ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Une introduction à
$S\!-\!Plus$

Version 3.1 pour Silicon Graphics Iris

Marcel BAUMGARTNER
Département de Mathématiques
Chaire de Statistique

Juillet 1994

Table des matières

1 Qu'est-ce que *S-PLUS* ?

Le logiciel statistique pour l'analyse de données *S-PLUS* est devenu un outil de plus en plus important dans l'application de la statistique tant dans le monde universitaire que dans l'industrie. Ce sont la simplicité du langage et les possibilités graphiques qui fascinent les statisticiens et les étudiants.

Cette introduction s'inspire de deux autres introductions écrites en anglais, aussi disponibles sur *StatLib* (cf. chapitre 12):

- *Notes on S-PLUS: A Programming Environment for Data Analysis and Graphics*, par Bill Venables et Dave Smith, University of Adelaide
- *Introductory Guide to S-Plus*, par B. D. Ripley, University of Oxford

L'idée de cette introduction est d'initier les nouveaux utilisateurs à *S-PLUS* en peu de temps. Elle ne peut en aucun cas remplacer les références principales suivantes:

- *The New S Language: A Programming Environment for Data Analysis and Graphics*, par R. A. Becker, J. M. Chambers et A. R. Wilks, Wadsworth & Brooks, 1988: il s'agit de la documentation originale, on l'appelle souvent "The Blue Book"
- *Statistical Models in S*, par J. M. Chambers et T. J. Hastie, Wadsworth & Brooks, 1992: ce livre introduit une nouvelle approche de modélisation, on l'appelle aussi "The White Book"
- Statistical Sciences, Inc. *S-PLUS User's Manual, Version 3.1*, Seattle: Statistical Sciences, Inc., 1993
- Statistical Sciences, Inc. *S-PLUS Reference Manual, Version 3.1*, Seattle: Statistical Sciences, Inc., 1993
- Statistical Sciences, Inc. *S-PLUS Programmer's Manual, Version 3.1*, Seattle: Statistical Sciences, Inc., 1993

Cette introduction est destinée aux étudiantes et étudiants de l'Ecole Polytechnique Fédérale de Lausanne (EPFL), Suisse; elle peut être copiée et redistribuée pour des cours d'enseignement (des cours commerciaux inclus) sans conditions ultérieures si l'auteur est mentionné. Le fichier LaTeX est disponible auprès de l'auteur et il peut être modifié et incorporé dans des versions locales. Pour tout commentaire ou suggestion, contactez l'auteur via e-mail à l'adresse

`baumgart@dma.epfl.ch`

ou `helbling@dma.epfl.ch` (Adresse: Dr. Jean-Marie Helbling, Chaire de Statistique, EPFL-DMA, 1015 **Lausanne**, Suisse).

2 Organisation d'une session

Les commentaires suivants concernent l'organisation d'une session *S-PLUS* avec un environnement UNIX. On suppose que le lecteur possède quelques connaissances de base de UNIX.

Suivez les points ci-dessous si vous allez utiliser *S-PLUS* pour la première fois:

1. Créez tout d'abord - au niveau UNIX - un nouveau directoire; par exemple *essai*:

   `$ mkdir essai`

2. Puis allez dans ce directoire et créez un nouveau sous-directoire, qui va contenir tous les objets *S-PLUS* que vous allez former:

   `$ cd essai`

   `$ mkdir .Data` (c'est un nom réservé, il faut l'écrire de cette manière)

3. Restez dans le directoire *essai* et **n'allez pas** dans le sous-directoire *.Data*. Lancez *S-PLUS* à l'aide de la commande suivante:

   `$ Splus`[1]

   Vous devriez voir 4 lignes similaires aux suivantes qui indiquent que *S-PLUS* est prêt.

   ```
   S-PLUS : Copyright (c) 1988, 1993 Statistical Sciences, Inc.
   S : Copyright AT&T.
   Version 3.1 Release 1 for Silicon Graphics Iris, IRIX 4.x : 1993
   Working data will be in .Data
   >
   ```

4. Le logiciel attend maintenant vos commandes. Pour quitter le programme:

   `> q()`

Si vous utilisez à nouveau *S-PLUS* dans ce même directoire, vous n'avez pas besoin de recréer le directoire *.Data*. Remarquez que tous les objets créés jusqu'alors seront sauvegardés: ceci est un avantage majeur de *S-PLUS*.

Dans le travail quotidien avec *S-PLUS*, vous serez souvent confrontés au problème de la nomenclature des variables: par exemple `x` et `y` sont des noms fréquemment utilisés, qui peuvent avoir des significations différentes d'une analyse à l'autre. Nous vous recommandons ainsi de créer pour chaque nouvelle tâche un nouveau directoire (comme *essai* auparavant), d'y entrer

[1] Cette commande peut être légèrement différente d'un système à l'autre, par exemple `splus` ou bien `Splus3.1`. Contactez votre expert UNIX local.

et de créer le sous-directoire *.Data*, qui ne va contenir que les variables concernant la tâche en question. Consultez à ce sujet aussi le chapitre 11.

Les deux fonctions suivantes sont aussi importantes en ce qui concerne l'organisation du travail dans *S-PLUS*: `ls()` et `objects()`. La commande `ls()` affiche toutes les variables dans le directoire *.Data* courant. Or, dorénavant on recommande d'utiliser plutôt la fonction `objects()`, une version plus sophistiquée de `ls()`. Elle permet entre autres de n'afficher qu'une partie des objets: écrivez `objects(pattern="^x")` pour voir toutes les variables commençant avec la lettre x.

3 L'apéritif

Suivez cette session "apéritif" pour avoir une première idée de ce que vous allez pouvoir produire avec *S-PLUS*. Par la suite nous omettrons le "`>`" pour rendre la lecture plus aisée. Les sessions ne contiennent que les instructions et non pas ce que *S-PLUS* affiche. Tapez la touche "return" ou "enter" après chaque ligne. *S-PLUS* contient quelques jeux de données qui nous servirons à titre illustratif, notamment `corn.rain` et `corn.yield`.

3.1 Les bases: vecteurs, matrices

`$ Splus`	Lance *S-PLUS*
`x <- c(2,5,4,3,-2)`	Définit un vecteur de composantes $(2, 5, 4, 3, -2)$
`x`	Affiche x
`length(x)`	Longueur du vecteur x
`moy.x <- mean(x)`	Moyenne de x mise dans la nouvelle variable *moy.x*
`moy.x`	Affiche la variable *moy.x*
`y <- c(1:5)`	Définit un vecteur de composantes 1 à 5
`y`	Affiche y
`exp(x)`	e^x (composante par composante)
`x+y`	Somme de 2 vecteurs (comp. par comp.)
`x[3]`	Retourne la troisième composante de x
`x[c(2,4)]`	Retourne la 2ème et la 4ème composante
`X <- cbind(x,y)`	Construit la matrice X avec colonnes x et y
`x`	Affiche x
`X`	Affiche X
	Notez la différence entre lettres majuscules et minuscules
`xtx <- t(X) %*% X`	Multiplication matricielle: `t()` est est la transposée d'une matrice
`solve(xtx)`	Inverse une matrice carrée
`solve(xtx) %*% xtx`	Contrôle!
`help(solve)`	Documentation "on-line" sur la commande entre parenthèses
`r <- round(runif(10,0,20),0)`	Dix nombres aléatoires entiers entre 0 et 20
`sort(r)`	Ordonne les valeurs de r (croissant)
`rev(sort(r))`	Ordonne les valeurs de r (décroissant)
`order(r)`	Permutation qui permet d'ordonner r
`(1+sqrt(5))/2`	*S-PLUS* est aussi une calculatrice!

3.2 Régression et représentation graphique

`motif()`	Ouvre une fenêtre graphique
`plot(corn.rain,corn.yield)`	“Scatterplot” de données bidimensionelles
`reg <- lsfit(corn.rain,corn.yield)`	Régression: prédicteur = *corn.rain*, réponse = *corn.yield*
`ls.print(reg)`	Sortie usuelle d'une régression
`abline(reg)`	Ajoute la droite de régression au “scatterplot”
`abline(lmsreg(corn.rain, corn.yield),lty=2)`	Ajoute une droite de régression robuste avec un autre style de ligne
`title("Deux regressions")`	Ajoute un titre au graphique

3.3 Graphiques plus sophistiqués

`x <- seq(0,4,length=50)`	Construit une grille régulière entre 0 et 4 de longueur 50
`y <- x`	
`f <- function(x,y) {sin(x*y)}`	Définit une fonction à 2 variables
`persp(x,y,outer(x,y,f))`	Graphique en 3 dimensions
`persp(x,y,outer(x,y,f),eye=c(6,8,-5))`	Changement du point de vue
`contour(x,y,outer(x,y,f))`	Graphique des contours (lignes de niveaux)
`x <- seq(-5,5,length=100)`	
`plot(dnorm(x),type="l")`	Dessine la densité d'une loi $N(0,1)$
`lines(dt(x,df=4),lty=2)`	Ajoute la densité d'une loi t_4 avec un autre style de ligne
`lines(dcauchy(x),lty=3)`	Ajoute la densité d'une loi de Cauchy
`legend(700,0.35,c("Normal","t4", "Cauchy"),lty=1:3)`	Ajoute une légende sur le graphe

3.4 Modèles et analyse de variance

Comparez les instructions suivantes avec celles du chapitre 3.2.

`lm.corn <- lm(corn.yield ~ corn.rain)`	Modèle linéaire: prédicteur = *corn.rain*, réponse =*corn.yield*
`summary(lm.corn)`	Sortie usuelle d'un modèle linéaire
`par(mfrow=c(2,2))`	Définit plusieurs graphiques (ici 2×2) sur une page
`plot(lm.corn)`	Dessine les données et la droite de régression ainsi qu'un graphique des résidus
`residuals(lm.corn)`	Résidus
`qqnorm(residuals(lm.corn))`	Q-Q plot des résidus

Pour l'exemple suivant, utilisez un éditeur de texte extérieur à *S-PLUS* et écrivez les données suivantes dans un fichier appelé *chemical.data*. Placez ce fichier dans le directoire à partir duquel vous lancez *S-PLUS*.

```
       Temp Conc Catalyseur Recolte
 obs.1  160   20         C1      59
 obs.2  160   20         C2      50
 obs.3  160   20         C1      61
 obs.4  160   20         C2      54
 obs.5  160   40         C1      50
 obs.6  160   40         C2      46
 obs.7  160   40         C1      58
 obs.8  160   40         C2      44
 obs.9  180   20         C1      74
obs.10  180   20         C2      81
obs.11  180   20         C1      70
obs.12  180   20         C2      85
obs.13  180   40         C1      69
obs.14  180   40         C2      79
obs.15  180   40         C1      67
obs.16  180   40         C2      81
```

Puis continuez avec les commandes suivantes:

`chem <- read.table("chemical.data", header=T)`	Lit les données dans un tableau de données

`chem[,1] <- as.factor(chem[,1])`	Définit la variable *Temp* en tant que facteur
`chem[,2] <- as.factor(chem[,2])`	De même pour la variable *Conc*
`attach(chem)`	Permet de travailler directement avec les variables *Temp*, *Conc*, ...
`Temp`	Affiche la variable *Temp*
`aov.chem <- aov(Recolte ~ Temp+Conc +Catalyseur)`	Fait une analyse de variance à trois voies
`motif() ; par(mfrow=c(2,2))`	Fenêtre graphique
`plot(aov.chem)`	Deux représentations graphique du modèle
`aov.chem.2 <- aov(Recolte ~ Temp*Conc *Catalyseur)`	Analyse de variance avec toutes les interactions
`summary(aov.chem.2)`	Tableau ANOVA
`plot(aov.chem.2)`	Deux représentations graphique du modèle
`scatter.smooth(1:16,residuals(aov.chem.2))`	Représentation des résidus contre i avec un lissage
`abline(h=0,lty=2)`	Ajoute une ligne horizontale
`interaction.plot(Temp,Conc,Recolte)`	Graphiques pour détecter des interactions
`interaction.plot(Temp,Catalyseur,Recolte)`	
`interaction.plot(Conc,Catalyseur,Recolte)`	

4 Documentation "on-line" des fonctions

L'aide principale dans *S-PLUS* est sa documentation "on-line". Pour n'importe quelle fonction (par exemple `mean()`), il existe un fichier qui comporte sa documentation. On peut l'appeler en tapant la commande `help(mean)` ou plus simplement `?mean`. Si vous vous intéressez, par exemple, à la multiplication matricielle, écrivez `help("%*%")`. Ces documentations sont aussi imprimées dans le *Reference Manual*.

Une documentation bien plus sophistiquée est fournie par la commande

```
help.start(gui="motif").
```

Cliquez dans la partie droite de la fenêtre sur un domaine qui vous intéresse (par exemple "Linear Algebra") et vous verrez apparaître à gauche de la fenêtre une liste de fonctions liées à l'algèbre linéaire. Choisissez-en une, cliquez sur le nom et alors une fenêtre s'ouvrira avec la documentation désirée. Pour une liste de tous les jeux de données inclus dans *S-PLUS*, cliquez sur le domaine "All Datasets".

Ce "*S-PLUS* Help" donne entre autres la possibilité de <u>chercher à l'aide de mots-clés</u>: cliquez dans la boîte "Topic" et écrivez "normal" et *S-PLUS* vous retournera une liste de toutes les fonctions avec le mot clé "normal" dans leur description.

Dans le cadre d'un projet plus élaboré, vous seriez obligés de documenter vos propres fonctions. Le *Programmer's Manual* décrit dans le chapitre 11 comment le faire conformément au style de *S-PLUS*.

5 Vecteurs, matrices, listes, tableaux de données

5.1 Vecteurs

Le vecteur est la structure la plus simple dans *S-PLUS*. Pour créer un vecteur `x` de longueur 4 avec composantes (5, 3.6, 1, -4), utilisez

`x <- c(5,3.6,1,-4)`[2].

Il y a deux autres façons de créer ce même vecteur `x`:

`c(5,3.6,1,-4) -> x`

`x _ c(5,3.6,1,-4)`[3]

C'est la dernière manière qui est le plus fréquemment utilisée, mais au cours de cette introduction, nous allons utiliser la première, parce qu'elle est la plus lisible. Il faut souligner qu'en écrivant simplement `2*x`, *S-PLUS* donne `[1] 10.0 7.2 2.0 -8.0`, mais ce nouveau vecteur est *perdu*. Si vous voulez garder la nouvelle variable, il faut la stocker via l'utilisation de "`<-`" ou bien "`_`".

L'arithmétique sur les vecteurs se fait composante par composante. Essayez `x*x`, `x+2` et `x/x`. Les opérations élémentaires sont `+`, `-`, `*`, `/` et `^` ou `**` pour la puissance. Les symboles "`%/%`" et "`%%`" représentent respectivement la divison entière et l'opérateur *modulo*. *S-PLUS* offre toutes les fonctions usuelles comme `exp()`, `log()`, `log10()`, `sin()`, `cos()`, `tan()`, `sqrt()` (racine carrée). Essayez en donnant comme argument le vecteur `x` les fonctions `min()`, `max()`, `range()`, `sum()`, `prod()`, `length()`, `cumsum()`, `cumprod()` et `summary()`. Les statistiques de base sont `mean()` pour la moyenne arithmétique, `median()` pour la médiane et `var()` pour l'estimateur usuel de la variance, qui n'est rien d'autre que

`sum((x-mean(x))^2)/(length(x)-1)`.

Ceci vous donne déjà une idée de la manière dont vous allez pouvoir écrire vos fonctions personnelles dans *S-PLUS*.

Pour générer des séquences de nombre, *S-PLUS* offre plusieurs possibilités. `x <- 1:10` est le vecteur (1,2,...,10). `10:1` est le même vecteur dans l'autre sens. Essayez `4*1:15`. Posez maintenant `n <- 10` et notez la différence entre les expressions

`1:n-5` et

[2] Notez ici que `c()` est un nom réservé dans *S-PLUS*, c'est-à-dire qu'il ne faut pas créer d'objets qui ont pour nom `c`. La même remarque s'applique à `t()`, la fonction qui transpose une matrice; `T`, le "True" dans des objets booléens; `F`, le "False"; `q()`, qui sert à quitter *S-PLUS*; `C()` qui définit des facteurs pour des modèles; `I()` qui sert à transformer des variables dans les formules et finalement `s()`, qui est une fonction de lissage utilisée avec `gam()`.

[3] Cette notation a comme effet qu'il ne faut pas utiliser des noms du type "`var_test_1`" comme cela se fait couramment dans la programmation avec d'autres langages. Utilisez plutôt la notation "`var.test.1`" ou bien "`VarTest1`".

```
1:(n-5).
```

Vous devriez vous apercevoir que l'opérateur "`:`" a une plus haute priorité.

Utilisez la fonction `c()` pour la concaténation de vecteurs. Posez `x <- c(1,1,1)` et `y <- c(0,0,0)`, puis écrivez `c(x,y,x)`: vous avez ainsi crée le vecteur $(1,1,1,0,0,0,1,1,1)$. D'autres fonctions plus pratiques pour créer des vecteurs particuliers sont `seq()` et `rep()`: `seq(-2,3,by=0.5)` est (-2.0, -1.5, ..., 2.5, 3.0) et `seq(0,1,length=4)` est $(0,\frac{1}{3},\frac{2}{3},1)$. La commande `rep(2,10)` donne un vecteur contenant 10 composantes, toutes égales à 2.

Un vecteur peut aussi admettre pour composantes des caractères: tapez

```
types <- c("A","B","O","OO")
```

pour créer un vecteur contenant les composantes *A, B, O* et *OO*. La commande `paste()` est utile pour construire des vecteurs particuliers. Essayez la commande suivante:

```
niveaux <- paste(rep("X",10),1:10,sep=".")
```

Pour extraire des éléments d'un vecteur, il faut utiliser les crochets: `x[3]` est la 3ème composante de `x`, `x[1:3]` donne les 3 premières, `x[c(2,4)]` la 2ème et 4ème. `x[-3]` supprime la 3ème composante; pour supprimer la 1ère et la 4ème, écrivez `x[-c(1,4)]`.

5.2 Opérateurs booléens

S-PLUS permet aussi de travailler avec des quantités booléennes ou logiques, c'est-à-dire des vecteurs dont les éléments ne peuvent prendre que les 2 valeurs *faux* (notée `F` "False") et *vrai* (notée `T` "True"). Créez `y <- c(4,-2,3,1,-5)`: l'expression `y>0` est le vecteur booléen (`T F T T F`) et l'expression `y[y<0]` est le vecteur numérique (`-2 -5`). Les opérateurs logiques "`<`", "`<=`", "`>`", "`>=`" signifient "plus petit que", "plus petit ou égal", "plus grand que" et "plus grand ou égal". Afin de tester l'égalité entre deux objets, vous devez utiliser le symbole "`==`". Voici un exemple fréquent: `c(1:length(y))[y==-2]` donne la composante de l'observation -2. Le symbole pour l'inégalité est "`!=`". En plus, si `b1` et `b2` sont deux expressions booléennes, `b1 & b2` est leur intersection, `b1 | b2` leur union et {`!b1`}[4] la négation. Pour la documentation, référez-vous à `help(Logic)`.

5.3 Matrices

Pour créer une matrice dans *S-PLUS*, utilisez `matrix()` ou, pour des matrices de dimension plus grande que 2, `array()`. Supposez `x <- c(2,4,3,1,0,1)` et comparez

`Z1 <- matrix(x,ncol=3)` et

[4] Les "{}" sont seulement nécessaires si l'expression est au début de la commande car le "`!`" est réservé pour retourner dans l'environnement UNIX (par exemple `!ls` pour une liste de fichiers du directoire à partir duquel vous avez lancé *S-PLUS*).

```
Z2 <- matrix(x,ncol=3,byrow=T).
```

Les deux sont des matrices 2×3, mais l'attribution des données est différente! Dans la plupart des cas vous allez lire vos matrices ligne par ligne et vous devez alors préciser l'argument `byrow=T`, car `byrow=F` est la valeur par défaut dans `matrix()`. La commande `dim(Z1)` vous donne les dimensions de la matrice. Essayez aussi `Z1+Z2` et `t(Z1)%*%Z2` (il s'agit en fait de la multiplication matricielle $Z_1^t \cdot Z_2$). Si `x` et `y` sont deux vecteurs de même longueur, `as.vector(x%*%y)` donne alors le produit scalaire usuel. Pour extraire des éléments d'une matrice: `Z1[1,2]` donne l'élément (1,2), `Z1[,2]` la 2ème colonne, `Z1[1,]` la 1ère ligne, `Z1[,c(1,3)]` les colonnes 1 et 3. Les fonctions `rbind()` et `cbind()` sont utiles pour construire des matrices: écrivez `nouveau <- c(4,5)` et regardez `cbind(Z1,nouveau)`: `cbind()` peut coller des colonnes, `rbind()` de manière similaire des lignes. Consultez la documentation de `dimnames()` pour voir comment on peut donner des noms aux lignes et colonnes.

Voici quelques autres fonctions concernant les matrices: `t()` pour la transposée, `eigen()` pour calculer des valeurs et vecteurs propres d'une matrice carrée, `chol()` pour la décomposition de Choleski, `solve()` pour trouver l'inverse et finalement `svd()` pour la décomposition en valeurs singulières. Souvent, vous avez besoin de calculer des statistiques sur des matrices, comme par exemple trouver la moyenne sur toutes les lignes. Supposez `Z` de dimension 25×3: `apply(Z,1,mean)` vous donne un vecteur contenant les 25 moyennes par ligne. De la même manière, `apply(Z,2,var)` donne les variances estimées des 3 colonnes.

Il est possible de créer des matrices de n'importe quelle dimension. Posez `z <- 1:30` et `Z <- array(z,c(2,3,2))` et regardez comment *S-PLUS* présente une matrice de dimension $2 \times 3 \times 2$. Comparez `Z[1,,1]` et `Z[1,,]`.

5.4 Listes

Une liste est une collection d'objets qui peuvent être de nature différente. En voici un exemple: effectuez une régression simple avec comme prédicteur la quantité de pluie (en "inches") dans six états des USA de 1890 à 1927 (`corn.rain`) et, comme réponse, la récolte de maïs en "bushels per acre" (`corn.yield`). La commande suivante le fait:

```
corn.reg <- lsfit(corn.rain,corn.yield)
```

La variable `corn.reg` contient les résultats de la régression et elle a la forme d'une liste. Ecrivez `names(corn.reg)` pour voir les 4 composantes `"coef"`, `"residuals"`, `"intercept"` et `"qr"`. Vous pouvez accéder aux coefficients estimés via la commande `corn.reg$coef` et le résultat est un vecteur de longueur 2 qui contient ainsi une estimation de la pente et de l'ordonnée à l'origine. `corn.reg$residuals` contient les résidus de la régression, dans notre exemple un vecteur de longueur 38. Parfois les composantes d'une liste n'ont pas de noms. Dans ce cas vous pouvez accéder à la deuxième composante avec la commande `corn.reg[[2]]` (tapez `help("[[")` pour plus de détails). Si vous voulez seulement les résidus des observations 18, 20 et 25, tapez

```
corn.reg[[2]][c(18,20,25)].
```

Examinez les autres composantes: celle nommée "`intercept`" vaut "`F`" (pour "False") si la régression a été forcée par l'origine, sinon elle vaut "`T`" ("True"). Celle nommée "`qr`" est elle-même une liste et elle contient une décomposition numérique de la matrice des prédicteurs (voir `help(qr)`).

Pour créer une liste, il existe deux possibilités:

1. Si vous connaissez les éléments à l'avance, écrivez

   ```
   res.1 <- list(val.propres=vp1, vec.propres=vp2).
   ```

   La commande `res.1$val.propres` contient donc `vp1`.

2. Si vous ne savez pas à l'avance quels en seront les éléments, écrivez

   ```
   res.2 <- vector("list",3).
   ```

   L'argument "3" indique la longueur de la liste. L'attribution se fait avec les doubles crochets: par exemple `res.2[[1]] <- c(1:10)`. Pour donner des noms aux composantes, on utilise:
   `names(res.2) <- c("c1","c2","c3")`.

Notez ici une subtilité dans l'utilisation des listes: le résultat de la commande `corn.reg[1]` est différent de celui de `corn.reg[[1]]`. L'utilisation de "`[[`" donne la première composante de `corn.reg` en tant que vecteur, tandis qu'avec "`[`" vous l'obtenez en tant que liste. L'utilisation des crochets simples "`[`" est utile quand vous devez extraire plusieurs composantes d'une liste (par exemple `corn.reg[c(1,3)]`) ou attribuer des éléments à la liste via une boucle: imaginez que vous ayez 10 prédicteurs et vous aimeriez mettre dans un seul objet les 10 régressions simples possibles. Pour faire cela, créez tout d'abord la liste (`reg.10 <- vector("list",10)`) et puis attribuez les résultats des 10 régressions de la manière suivante:

```
for(i in 1:10) reg.10[i] <- lsfit(x[,i],y)
```

5.5 Tableaux de données

Les tableaux de données (en anglais "data frames") sont une généralisation des matrices; un inconvénient des matrices est qu'elles ne permettent pas d'avoir parmi des colonnes numériques une colonne de caractères. Or, dans les applications on est souvent confronté à des variables qui décrivent les différents niveaux d'un facteur (souvent par des caractères). Les tableaux de données surmontent ce problème et facilitent donc énormément le travail. Prenez l'exemple suivant: `catalyst` est un tableau de données avec 4 variables et 8 observations qui fait partie des données existantes dans *S-PLUS*.

```
> catalyst
  Temp Conc Cat Yield
1  160   20   A    60
2  180   20   A    72
3  160   40   A    54
```

```
4  180  40  A  68
5  160  20  B  52
6  180  20  B  83
7  160  40  B  45
8  180  40  B  80
```

Vous pouvez accéder à la variable "`Temp`" de différentes manières:

1. `catalyst[,1]` (type matrice)
2. `catalyst$Temp` (type liste)
3. `attach(catalyst)` et ensuite simplement écrire `Temp`[5]

Il est possible d'utiliser les fonctions `cbind()` et `rbind()` avec des tableaux de données. Lisez à ce sujet le chapitre 10 et surtout le "Blue Book" qui traite en détail tous les aspects des tableaux de données.

[5] Notez qu'il est aussi possible d'utiliser `attach()` avec une liste.

6 Lecture et écriture de données

6.1 Lecture de données

Les données à analyser se trouvent normalement dans des fichiers externes à *S-PLUS*. Pour les lire, on utilise les fonctions `scan()` et `read.table()`.

Dans le cas le plus simple où vous voulez créer un vecteur, vous avez les possibilités suivantes: soit écrire `z <- c(5, 2 ,8 ,10)` ou bien `z <- scan()`, puis entrer les données une à une avec des "return"; pour terminer, tapez "return" deux fois. Pour lire des vecteurs de caractères: `z <- scan(what="")`.

Si, par exemple, vos données se trouvent dans un fichier "*test.dat*", placez-le dans le directoire dans lequel vous voulez travailler avec *S-PLUS*. Vous pouvez lire ce vecteur par la commande `z <- scan("test.dat")`. Dans le cas où votre fichier est structuré comme une matrice (disons avec 10 colonnes) entièrement numérique, la commande typique est:

```
z <- matrix(scan("nom.dat"), ncol=10, byrow=T)
```

Il est aussi possible de lire le fichier directement dans une liste (lisez à ce sujet le paragraphe "Details" dans `help(scan)`).

La fonction `read.table()` permet de lire des données d'une structure plus générale. Supposez que votre fichier "*test.dat*" a l'allure suivante:

	Var1	Var2	Var3
1	5	A	3
2	1	B	4
⋮	⋮	⋮	⋮
19	0	B	-2
20	1	C	1.5

`X <- read.table("test.dat",header=T)` crée le tableau de données `X` (cf. chapitre 5.5). Consultez `help(read.table)` si vous avez des données plus particulières, la fonction a une multitude d'options.

Pour contrôler vos données ou pour faire des changements, utilisez `data.ed()`: une fenêtre s'ouvre avec votre jeu de données. Cliquez sur l'observation que vous aimeriez changer et puis cliquez sur l'option "Commit". Cependant, cet éditeur est un peu limité (vous ne pouvez pas, par exemple, ajouter de nouvelles colonnes ou lignes), mais il est très utile pour de grands jeux de données où il est parfois difficile d'identifier la bonne observation à éditer.

6.2 Ecriture de données

La fonction `sink()` est utile pour sauver rapidement les résultats d'une session de *S-PLUS* dans un fichier. Ecrivez simplement `sink("output.session1")` et tout ce que vous faites par la suite dans *S-PLUS* sera écrit dans le fichier "*output.session1*". Pour revenir dans le programme, tapez `sink()`.

Pour sortir des matrices ou des tableaux de données de façon à ce qu'ils puissent être utilisés par d'autres programmes, écrivez

> `write(format(t(x)),"x.dat",ncol=ncol(x))`
> (`x` est la matrice ou le tableau de données en question, `x.dat` le nom du fichier à créer et `ncol(x)` le nombre de colonnes de `x`)

Cette dernière commande est un peu lourde, mais la version 3.2 de *S-PLUS* fournira une fonction plus sophistiquée appelée `write.table()`.

7 Les graphiques

Les possibilités graphiques de *S-PLUS* impressionnent beaucoup l'utilisateur. Elles sont particulièrement simples à utiliser et offrent une multitude d'options pour les personnaliser. On va présenter l'utilisation des fonctions graphiques dans un environnement "Workstation" où vous pouvez travailler simultanément avec plusieurs fenêtres. Dans le cas où vous ne pouvez travailler que sur un terminal simple, référez-vous au chapitre 2.4.8 du *User's Manual* (et aussi à la documentation sur `postscript()`).

7.1 Fenêtre graphique et impression

Les fenêtres graphiques les plus communes sont `X11()` et `motif()`. Pour cette introduction, utilisez `motif()` car il s'avère que l'utilisation de `X11()` pourrait être supprimée. A titre illustratif, exécutez les commandes suivantes:

- `motif()`
- `x <- seq(-2*pi,2*pi,length=100)`
- `plot(sin(x),ylim=c(-1.5,1.5),xlab="x",main="Premiere impression")`

Un sinus pointillé devrait apparaître sur votre écran. Cliquez sur le menu "Options" et choisissez le menu "Printing". Dans la boîte "Command" vous devez maintenant inscrire la commande pour imprimer depuis UNIX: par exemple `lp -dxxx`, où `xxx` est le nom d'une imprimante laser reliée à votre ordinateur. Cliquez sur "Save" et puis "Print" et *S-PLUS* crée un fichier postscript (nommé `ps.out.0001.ps` ou similaire) qui sera sauvegardé dans le directoire à partir duquel vous avez lancé *S-PLUS*[6]. Une fois l'imprimante choisie, vous n'avez plus qu'à cliquer sur "Print" dans le menu "Graph".

7.2 Les principales fonctions graphiques

Nous présentons dans ce chapitre les principales fonctions graphiques. Toutes ces commandes admettent des paramètres servant à particulariser le graphique: ils seront présentés dans le chapitre suivant.

`plot(x,y,...)`	Cette fonction va réagir très différemment selon l'argument que vous lui donnez: `plot(x)` fait un graphique de x versus l'indice de l'observation; `plot(x,y)` produit un "scatterplot" de x et y; la fonction `plot()` est - comme le sont beaucoup d'autres - *générique*, ce qui signifie que le résultat dépend essentiellement de l'argument que l'on donne (cf. le chapitre 10).

[6] Ceci vous donne la possibilité de garder vos graphiques pour des impressions ultérieures. Important: changez le nom du fichier en un nom plus parlant (par exemple, dans UNIX, `mv ps.out.1.ps Temp.vs.Recolte.ps`). Si vous voulez seulement créer un fichier en postscript, sans l'imprimer, donner un nom "bidon" pour l'imprimante; *S-PLUS* va créer le fichier postscript, mais ne pourra pas l'imprimer.

`points(x,y,...)`	Ajoute des points à un graphique déjà existant (doit suivre une commande `plot()`).
`lines(x,y,...)`	Ajoute des lignes à un graphique déjà existant, similairement à `points()`.

`abline(a,b,...)`
`abline(v=x,...)`
`abline(h=y,...)`
	Ajoute une ligne de pente b et d'ordonnée à l'origine a, une (des) ligne(s) verticale(s) de coordonnée(s) x ou une (des) ligne(s) horizontale(s) de coordonnée(s) y à un graphique déjà existant.

`text(x,y,labels,...)`	Permet d'ajouter du texte au point(s) donné(s) par x et y. Essayez • `x <- seq(0,2*pi,length=26)` • `plot(sin(x),cos(x),type="n")` • `text(sin(x),cos(x),labels=letters)` Utilisez `locator(1)` à la place de x et y, si vous voulez choisir l'endroit du texte avec la souris.
`legend(x,y,legend,...)`	Permet d'ajouter une légende à un graphique. Consultez `help(legend)` pour des exemples d'utilisation.

`tsplot(x1,x2,...)`	Dessine un nombre quelconque de séries chronologiques à la même échelle, avec différents styles de lignes.
`qqnorm(x,...)` `qqplot(x,y,...)`	Sert à comparer les quantiles d'une loi quelconque avec ceux de la loi Normale (`qqnorm()`) et entre deux vecteurs quelconques (`qqplot()`).
`hist(x,...)`	Histogramme d'un vecteur. Consultez `help(hist)` pour voir comment changer certains paramètres.
`boxplot(x,y,...)`	Boxplot d'un ou plusieurs vecteurs à la même échelle. Lisez aussi la documentation `help(bxp)`; c'est la fonction qui dessine réellement les boxplots.
`pie(x,names=noms,...)`	Produit des "gâteaux" pour répartir un tout en morceaux proportionnels aux tranches. Voir les exemples dans `help(pie)` pour personnaliser la sortie.

`persp(x,y,z,...)` `contour(x,y,z,...)`	Graphiques pour fonctions bidimensionelles: la surface avec `persp()` et les lignes de niveaux avec `contour()`.

Consultez la rubrique "High-Level Plots" dans `help.start()` pour d'autres fonctions intéressantes. Notamment `pairs()` pour un tableau de "scatterplots" si vous avez des données multivariées, `brush()` et `spin()` pour des graphiques dynamiques et `coplot()` pour des graphiques conditionnels.

7.3 Quelques paramètres graphiques

Un exemple d'un paramètre graphique: `type=""`. Dans la figure 1 sont représentées les différentes possibilités d'affichages.

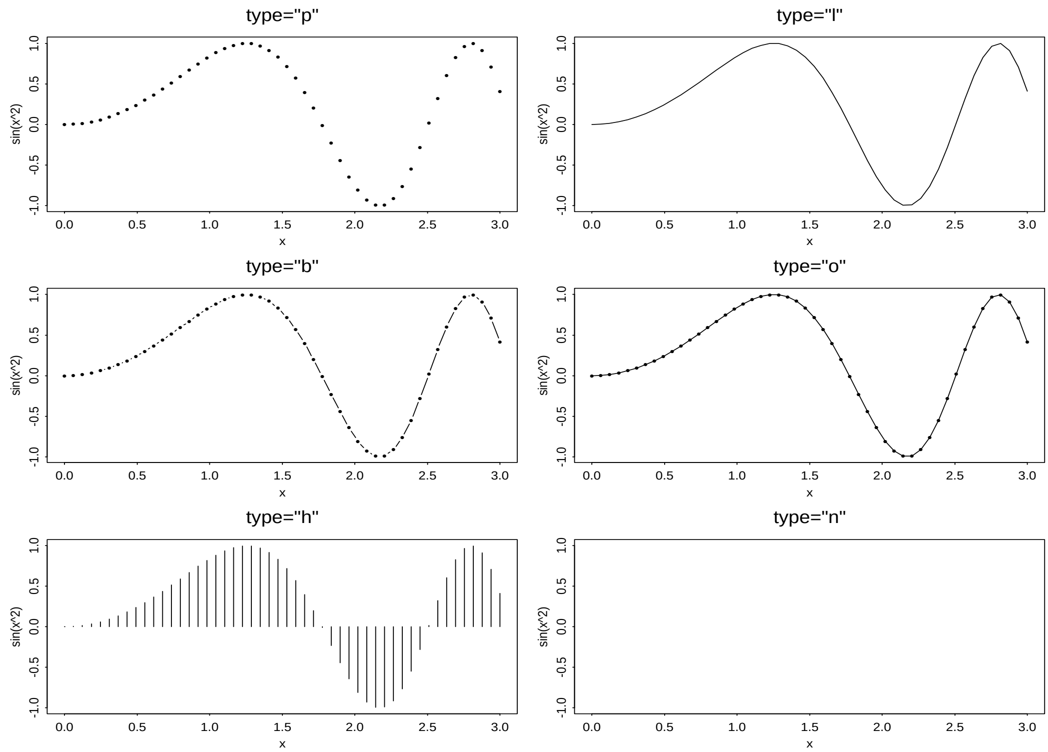

Figure 1: Applications du paramètre `type=""`

Vous trouvez la <u>documentation de tous les paramètres graphiques</u> via `help(par)`. La syntaxe en est la suivante: soit vous mettez les paramètres comme arguments dans la fonction graphique (par exemple `plot(x,f(x),type="l")` pour relier les points par des lignes), soit vous les utilisez avec la fonction `par()` (exemple: `par(pch="+")` pour changer le symbole dans les graphiques). Voici une liste d'arguments que l'on utilise fréquemment.

`type=" "` — Contrôle le type de dessin: `"p"` pour points (par défaut), `"l"` pour ligne, `"b"` pour points et lignes non reliés, `"o"` pour points et lignes reliés, `"h"` pour des lignes verticales, `"s"` et `"S"` pour des "escaliers" et finalement `"n"` pour aucune représentation. La dernière option est utile quand vous devez superposer des graphiques.

`pch=" "` ou `pch=n` — Contrôle le symbole graphique: `pch="k"` va dessiner des lettres "k" à la place des points habituels. Vous pouvez mettre n'importe quel symbole de votre clavier. (Remarque: `pch="."` donne des points très petits, ceci est souvent utile quand vous avez un nombre élevé de points). `pch=n` (`n=0,1,...,18`) donne un symbole particulier. Faites

```
legend(locator(1), as.character(0:18),
marks=0:18, cex=1.5)
```

pour consulter une liste de tous les symboles disponibles (voir aussi le chapitre 14.1).

`lty=n` — Contrôle le style de ligne (`n=1,2,...,8`).

```
legend(locator(1), as.character(1:8), lty=1:8,
cex=2)
```

vous donne une liste de tous les styles disponibles (voir aussi le chapitre 14.1).

`cex=x` — Contrôle la taille des caractères. Si `x=0.6`, tous les caractères apparaissent 0.6 fois plus petits.

`xlim=c(a,b)`
`ylim=c(a,b)` — Contrôle les limites pour les axes x et y. `xlim=c(2,5)` limite l'axe des x entre 2 et 5. Souvent utilisé pour faire plusieurs graphiques à la même échelle.

`xlab=" "`
`ylab=" "` — Permet de donner des noms aux axes. Par défaut, ce sont les noms des variables `x` et `y` qui sont utilisés.

`title(" ")`
`title(sub=" ")` — Affiche des titres, par défaut en haut de la page; en choisissant `title(sub=" ")`, vous pouvez mettre un titre en bas de la page.

`new=T`	Force la superposition d'un nouveau graphique sur le graphique courant. Peut seulement être utilisé avec la commande `par()` et non pas à l'intérieur d'une autre fonction.

`axes=F`
`xaxt=" "`
`yaxt=" "`
	Ces commandes permettent de particulariser les axes. Utilisez `xaxt="n"` si vous changer la légende de l'axe des x. Pour l'accomoder par la suite, consulter la documentation sur `axis()`.

Vous pouvez identifier interactivement avec la souris des points sur un graphique. Une illustration:

- `attach(cu.summary)` (un jeu de données de caractéristiques de 117 voitures)
- `plot(Mileage,Price)`
- `identify(Mileage,Price,labels=row.names(cu.summary),n=4)`
  Puis cliquez près des 4 points dont vous aimeriez savoir à quelle voiture ils correspondent.

Pour faire plusieurs dessins sur une page (un exemple est la figure 1), utilisez `par(mfrow=c(3,2))`: la page sera subdivisée en 3×2 graphiques qui sont remplis ligne par ligne. Si vous désirez faire les graphiques colonne par colonne, utilisez `par(mfcol=c(3,2))`. La commande `split.screen()` permet de subdiviser la fenêtre graphique de manière plus générale et aussi de manière interactive. La figure 5 du chapitre 13.2 est un exemple d'application de `split.screen()`. Vous trouvez une description de la commande dans le chapitre 9.2.2 du "White Book" et naturellement "on-line" avec `help(split.screen)`. Consultez aussi dans `help(par)` les paragraphes sur `fig, fty, mai, mar, oma` et `omi` si vous devez changer la taille, la forme ou la position des graphiques.

8 Comment écrire des fonctions?

La référence pour ce chapitre est le *Programmer's Manual* de Statistical Sciences, Inc. Nous vous conseillons vivement la lecture de ce livre au cas où vous aimeriez entreprendre un sérieux travail de programmation en *S-PLUS*.

8.1 Un exemple

Nous avons déjà vu plusieurs fois que *S-PLUS* offre la possibilité d'écrire ses propres fonctions, c'est-à-dire que vous pouvez programmer vous-même plus ou moins tout ce que vous voulez. Pour avoir une idée de la syntaxe, tapez `mean` dans *S-PLUS*; vous pouvez voir ce qui suit:

```
> mean
function(x, trim = 0, na.rm = F)
{
        if(na.rm)
                x <- x[!is.na(x)]
        else if(any(is.na(x)))
                return(NA)
        if(mode(x) == "complex") {
                if(trim > 0)
                        stop("trimming not allowed for complex data")
                return(sum(x)/length(x))
        }
        x <- as.double(x)
        if(trim > 0) {
                if(trim >= 0.5)
                        return(median(x))
                n <- length(x)
                i1 <- floor(trim * n) + 1
                i2 <- n - i1 + 1
                x <- sort(x, unique(c(i1, i2)))[i1:i2]
        }
        sum(x)/length(x)
}
```

La ligne principale se trouve à la fin du programme, où `sum(x)/length(x)` calcule la moyenne de `x` et c'est aussi la sortie de la fonction. L'argument `trim = 0` est une option: elle enlève les valeurs les plus extrêmes dans `x`, ce qui donne un estimateur plus robuste. La ligne avec `x <- x[!is.na(x)]` vous montre la façon standard d'enlever toutes les données manquantes[7] d'un vecteur.

8.2 Syntaxe et `fix()`

La syntaxe générale pour les fonctions est:

[7] Le symbole pour les données manquantes est "`NA`" (Non-Available).

`noms <- function( ... ) { ... }`
Les parenthèses après `function` contiennent les paramètres de la fonction; les accolades contiennent le programme principal.

Vous pouvez utiliser des fonctions à l'intérieur de fonctions et il est possible de les appeler de manière récursive.

Pour écrire vos fonctions personnelles, nous vous conseillons d'utiliser la commande `fix()`. Choisissez d'abord votre éditeur de texte préféré (par exemple *vi, ame, (x)emacs, xedit* ou bien *asedit* sur nos stations Silicon Graphics) avec la commande

`options(editor="ame")`[8]

puis tapez `fix(nom.de.fonction)`. Comme exercice, écrivez une fonction pour calculer le déterminant d'une matrice carrée, fonction qui n'existe pas encore dans *S-PLUS*. Tapez donc `fix(absdet)` et une fenêtre de l'éditeur choisi s'ouvre avec la structure de base des fonctions. Ecrivez maintenant dans le fichier le code suivant[9]:

```
function(x)
{
   if(!is.matrix(x))
           stop("Ce n'est pas une matrice")
   if((dim(x)[1])!=(dim(x)[2]))
           stop("La matrice n'est pas carree")
   val.singulieres <- svd(x)$d
   prod(val.singulieres)
}
```

Quittez après l'éditeur (seulement écrire sans quitter ne suffit pas) et la nouvelle fonction est prête à être utilisée. Tapez `absdet` pour la voir et `x <- matrix(1:4,ncol=2,byrow=T)` puis `absdet(x)` pour la tester. Avec `fix()` vous pouvez éditer à nouveau votre fonction.

Un autre moyen - certes moins sophistiqué - pour créer des nouvelles fonctions est le suivant: écrivez le code de la fonction dans votre éditeur en changeant la première ligne par

```
absdet <- function(x)
```

et sauvez le fichier sous le nom "*absdet.s*" dans le directoire à partir duquel vous lancez *S-PLUS*. Remarque importante: il est nécessaire de faire un saut de ligne après la dernière accolade du code, sinon *S-PLUS* ne pourra pas le lire! Exécuter après dans *S-PLUS* la commande

[8] On pourrait mettre cette commande dans la fonction `.First()` (voir chapitre 11).

[9] C'est en effet la valeur absolue du déterminant. Vous pourriez également utiliser le fait que le déterminant est le produit des valeurs propres, fournies par `eigen()`, mais cette méthode est moins stable. Voir aussi `qr()`.

```
source("absdet.s")
```

pour ainsi créer la nouvelle fonction. La fonction `source()` est aussi utile dans le cas suivant: vous avez une suite de commandes à exécuter et ceci plusieurs fois; écrivez alors ces commandes dans un fichier et puis utilisez `source("nom.du.ficher")` pour les exécuter.

8.3 D'autres remarques concernant les fonctions

Les options dans `function()`: la fonction `args()` vous donne les options de n'importe quelle fonction. Faites `args(mean)` et *S-PLUS* vous donne:

```
function(x, trim = 0, na.rm = F)
```

`x` est donc un argument nécessaire dans l'appel de `mean()`, tandis que `trim = 0` et `na.rm = F` indiquent les valeurs par défaut d'options que vous pouvez adapter vous-même. Regardez maintenant `args(hist)`:

```
function(x, nclass, breaks, plot = TRUE, probability = FALSE,
..., xlab=deparse(substitute(x)))
```

`x` est toujours un argument nécessaire, mais `nclass` et `breaks` sont seulement des options sans valeur par défaut. Notez l'argument particulier "`...`": ceci permet d'ajouter d'autres arguments, notamment des options graphiques. Un exemple est `hist(x, main="Titre")` qui ajoute directement un titre à votre graphique.

Si vous écrivez de longues fonctions, vous devez souvent chercher des erreurs dans le programme. Mettez alors simplement la ligne `browser()` dans votre code et à chaque fois que le programme arrive à cet endroit, il s'arrête et vous donne la possibilité de tester si vos variables créées sont correctes. Tapez le nom de la variable et *S-PLUS* vous montre son contenu. Un exemple d'une application plus sophistiquée du `browser()`:

```
teste <- function(x,y,w=rep(1,length(x)),debug=F)
{     [votre programme]
      if(debug) browser()
      [reste du programme]
}
```

Cette fonction permet de trouver des erreurs possibles; appelez `teste` avec `teste(x1,y1, debug=T)`. Remarque: écrivez `0` pour quitter le `browser()`.

D'autres fonctions utiles dans la programmation:

- `for(), For()` pour faire des boucles:
  Un exemple: l'estimation de l'écart-type de la moyenne par la méthode du "jackknife".

Un code simple serait:

```
jackknife <- function(x)
 {n <- length(x)
  u <- rep(0,n)
  for(i in 1:n) {
        u[i] <- mean(x[-i])
  }
ecart.type <- sqrt(((n-1)/n) * sum((u-mean(u))^2))
ecart.type
}
```

Le `for()` peut aussi être utilisé au niveau principal: supposez que vous avez une matrice `x` qui contient dans ses colonnes 9 séries chronologiques. Pour les imprimer dans 9 fenêtres graphiques différentes sur une page, vous pourriez écrire:

```
par(mfrow=c(3,3))
for(i in 1:9) plot(x[,i],main=paste("Serie ",i))
par(mfrow=c(1,1))
title("Presentation graphiques de 9 series chronologiques")
```

Remarque: *S-PLUS* peut avoir des problèmes de mémoire si vous faites un nombre très élevé de boucles, même si elles contiennent des instructions très simples. A partir de la version 3.1, *S-PLUS* offre une option: la fonction `For()`. Bien que l'utilisation soit un peu plus compliquée que celle de `for()`, `For()` est plus rapide pour des boucles contenant des instructions plus complexes car elle calcule chaque itération séparément, ce qui évite quelques problèmes de mémoire. La stratégie principale consiste néanmoins à éviter les `for()` le plus souvent possible en utilisant d'autres fonctions de *S-PLUS* (regardez plus particulièrement sous `apply()`, `tapply()`, `sapply()` et `lapply()`). Les chapitres 12 et 13 du *Programmer's Manual* contiennent plusieurs autres remarques concernant ce sujet. Si ceci n'est pas possible, il faudra programmer les instructions dans un langage autre que *S-PLUS* [10].

- `if(), ifelse()`:
  La syntaxe est:

  `if(test) expr.TRUE` ou bien

[10] *S-PLUS* permet de faire des interfaces avec *Fortran* et *C*. Consultez le *Programmer's Manual* et la documentation sur `.Fortran` et `.C`. Pour des exemples simples illustrant le travail avec *C*, cf. le livre de W. Haerdle, *Smoothing Techniques with Implementation in S*, Springer Verlag, 1991.

```
if(test) expr.TRUE else expr.FALSE
```

Dans le cas où vous avez plusieurs instructions à exécuter, mettez `expr.TRUE` entre accolades. La fonction `ifelse()` est pratique dans le cas où le conditionnement est simple. Sa syntaxe est:

```
ifelse(teste, yes, no)
```

La condition dans `if()` peut être divisée en plusieurs parties. Si `c1` et `c2` sont deux valeurs logiques, l'expression `c1 && c2` vaut "T" si les deux expressions valent "T" (si `c1` est "F", `c2` n'est pas évaluée). L'expression `c1 || c2` vaut "T" si une des deux expressions vaut "T" (si `c1` est "T", `c2` n'est pas évaluée).

- <u>`menu(), switch()`</u>:
  Les commandes `menu()` et `switch()` permettent d'évaluer une instruction parmi plusieurs autres. Un exemple:

```
menu.test <- function(x, y)
{
   choix <- c("Moindres carres", "Regression LMS", "Regression L1")
   reponse <- menu(choix)
   switch(reponse,
      lsfit(x, y),
      lmsreg(x, y),
      l1fit(x, y),
      stop("Mauvais choix!"))
}
```

Cette fonction vous donne le choix entre la régression des moindres carrés usuelle, la régression robuste LMS ("Least Median of Squares") et la régression en norme L_1.

- <u>`while(), repeat, break`</u>:
  La syntaxe est:

```
while(condition) expression
```

Si vous ne pouvez pas facilement exprimer la condition d'arrêt pour un `while()`, utilisez `repeat` avec `break`:

```
repeat {
   [votre programme]
   if(condition) break
   [reste du programme]
   }
```

9 Fonctions statistiques

9.1 Densités, probabilités, quantiles, nombres aléatoires

Avec *S-PLUS*, vous n'avez plus besoin de consulter les tables numériques qui d'ailleurs sont souvent incomplètes et difficiles à lire. Le programme fournit un système très simple pour calculer des densités, quantiles et probabilités de nombreuses variables aléatoires et aussi pour générer des nombres aléatoires qui sont à la base de toute simulation.

Nous allons d'abord expliquer les opérations principales en prenant l'exemple de la loi Normale. Ces techniques s'appliquent facilement aux autres lois (voir la liste en fin de chapitre). Tous les noms des fonctions de ce chapitre sont construits de la manière suivante: prenez tout d'abord comme racine le nom de la loi en question (`norm` pour la loi Normale, `binom` pour la loi Binomiale etc.) et ajoutez au début du nom une des 4 lettres suivantes:

- d: pour calculer des valeurs de la densité (`dnorm()`)
- p: pour calculer des probabilités (`pnorm()`)
- q: pour calculer des quantiles (`qnorm()`)
- r: pour générer des nombres aléatoires (`rnorm()`)

Regardez la figure suivante[11]:

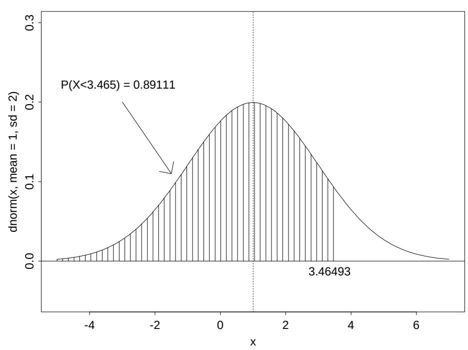

Figure 2: Loi Normale $N(\mu = 1, \sigma^2 = 4)$: quantiles et probabilités

[11] Elle a été construite par les commandes suivantes:

```
x <- seq(-5,7,length=500)
plot(x,dnorm(x,mean=1,sd=2),type="l",ylim=c(-0.05,0.3))
y <- seq(-5,x[353],length=50)
lines(y,dnorm(y,1,2),type="h")
abline(h=0)
abline(v=1,lty=2)
text(locator(1),"3.46493")
arrows(-3,0.2,-1.5,0.11,open=T,rel=T)
text(locator(1),"P(X<3.465) = 0.89111")
```

Soit une loi Normale de moyenne égale à 1 et de variance égale à 4. Vous pouvez calculer la valeur de la densité à chaque point x ($f(x) = \frac{1}{2\times\sqrt{2\pi}} e^{-\frac{1}{2\times 4}(x-1)^2}$) avec `dnorm(x,mean=1,sd=2)`. La probabilité qu'une valeur soit plus petite que x (c'est la surface hachurée dans la figure 2) se calcule via `pnorm(x,1,2)`. De manière similaire, vous obtenez le quantile correspondant à la probabilité p avec `qnorm(p,1,2)`.

En ce qui concerne la génération de nombres aléatoires, écrivez simplement `rnorm(20, mean=1, sd=2)` pour obtenir un échantillon d'une loi $N(\mu = 1, \sigma^2 = 4)$ de taille 20. Tous les générateurs sont basés sur un générateur de nombres aléatoires uniformes entre 0 et 1. Il est parfois important de pouvoir reproduire les mêmes nombres aléatoires: utilisez `set.seed()` pour initialiser le générateur et lisez la documentation `help(.Random.seed)`.

Finalement, voici la liste de toutes les densités et lois de probabilités dont vous pouvez calculer les probabilités et les quantiles et générer des échantillons, ainsi que leurs paramètres (on ne donnera pas toutes les fonctions de densité et leur support).

Nom	**Loi**	**Densité**	**Paramètre(s)**
`beta`	Bêta	$f(x) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}$	`shape1=`α, `shape2=`β
`binom`	Binomiale	$f(x) = \binom{n}{x} p^n(1-p)^{n-x}$	`size=`n, `prob=`p
`cauchy`	Cauchy	$f(x) = \beta/(\pi[\beta^2 + (x-\alpha)^2])$	`location=`α, `scale=`β
`chisq`	χ^2		`df=` degrés de liberté
`exp`	Exponentielle	$f(x) = \lambda e^{-\lambda x}$	`rate=`λ
`f`	Loi F de Fisher		`df1=` degrés de liberté du numérateur, `df2=` degrés de liberté du dénominateur
`gamma`	Gamma	$f(x) = e^{-x}x^{\alpha-1}/\Gamma(\alpha)$	`shape=`α
`geom`	Géométrique	$f(x) = (1-p)^n p$	`prob=`p
`hyper`	Hypergéométrique		`m=` # de boules blanches `n=` # de boules noires `k=` # de boules tirées
`lnorm`	Log-normale		`meanlog=` moyenne `sdlog=` écart-type
`logis`	Logistique	$\frac{1}{\beta} e^{-(x-\alpha)/\beta}[1 + e^{-(x-\alpha)/\beta}]^{-2}$	`location=`α, `shape=`β
`nbinom`	Binomiale négative		`size=` # de succès `prob=` probabilité d'un succès
`norm`	Normale	$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}$	`mean=`μ, `sd=`σ
`pois`	Poisson	$f(x) = e^{-\lambda}\lambda^x/x!$	`lambda=`λ
`t`	Loi t de Student		`df=` degrés de liberté
`unif`	Uniforme	$f(x) = 1/(\beta-\alpha)$	`min=`α, `max=`β
`weibull`	Weibull	$f(x) = cx^{c-1}\exp(-x^c)$	`shape=`c

9.2 Quelques tests standards, rééchantillonnage

S-PLUS contient quelques fonctions qui effectuent des tests statistiques classiques comme le test t de Student ou le test non paramétrique des rangs de Wilcoxon. L'utilisation des ces fonctions est simple; consultez entre autres les documentations concernant `t.test()`, `binom.test()`, `chisq.test()` et `wilcox.test()` (la rubrique "Statistical Inference" dans le "*S-PLUS* Help" donne une liste exhaustive).

En ce qui concerne les applications de techniques plus modernes comme le "bootstrap", installez le module `bootstrap.funs` qui se trouve sur `StatLib` (cf. chapitre 12) ou bien écrivez vos propres fonctions.

9.3 Analyse en composantes principales, discrimination, classification

On indique dans ce chapitre quelques fonctions concernant l'analyse de données multivariées.

La discrimination classique est incorporée dans la fonction `discr()`. Consultez la documentation sur `cancor()` si vous êtes intéressé à l'analyse canonique. Vous pouvez obtenir des arbres de classification via `kmeans()` et `hclust()`. La fonction `tree()` (utilisant les idées des modèles décrits dans le chapitre suivant) ajuste des arbres de régression et de classification (*CART: Classification and Regression Trees*, par Breiman, Friedman, Olshen et Stone, Wadsworth International Group). C'est une fonction exemplaire du point de vue de l'utilisation graphique.

La fonction `prcomp()` permet d'effectuer l'analyse en composantes principales (ACP). Nous allons l'illustrer sur un jeu de données concernant la structure fonctionnelle des dépenses de l'état français entre 1872 et 1971. Les données sont présentées en appendice (chapitre 14.2). Lisez-les dans un tableau de données *S-PLUS* avec la fonction `read.table()`, en lui donnant le nom `depenses`. Il est souvent préférable de standardiser les données afin de faire l'analyse en composantes principales sur la matrice des corrélations plutôt que sur la matrice de variances-covariances. Créez `dep.standard <- scale(depenses)`. La commande

```
depenses.acp <- prcomp(dep.standard)
```

effectue l'analyse en composantes principales. Faites la projection sur les deux axes principaux:

- `plot(depenses.acp$x[,1:2],pch="+",xlab="Axe 1",ylab="Axe 2")`
- `identify(depenses.acp$x[,1:2],label=row.names(depenses),n=24,cex=0.7)`
  Cliquez près des 24 points pour les identifier.
- `abline(h=0,v=0,lty=2)`

Vous obtenez le graphique suivant, avec un groupement évident des années correspondant aux périodes avant-, entre- et après les guerres mondiales:

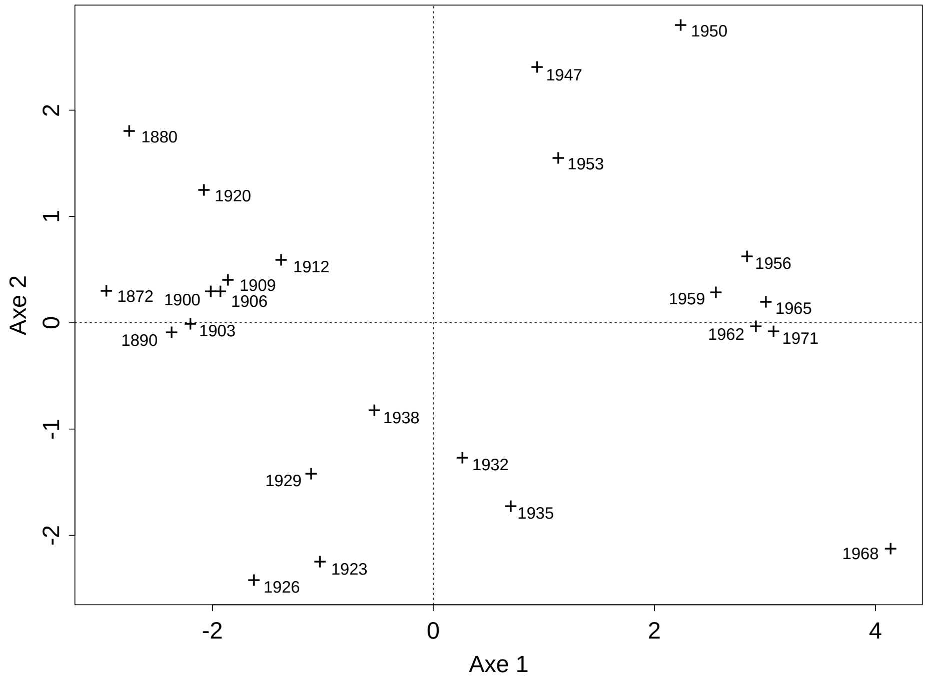

Figure 3: ACP des dépenses de l'état français

La projection sur un espace à deux dimensions explique 41.7% de la variance. Pour le vérifier mettez `vp <- depenses.acp$sdev` et calculez `cumsum(vp)/sum(vp)*100`.

9.4 Séries temporelles

Le domaine des séries temporelles devient de plus en plus important dans la recherche et l'application des statistiques. *S-PLUS* offre une trentaine de fonctions liées aux séries temporelles. On ne vous en indique ici que les fonctions principales.

Pour créer des objets *séries temporelles*, utilisez `ts()` et pour les représenter graphiquement, `tsplot()`. Vous pouvez modéliser des séries autorégressives avec `ar()`; `ar.gm()` utilise une technique robuste. `arima.mle()` permet d'ajuster des modèles ARIMA et `arima.sim()` de les simuler. Consultez la rubrique "Time Series" dans le "*S-PLUS* Help" pour une liste exhaustive de fonctions liées aux séries temporelles ainsi que le chapitre 17 dans le *S-PLUS User's Manual*.

10 Les modèles dans *S-PLUS*

S-PLUS s'est amélioré avec l'apparition de la version 3.0 qui incorpore pour la première fois une nouvelle approche de modélisation. Tout ce qui sera décrit dans ce chapitre se trouve en détail dans le "White Book", i.e. le livre de John M. Chambers et Trevor J. Hastie, *Statistical Models in S*. Nous allons essayer d'en ressortir les points principaux et de décrire les modèles les plus courants. Du point de vue informatique, il s'agit en effet de la programmation orientée objet ("object-oriented programming"). En pratique, ceci veut dire que certaines fonctions se comportent différemment selon l'argument qu'on leur donne. On les appelle aussi *génériques*. Vous trouvez un exemple d'application dans la dernière partie de l'apéritif (chapitre 3.4).

10.1 Formules

Une partie intégrale des modèles sont les formules. Ce sont des expressions symboliques qui traduisent la dépendance entre une variable *réponse* et plusieurs *prédicteurs*. Par exemple, la formule

```
Recolte ~ Temp + Conc
```

se lit: "La `Recolte` est modélisée en fonction de `Temp` (la température) et `Conc` (la concentration)". La modélisation est additive. La formule

```
Recolte/100 ~ sqrt(Temp) + poly(Conc,2)
```

se traduit par: "La `Recolte` divisée par 100 est modélisée en fonction de la racine carrée de `Temp` plus un polynôme quadratique en `Conc`". Notez que les variables qui représentent les facteurs sont directement converties en variables indicatrices[12].

Le tableau suivant présente un résumé (non exhaustif) de la syntaxe des formules. Supposez que `y` et `x, x1, x2, ...` sont des variables numériques, `X` une matrice et `A, B, C, ...` des facteurs.

`y ~ x` `y ~ 1 + x`	Modèles linéaires simples y sur x, avec ordonnée à l'origine
`y ~ -1 + x` `y ~ x - 1`	Modèles linéaires simples y sur x forcé par l'origine
`log(y) ~ x1 + x2`	Modèle linéaire multiple, avec $\log(y)$ sur x_1 et x_2
`y ~ poly(x1,2) + I(x2**2)`	Modéle linéaire multiple, polynomial de degré 2 sur x_1 et sur x_2^2

[12] La fonction `as.factor()` vous permet de définir qu'une variable est catégorique, `is.factor()` de tester si une variable est déjà un facteur ou non. Consultez `help(C)` pour définir quelle transformation en variables indicatrices (c'est-à-dire quel contraste) il faudra utiliser. Pour avoir les contraintes habituelles (qui sont différentes de celles par défaut) sur les paramètres d'une ANOVA, écrivez `options(contrasts=c("contr.sum","contr.treatment"))`.

`y ~ A`	Typiquement une analyse de variance à une voie (facteur A)
`y ~ A + x`	Analyse de covariance: A est le facteur, x la covariable

---

`y ~ A*B`	
`y ~ A+B+A:B`	Deux manières équivalentes pour une analyse de variance à deux voies avec interactions
`y ~ (A+B+C)**2`	
`y ~ A*B*C-A:B:C`	Deux manières équivalentes pour une analyse de variance à trois voies sans interactions de degré 3

Vous pouvez sauver vos formules avec `formula()`: écrivez `f1 <- formula(y ~ x+y)` et utilisez un des modèles décrits dans les deux prochains chapitres, par exemple `lm(f1,data=X)`.

10.2 Modèles linéaires: `lm()`

La fonction `lm()` est une version plus sophistiquée de `lsfit()`, régression usuelle des moindres carrés. Nous montrons dans ce chapitre de manière générale l'utilisation de ces modèles. La technique se transfère évidemment aux fonctions présentées dans les chapitres qui suivront.

Lisez le jeu de données suivant dans un tableau de données avec nom `voitures`. Il contient des données décrivant des caractéristiques de 18 voitures[13].

```
           Prix Puissance Vitesse
Voiture.1 30570        79     165
Voiture.2 39990        85     160
Voiture.3 29600        68     152
Voiture.4 28250        59     151
Voiture.5 34900        98     165
Voiture.6 35480        82     160
Voiture.7 32300        79     154
Voiture.8 32000        55     140
Voiture.9 47700       128     180
```

[13] Source: G. Saporta, *Probabilités, Analyse des Données et Statistique*, Technip, 1990

```
Voiture.10 26540       55     140
Voiture.11 42395      109     175
Voiture.12 33990       82     158
Voiture.13 43980      115     160
Voiture.14 35010       98     167
Voiture.15 39450       80     144
Voiture.16 27900       83     165
Voiture.17 32700      100     173
Voiture.18 22100       68     140
```

Afin d'analyser un premier modèle, vous avez les trois possibilités suivantes:

- Faites `lm.v1 <- lm(Prix ~ Puissance+Vitesse,data=voitures)`.
- Faites `attach(voitures)` et puis `lm.v1 <- lm(Prix ~ Puissance+Vitesse)`.
- Ou bien simplement: `lm.v1 <- lm(Prix ~ ., voitures)`.

Tapez `lm.v1` ou bien `summary(lm.v1)` pour obtenir un résumé des statistiques usuelles d'une régression. Vous pouvez extraire les coefficients via `coef(lm.v1)` et les résidus avec `resid(lm.v1)`. Des commandes équivalentes sont `lm.v1$coef` respectivement `lm.v1$res`. Faites `names(lm.v1)` pour voir une liste de tous les arguments de `lm.v1`. La commande `plot(lm.v1)` vous donne deux résumés graphiques du modèle, à savoir la réponse versus les valeurs ajustées, ainsi que les valeurs absolues des résidus versus les valeurs ajustées.

Dans cet exemple, il serait intéressant de comparer ce premier modèle avec un modèle plus simple, où l'on ne tient plus compte de `Vitesse`. Une manière élégante d'ajuster ce nouveau modèle (valable surtout pour des formules plus compliquées) est la suivante:

```
lm.v2 <- update(lm.v1, . ~ .  - Vitesse)
```

10.3 Autres fonctions liées aux modèles

Voici une liste regroupant d'autres modèles dans le même esprit que `lm()`. L'utilisation de ces fonctions est semblable avec néanmoins quelques spécialisations d'un cas à l'autre. Les descriptions sont très brèves.

- `aov()`:
  C'est la fonction pour l'analyse de variance classique, basée sur des plans d'expériences. Elle accepte aussi des plans d'expériences non équilibrés.
- `glm()`:
  Les modèles linéaires généralisés: c'est l'application des techniques présentées dans le livre *Generalized Linear Models*, de P. McCullagh et J. A. Nelder et dans le logiciel *GLIM*. Vous définissez le type de modèle à utiliser via l'argument `family=" "` par un appel

de `glm()`: `family="gaussian"` est le modèle usuel, `family="binomial"` celui pour la régression logistique où la variable réponse a la forme 0 ou 1 et `family="poisson"` pour les modèles log-linéaires dans l'analyse des tableaux de contingence.

- `gam()`:
  Les modèles additifs généralisés combinent `lm()` et `glm()` pour incorporer de possibles dépendances non linéaires dans les variables. Vous pouvez modéliser des polynômes et des fonctions de lissage non paramétrique plus complexes (`s()` basé sur des "splines", `lo()` basé sur `loess()` qui est une méthode de lissage par des régressions locales). L'analyse graphique joue un rôle primordial dans `gam()`.

- `loess()`:
  Modèle de régressions locales: un modèle de lissage non paramétrique. Un avantage par rapport à d'autres méthodes de lissage (comme les "splines", la méthode des noyaux ou bien les ondelettes) est son approche simple et la possibilité de lisser dans des espaces de dimension plus élevée.

11 L'organisation du travail avec *S-PLUS*

A la fin du chapitre 1, nous avons mentionné qu'il est vivement recommandé de séparer les travaux et que *S-PLUS* le permettait aisément. Ainsi, nous allons détailler dans ce chapitre l'organisation du travail avec *S-PLUS*.

Tout d'abord, quittez *S-PLUS* puis retournez-y et exécutez la fonction `search()`. Vous devez voir sur votre écran une liste de directoires similiaire à la suivante:

```
> search()
[1] ".Data"
[2] "/usr/people6/splus3.1/splus/.Functions"
[3] "/usr/people6/splus3.1/stat/.Functions"
[4] "/usr/people6/splus3.1/s/.Functions"
[5] "/usr/people6/splus3.1/s/.Datasets"
[6] "/usr/people6/splus3.1/stat/.Datasets"
[7] "/usr/people6/splus3.1/splus/.Datasets"
```

Vous pouvez alors remarquer que vous êtes relié à différents directoires, notamment au premier `.Data` qui contient toutes les variables créées durant la session courante. La commande `objects()` (ou bien `ls()`) en montre son contenu. Pour voir le contenu du deuxième directoire, tapez `objects(where=2)`.

Les fonctions qui permettent de se relier à d'autres directoires et de s'en délier sont `attach()` et `detach()`. Nous avons déjà rencontré la commande `attach()` permettant un travail plus sophistiqué avec les tableaux de données. Faites par exemple `attach(cu.summary)` (le tableau de données mentionné au chapitre 7.3). En faisant à noveau `search()`, vous verrez la liste suivante:

```
> search()
[1] ".Data"
[2] "cu.summary"
[3] "/usr/people6/splus3.1/splus/.Functions"
[4] "/usr/people6/splus3.1/stat/.Functions"
[5] "/usr/people6/splus3.1/s/.Functions"
[6] "/usr/people6/splus3.1/s/.Datasets"
[7] "/usr/people6/splus3.1/stat/.Datasets"
[8] "/usr/people6/splus3.1/splus/.Datasets"
```

En tapant `objects(where=2)`, vous pouvez voir les noms des variables que vous pouvez appeler directement. La fonction `detach()` permet de se délier d'une position: faites `detach(what=2)` pour enlever `cu.summary` de votre liste.

Dans un travail régulier avec *S-PLUS*, vous allez créer des fonctions et des jeux de données que vous voudrez utiliser à partir de n'importe quel directoire. Nous vous conseillons de procéder de la manière suivante:

1. Créez au niveau principal de votre compte un directoire nommé `mon.Splus`.
2. A l'intérieur de ce directoire, créez les deux sous-directoires `.Functions` et `.Datasets`.
3. Si vous avez écrit une fonction et que vous voulez l'utiliser pour différentes tâches, copiez-la au niveau UNIX dans le sous-directoire `.Functions`. Un exemple:

   ```
   cp .Data/absdet ~dupont/mon.Splus/.Functions/.
   ```

   De même pour des jeux de données: copiez-les dans le sous-directoire `.Datasets`.
4. Utilisez `attach()` pour se relier à ces directoires: exécutez d'abord au niveau UNIX la commande `pwd` pour connaître le “path” entier de votre compte et puis au niveau *S-PLUS* (avec un “path” fictif):

   ```
   attach("/usr/student/dupont/mon.Splus/.Functions")
   attach("/usr/student/dupont/mon.Splus/.Datasets")
   ```

Afin d'éviter la réécriture de commandes utilisées systématiquement à chaque lancement de *S-PLUS*, il est possible de composer une fonction particulière `.First()` reconnue par le programme. Cette fonction sera exécutée automatiquement à chaque utilisation de *S-PLUS*. Un exemple typique pourrait être le suivant:

```
.First <- function()
{
   cat("B O N J O U R  !")
   motif()
   help.start(gui = "motif")
   options(prompt="... >",editor="ame")
   options(contrasts = c("contr.sum","contr.treatment"))
   attach("/usr/student/dupont/mon.Splus/.Functions")
   attach("/usr/student/dupont/mon.Splus/.Datasets")
}
```

De manière similaire vous pouvez écrire une fonction qui sera exécutée à chaque sortie de *S-PLUS*: elle doit s'appeler `.Last()`.

12 La librairie électronique *StatLib*

12.1 Introduction

La librairie électronique *StatLib* gérée par l'Université de Carnegie Mellon à Pittsburgh, USA, est devenue un point de rencontre de nombreux statisticiens et autres scientifiques intéressés aux statistiques. Elle contient une multitude de programmes statistiques, de jeux de données et de macros relatives à différents langages. Comme beaucoup de chercheurs soumettent à cette librairie leurs travaux en *S-PLUS*, les utilisateurs de ce logiciel bénéficieront d'un grand choix de modules contenant des techniques récentes. L'accès le plus simple à *StatLib* se fait via *ftp* ("File Transfer Protocol"). Exécutez - au niveau UNIX - la commande suivante:

- `ftp temper.stat.cmu.edu`

Le "login" est `statlib`; donnez ensuite votre adresse e-mail complète comme mot de passe. La commande "`ls`" vous donne une liste des directoires de *StatLib*. Copiez maintenant - avec la commande "`get GET.ME.FIRST`" - le fichier contenant les informations pour les nouveaux utilisateurs. Pour voir les modules concernant *S-PLUS*, faites "`cd S`" puis "`get index`" pour une liste de tous les modules disponibles. Pour quitter *ftp*, écrivez "`quit`". Vous pouvez aussi accéder à *StatLib* via *Mosaic*. L'adresse est la suivante:

`http://lib.stat.cmu.edu/`

En ce qui concerne l'installation sur votre système de modules copiés, nous vous conseillons de consulter votre expert UNIX local.

12.2 Quelques modules intéressants

Voici une liste subjective de modules intéressants (Juin 1994):

- `bootstrap.funs`:
  Plusieurs fonctions pour faire du "bootstrap" et du "jackknife", expliqué dans le livre *An Introduction to the Bootstrap*, par B. Efron et R. Tibshirani, 1993, Chapman and Hall.

- `domouSe`:
  Détection de valeurs aberrantes dans le contexte multivarié.

- `faq`:
  Les "Frequently Asked Questions": Questions courantes sur *S-PLUS*.

- `gnuemacs4`:
  Facilite énormément l'utilisation de *S-PLUS* : à utiliser avec l'éditeur *xemacs* (ou bien *gnuemacs*). Contactez votre spécialiste "*emacs*" local.

- `haerdle`:
  Les fonctions du livre de W. Haerdle, mentionné dans le chapitre 8.3.

- `latex.table` et `s.to.latex`:
  Pour convertir des matrices ou tableaux de données de *S-PLUS* en code LaTeX, ainsi qu'un style LaTeX pour *S-PLUS*.

- `postscriptfonts`:
  Vous voulez des symboles grecs pour nommer vos axes? Ce module explique comment le faire.

- `robeth`:
  Code en Fortran et un interface avec *S-PLUS* pour la famille de fonctions statistiques robustes ROBETH.

- `sguide`:
  L'introduction à *S-PLUS* de B. Ripley, mentionnée dans le chapitre 1.

- `splusnotes`:
  L'introduction à *S-PLUS* de B. Venables, mentionnée dans le chapitre 1.

- `wavethresh`:
  Les ondelettes dans *S-PLUS*; une autre approche de lissage.

13 Le dessert: une analyse statistique avec *S-PLUS*

Dans ce chapitre, nous vous proposons une analyse statistique d'un jeu de données réelles décrivant des observations phénologiques en Suisse. Nous aimerions surtout illustrer le travail avec *S-PLUS* et nous allons mettre l'accent sur la présentation graphique des données ainsi que sur la modélisation de dépendance entre variables.

13.1 Description du jeu de données et buts de l'étude

L'analyse portera sur des observations recueillies par la *Station Centrale Suisse de Météorologie*. Il s'agit de données phénologiques, c'est-à-dire portant sur l'influence du climat sur les phénomènes biologiques saisonniers végétaux (feuillaison, floraison, etc.) et animaux (migration, hibernation, etc.). Essentiellement, on a relevé en un lieu précis le jour de l'année où l'on assiste au premier déploiement des feuilles, à la pleine floraison ou à la chute des feuilles de certaines plantes. Ces relevés sont en correspondance avec les températures moyennes mensuelles ainsi que les précipitations mensuelles d'un des lieux en question.

Nous possédons des données des 4 endroits suivants: Versoix (canton de Genève), Cartigny (canton de Neuchâtel), Liestal (canton de Bâle-Campagne) et Rafz (canton de Zurich). En principe, nous devrions avoir 69 différentes dates (du déploiement des feuilles du marronier jusqu'à la fin de la récolte des pommes de terre) pour les années 1952 à 1992, mais malheureusement les fichiers contiennent beaucoup de données manquantes. Ceci nous a contraints à nous limiter à 8 variables qui sont représentées plus ou moins exhaustivement pour les 4 endroits et pour toutes les années.

Etant donné que nous connaissons les températures moyennes mensuelles ainsi que les précipitations mensuelles à Versoix, il serait intéressant de décrire les dépendances entre les observations biologiques et les observations météorologiques. Notons ici que nous considérons les observations faites d'une année à l'autre comme indépendantes. Ceci est une hypothèse primordiale dans l'application des techniques que nous allons présenter.

Voici les variables que nous avons retenues pour Versoix. Vous trouvez les données en appendice (chapitre 14.3). Le jour le plus tardif de l'année donnant une information biologique est le 151ème, c'est-à-dire le 31 Mai. Nous n'avons donc inclus que les températures moyennes mensuelles (en *degré Celsius*) et les précipitations mensuelles (en *mm*) des mois de Janvier à Mai (appellées `T.JANVIER`, ..., `T.MAI` et `P.JANVIER`, ..., `P.MAI`). Le code pour les plantes est le suivant:

Code	Description
`X3`	Déploiement des feuilles Noisetiers
`X4`	Pleine floraison Anémone sylvie
`X9`	Pleine floraison Dent de lion
`X12`	Pleine floraison Lilas
`X13`	Pleine floraison Renoncules, Ficaire
`X16`	Pleine floraison Grandes Marguerites
`X17`	Déploiement des feuilles Sapin rouge
`X18`	Pleine floraison Epine noire

La figure suivante présente l'évolution des phénomènes au cours des années.

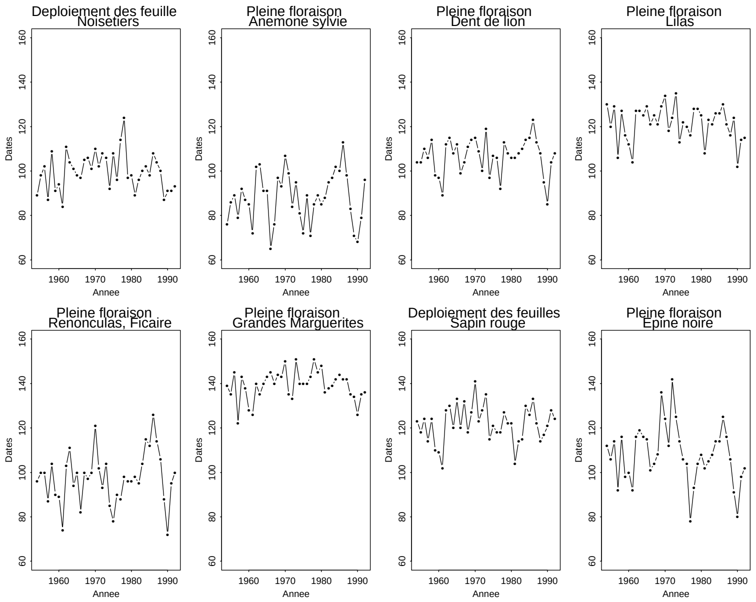

Figure 4: Evolution des 8 phénomènes entre 1954 et 1992

13.2 Analyse statistique

Nous sommes intéressés à modéliser la date d'un phénomène en fonction des températures et précipitations. Il s'agit maintenant de trouver lesquelles des 10 variables (`T.JANVIER` à `T.MAI` et `P.JANVIER` à `P.MAI`) influencent vraiment la date du déploiement des feuilles, respectivement de la pleine floraison. Une technique possible est de générer toutes les régressions possibles et d'en choisir les "meilleures" en utilisant comme critère par exemple le C_p de Mallows. Nous utilisons la fonction *S-PLUS* `leaps()`. Voici une illustration pour la variable `X12`, *la pleine floraison du lilas*. La commande est:

```
X12.Cp <- leaps(versoix[,9:18],X12,nbest=4)
```

Puis nous trouvons les modèles les plus intéressants d'une manière graphique en dessinant le C_p versus le nombre de paramètres du modèle. Nous conservons les modèles qui ont une valeur C_p petite et qui sont proches de la droite $y = x$. Le graphique obtenu est le suivant. La partie en haut à droite est un agrandissement des modèles ayant une valeur C_p plus petite que 20. Pour produire ce dessin, nous avons utilisé la commande `split.screen()` (voir chapitre 7.3).

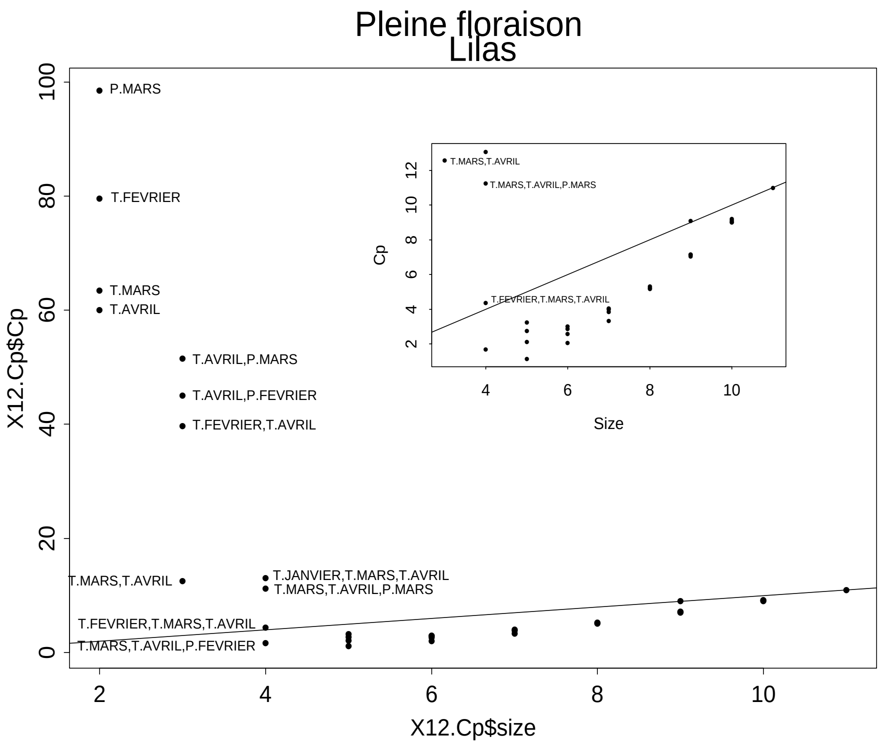

Figure 5: C_p-plot pour Pleine floraison Lilas

Nous constatons clairement que les températures en Mars et en Avril sont les deux variables les plus importantes. Si nous voulons une troisième variable, nous devrions prendre soit les précipitations en Février soit les températures dans ce même mois. Nous avons effectué les mêmes procédures pour les 7 autres variables. Le tableau suivant contient pour chacune des 8 variables les meilleures régressions (signifiant plus petite valeur C_p) pour deux et trois régresseurs.

Variable	**2 régresseurs**	**3 régresseurs**
X3	T.MARS, T.AVRIL T.MARS, P.AVRIL	T.MARS, T.AVRIL, T.MAI T.MARS, T.AVRIL, P.MAI
X4	T.FEVRIER, T.MARS T.MARS, P.MARS	T.FEVRIER, T.MARS, P.MARS T.JANVIER, T.FEVRIER, T.MARS
X9	T.FEVRIER, T.MARS T.MARS, T.AVRIL	T.FEVRIER, T.MARS, T.AVRIL T.JANVIER, T.FEVRIER, T.MARS
X12	T.MARS, T.AVRIL T.FEVRIER, T.AVRIL	T.MARS, T.AVRIL, P.FEVRIER T.FEVRIER, T.MARS, T.AVRIL
X13	T.FEVRIER, T.MARS T.MARS, T.AVRIL	T.FEVRIER, T.MARS, P.FEVRIER T.FEVRIER, T.MARS, T.AVRIL
X16	T.MARS, T.AVRIL T.AVRIL, P.FEVRIER	T.MARS, T.AVRIL, P.FEVRIER T.MARS, T.AVRIL, P.MAI
X17	T.MARS, T.AVRIL T.JANVIER, T.AVRIL	T.MARS, T.AVRIL, P.JANVIER T.JANVIER, T.MARS, T.AVRIL
X18	T.FEVRIER, T.MARS T.MARS, P.FEVRIER	T.MARS, T.AVRIL, P.FEVRIER T.FEVRIER, T.MARS, P.JANVIER

On remarque que ce sont surtout les températures qui influencent les phénomènes et assez rarement les précipitations. En plus, nous voyons clairement que les mois de Mars, Avril et Février sont déterminants. Pour mieux discerner ces dépendances, nous avons representé à l'aide de "scatterplot" nos 8 variables versus les différentes températures, les points représentant des années. Nous ajoutons un lissage par la méthode "*LOESS*" (en utilisant la fonction *S-PLUS* `scatter.smooth()`) pour visualiser plus clairement la forme du nuage des points. Référez-vous au chapitre 8 du "White Book" pour une description plus détaillée de `scatter.smooth()` et `loess()`.

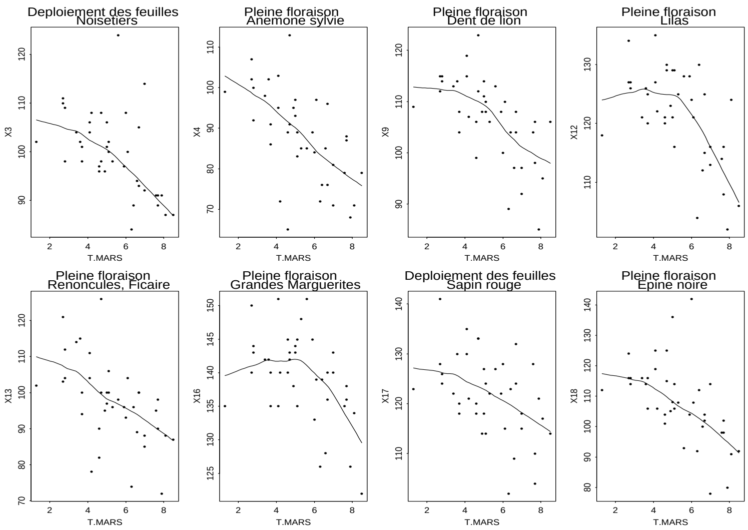

Figure 6: Les 8 phénomènes versus la température en Mars

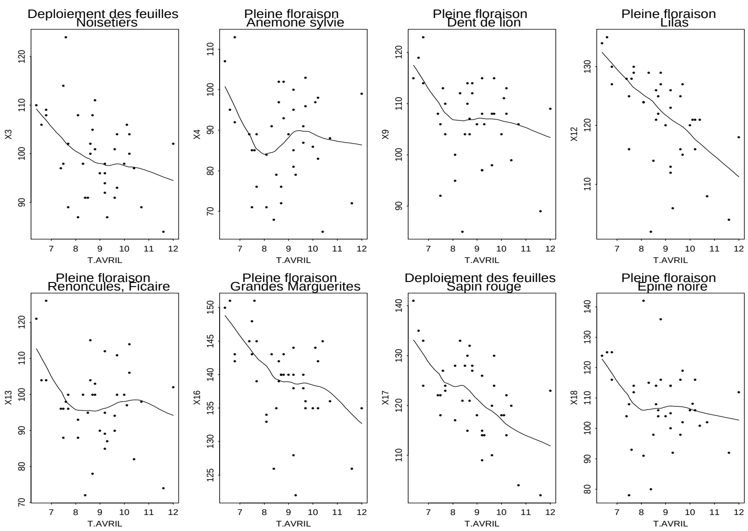

Figure 7: Les 8 phénomènes versus la température en Avril

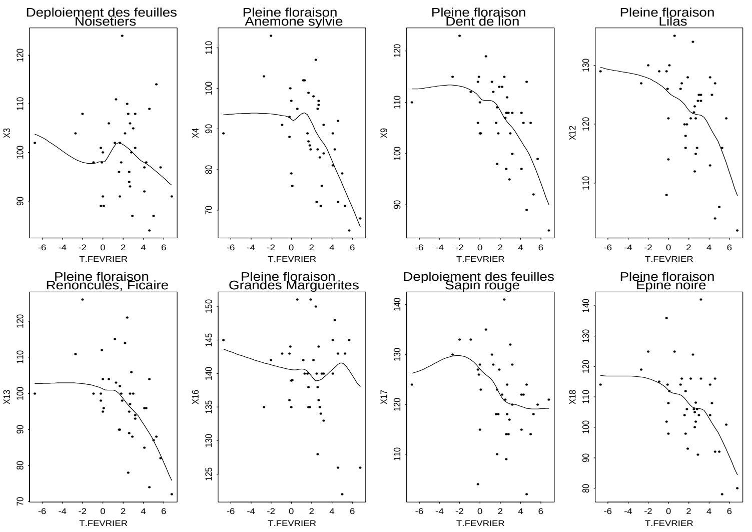

Figure 8: Les 8 phénomènes versus la température en Février

Généralement, la tendance est que, plus la température moyenne est haute, plus le phénomène se passe tôt dans l'année; c'est en fait ce à quoi il fallait s'attendre. Or, on voit très bien que cette tendance n'est pas forcément linéaire; la courbure de certaines courbes de lissage est évidente. Il est maintenant important d'estimer la variabilité du lissage. La fonction `loess()` incorpore une option qui calcule et dessine des intervalles de confiance pour la courbe estimée. Nous allons présenter une autre approche, celle du "bootstrap". L'idée est de rééchantillonner les résidus d'un modèle donné, de reconstruire une nouvelle variable dépendante et de refaire le lissage avec `loess()`. La fonction suivante, appelée `loess.boot`, montre une façon de programmer ceci dans *S-PLUS*.

```
loess.boot <- function(x, y, B = 50, var.nom)
{
x.nom <- deparse(substitute(x))
y.nom <- deparse(substitute(y))
loess.original <- loess(y ~ x)
plot(loess.original, ylim = range(y), xlab = x.nom, ylab = y.nom, main = var.nom)
points(x, y)
residus <- residuals(loess.original)
for(i in 1:B) {
         res.bootstrap <- sample(residus, replace = T)
         y.bootstrap <- res.bootstrap + fitted(loess.original)
         par(new = T)
         plot(loess(y.bootstrap ~ x), xaxt = "n", yaxt = "n", ylim =
              range(y), xlab = x.nom, ylab = y.nom, main = var.nom)
    }
invisible(par(new = F))
}
```

Dans les figures suivantes, nous présentons les "scatterplots" de la figure 6 et 7, en ajoutant 50 lissages obtenus en rééchantillonnant les résidus de l'ajustement original. Ces lissages fournissent une fois réunis une sorte d'intervalles de confiance.

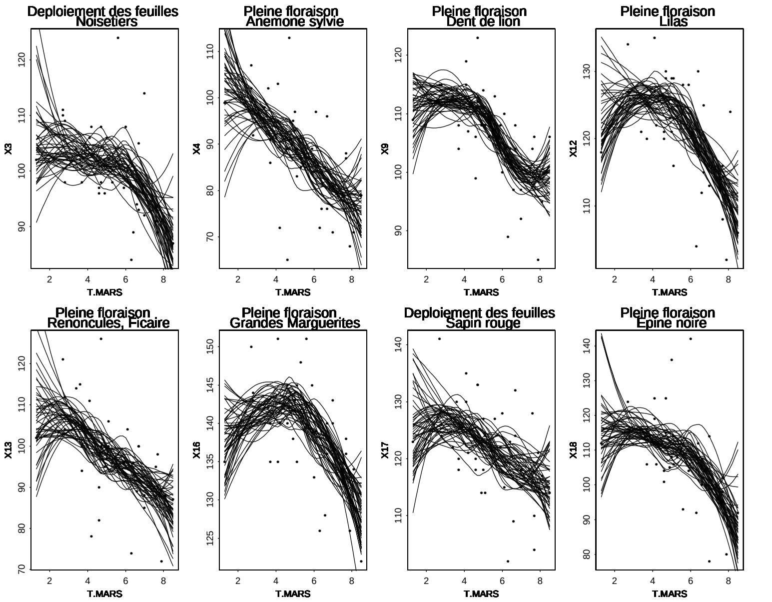

Figure 9: les 8 phénomènes versus la température en Mars avec 50 lissages "bootstrap"

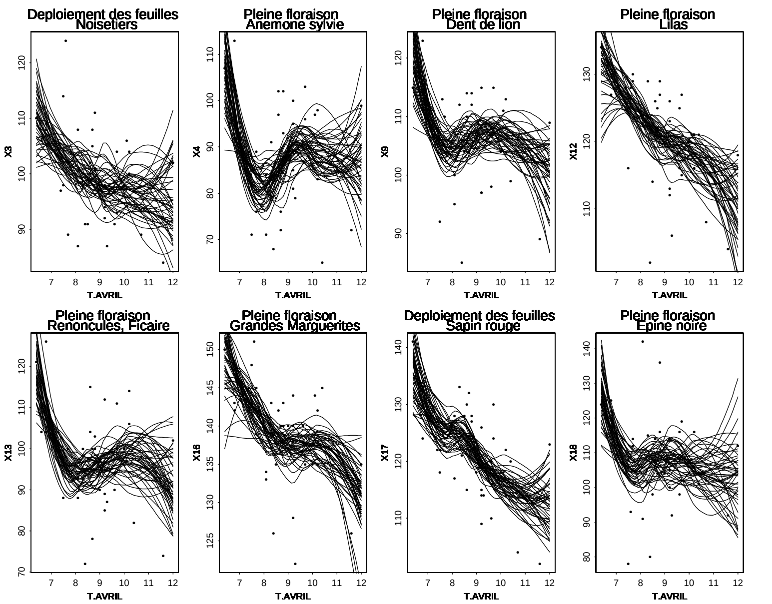

Figure 10: Les 8 phénomènes versus la température en Avril avec 50 lissages "bootstrap"

Notez la dépendance particulière entre la variable `X4` (Pleine floraison de l'Anémone Sylvie) et la température en Avril: les régressions multiples linéaires n'ont pas tenu compte de cette température, certainement du fait de la non linéarité de la dépendance. Des effets de multi-colinéarité entre variables peuvent être exclus car les températures ne sont pas significativement corrélées (voir le tableau ci-dessous des corrélations pour les variables météorologiques):

	T.JANVIER	T.FEVRIER	T.MARS	T.AVRIL	T.MAI	P.JANVIER	P.FEVRIER	P.MARS	P.AVRIL	P.MAI
T.JANVIER	1.00	0.06	0.21	-0.16	0.09	0.23	-0.04	-0.14	-0.01	0.17
T.FEVRIER	0.06	1.00	0.22	0.12	0.07	-0.15	0.57	-0.14	-0.12	-0.22
T.MARS	0.21	0.22	1.00	-0.03	0.02	-0.29	0.08	-0.21	0.01	-0.12
T.AVRIL	-0.16	0.12	-0.03	1.00	-0.05	-0.05	-0.21	-0.02	-0.25	0.05
T.MAI	0.09	0.07	0.02	-0.05	1.00	-0.17	-0.02	-0.04	0.18	-0.15
P.JANVIER	0.23	-0.15	-0.29	-0.05	-0.17	1.00	0.14	0.12	0.04	0.22
P.FEVRIER	-0.04	0.57	0.08	-0.21	-0.02	0.14	1.00	-0.25	0.06	-0.12
P.MARS	-0.14	-0.14	-0.21	-0.02	-0.04	0.12	-0.25	1.00	0.10	-0.01
P.AVRIL	-0.01	-0.12	0.01	-0.25	0.18	0.04	0.06	0.10	1.00	-0.10
P.MAI	0.17	-0.22	-0.12	0.05	-0.15	0.22	-0.12	-0.01	-0.10	1.00

Prenons le modèle avec `X4`, la *Pleine floraison de l'Anémone Sylvie* et la température en Avril. La figure suivante montre trois façons de représenter les résidus pour vérifier si les hypothèses dans la modélisation sont bien vérifiées.

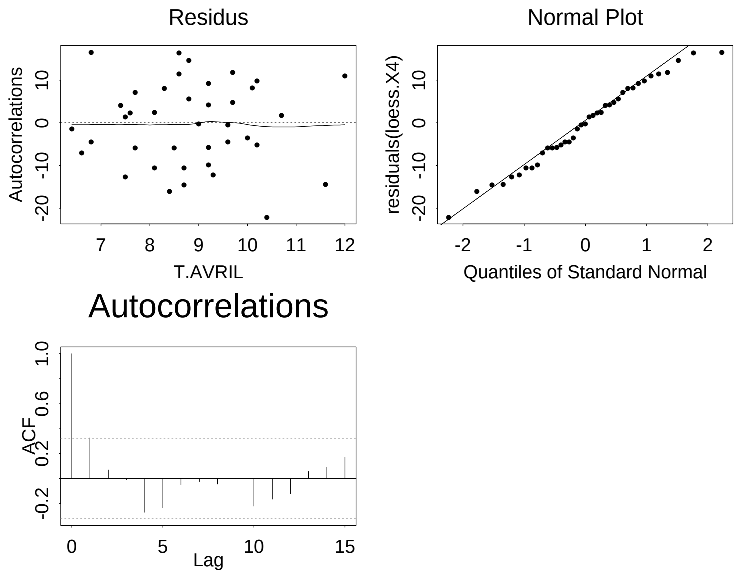

Figure 11: Résidus pour `loess(X4 ~ T.AVRIL)`

Les graphiques ne revèlent rien de particulier. Le troisième dessin montre les autocorrélations estimées; elles ne sont pas significatives. Nous avons mentionné que la fonction `loess()` permet de calculer et de visualiser des intervalles de confiance approximatifs de la courbe ajustée. La figure suivante permet de comparer ces intervalles (avec seuil de confiance 0.95)[14] avec la représentation de la variabilité obtenue par le "bootstrap".

[14] La commande est la suivante: `scatter.smooth(X4 ~ T.AVRIL, confidence=20, coverage=0.95)`

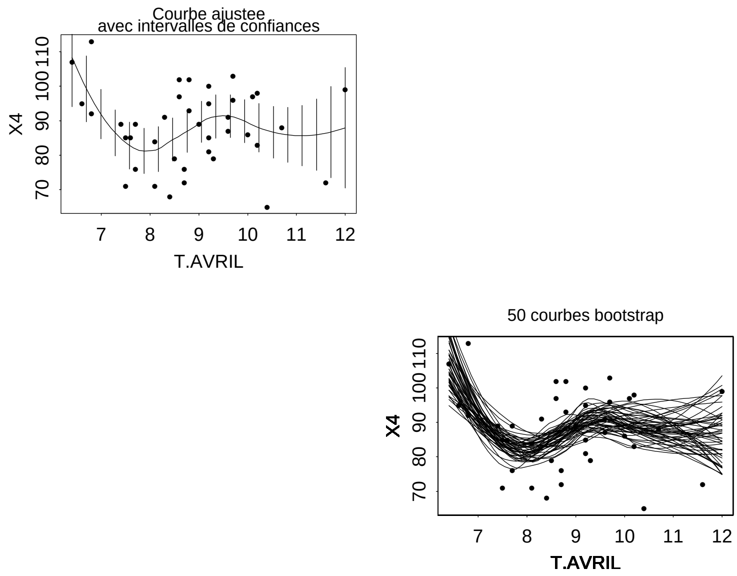

Figure 12: Comparaison des intervalles approximatifs avec la représentation par "bootstrap"

Nous constatons que les deux représentations sont très similaires.

13.3 Conclusion

L'analyse statistique effectuée sur les données de Versoix est évidemment incomplète. Nous avons voulu par cette analyse illustrer l'utilisation du logiciel *S-PLUS*. Plusieurs autres techniques peuvent être utilisées sur ces données. On peut citer:

- Des statistiques descriptives plus complètes, par exemple en utilisant les commandes `boxplot()`, `pairs()` et `coplot()`. La dernière pourrait être utile afin de détailler les dépendances décrites conditionnées par exemple aux précipitations.
- L'analyse canonique pour lier les 8 variables biologiques avec les variables météorologiques intéressantes.
- Les modèles *ARIMA* ou *AR* de séries temporelles sur chacune des variables.
- etc.

Nous avons effectué certaines de ces techniques mais par souci de concisions nous ne les incorporons pas dans ce texte. Toutefois nous apprécierons que toute personne qui effectuerait ces analyses et tirerait des conclusions intéressantes nous communique ses résultats et commentaires.

De plus, pour être complet on doit répéter l'analyse effectuée pour les 3 autres endroits, à savoir Cartigny, Rafz et Liestal et éventuellement faire une analyse simultanée des résultats dans les 4 endroits.

14 Appendice

14.1 Appendice pour les graphiques

Comme référence, voici les listes de symboles et styles de lignes que vous pouvez utiliser dans *S-PLUS*. Les légendes ont été produites avec les commandes présentées au chapitre 7.3.

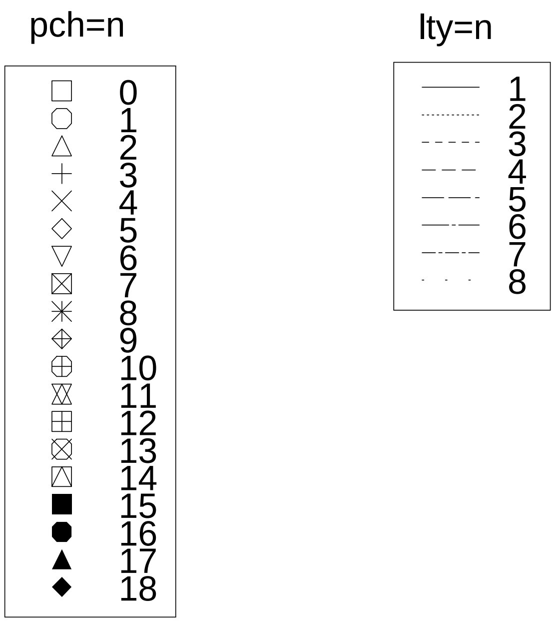

Figure 13: Symboles pour `pch=n` *et styles de lignes pour* `lty=n`

14.2 Les dépenses de l'état français

Voici le jeu de données utilisé dans le chapitre 9.3. Pour chaque année, on donne les pourcentages du budget dépensés dans les différents ministères. Tout d'abord une description des variables:

PVP	=	pouvoir public
AGR	=	agriculture
CMI	=	commerce et industrie
TRA	=	transports
LOG	=	logement et aménagement du territoire
EDU	=	éducation et culture
ACS	=	action sociale
ACO	=	anciens combattants
DEF	=	défense nationale
DET	=	dette
DIV	=	divers

	PVP	AGR	CMI	TAR	LOG	EDU	ACS	ACO	DEF	DET	DIV
1872	18.0	0.5	0.1	6.7	0.5	2.1	2.0	0.0	26.4	41.5	2.1
1880	14.1	0.8	0.1	15.3	1.9	3.7	0.5	0.0	29.8	31.3	2.5
1890	13.6	0.7	0.7	6.8	0.6	7.1	0.7	0.0	33.8	34.4	1.7
1900	14.3	1.7	1.7	6.9	1.2	7.4	0.8	0.0	37.7	26.2	2.2
1903	10.3	1.5	0.4	9.3	0.6	8.5	0.9	0.0	38.4	27.2	3.0
1906	13.4	1.4	0.5	8.1	0.7	8.6	1.8	0.0	38.5	25.3	1.9
1909	13.5	1.1	0.5	9.0	0.6	9.0	3.4	0.0	36.8	23.5	2.6
1912	12.9	1.4	0.3	9.4	0.6	9.3	4.3	0.0	41.1	19.4	1.3
1920	12.3	0.3	0.1	11.9	2.4	3.7	1.7	1.9	42.4	23.1	0.2
1923	7.6	1.2	3.2	5.1	0.6	5.6	1.8	10.0	29.0	35.0	0.9
1926	10.5	0.3	0.4	4.5	1.8	6.6	2.1	10.1	19.9	41.6	2.3
1929	10.0	0.6	0.6	9.0	1.0	8.1	3.2	11.8	28.0	25.8	2.0
1932	10.6	0.8	0.3	8.9	3.0	10.0	6.4	13.4	27.4	19.2	0.0
1935	8.8	2.6	1.4	7.8	1.4	12.4	6.2	11.3	29.3	18.5	0.4
1938	10.1	1.1	1.2	5.9	1.4	9.5	6.0	5.9	47.0	18.2	0.0
1947	15.6	1.6	10.0	11.4	7.6	8.8	4.8	3.4	32.2	4.6	0.0
1950	11.2	1.3	16.5	12.4	15.8	8.1	4.9	3.4	20.7	4.2	1.5
1953	12.9	1.5	7.0	7.9	12.1	8.1	5.3	3.9	36.1	5.2	0.0
1956	10.9	5.3	9.7	7.6	9.6	9.4	8.5	4.6	28.2	6.2	0.0
1959	13.1	4.4	7.3	5.7	9.8	12.5	8.0	5.0	26.7	7.5	0.0
1962	12.8	4.7	7.5	6.6	6.8	15.7	9.7	5.3	24.5	6.4	0.1
1965	12.4	4.3	8.4	9.1	6.0	19.5	10.6	4.7	19.8	3.5	1.8
1968	1.4	6.0	9.5	5.9	5.0	21.1	10.7	4.2	20.0	4.4	1.9
1971	12.8	2.8	7.1	8.5	4.0	23.8	11.3	3.7	18.8	7.2	0.0

14.3 Les données phénologiques de Versoix

Voici les données décrites dans le chapitre 13.1. Les chiffres indiquent le jour de l'année lors duquel on a observé le phénomène. Il y avait une donnée manquante pour la variable `X18` et l'année 1968: nous l'avons remplacée par la moyenne des dates disponibles pour `X18`.

	X3	X4	X9	X12	X13	X16	X17	X18	T.JANVIER	T.FEVRIER	T.MARS	T.AVRIL
1954	89	76	104	130	96	139	123	112.0	-0.8	0.1	6.4	7.7
1955	98	86	104	120	100	135	118	106.0	2.3	1.8	3.7	10.0
1956	102	89	110	129	100	145	124	114.0	3.0	-6.7	5.1	7.7
1957	87	79	106	106	87	122	114	92.0	-1.3	5.0	8.5	9.3
1958	109	92	114	127	104	143	124	116.0	0.6	4.6	2.8	6.8
1959	91	87	98	116	90	138	110	98.0	1.4	1.7	7.7	9.6
1960	94	85	97	112	89	128	109	100.0	0.4	2.6	6.6	9.2
1961	84	72	89	104	74	126	102	92.0	1.6	4.6	6.3	11.6
1962	111	102	112	127	103	140	128	116.0	2.9	1.3	2.7	8.8
1963	104	103	115	127	111	135	130	119.0	-4.3	-2.7	4.1	9.7
1964	101	91	108	125	94	140	120	116.0	-1.9	3.2	3.7	9.6
1965	98	91	112	129	100	143	133	115.0	0.7	-0.9	4.7	8.3
1966	97	65	99	121	82	145	120	101.0	-0.3	5.7	4.6	10.4
1967	105	76	104	125	100	140	132	104.0	1.0	3.0	6.7	8.7
1968	106	97	111	121	97	144	118	108.1	0.7	2.7	5.0	10.1
1969	101	93	114	129	100	143	127	136.0	1.3	-0.2	5.0	8.8
1970	110	107	115	134	121	150	141	124.0	0.4	2.4	2.7	6.4
1971	102	99	109	118	102	135	123	112.0	-1.2	1.7	1.3	12.0

1972	108	84	100	124	93	133	128	142.0	0.8	3.2	6.0	8.1
1973	106	95	119	135	104	151	135	125.0	0.1	0.6	4.1	6.6
1974	92	81	97	113	85	140	115	114.0	3.2	4.1	7.0	9.2
1975	108	72	107	122	78	140	121	106.0	3.6	2.5	4.2	8.7
1976	96	89	106	120	90	140	118	104.0	1.6	1.6	4.6	9.0
1977	114	71	92	116	88	143	118	78.0	1.6	5.3	7.0	7.5
1978	124	85	113	128	98	151	127	93.0	1.3	1.9	5.6	7.6
1979	97	89	108	128	96	145	122	104.0	-0.8	4.1	5.9	7.4
1980	98	85	106	125	96	148	122	108.0	0.6	4.3	5.3	7.5
1981	89	88	106	108	98	136	104	102.0	-0.9	-0.2	7.7	10.7
1982	96	95	108	123	95	138	114	105.0	3.0	2.6	4.9	9.2
1983	100	97	110	121	104	139	115	108.0	2.5	0.0	6.1	8.6
1984	102	102	114	126	115	142	130	114.0	1.7	1.2	3.6	8.6
1985	98	100	115	126	112	144	126	114.0	-3.8	-0.1	2.8	9.2
1986	108	113	123	130	126	142	133	125.0	1.6	-2.0	4.7	6.8
1987	104	98	113	121	114	142	122	116.0	-2.0	2.2	3.4	10.2
1988	100	83	108	116	106	135	114	106.0	4.4	2.8	5.1	10.2
1989	87	71	95	124	88	134	117	91.0	1.8	2.9	8.1	8.1
1990	91	68	85	102	72	126	121	80.0	1.8	6.8	7.9	8.4
1991	91	79	104	114	95	135	128	98.0	2.3	0.0	7.6	8.5
1992	93	96	108	115	100	136	124	102.0	0.1	2.7	6.7	9.7

	T.MAI	P.JANVIER	P.FEVRIER	P.MARS	P.AVRIL	P.MAI
1954	12.5	72	93	41	56	35
1955	13.0	188	166	26	2	108
1956	14.0	137	2	40	85	122
1957	11.3	26	191	21	29	31
1958	15.1	133	160	52	39	107
1959	13.8	118	0	158	103	18
1960	14.4	120	115	101	7	69
1961	12.3	109	90	8	104	36
1962	11.8	133	90	184	65	67
1963	12.4	58	63	141	73	58
1964	14.7	25	35	192	75	34
1965	13.2	87	11	144	72	107
1966	14.1	122	125	58	93	64
1967	12.5	51	108	77	19	137
1968	12.4	114	103	36	63	72
1969	13.6	65	43	67	92	103
1970	12.6	90	211	153	140	33
1971	13.9	75	40	96	32	79
1972	12.3	22	77	48	60	31
1973	13.7	34	51	6	50	63
1974	12.1	66	75	46	33	113
1975	12.7	115	11	72	62	97
1976	14.5	36	94	22	11	40
1977	11.9	138	188	61	129	76
1978	12.0	122	126	120	63	78
1979	12.6	143	119	155	46	60
1980	11.7	89	66	112	8	117
1981	12.2	123	46	109	8	124
1982	13.5	67	22	87	6	46
1983	10.7	65	70	84	110	196
1984	10.2	181	112	69	26	66
1985	12.4	109	94	82	92	116
1986	15.1	169	57	75	152	67
1987	10.9	61	64	52	48	89
1988	14.4	119	105	139	45	130
1989	15.6	17	93	74	96	38
1990	15.9	42	234	9	88	57
1991	11.6	49	29	90	36	27
1992	15.4	16	37	80	86	101