à rentrer une dizaine de décimales de ce nombre à la main et sans se soucier de la précision numérique.

A.4 chapitre5

A.4.1 exercice 1

```
 1.  #!/usr/bin/perl
 2.
 3.  $motif = $ARGV[0];
 4.  if (( !(defined($motif))) || ( !($motif =~ /^\w+$/)))
 5.    {
 6.      die ("syntaxe : cherche.pl motif\n");
 7.    }
 8.
 9.  foreach $fichier (<*.txt>)
10.    {
11.      if (open (INFILE, $fichier))
12.        {
13.          $drapeau = 0;
14.          while (($drapeau == 0) && (defined($ligne = <INFILE>)))
15.            {
16.              if ($ligne =~ /\b$motif\b/i)
17.                {
18.                  print ($fichier."\n");
19.                  $drapeau = 1;
20.                }
21.            }
22.          close (INFILE);
23.        }
24.      else
25.        {
26.          print ("impossible d'ouvrir le fichier ".$fichier."\n");
```

```
27.         }
28.     }
```

Il est toujours bon de vérifier que les paramètres passés qu programme par l'utilisateur soient au bon format. La ligne 4 se charge de s'assurer qu'un paramètre a été (par le *defined*) et que celui-ci n'est composé que de caractères alphanumériques.

A priori, lors du parcours d'un fichier, nous pouvons nous arrêter la recherche du motif dès la première occurence de celui-ci. Pour cela nous utilisons la variable *$drapeau* qui permet de stopper la boucle *while* de la ligne 14 dès que le motif a été trouvé. Notez aussi que sur la recherche du motif (ligne 16), nous avons mis un *i* après l'expression régulière qui permet d'indiquer que cette dernière ne doit pas tenir compte de la casse.

A.4.2 exercice 2

```
 1.  #!/usr/bin/perl
 2.
 3.  $nbrfichiers = $ARGV[0];
 4.  if ((!(defined($nbrfichiers))) || (!($nbrfichiers =~ /^\d+$/)) || ($nbrfichiers == 0))
 5.    {
 6.      die ("syntaxe : fichier.pl nombre_de_fichiers\n");
 7.    }
 8.
 9.  for ($index = 0; $index < $nbrfichiers; $index++)
10.    {
11.      unless (open (OUTFILE, ">qhe".$index.".dat"))
12.        {
13.          die ("impossible de creer le fichier qhe".$index.".dat\n");
14.        }
15.      print OUTFILE ("qhe".$index.".dat");
16.      close (OUTFILE);
17.    }
```

```
1.  #!/usr/bin/perl
2.
3.  foreach $fichier (<*.dat>)
4.    {
5.      if ($fichier =~ /^qhe\d*\.dat$/)
6.        {
7.          rename ($fichier, "f".$fichier);
8.        }
9.    }
```

A.4.3 exercice 3

```
1.  #!/usr/bin/perl
2.
3.  if ((!(defined($ARGV[2]))) || (!($ARGV[0] =~ /^\d+$/)) ||
4.  (!($ARGV[1] =~ /^\d+$/)) || ($ARGV[0] == 0) || ($ARGV[1] == 0))
5.    {
6.      die ("syntaxe : matrice.pl nbr_lignes nbr_colonnes nom_fichier\n");
7.    }
8.
9.  $ligne = $ARGV[0];
10. $colonne = $ARGV[1];
11. $fichier = $ARGV[2];
12.
13. unless (open (OUTFILE, ">".$fichier))
14.   {
15.     die ("impossible de creer le fichier ".$fichier."\n");
16.   }
17.
18. while ($ligne > 0)
19.   {
20.     print OUTFILE (rand());
21.     for ($index = 1; $index < $colonne; $index++)
22.       {
23.         print OUTFILE (" ".rand());
24.       }
25.     print OUTFILE ("\n");
26.     $ligne--;
27.   }
28.
29. close (OUTFILE);
```

Comme souvent avec les sorties formattées en colonne, il faut prendre attention au fait que la séparation ne doit être présente qu'entre les éléments. Ceci explique le découpage de l'écriture d'une ligne en deux morceaux (ligne 20 puis des lignes 21 à 24).

```
1.  #!/usr/bin/perl
2.
3.  $fichier = $ARGV[0];
4.  unless (open (INFILE, $fichier))
5.    {
6.      die ("impossible d'ouvrir le fichier ".$fichier."\n");
7.    }
8.  if (defined($ligne = <INFILE>))
9.    {
10.     @valeurs = split (/ /, $ligne);
```

```
11.      $maximum = $valeurs[0];
12.    }
13.  else
14.    {
15.      die ("le fichier ".$fichier." est vide\n");
16.    }
17.  close (INFILE);
18.
19.
20.  unless (open (INFILE, $fichier))
21.    {
22.      die ("impossible d'ouvrir le fichier ".$fichier."\n");
23.    }
24.  while (defined($ligne = <INFILE>))
25.    {
26.      @valeurs = split (/ /, $ligne);
27.      foreach $element (@valeurs)
28.        {
29.          if ($element > $maximum)
30.            {
31.              $maximum = $element;
32.            }
33.        }
34.    }
35.  close (INFILE);
36.
37.  $temporaire = "";
38.  unless (open (INFILE, $fichier))
39.    {
40.      die ("impossible d'ouvrir le fichier ".$fichier."\n");
41.    }
42.  while (defined($ligne = <INFILE>))
43.    {
44.      @valeurs = split (/ /, $ligne);
45.      for ($index = 0; $index <= $#valeurs; $index++)
46.        {
47.          $valeurs[$index] /= $maximum;
48.        }
49.      $temporaire .= join (" ", @valeurs)."\n";
50.    }
51.  close (INFILE);
52.
53.  unless (open (OUTFILE, ">".$fichier))
54.    {
55.      die ("impossible d'ouvrir le fichier ".$fichier."\n");
56.    }
57.  print OUTFILE $temporaire;
58.  close (OUTFILE);
```

A.4.4 exercice 4

```
 1.  #!/usr/bin/perl
 2.
 3.  $fichier = $ARGV[0];
 4.  unless (open (INFILE, $fichier))
 5.    {
 6.      die ("impossible d'ouvrir le fichier ".$fichier."\n");
 7.    }
 8.  $contenu = "";
 9.  while (defined($ligne = <INFILE>))
10.    {
11.      $ligne =~ s/\r//;
12.      $contenu .= $ligne;
13.    }
14.  close (INFILE);
15.  unless (open (OUTFILE, ">".$fichier))
16.    {
17.      die ("impossible d'ecrire le fichier ".$fichier."\n");
18.    }
19.  print OUTFILE ($contenu);
20.  close (OUTFILE);
```

```
 1.  #!/usr/bin/perl
 2.
 3.  $fichier = $ARGV[0];
 4.  unless (open (INFILE, $fichier))
 5.    {
 6.      die ("impossible d'ouvrir le fichier ".$fichier."\n");
 7.    }
 8.  $contenu = "";
 9.  while (defined($ligne = <INFILE>))
10.    {
11.      $ligne =~ s/^\n/\r\n/;
12.      $ligne =~ s/([^\r])\n/$1\r\n/;
13.      $contenu .= $ligne;
14.    }
15.  close (INFILE);
16.  unless (open (OUTFILE, ">".$fichier))
17.    {
18.      die ("impossible d'ecrire le fichier ".$fichier."\n");
19.    }
20.  print OUTFILE ($contenu);
21.  close (OUTFILE);
```

La conversion DOS vers UNIX ne pose pas de problème puisqu'il suffit d'enlever un caractère. Par contre dans l'autre sens, nous devons ajouter un $\backslash r$ devant le $\backslash n$. Il nous faut vérifier que celui-ci n'est pas déja présent.

A.5 chapitre6

A.5.1 exercice 1

```
 1. #!/usr/bin/perl
 2.
 3. &renomme();
 4.
 5.
 6. sub renomme()
 7.   {
 8.     foreach $fichier (<*>)
 9.       {
10.         if (-d $fichier)
11.           {
12.             chdir ($fichier);
13.             &renomme();
14.             chdir ("..");
15.           }
16.         else
17.           {
18.             if ($fichier =~ /^qhe\d*\.dat$/)
19.               {
20.                 rename ($fichier, "f".$fichier);
21.               }
22.           }
23.       }
24.   }
```

A.5.2 exercice 2

```
 1. sub minmax
 2.   {
 3.     $tableau = $_[0];
 4.     $minimum = $$tableau[0];
 5.     $maximum = $minimum;
 6.     foreach $valeur (@$tableau)
 7.       {
 8.         if ($valeur < $minimum)
 9.           {
```

```
10.             $minimum = $valeur;
11.           }
12.         else
13.           {
14.             if ($valeur > $maximum)
15.               {
16.                 $maximum = $valeur;
17.               }
18.           }
19.       }
20.     return ($minimum, $maximum);
21.   }
```

```
 1.  sub minmax
 2.    {
 3.      $tableau = $_[0];
 4.      $minimum = $_[1];
 5.      $maximum = $_[2];
 6.      $$minimum = $$tableau[0];
 7.      $$maximum = $$minimum;
 8.      foreach $valeur (@$tableau)
 9.        {
10.          if ($valeur < $$minimum)
11.            {
12.              $$minimum = $valeur;
13.            }
14.          else
15.            {
16.              if ($valeur > $$maximum)
17.                {
18.                  $$maximum = $valeur;
19.                }
20.            }
21.        }
22.    }
```

A.6 chapitre8

A.6.1 exercice 1

```
1.  #!/usr/bin/perl -w
2.
3.  use strict 'vars';
4.
```

```
 5.  use MIME : :Lite ;
 6.
 7.  my $ServeurSMTP = "lumiere.relativite.fr" ; ;
 8.  my $Expediteur = "albert\@relativite.fr" ;
 9.  my $Destinaire = "werner\@quantique.org" ;
10.
11.  unless (open (INFILE, "fichier.log"))
12.    {
13.      die ("impossible d'ouvrir fichier.log\n") ;
14.    }
15.  my $Ligne ;
16.  my $Drapeau = 0 ;
17.  my $Fichier = "" ;
18.  while (defined($Ligne = <INFILE>))
19.    {
20.      if ($Ligne =~ /alert/)
21.        {
22.          $Drapeau = 1 ;
23.        }
24.      $Fichier .= $Ligne ;
25.    }
26.  close (INFILE) ;
27.
28.  if ($Drapeau == 1)
29.    {
30.      my @Date = gmtime(time) ;
31.      $Date[5] += 1900 ;
32.      $Date[4] += 1 ;
33.      my $Jour = $Date[3]."/".$Date[4]."/".$Date[5] ;
34.      my $Heure = $Date[2]."h".$Date[1] ;
35.      MIME : :Lite->send("smtp", $ServeurSMTP, Timeout=>60) ;
36.      my $msg = new MIME : :Lite
37.      From =>$Expediteur,
38.      To =>$Destinaire,
39.      Subject => "Message d'alerte de mail.pl le ".$Jour." a ".$Heure,
40.      Type =>"TEXT",
41.      Data =>$Fichier ;
42.      $msg->send ;
43.    }
```

A.6.2 exercice 2

```
1.  # donne un nom pour un fichier temporaire (sans extension)
2.  #
3.  # valeur de retour = nom de fichier
4.
5.  sub ObtenirNomFichierTemporaire
```

```
 6.    {
 7.      my $NomFichier = "tmp".time();
 8.      my $Motif1 = "^".$NomFichier.".*";
 9.      my $Motif2 = "^".$NomFichier."(\\d*).*";
10.      my $TmpFichier;
11.      my $Maximum = -1;
12.      foreach $TmpFichier (<tmp*>)
13.        {
14.          if ((-f $TmpFichier) && ($TmpFichier =~ /\b$Motif1\b/))
15.            {
16.              $TmpFichier =~ /\b$Motif2\b/;
17.              if (($1 ne "") && ($1 > $Maximum))
18.                {
19.                  $Maximum = $1;
20.                }
21.            }
22.        }
23.      if ($Maximum != -1)
24.        {
25.          $Maximum++;
26.          $NomFichier .= $Maximum;
27.        }
28.      else
29.        {
30.          $NomFichier .= "0";
31.        }
32.      return $NomFichier;
33.    }
```

Index