Ects

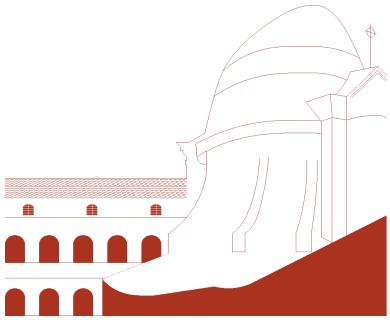
Utilitaire d'Économétrie
Version 2

Russell Davidson

Mars 1993

Ects, Version 2
© Russell Davidson, Mars 1993.
Tous droits de reproduction, de traduction, d'adaptation, et d'exécution réservés pour tous les pays.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".



AVANT PROPOS

Cet utilitaire d'Économétrie est destiné à l'usage des étudiants du Magistère Ingénieur-Économiste de l'Université d'Aix-Marseille II ainsi que du D.E.A. du GREQE, à Marseille. Son but n'est pourtant pas exclusivement pédagogique. La première version d'***Ects*** s'est avérée utile, par sa commodité et par sa rapidité, non seulement pour les exercices de cours, mais aussi pour un certain nombre d'applications pratiques. Il reste néanmoins que l'utilitaire fut conçu à l'origine dans le cadre de l'apprentissage de l'économétrie. C'est pour cette raison qu'il n'essaie pas de tout faire. Après une estimation, l'utilisateur aura parfois encore un petit travail de programmation afin de calculer exactement l'estimateur ou la statistique de test souhaité . . . tout cela fait partie justement de l'apprentissage. Toutefois, beaucoup de choses sont possibles pour ceux et celles qui seront prêts à entreprendre ce petit travail supplémentaire: les estimations non-linéaires, les régressions artificielles, la méthode du maximum de vraisemblance, jusqu'à la méthode des moments généralisée. Toutes ces procédures sont parfaitement abordables, à condition, bien entendu, qu'il y ait assez de place dans la mémoire vive de votre ordinateur pour contenir les données.

La nouvelle version de l'utilitaire permet un nombre de manipulations et d'opérations largement supérieur au nombre relativement restreint offert par la première version. En particulier, toute une gamme d'opérations matricielles a été introduite. Une conséquence en est que la première version tournera un peu plus rapidement si les commandes à exécuter n'utilisent pas les fonctionnalités nouvelles. En revanche, il existe une version de l'utilitaire qui ne tourne que sur des ordinateurs 386 ou 486, avec coprocesseur numérique. Cette version, plus rapide que la version « ordinaire », permet aussi l'utilisation de la totalité de la mémoire étendue dont dispose l'ordinateur.

Depuis quelques années, mes enseignements d'économétrie s'appuient sur les versions successives d'un ouvrage que j'ai rédigé avec mon collègue J.G. MacKinnon. Cet ouvrage est publié par la Oxford University Press de New-York sous l'intitulé *Estimation and Inference in Econometrics*. Dans ce manuel, vous trouverez de nombreux renvois aux chapitres et aux sections de ce livre, que j'appelle désormais simplement DM.

Pour Émmanuel et Christophe, mes « esclaves », et pour Dirk, qui a dû attendre si longtemps.

Ce volume a été légèrement modifié par rapport à la documentation d'***Ects*** 2 de mars 1993, afin qu'il puisse servir tout particulièrement aux utilisateurs d'***Ects*** 4. Le texte est quasiment inchangé, mais j'ai rajouté quelques notes où le texte d'origine peut être trompeur. Il est à noter aussi que, à la différence de la documentation de mars 1993, ce volume est protégé par la licence GFDL; voir la section pertinente de la documentation de la version 4.

Table des Matières

Introduction

Les utilisateurs de la première version du logiciel ***Ects*** remarqueront tout de suite que ce manuel, la documentation de la version 2, est beaucoup plus volumineux que le petit manuel fourni avec la première version. Il y a des fonctionnalités nouvelles dans la version actuelle, certes, mais pas dans la proportion que le pourrait faire entendre le gonflement de la documentation. Évidemment, si je me suis donné la peine d'écrire une petite centaine de pages à la place des vingt et quelques du manuel précédent, c'est parce que je voulais, cette fois-ci, expliciter beaucoup plus les possibilités diverses offertes par le logiciel. J'espère ainsi non seulement m'épargner les explications orales trop nombreuses nécessitées par une documentation trop maigre, mais aussi faciliter la mise en œuvre des procédures économétriques par les utilisateurs du logiciel. Je serai content si les explications plus détaillées données dans les pages qui suivent vous inspirent à essayer des études empiriques de plus en plus ambitieuses.

Je connais très bien le risque couru en augmentant ainsi le volume de la documentation. Plus je vous donne à lire, moins vous lisez ! C'est pourquoi je n'ai pas essayé d'exposer les commandes et les fonctions d'***Ects*** d'une façon logique et systématique : ç'aurait été trop ennuyeux à la lecture. J'ai essayé plutôt de commencer par les commandes les plus utiles, et les plus faciles à expliquer, et de progresser ensuite vers les choses plus sophistiquées. Par conséquent, quelques commandes sont considérées plusieurs fois, dans des contextes différents. Pour être sûr d'obtenir la totalité des informations disponibles sur une commande, on doit utiliser l'un des deux Index. Dans le premier Index, général, on trouve tout, mais pour la commodité des lecteurs qui cherchent une information précise sur une commande ou sur une fonction ***Ects***, il y a aussi l'Index ***Ects***, où les choses se trouvent plus facilement.

Il devrait être possible d'implémenter presque toutes les procédures qui relèvent de la régression linéaire après la lecture des deux premiers chapitres. Au troisième chapitre, j'expose les fonctionnalités nouvelles qui permettent de faire la quasi-totalité des opérations courantes du calcul matriciel. Dans la dernière section de ce chapitre, pourtant, il a été nécessaire de voir de près des opérations qui sont loins d'être élémentaires. Au début de cette section, j'ai mis un avertissement à cet effet, et, en effet, on peut se passer de la lecture de cette section jusqu'au jour où il sera nécessaire d'effectuer l'une de ces opérations non élémentaires, dans un avenir lointain peut-être.

Le chapitre 4 est l'endroit où j'expose l'utilisation des commandes ***Ects*** qui permettent d'effectuer les estimations non-linéaires. Même si l'on ne souhaite

pas entrer dans les détails des opérations matricielles, il est vivement conseillé de lire au moins les deux premières sections de ce chapitre, afin de pouvoir employer les méthodes des moindres carrés non-linéaires et du maximum de vraisemblance. Dans le chapitre 5, vous trouverez un exposé de quelques manipulations qui peuvent faciliter l'usage d'***Ects***. Il n'y a rien là d'essentiel, mais un certain nombre de choses amusantes. Pour ceux et celles qui s'intéressent à l'informatique, ce chapitre vous enseignera à jouer avec ***Ects***.

Dans un souci de faire d'***Ects*** un outil de recherche, j'ai introduit dans la version actuelle des fonctionnalités permettant de faire des simulations. Les possibilités sont exposées dans le chapitre 6. Clairement, la lecture de ce chapitre ne s'impose qu'aux lecteurs qui souhaitent faire des simulations.

Malgré mes efforts, il n'a pas été possible d'expliquer tout ce que peut faire ***Ects*** dans les contextes des six premiers chapitres, comme l'atteste le nom du chapitre 7. J'ai profité de la nécessité de ce chapitre pour y fourrer tous les détails dont l'intérêt est très faible. Normalement, à moins que vous ne soyez un grand amateur de l'informatique, vous pouvez vous passer complètement de la lecture de ce chapitre. Toutefois, si vous ne parvenez absolument pas à comprendre pourquoi vos résultats ne sont pas conformes à vos attentes, c'est probablement dans ce chapitre que vous trouverez l'explication.

La dernière section du chapitre 7 ne concerne que les utilisateurs qui souhaitent faire tourner ***Ects*** sous Unix. À la différence de DOS, ce système d'exploitation peut être implanté sur beaucoup d'ordinateurs qui ont des caractéristiques matérielles très différentes. Pour cette raison, il n'est pas possible de distribuer un seul fichier exécutable susceptible de fonctionner correctement sur des machines différentes. En fait, pour chaque machine différente, il faut recompiler ***Ects*** à partir du code source. Les instructions pertinentes sont fournies dans la section 7.4.

Je suis très reconnaissant des efforts de tous les étudiants qui m'ont fait comprendre les limitations, voire les erreurs, de la première version du logiciel, et qui m'ont fait des suggestions pour la version présente. Sans la demande de Christophe Flachot, ***Ects*** serait muet ou presque.[1] Si je n'ai pas suivi toutes les suggestions que l'on m'a faites, veuillez m'en excuser ; quand j'aurai le temps, j'essaierai de les incorporer dans une version ultérieure. À l'heure actuelle, les demandes les plus urgentes réclament le graphisme, et les macrocommandes. S'il y a autre chose, j'attends vos suggestions ...

[1] Note de la version 4 : Il l'est redevenu.

Chapitre Premier
Comment Utiliser *Ects*

1. Installation

Le logiciel ***Ects*** a été conçu comme un petit utilitaire plutôt qu'un outil performant mais lourd. L'une des versions de ***Ects*** peut tourner sur tous les PC, même s'ils n'ont ni disque dur ni coprocesseur numérique ; le fichier exécutable de cette version s'appelle `ects86.exe`. L'autre version, `ects.exe`, ne peut tourner que sur les machines 386 ou 486 avec coprocesseur numérique.

Si vous ne souhaitez pas installer ***Ects*** sur votre disque dur, ou si vous n'avez pas de disque dur, il n'y a rien à faire, sauf, évidemment, faire une copie de sauvegarde de la disquette d'origine. L'installation sur un disque dur n'est guère plus difficile. Après avoir fait la copie de sauvegarde, mettez-vous dans le répertoire où vous souhaitez installer ***Ects***. Au besoin, vous pouvez créer un répertoire par la commande DOS `md ects`. Vous avez le droit, bien entendu, de choisir un autre nom que ECTS. Ensuite, si la disquette d'origine, ou, de préférence, la disquette de sauvegarde, est dans le lecteur A:, il suffira de taper `copy a:*.*` pour que tous les fichiers soient transférés sur le disque dur.

Il serait prudent maintenant de faire un `dir` pour vous assurer que tout s'est bien déroulé. Tous les fichiers suivants devraient se trouver dans le répertoire :

```
2sls.dat
ar.dat
ivnls.dat
nls.dat
ols.dat
sur.dat
2sls.ect
ar.ect
arnls.ect
gen.ect
ivnls.ect
logit.ect
ml.ect
nearunit.ect
nls.ect
nlsols.ect
```

```
ols.ect
season.ect
sur.ect
ects86.exe
ects.exe
```

Vous pouvez vérifier que les fichiers `*.dat` contiennent des données et les fichiers `*.ect` des programmes. `ects86.exe` est le fichier exécutable qui tourne n'importe où ; `ects.exe` est le fichier exécutable qui ne tournera que sur les 386 et 486 equipés d'un coprocesseur numérique.[2]

Les fichiers `*.ect` sont fournis, à titre illustratif, pour que vous puissiez utiliser ***Ects*** sans attendre. Toutefois, il est important de comprendre que les fichiers contenant les programmes ne peuvent pas être créés par le logiciel ***Ects*** lui-même. Normalement, on utilise un éditeur de texte à cette fin. On peut également se servir de *certains* traitements de texte, à condition qu'ils sachent créer des fichiers ASCII. En effet, un fichier de commandes ***Ects*** est un simple fichier ASCII.

La façon la plus simple de faire tourner ***Ects*** est de se mettre sous DOS (Windows, OS/2, etc., sont inutiles à cet égard)[3] et de taper `ects` suivi du nom d'un fichier de commandes. Si, comme c'est le cas de tous les fichiers de commandes fournies, l'extension est `ect`, il suffit de donner le nom du fichier sans l'extension – pour faire exécuter `ols.ect` il suffit de taper `ects ols`. Si l'extension n'est pas `ect`, il faut la préciser. Si l'on tape `ects` sans le nom d'un fichier, on se trouve dans le mode interactif, dont nous parlerons plus tard. (Ce mode n'existait pas dans la première version.)

2. La Commande `ols`

Il s'agit du modèle le plus utilisé en économétrie, à savoir

$$\boldsymbol{y} = \boldsymbol{X\beta} + \boldsymbol{u}.$$

* * * *

Voir le Chapitre 1 de DM pour un exposé théorique.

* * * *

La **variable dépendante**, appelée parfois la **régressande**, est $\boldsymbol{y}$, représentée algébriquement par un vecteur à n composantes scalaires, et géométriquement par ce même vecteur considéré comme un point de l'espace $\mathbb{R}^n$. De même, les k colonnes de la matrice $\boldsymbol{X}$, de la forme $n \times k$, représentent les **explicatives**, ou les **régresseurs**.

[2] Note de la version 4 : Tout ce qui concerne les processeurs est actuellement caduc. On n'a besoin que d'un seul exécutable.

[3] À considérer comme une remarque d'intérêt historique uniquement !

Les **paramètres** $\boldsymbol{\beta}$ du modèle seront estimés par la méthode des **moindres carrés**, ici les moindres carrés linéaires, que l'on appelle les **moindres carrés ordinaires**, ou les MCO, ou les OLS, selon l'appellation anglaise, *ordinary least squares*. C'est pourquoi la commande de l'utilitaire qui sert à effectuer une estimation par les OLS est `ols`.

Regardez maintenant le fichier de commandes, `ols.ect`. Le contenu du fichier est comme suit :

```
sample 1 100
read ols.dat y x1 x2 x3
ols y c x1 x2 x3
quit
```

Bien que très simple, ce fichier nous permet de voir l'opération de quatre commandes de l'utilitaire ***Ects***.

La première, `sample` (*échantillon* en anglais), sert en effet à définir la taille de l'échantillon. Du point de vue de l'ordinateur, deux entiers sont définis par `sample` ; le premier, que nous noterons t_1, étant l'indice de la première observation à prendre en compte dans les opérations qui suivront, le deuxième, t_2, l'indice de la dernière. Dans notre cas actuel, on commence par la toute première observation sur l'ensemble des variables, et on finit par la centième. En général, la commande a la forme `sample` $\langle expn_1 \rangle$ $\langle expn_2 \rangle$. Les deux expressions, $\langle expn_i \rangle$, $i = 1, 2$, peuvent être, comme c'est le cas ici, des entiers explicites, mais elles peuvent être aussi des expressions algébriques, que nous considérerons plus loin, auxquelles l'ordinateur peut attribuer une valeur numérique. Il faut que $\langle expn_1 \rangle \geq 1$, et que $\langle expn_2 \rangle \geq \langle expn_1 \rangle$; sinon les résultats de la commande peuvent être imprévisibles. Attention ! Les valeurs initiales de t_1 et de t_2, définies par l'ordinateur, sont toutes deux égales à 1. Ceci implique que notre échantillon ne contient qu'une seule observation. En général, il y en aura $t_2 - t_1 + 1$.

À ce stade, la taille de l'échantillon est définie. Il y a 100 observations. Jusqu'ici aucune variable n'a été créée. Cette tâche est accomplie par la commande suivante du fichier, `read`. La syntaxe de cette commande est simple : le mot `read` sera suivi du nom d'un fichier accessible à l'utilitaire, et ensuite des noms d'une ou de plusieurs variables. Si le fichier n'existe pas, ou s'il est introuvable, un message d'erreur sera affiché. Les noms de variables doivent être des chaînes de moins de 11 caractères alphanumériques (c'est-à-dire, un des $26 + 26 + 10 = 62$ caractères A–Z, a–z, 0–9), sans espace blanc, commençant par un caractère alphabétique. Des noms comme `x1` sont donc admissibles ; `1x` ne le serait pas. Un nom potentiel qui a plus de 10 caractères sera tronqué : seuls les 10 premiers caractères seront retenus.

Avant de regarder la commande `ols` de plus près, nous remarquons que la commande `quit`, comme son nom l'indique, sert à terminer l'exécution du fichier de commandes. Si d'autres commandes suivent la commande `quit`, elles ne seront pas exécutées. Toutefois, l'usage de `quit` n'est pas obligatoire.

On aurait pu le supprimer dans le cas actuel, car de toute façon l'utilitaire arrêtera l'exécution de commandes dès qu'il constate qu'il est arrivé à la fin du fichier.

Et maintenant, voyons `ols`. Encore une fois la syntaxe est très simple. Après `ols` on trouve d'abord le nom de la variable dépendante, la régressande, ici `y`. Ensuite, outre les noms des trois autres variables trouvées dans le fichier `ols.dat`, il y a la constante, notée `c`. Cette variable est définie de manière automatique par ***Ects*** chaque fois qu'on lance une commande comme `ols` qui effectue une régression. Pour chaque observation de l'échantillon défini, la valeur de `c` égale 1. Attention ! Il est entièrement possible de redéfinir `c`, mais très peu souhaitable. Les conséquences d'une telle redéfinition seraient difficilement prévisibles et éventuellement désastreuses.

L'écriture algébrique correspondant à la commande

```
ols y c x1 x2 x3
```

est donc

$$\boldsymbol{y} = \alpha + \beta_1 \boldsymbol{x}_1 + \beta_2 \boldsymbol{x}_2 + \beta_3 \boldsymbol{x}_3 + \boldsymbol{u}, \tag{1}$$

où on note α la constante, le paramètre associé à la « variable » `c`.

C'est le moment d'essayer le programme `ols.ect`. Tapez l'instruction

```
ects ols
```

suivie d'un retour chariot. Sauf erreur quelque part, vous verrez affichées à l'écran de votre ordinateur les lignes successives du fichier `ols.ect` suivies de la salutation de l'utilitaire, `Processing terminated`, ce qui signifie que le programme a été exécuté. Mais où sont les résultats ? Vous les trouverez dans un fichier dont le nom est `ols.out`. Ce fichier peut être imprimé, ou visualisé, ou incorporé dans un traitement de texte ; comme le fichier de commandes, c'est un simple fichier ASCII. Voici son contenu :

```
sample 1 100
Sample set from observation 1 to observation 100

read ols.dat y x1 x2 x3
Variables y x1 x2 x3 now read in

ols y c x1 x2 x3

Ordinary Least Squares:

Variable  Parameter estimate  Standard error  T statistic

constant     87.053652          24.944936       3.489833
x1            1.579244           0.224144       7.045661
x2           -3.116123           0.194432     -16.026823
x3            1.169272           0.161331       7.247679
```

```
Number of observations = 100   Number of regressors = 4
Mean of dependent variable = 133.296804
Sum of squared residuals = 210392.502407
Explained sum of squares = 2.644800e+06
Estimate of residual variance
  (with d.f. correction) = 2191.588567
Standard error of regression = 46.814406
R squared (uncentred) = 0.926312  (centred) = 0.804901

Estimated covariance matrix:

622.249817  -5.188451   0.364278   3.064586
 -5.188451   0.050241  -0.007466  -0.019113
  0.364278  -0.007466   0.037804  -0.000765
  3.064586  -0.019113  -0.000765   0.026028

quit
```

On voit les réponses de l'utilitaire aux commandes `sample` et `read` : elles sont de simples constatations affirmant que la commande pertinente a été exécutée. En ce qui concerne `ols`, dans le premier tableau on peut lire, correspondant à chaque régresseur, le **paramètre estimé** (`Parameter estimate`), l'**écart-type estimé** (`Standard error`), et le ***t* de Student** (`T statistic`). Ensuite viennent quelques grandeurs scalaires associées à la régression : `Number of observations`, la taille de l'échantillon (c'est-à-dire, $t_2 - t_1 + 1$), `Number of regressors`, le nombre de régresseurs de la régression, `Sum of squared residuals`, la somme des carrés des résidus, `Explained sum of squares`, la somme des carrés expliqués, `Estimate of residual variance`, le $\hat{\sigma}^2$ de la régression, calculé avec prise en compte des degrés de liberté « usés » par l'estimation des paramètres, et finalement `R squared`, le R^2 de la régression. Ce dernier est `uncentred`, c'est-à-dire, non-centré. On trouve ensuite un deuxième tableau, qui contient la matrice de covariance estimée, la matrice que l'on note habituellement $\hat{\sigma}^2(\boldsymbol{X}^\top \boldsymbol{X})^{-1}$.

Les informations contenues dans le fichier de sortie `ols.out` sont également disponibles à l'intérieur d'***Ects***. Comme nous le verrons plus loin, il est possible d'utiliser ces informations pour construire des statistiques de test. Après chaque commande `ols`, un certain nombre de variables sont créées ou mises à jour. Correspondant aux quatre lignes qui suivent le premier tableau ci-dessus, ***Ects*** définit quatre variables. D'abord, `nobs` est une variable scalaire qui contient le nombre d'observations utilisées par `ols`. (Rapidement, une **variable scalaire** est comme une variable dont la définition est précédée de la commande `sample 1 1`, en fait, une matrice 1×1 : pour les détails, voir la section 2.1.) De manière similaire, la variable scalaire `nreg` contient le nombre de régresseurs. Viennent ensuite `ssr` et `sse`, variables auxquelles sont affectées respectivement la somme des carrés des résidus et la somme des carrés expliqués. En fait, une cinquième variable est définie, `sst`, qui contient

la somme des carrés totaux. En plus, les variables scalaires `R2` et `errvar` égalent le R^2 et le $\hat{\sigma}^2$ de la régression.[4]

Outre ces variables scalaires, ***Ects*** met à jour après chaque `ols` trois séries. (Une **série**, à la différence d'une variable scalaire, a plusieurs composantes. Si elle en a n, la série est représentée par un vecteur colonne $n \times 1$.) `fit` est une série qui contient les valeurs ajustées, `res` contient les résidus. Ces deux séries ont chacune t_2 éléments. La série `hat` contient les $t_2 - t_1 + 1$ éléments diagonaux de la matrice $\boldsymbol{P_X} \equiv \boldsymbol{X}(\boldsymbol{X}^\top\boldsymbol{X})^{-1}\boldsymbol{X}^\top$ – voir le premier chapitre de DM ; plus particulièrement la section 1.6.

Trois séries généralement moins longues sont aussi créées. Elles sont `coef`, qui a `nreg` éléments, et qui contient les paramètres estimés ; `stderr` de la même taille et qui contient les `nreg` écarts-type estimés ; `student`, toujours de la même taille et qui contient les t de Student. Finalement, deux *matrices*[5] sont créées ; `XtXinv` contient la matrice $(\boldsymbol{X}^\top\boldsymbol{X})^{-1}$, et `vcov` contient la même matrice multipliée par $\hat{\sigma}^2$, c'est-à-dire la matrice de variance estimée des paramètres estimés.

EXERCICES:

Essayez des commandes `sample` différentes dans le fichier `ols.ect`. Par exemple, quels sont les résultats des commandes `sample 50 100`, `sample 1 50`, `sample 1 200` ? Comprenez-vous pourquoi les résultats sont ce qu'ils sont ?

Faites lire les données en deux temps. Il est important de savoir pourquoi

```
sample 1 50
read ols.dat y x1 x2 x3
sample 51 100
read ols.dat y x1 x2 x3
sample 1 100
```

et

```
sample 51 100
read ols.dat y x1 x2 x3
sample 1 50
read ols.dat y x1 x2 x3
sample 1 100
```

ont des conséquences différentes. Avant de l'essayer sur l'ordinateur, devinez le résultat de

```
sample 1 100
read ols.dat y x1 x2 x3
sample 1 50
read ols.dat y x1 x2 x3
```

[4] Comme nous verrons plus loin, ces variables « scalaires » seront des matrices si la commande `ols` est utilisée pour faire plusieurs régressions à la fois.

[5] On verra comment on peut faire des opérations matricielles dans ***Ects*** au chapitre 3.

Même si les données sont introduites normalement, il est possible de scinder en deux la régression elle-même. Essayez

```
sample 1 45
ols y c x1 x2 x3
sample 46 100
ols y c x1 x2 x3
```

ou même

```
sample 1 33
ols y c x1 x2 x3
sample 34 67
ols y c x1 x2 x3
sample 68 100
ols y c x1 x2 x3
```

3. Lecture et Écriture de Données

Nous avons vu dans la section précédente comment ***Ects*** peut lire des données au moyen de la commande `read`. Cette commande est à utiliser avec précaution, parce qu'elle n'est pas très souple. En effet, il faut veiller à ce que les données soient rangées correctement dans le fichier. Chaque ligne du fichier doit contenir une observation sur chacune des variables dont le nom figure dans la liste fournie à la commande `read`. Par exemple, la toute première ligne du fichier `ols.dat` est comme suit :

```
231.3227    91.52508    -7.879150    -22.82027
```

et les 99 lignes qui suivent celle-ci ont un aspect pareil.

La règle est la suivante : si la dernière commande `sample` était `sample` t_1 t_2, le contenu de la première ligne du fichier sera affecté à l'observation t_1 de chacune des variables de la liste, et ainsi de suite : le contenu de la $i^{\text{ième}}$ ligne du fichier deviendra l'observation $t_1 + i - 1$. La lecture du fichier s'arrête soit à la fin du fichier soit après la lecture de l'observation t_2, si l'on y arrive avant la fin du fichier.

Si l'on essaie la commande

```
read ols.dat y x1 x2 x3 x4
```

que se passera-t-il ? ***Ects*** répondra tout gentiment que les *cinq* variables `y x1 x2 x3 x4` ont été saisies. Dans un sens, c'est vrai – ***Ects*** ne ment que rarement – mais la variable `x4` aura 100 observations (si la commande `read` est précédée de `sample 1 100`), chacune égale à zéro. Ce n'est probablement pas très gênant.

Et si l'on essaie

```
read ols.dat y x1 x2
```

quel sera notre sort ? Cette fois-ci, ***Ects*** répondra tout à fait correctement que les variables `y x1 x2` ont été saisies. Ce qui se passe est simple : si on demande trois variables seulement, ***Ects*** sautera à la ligne suivant après avoir trouvé trois chiffres.

Les lignes blanches ne sont pas admissibles, même au début du fichier, sous peine de trouver des zéros dans les variables saisies. Par exemple, si une ligne blanche se glissait dans `ols.dat` avant la première ligne de données, la commande

```
read ols.dat y x1 x2 x3
```

mettrait 0 pour la première observation sur les 4 séries, et la dernière ligne du fichier ne serait pas lue. L'effet serait identique à ce qu'on obtiendrait par les commandes

```
sample 2 100
read ols.dat y x1 x2 x3
```

Pourquoi ? Parce que l'effet de la commande `sample 2 100` est que `read` ne lira que $100-2+1=99$ lignes du fichier, qui seront affectées aux observations 2 à 100.

Si ***Ects*** arrive à la fin d'un fichier de données avant de trouver toutes les observations comprises dans l'échantillon défini par `sample`, les observations manquantes seront remplacées par zéro. Les commandes

```
sample 1 200
read ols.dat y x1 x2 x3
```

créent 4 variables de 200 composantes chacune, les 100 dernières composantes étant nulles.

Les données doivent obligatoirement être rangées par *observation* et non pas par *variable*. Certains logiciels d'économétrie permet à l'utilisateur de choisir le format de son fichier de données ; ce n'est pas le cas d'***Ects***. Normalement, un fichier dont le format n'est pas adapté à ***Ects*** peut être modifié sans trop de difficulté au moyen d'un éditeur de texte. Sinon, c'est le moment de vous servir de vos connaissances d'un langage de programmation comme PASCAL, ou C, ou même C++, afin de créer un nouveau fichier de la forme exigée par ***Ects***.

Au moment de faire un `read`, il se peut qu'il y ait parmi les variables de la liste soumise à `read` des variables déjà existantes. Dans ce cas, il y a deux choses différentes qui peuvent se produire. Si la variable pré-existante a déjà au moins t_2 observations, les valeurs des observations 1 à t_1-1, et après t_2+1, seront conservées telles quelles, les valeurs t_1 à t_2 étant remplacées par celles du fichier. Mais si la variable pré-existante a moins de t_2 observations, elle sera entièrement effacée et remplacée.

EXERCICES:

Faites des expériences devant l'ordinateur pour vous assurer de tout ce qu'on a vu jusqu'ici. Pour voir l'effet des commandes que vous mettez dans les fichiers de commandes, vous pouvez utiliser la commande `write`, qui sera décrite dans le paragraphe suivant. Pour voir les 10 premières observations des variables `y x1`, par exemple, on peut faire

```
sample 1 10
write temp.dat y x1
```

Un fichier `temp.dat` sera créé (s'il existe déjà, il sera écrasé) pour contenir les observations 1 à 10 de `y` et `x1`.

La commande `read` est la seule commande de lecture de données. En revanche, il existe quatre commandes permettant de les écrire ou de les afficher à l'écran. La première, déjà évoquée dans l'exercice ci-dessus, est `write`. La syntaxe est identique à celle de `read`; l'effet est exactement l'inverse de celui de `read`. Par exemple, la commande

```
write ols2.dat y x1 x2
```

sert à créer un fichier `ols2.dat` et de mettre dedans les $t_2 - t_1 + 1$ observations de `y`, `x1`, et `x2` de l'échantillon courant. Comme on l'a remarqué plus haut, si un fichier `ols2.dat` existe, il sera écrasé par la commande `write`. Si $t_1 = 1$ et $t_2 = 100$, le contenu du fichier `ols2.dat` serait comme celui de notre fichier de départ, `ols.dat`, sans la dernière colonne. En combinaison avec la commande `sample`, on peut aussi supprimer des lignes de `ols.dat`.

Les autres commandes d'écriture sont `print`, `show`, et `put`. En fait, les quatre commandes d'écriture sont implémentées par une seule fonction à l'intérieur d'***Ecta***, qui permet deux arguments. C'est le choix de ces deux arguments qui sert à distinguer les commandes. Le premier argument ne prend que deux valeurs possibles, VRAI et FAUX. Si la valeur est VRAI, le nom de chaque variable sera imprimé avant ses composantes; si la valeur est FAUX, seules les composantes seront imprimées. La valeur est VRAI pour `print` et pour `show`, elle est FAUX pour `write` et pour `put`. Si on fait

```
sample 1 4
write bidon y x1
print y x1
```

(`bidon` est le nom d'un fichier DOS!) le résultat de la commande `write` est

```
231.322700      91.525080
168.226300      58.321080
5.600930        71.192640
121.004900      85.375360
```

tandis que le résultat de `print` est

```
y               x1
231.322700      91.525080
```

```
168.226300      58.321080
5.600930        71.192640
121.004900      85.375360
```

Les résultats sont légèrement différents si les variables à imprimer sont des variables scalaires.

Le deuxième argument est plus intéressant, car il spécifie l'endroit où le résultat sera imprimé. Si on utilise `write`, il faut spécifier explicitement le nom d'un fichier DOS qui recevra le résultat de la commande. Comme on peut le constater de l'exemple précédent, il n'est pas nécessaire d'expliciter la destination d'une commande `print`. En effet, le résultat se trouvera dans le fichier de sortie, le même qui reçoit les résultats d'une commande `ols`, par exemple. Par défaut, le fichier de sortie porte le même nom que le fichier de commandes, avec l'extension `.out` (voir la section 1.2).[6] Si l'on souhaite que les résultats de l'écriture soient simplement affichés à l'écran, la commande qui sert est

```
show y x1
```

dont l'effet est identique à celui de

```
print y x1
```

à la différence près que ce qui serait destiné au fichier de sortie est maintenant affiché. La commande `put` écrit aussi dans le fichier de sortie. Nous ne verrons que plus tard pourquoi on a besoin de cette commande.

EXERCICES:

Amusez-vous à apprendre la pratique des commandes `show` et `write`. Introduisez les variables du fichier `ols.dat`, ou d'un autre fichier, par `read`, et faites-les afficher à l'écran par `show`. Ensuite créez un nouveau fichier par `write` pour contenir les variables. Attention ! Si vous choisissez le nom d'un fichier existant, il sera effacé. Visualisez le fichier `ols.dat` et votre nouveau fichier pour vous assurer que les variables sont les mêmes.

Astuce : on peut obtenir l'effet de `show y` sans l'impression des noms des variables par la commande

```
write con y
```

.DOS accepte le nom `con`[7] pour désigner la console. Une écriture sur `con` affiche à l'écran ; une lecture de `con` accepte ce que l'on tape au clavier. Si vous mettez la commande

```
read con y
```

dans un fichier `.ect`, l'ordinateur acceptera les données que vous taperez. Chaque observation doit être suivie d'un retour chariot.

Observez les conséquences si dans une commande `print`, `write` ou `show` vous mélangez des variables scalaires et des séries.

[6] Le fichier de sortie peut être choisi différemment. Voir la section 5.1.

[7] Effectivement, le nom ne convient pas du tout dans le monde francophone !

Chapitre 2
La Régression Linéaire

1. Transformation de Variables

Pour de multiples raisons, on a souvent envie de transformer les variables qu'on a introduites dans l'ordinateur, avant d'effectuer ou après avoir effectué une régression. Par conséquent, ***Ects*** donne la possibilité d'effectuer un grand nombre de manipulations algébriques, au moyen des commandes `gen` (pour *générer*) et `set` (de l'anglais pour *poser*). La distinction est très simple : on utilise `gen` s'il s'agit d'une **série**, c'est-à-dire une variable possédant une valeur pour chaque observation comprise entre t_1 et t_2, et `set` s'il s'agit une simple **variable scalaire**.

À titre illustratif, étudions le fichier de commandes, `gen.ect`. Voici son contenu :

```
sample 1 100
read ols.dat y x1 x2 x3
ols y c x1 x2 x3
set ssr0 = ssr
print ssr0
gen yy1 = y - 2*x1
ols yy1 c x2 x3
set ssr1 = ssr
set F1 = 96*(ssr1 - ssr0)/ssr0
print ssr1 F1
gen xx1 = x1 - x3
gen xx2 = x2 - x3
ols y c xx1 xx2
set ssr2 = ssr
set F2 = 96*(ssr2 - ssr0)/ssr0
print ssr2 F2
gen yy2 = y + 2*x2
ols yy2 c x1 x3
set ssr3 = ssr
set F3 = 96*(ssr3 - ssr0)/ssr0
print ssr3 F3
gen yy3 = y - 2*x1 + 2*x2 - 2*x3
```

```
ols yy3 c
set ssr4 = ssr
set F4 = 96*(ssr4 - ssr0)/(3*ssr0)
print ssr4 F4
quit
```

On commence par la lecture des données du fichier `ols.dat`, et on refait l'estimation par OLS de la régression (1) :

$$\boldsymbol{y} = \alpha + \beta_1 \boldsymbol{x}_1 + \beta_2 \boldsymbol{x}_2 + \beta_3 \boldsymbol{x}_3 + \boldsymbol{u}$$

L'idée de la suite du fichier est d'implémenter des tests en F des hypothèses suivantes :

$$\begin{array}{rll} \beta_1 = 2; & & \text{statistique F1} \\ \beta_1 + \beta_2 + \beta_3 = 0; & & \text{statistique F2} \\ \beta_2 = -2; & & \text{statistique F3, et} \\ \beta_1 = 2, \beta_2 = -2, \beta_3 = 2; & & \text{statistique F4} \end{array} \tag{2}$$

* * * *

Rappel : le Fisher employé pour le test en F d'un ensemble de contraintes a la forme

$$F = \frac{(\mathrm{SSR}_c - \mathrm{SSR}_{nc})/r}{\mathrm{SSR}_{nc}/(n-k)},$$

où SSR_c est la somme des carrés des résidus de la régression contrainte, SSR_{nc} est la même pour la régression non-contrainte, n est la taille de l'échantillon, k est le nombre de régresseurs dans la régression *non-contrainte*, et r est le nombre de restrictions.

* * * *

La commande

```
set ssr0 = ssr
```

crée une nouvelle variable scalaire, nommée `ssr0`, à laquelle est affectée la valeur de la variable `ssr`. Comme nous l'avons vu dans la section 1.2, `ssr` est une variable scalaire qui est mise à jour après chaque commande `ols`. Sa valeur est, bien entendu, la somme des carrés des résidus de la régression effectuée. La commande suivante

```
print ssr0
```

fera apparaître la ligne

```
ssr0 = 210392.502407
```

dans le fichier de sortie, `gen.out`.

Le prochain groupe de commandes sert à tester la première hypothèse de (2). Une nouvelle régressande est créée par

```
gen yy1 = y - 2*x1
```

Elle est ensuite régressée sur la constante, $\boldsymbol{x}_2$, et $\boldsymbol{x}_3$. Ensuite, le Fisher est calculé et enregistré dans le fichier de sortie. Les autres hypothèses sont testées

de la même façon. Observez que le symbole de multiplication est `*`, et qu'il ne peut être supprimé, à la différence de la notation algébrique habituelle.

Les manipulations et transformations effectuées dans `gen.ect` sont très simples. Il est possible de demander des opérations algébriques bien plus compliquées. Avant de considérer les multiples fonctions connues à ***Ects***, remarquons que les quatre opérations arithmétiques de base s'obtiennent très simplement, par l'usage des symboles `+`, `-`, `*`, et `/`. La **priorité** de `*` et `/` est plus élevée que celle de `+` et `-`. Par cela, on entend par exemple que, dans l'expression

```
y + x*z
```

le produit `x*z` sera calculé d'abord, et le résultat sera ensuite ajouté à `y`. En fait, le résultat est simplement $y + xz$. ***Ects*** respecte les conventions de l'écriture algébrique habituelle.

La commande `gen` effectue les opérations arithmétiques observation par observation. Par exemple, la commande

```
gen y = x + z
```

fait que

$$y_t = x_t + z_t \text{ pour } t = t_1, \ldots, t_2.$$

L'opération s'exprime également par une notation vectorielle :

$$\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{z}.$$

De même, l'effet de

```
gen y = x*z
```

est que

$$y_t = x_t z_t \text{ pour } t = t_1, \ldots, t_2. \tag{3}$$

Il est à noter que ceci ne correspond pas à une opération vectorielle standard. Cependant, on parle parfois du **produit Schur**, noté $\boldsymbol{y} * \boldsymbol{z}$, et défini par (3). L'opération donnée par la commande

```
gen y = x/z
```

à savoir

$$y_t = x_t / z_t \text{ pour } t = t_1, \ldots, t_2,$$

quoique très utile en économétrie, n'est pas connue à l'algèbre standard.

Il existe une autre opération qui a une priorité encore plus élevée que les quatre opérations de base. Il s'agit de l'opération « puissance. » Si l'on écrit

```
gen z = y^w
```

on obtient le résultat

$$z_t = y_t^{w_t}.$$

Il n'est pas nécessaire que l'exposant w_t soit un entier positif, mais sinon on risque d'avoir des ennuis si y_t est négatif. Dans un tel cas, z_t serait égal à zéro.

Regardons à présent quelques fonctions connues à ***Ects***. Nous verrons d'abord des fonctions d'un seul argument, qui peut être soit une variable scalaire soit une série, selon si l'on utilise `set` ou `gen`. Le résultat de l'application de chacune de ces fonctions est une nouvelle variable scalaire ou une nouvelle série, selon le cas. Les fonctions sont les suivantes : `log`, `sqrt`, `sign`, `abs`, `exp`, `sin`, `cos`, `tan`, `sum`, `time`, et `phi`. Le sens de quelques-unes de ces opérations est plus ou moins évident : voici les définitions algébriques :

$$\begin{aligned}
\texttt{log}(\texttt{z}) &= \log z;\\
\texttt{sqrt}(\texttt{z}) &= \sqrt{z};\\
\texttt{sign}(\texttt{z}) &= 1 \text{ si } z \geq 0, \text{ et } -1 \text{ sinon};\\
\texttt{abs}(\texttt{z}) &= z \text{ si } z \geq 0, \text{ et } -z \text{ sinon};\\
\texttt{exp}(\texttt{z}) &= e^z;\\
\texttt{sin}(\texttt{z}) &= \sin z;\\
\texttt{cos}(\texttt{z}) &= \cos z;\\
\texttt{tan}(\texttt{z}) &= \tan z.
\end{aligned}$$

Attention ! Si vous essayez d'utiliser les fonctions `log` et `sqrt` avec un argument négatif ou zéro, le résultat sera zéro, et un message d'erreur s'affichera.

La fonction `phi` est la fonction de répartition de la loi normale centrée réduite, c'est-à-dire la loi $N(0,1)$. La définition formelle en est

$$\Phi(x) = \frac{1}{(2\pi)^{1/2}} \int_{-\infty}^{x} e^{-y^2/2}\, dy. \tag{4}$$

La valeur de l'intégrale ne peut être exprimée sous forme analytique.

La fonction `sum` sert à calculer les valeurs cumulées d'une série. Si `z` est défini par

```
gen z = sum(y)
```

l'observation z_t égale $\sum_{i=1}^{t} y_i$. `time` sert normalement à définir des tendances temporelles, avec un argument constant. Si

```
gen z = time(0)
```

alors $z_t = t$. En général, quel que soit y, le résultat de la commande

```
gen z = time(y)
```

est que $z_t = y_t + t$.

Une opération de la toute première importance dans la manipulation des séries temporelles est l'opération de **retard**. A cet effet, la fonction `lag` existe. La syntaxe se décrit comme suit. Le résultat de la commande

```
gen z = lag(⟨expn1⟩,⟨expn2⟩)
```

est que, pour tout $t = t_1, \ldots, t_2$,

$$z_t = \begin{cases} y_{t-n} & \text{si } t - n > 0,\\ 0 & \text{si } t \leq n. \end{cases} \tag{5}$$

Le sens de l'équation (5) est que $\boldsymbol{y}$ est la *série* qui résulte de l'évaluation de l'expression $\langle expn_2 \rangle$, tandis que n est l'entier, positif ou négatif, qui résulte de l'opération suivante : soit $\boldsymbol{n}$ la série qui résulte de l'évaluation de $\langle expn_1 \rangle$, n est alors le plus grand entier tel que $n < (n_1 + 0.1)$, où n_1 est la première composante de $\boldsymbol{n}$.

* * * *

Par exemple, si l'on écrit `lag(n,y)`, et si `n` représente une série dont la première composante égale 4.5678, n est égal à 4, qui est le plus grand entier tel que $n < 4.5678 + 0.1 = 4.6678$.

* * * *

Normalement, on s'arrangera à ce que $\langle expn_1 \rangle$ soit un entier explicite, comme 2 ou -3, ou bien une variable scalaire à valeur entière.

EXERCICES:

Quel est le résultat de la commande `gen` sur des séries déjà existantes ? Essayez par exemple

```
sample 1 100
read ols.dat y x1
sample 1 50
gen z = y
sample 45 70
gen z = x1
```

et trouvez comment `gen` et `sample` interagissent.

Le fichier `ar.ect` présente un exemple étendu de manipulations par `gen` et `set`. Il s'agit d'une estimation non-linéaire effectuée au moyen de la régression artificielle de Gauss-Newton. Observez que l'on peut faire comme si une observation sur une série était une variable scalaire. En effet, `beta(2)` est l'observation 2 de la série `beta`.

2. La Colinéarité

Si, dans le modèle de régression linéaire

$$\boldsymbol{y} = \boldsymbol{X\beta} + \boldsymbol{u},$$

il existe une combinaison linéaire des colonnes de la matrice $\boldsymbol{X}$ égale à zéro ou presque, le modèle n'est plus identifié. Les ordinateurs ne sont pas capables d'effectuer les opérations arithmétiques à une précision infinie, et par conséquent, même si théoriquement la matrice $\boldsymbol{X}$ a le rang plein, une colinéarité approximative peut être tout aussi gênante qu'une colinéarité exacte.

Pour cette raison, avant d'exécuter une commande `ols`, ***Ects*** supprime tout régresseur qui peut s'exprimer comme une combinaison linéaire des régresseurs déjà définis à un « petit » écart près. Les détails de ce « petit » écart sont assez

techniques, mais il existe une variable scalaire pré-définie par ***Ects***, nommé `TOL` (pour tolérance), dont la valeur de défaut égale 10^{-8}, et qui sert à donner un sens précis au terme « petit. » Bien que la valeur de `TOL` puisse être modifiée par la commande `set`, il est fortement déconseillé de s'écarter de très loin de la valeur pré-définie, sous peine non seulement d'obtenir des résultats erronés mais aussi de déranger le fonctionnement de l'ordinateur, et en particulier celui du coprocesseur numérique.

Une façon classique de créer une colinéarité exacte est d'utiliser quatre variables saisonnières en même temps qu'une constante. Donc, si on essaie d'effectuer la régression suivante :

```
ols y c x s1 s2 s3 s4
```

où on fait intervenir les quatre variables `s1`, `s2`, `s3`, et `s4`, on s'attend à ce que ***Ects*** supprime `s4`. Pourquoi `s4` ? Parce que `s4` est la première variable de la liste de régresseurs susceptible d'être exprimée en une combinaison des régresseurs déjà définis.

Regardez maintenant le fichier `season.ect`. Après la lecture d'un sous-échantillon des données contenues dans `ols.dat` vous verrez les commandes suivantes :

```
gen s1 = seasonal(4,1)
gen s2 = seasonal(4,2)
gen s3 = seasonal(4,3)
gen s4 = seasonal(4,4)
```

C'est ainsi que l'on peut générer les variables saisonnières. La « fonction » `seasonal` prend deux arguments, dont chacun doit être un entier positif. En fait, les arguments sont évalués exactement comme le premier argument de la fonction `lag`, c'est-à-dire que l'on utilise le plus grand entier qui est plus petit que la première composante de l'argument, plus 0.1. L'expression `seasonal(`$\langle n \rangle$`,`$\langle m \rangle$`)` signifie une variable dont chaque « observation » égale zéro sauf celles dont l'indice a la forme $m + kn$, pour $k = 0, 1, \ldots$, pour lesquelles la valeur est 1. m est l'indice de la première observation pour laquelle la valeur est 1, et n est la périodicité de la saisonnalité. Si le premier argument n est plus petit que 1, il est remplacé par 1. Il en suit que `s1` est différent de zéro uniquement pour les observations $1, 5, 9, \ldots$. Pour `s2` c'est $2, 6, 10 \ldots$, et ainsi de suite. Vous pouvez insérer la commande `print s1 s2 s3 s4` dans le fichier de commandes après la génération des variables saisonnières pour vérifier qu'elles ont été générées correctement.

Viennent ensuite les commandes

```
ols y c x s1 s2 s3
ols y c x s1 s2 s3 s4
ols y x s1 s2 s3 s4
```

Si maintenant vous faites exécuter le fichier `season.ect`, vous verrez ce que cela a donné. La première et la dernière régression n'ont rien d'anormal, mais

la seconde, où il y a en effet une variable de trop, a produit un listing identique à celui de la première, à l'exception de la toute dernière ligne. Là, ***Ects*** a fait la remarque suivante :

```
The variable s4 was excluded as collinear with other variables.
```

En français, la variable `s4`, étant colinéaire avec d'autres variables, fut supprimée. Observez que les t de Student, ainsi que tous les éléments de la matrice de variance-covariance, sont identiques dans les listings des deux premières régressions. Ceci implique que ***Ects*** a bien calculé l'estimation de $\hat{\sigma}^2$, ne tenant compte que de 5 régresseurs.

EXERCICES:

Essayez d'avoir une idée du degré d'inexactitude d'une colinéarité qui fera supprimer une variable par ***Ects***. Générez deux variables presque colinéaires, et ensuite utilisez-les comme régresseurs dans une même régression, comme suit :

```
gen x2 = x1 + MINUSCULE
ols y c x1 x2
```

où `MINUSCULE` est une grandeur que vous pouvez modifier par la commande `set`. Exceptionnellement, jouez aussi sur la valeur de la variable pré-définie `TOL`. La valeur de défaut est 10^{-8}, que vous exprimerez comme `1.0E-8` en « informatique ». Essayez des valeurs de `TOL` plus grandes, et trouvez en fonction de `TOL` la valeur critique de `MINUSCULE` qui permet à ***Ects*** de distinguer les deux régresseurs.

3. Les Variables Instrumentales

Les raisons sont nombreuses pour lesquelles on est conduit à employer des variables instrumentales ; voir le Chapitre 7 de DM. Rappelons à présent les éléments de la procédure. L'estimateur des variables instrumentales (IV) de la régression linéaire

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{u}$$

pour une matrice d'instruments $\boldsymbol{W}$ s'écrit comme

$$\hat{\boldsymbol{\beta}}_{\text{IV}} = (\boldsymbol{X}^\top \boldsymbol{P}_W \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{P}_W \boldsymbol{y}. \tag{6}$$

Il faut que le nombre l d'instruments soit au moins égal au nombre k de régresseurs. Supposons que $k = 3$, et $l = 5$. Alors la traduction de (6) qu'il faut faire pour que ***Ects*** puisse la lire est comme suit :

```
iv y x1 x2 x3 (w1 w2 w3 w4 w5)
```

Si l est inférieur à k, un message d'erreur sera affiché, et ***Ects*** sautera à la commande suivante.

Il est très souvent le cas que parmi les instruments $\boldsymbol{W}$ on retrouve des régresseurs. C'est normal, `iv` n'est nullement dérangé par ce phénomène. En effet, tout régresseur exogène doit être instrument aussi.

* * * *

En particulier, la constante et d'autres variables artificielles, telles les variables saisonnières, autres variables indicatrices,...

* * * *

La définition de $\hat{\sigma}^2$ utilisée par `iv` est la bonne, à savoir

$$\hat{\sigma}^2 = \frac{1}{n-k}\sum_{t=0}^{n}(y_t - X_t\hat{\boldsymbol{\beta}}_{\text{IV}})^2.$$

Cette définition ne correspond pas à celle de la procédure des doubles moindres carrés, qui est erronée.

Le fichier de commandes `2sls.ect` sert à illustrer l'utilisation de `iv`. À partir des données contenues dans `2sls.dat`, on effectue, par IV, une estimation du système simultané

$$\begin{aligned} q_t &= \alpha_d - \beta_{11}p_t + \beta_{12}Y_t + u_{1t}, \text{ et} \\ q_t &= \alpha_o + \beta_{21}p_t - \beta_{22}C_t + u_{2t}. \end{aligned} \tag{7}$$

Les variables q et p sont respectivement les quantités échangées et les prix ; les exogènes Y et C sont les revenus des consommateurs et les coûts de production.

Exercices:

Refaites l'estimation du système (7) par les doubles moindres carrés. Essayez de calculer la bonne matrice de variance-covariance des paramètres estimés sans passer par `iv`.

Un problème de colinéarité peut survenir dans le contexte d'une estimation par variables instrumentales, comme dans le contexte d'un OLS. En effet, l'élimination de régresseurs ou d'instruments redondants par ***Ects*** procède en deux étapes. D'abord, s'il y a colinéarité entre les instruments, les instruments inutiles seront écartés de la liste ; ensuite, après la projection des régresseurs sur l'espace des instruments, (création de la matrice $\boldsymbol{P}_W\boldsymbol{X}$), les colonnes éventuellement redondantes de cette matrice seront éliminées.

La commande `iv`, comme la commande `ols`, crée ou met à jour un ensemble de variables. Les scalaires `ssr`, `sse`, `sst` ont le même sens que dans le contexte d'une régression effectuée par `ols`. En général, la relation `sse` + `ssr` = `sst`, qui est vérifiée dans le cas d'un `ols`, n'est plus vraie dans le cas d'un `iv`, car la projection utilisée est oblique, à la différence de la projection orthogonale utilisée par `ols` : voir le chapitre 7 de DM.

Exercices:

Refaites l'estimation du modèle de la section 1.2 par `iv`. Vérifiez que, si les instruments et les régresseurs sont les mêmes variables, tous les résultats sont identiques à ceux donnés pas `ols`.

Le scalaire `errvar` contient $\hat{\sigma}^2$, et les séries `coef`, `stderr`, `student` contiennent les paramètres estimés, leurs écarts-type estimés, et les Students, exactement comme dans le cas `ols`.

Aux scalaires `nobs`, (n = la taille de l'échantillon), `nreg`, (k =le nombre de régresseurs), s'ajoute cette fois-ci `ninst`, le nombre d'instruments, l. Les séries `fit` et `res` contiennent les valeurs ajustées et les résidus de la régression par variables instrumentales.

À la place de la matrice `XtXinv`, qui n'a aucun intérêt particulier dans le contexte IV, une matrice `XtPwXinv` est définie. Comme la notation l'indique, c'est la matrice $(\boldsymbol{X}^\top \boldsymbol{P}_W \boldsymbol{X})^{-1}$. Encore une fois la matrice `vcov` contient l'estimation de la matrice de covariance estimée des paramètres estimés, $\hat{\sigma}^2(\boldsymbol{X}^\top \boldsymbol{P}_W \boldsymbol{X})^{-1}$.

Chapitre 3
Les Opérations Matricielles

1. Opérations Simples

Dans la pratique de l'économétrie, il est très souvent nécessaire de faire des **calculs matriciels**. Pour répondre à ce besoin, ***Ects*** possède une commande `mat`, qui fonctionne comme `gen` et `set`, mais qui opère sur les matrices. À l'intérieur d'***Ects***, toute variable est une matrice. Un scalaire est simplement une matrice 1×1, et une série est un vecteur colonne, c'est-à-dire une matrice $n \times 1$, pour un entier positif n donné. Selon le choix de `mat`, `gen`, ou `set`, les opérations arithmétiques qui suivent la commande sont traitées différemment.

En ce qui concerne `gen`, tout dépend des valeurs courantes des variables internes `smplstart` et `smplend`. Ce sont les variables que nous avons notées t_1 et t_2 dans la section 1.2. On peut avoir accès direct à ces variables (par `print`, `show`, par exemple) en passant par des variables « externes » qui portent les mêmes noms. Mais attention ! Pour changer les valeurs des variables internes, il faut obligatoirement utiliser la commande `sample`. Un `set` changerait bien les variables externes, mais non pas les valeurs internes utilisées pour déterminer la taille de l'échantillon. Quand on lance une commande `set`, les valeurs internes sont posées égales à 1, de sorte que toute série devienne scalaire. Les anciennes valeurs sont reprises pour les commandes suivantes.

EXERCICES:

Pour bien comprendre les conséquences de ces manipulations, faites tourner les commandes suivantes :

```
sample 1 10
gen y = time(0)
sample 3 3
gen i = y
set ii = y
set iii = y(3)
print i ii iii
```

Expliquez pourquoi les résultats sont ce qu'ils sont.

Mais pour la durée d'une commande `mat`, ni les valeurs internes ni les valeurs externes ne sont prises en compte. Toute variable est considérée comme une matrice, ce qui implique qu'à chaque variable on attribue deux dimensions, n, le nombre de lignes, et m, le nombre de colonnes. Normalement, on s'arrangera à ce que ces deux dimensions soient correctes pour les opérations que l'on demande. Ainsi, si on veut additionner deux matrices, les deux dimensions des deux matrices doivent être les mêmes ; pour les multiplier, il faut que le nombre de colonnes du facteur de gauche soit égal au nombre de lignes du facteur de droite.

* * * *

La règle générale n'est guère plus compliquée : pour les détails intimes, voir la section 7.2. En cas d'addition ou de soustraction, les dimensions de la première matrice sont utilisées. Ceci veut dire que si la deuxième matrice a moins de lignes que la première, les lignes manquantes sont remplacées par zéro ; si elle en a plus, elles ne sont pas retenues dans le résultat. En cas de multiplication matricielle, le nombre de lignes du produit matriciel égale le nombre de lignes du facteur de gauche, le nombre de colonnes du produit égale le nombre de colonnes du facteur de droite. Si une ligne ou une colonne n'existe pas, elle est remplacée par une ligne ou une colonne de zéros.

* * * *

Comment créer des matrices autres que les scalaires et les séries ? Une façon très simple serait d'employer un produit extérieur de deux vecteurs, c'est-à-dire, de deux séries. Par exemple, si deux séries `x` et `y` ont été créées, pour un échantillon de taille n, l'opération

```
mat Z = x*y'
```

donnera une matrice `Z`, de forme $n \times n$. On aura en effet

$$\boldsymbol{Z} = \boldsymbol{x}\boldsymbol{y}^{\top}.$$

On voit que le signe `*` indique ici la multiplication matricielle.
Dans l'exemple ci-dessus, on a fait appel à une autre opération très utile, à savoir la **transposition**. On note couramment $\boldsymbol{Z}'$ la transposée de $\boldsymbol{Z}$.[7] C'est pourquoi, à l'intérieur d'une commande `mat`, on peut utiliser la prime $'$ pour effectuer la transposition d'une matrice.
Mais il est évident que la plupart des matrices ne sont pas des produits extérieurs. Deux commandes permettent de construire une matrice à partir de ses colonnes ou de ses lignes. Si nous reprenons l'exemple de la régression linéaire considérée dans le fichier `ols.ect`, on pourra construire une matrice 100×4 par la commande

```
mat X = colcat(c, x1, x2, x3)
```

La fonction `colcat` sert à concaténer les colonnes qui lui sont fournies.

[7] Cette notation n'est pas utilisée dans ce manuel : on préfère la notation $\boldsymbol{Z}^{\top}$.

* * * *

La commande ci-dessus ne marchera qu'*après* la commande `ols`. C'est parce que la constante, `c`, n'est pas créée avant qu'une régression ne soit demandée.

* * * *

Les arguments de la fonction `colcat` peuvent être des matrices. Ainsi, dans l'exemple, on aurait pu procéder en deux fois ou en trois fois. En fait, les matrices `X1`, `X2`, et `X3` définies par les commandes suivantes seront identiques :

```
mat X1 = colcat(c, x1, x2, x3)
mat Y = colcat(c, x1, x2)
mat X2 = colcat(Y, x3)
mat Y1 = colcat(c, x1)
mat Y2 = colcat(x2, x3)
mat X3 = colcat(Y1, Y2)
```

La fonction `rowcat` (une *ligne* est un *row* en anglais) fonctionne de la même manière. Voici trois méthodes équivalentes de construire la transposée de la matrice `X` créée ci-dessus.

```
mat ctr = c'
mat x1tr = x1'
mat x2tr = x2'
mat x3tr = x3'
mat X1tr = rowcat(ctr, x1tr, x2tr, x3tr)
mat X2tr = X1'
mat X3tr = rowcat(c', x1', x2', x3')
```

* * * *

Les fonctions `colcat` et `rowcat` créent des matrices aussi grandes que possibles. Si toutes les colonnes d'un `colcat` n'ont pas la même dimension, la dimension du résultat sera la plus grande de celles de l'ensemble des arguments. Une propriété analogue est vraie pour `rowcat`.

* * * *

Il est à remarquer que `colcat` et `rowcat` peuvent être utilisés également à l'intérieur d'une commande `gen`. Les résultats seront identiques à un détail près : le nombre de lignes d'un `colcat` ne peut jamais dépasser le `smplend` sous `gen`. Ainsi, si l'on remplace `mat` par `gen`, et si on définit un échantillon plus petit, on obtiendra une matrice plus petite. Supposons que l'état courant de notre échantillon soit donné par

```
sample 1 100
```

Ensuite considérez :

```
sample 1 10
mat X1 = colcat(c, x1, x2, x3)
gen X2 = colcat(c, x1, x2, x3)
```

La matrice `X1` aura 100 lignes, mais `X2` n'en aura que 10.

Il n'existe pas d'opération de division matricielle. En revanche, une matrice carrée non-singulière peut être **inversée**. Pour obtenir l'inverse d'une matrice carrée non-singulière, on utilise la commande suivante :

```
mat XX = X inv
```

EXERCICES:

La fonction `inv` permet finalement d'obtenir l'estimateur des OLS sans passer par la commande `ols` ! En effet, on peut vérifier que le résultat des opérations suivantes :

```
mat X = colcat(c, x1, x2, x3)
mat betahat = (X'*X) inv * (X'*y)
```

est identique à celui de

```
ols y X
```

L'exercice précédent démontre que les matrices sont reconnues par la commande `ols`. Comme c'était le cas pour `colcat`, on peut mélanger séries et matrices dans les arguments de `ols`. C'est vrai également pour la commande `iv`.

EXERCICES:

Faites tourner les commandes suivantes :

```
mat X = colcat(c, x1, x2)
ols y X x3
```

et voyez comment ***Ects*** désigne les trois régresseurs de la matrice `X`. On risque une certaine confusion s'il y a une autre variable `X1` !
Essayez le même genre d'exercice pour `iv`. Vous verrez que les instruments ainsi que les régresseurs peuvent être regroupés à l'intérieur d'une matrice.

Que se passe-t-il si l'on demande à ***Ects*** d'inverser une matrice qui n'est pas carrée, ou qui est carrée mais singulière ? On peut répondre aux deux questions en même temps. On obtiendra ce qu'on appelle une **inverse généralisée** de la matrice. Soit $\boldsymbol{A}$ une matrice $n \times m$. Une inverse généralisée de $\boldsymbol{A}$ est une matrice $\boldsymbol{A}^+$ de la forme $m \times n$ qui vérifie les propriétés suivantes (voir par exemple la section R.F. du Gouriéroux-Monfort (1989)) :

$$\begin{aligned} \boldsymbol{A}\boldsymbol{A}^+\boldsymbol{A} &= \boldsymbol{A}; \\ \boldsymbol{A}^+\boldsymbol{A}\boldsymbol{A}^+ &= \boldsymbol{A}^+ \end{aligned} \tag{8}$$

La symétrie de ces deux conditions montre clairement que $\boldsymbol{A}$ est une inverse généralisée de $\boldsymbol{A}^+$. En plus, on voit que $(\boldsymbol{A}^+)^\top$ est une inverse généralisée de $\boldsymbol{A}^\top$. Il est clair aussi que, si $\boldsymbol{A}$ est une matrice carrée et non-singulière, $\boldsymbol{A}^{-1}$ vérifie les deux conditions (8). Considérez maintenant une matrice de régresseurs, $\boldsymbol{X}$, de la forme $n \times k$. On démontre aisément qu'une inverse

généralisée de $\boldsymbol{X}$ est la matrice $(\boldsymbol{X}^\top\boldsymbol{X})^{-1}\boldsymbol{X}^\top$. Il en découle que le vecteur `betahat` défini plus haut par la commande

```
mat betahat = (X'*X) inv * (X'*y)
```

aurait pu être défini plus simplement par

```
mat betahat = X inv * y
```

EXERCICES:

Reprenez l'exemple de la section 2.2 sur les variables colinéaires. Créez une matrice regroupant la constante et les quatre variables saisonnières, et calculez les « paramètres estimés » au moyen de l'inverse généralisée de cette matrice. Ensuite comparez les résultats que vous avez obtenus à ceux fournis par une commande `ols`. Normalement, les estimations paramétriques ne sont pas les mêmes. Mais si vous utilisez les deux vecteurs de paramètres pour calculer les valeurs ajustées, ou les résidus, les résultats devraient être identiques.

Pour inverser une matrice `A`, on a écrit jusqu'ici `A inv` plutôt que `A^(-1)`. Mais la deuxième écriture est possible : les deux commandes

```
mat Z = A inv
mat ZZ = A^(-1)
```

donne la même matrice, à condition que `A` soit une matrice carrée. Mais la notation est plus générale. En fait, si la valeur de la variable scalaire `n` est un entier, positif, négatif, ou nul, la valeur de l'expression `A^n`, si `A` est une matrice carrée, sera la matrice `A` à la puissance `n`. Ainsi `A^0` est une matrice identité, `A^1` est la matrice `A` inchangée, `A^2` est le carré de `A`, `A^(-2)` est le carré de `A inv`, et ainsi de suite. Si l'exposant `n` n'est pas entier, le plus grand entier plus petit (algébriquement) que `n + 0.1` sera utilisé.

Si `A` est une série, ou, de manière équivalente, une matrice à une seule colonne et à plus d'une ligne, alors `A^n` est une série composée des éléments de `A`, à la puissance `n`. Autrement dit, les effets des deux commandes `mat Z = A^n` et `gen Z = A^n` sont identiques.

* * * *

En particulier, si l'on essaie de calculer une puissance non entière d'un élément négatif, le résultat sera zéro.

* * * *

Si, à la place de l'exposant `n` on met une série, ou une matrice, plutôt qu'un scalaire, l'élément $(1, 1)$ de la série ou de la matrice sera utilisé. Finalement, si, dans la commande `mat Z = A^n`, `A` n'est ni une matrice carrée ni une série, un message d'erreur sera affiché et la commande ne sera pas exécutée.

La commande `gen Z = A^B` a toujours un sens, même si `A` et `B` sont des matrices. La règle générale est très simple : chaque colonne de `A` est traitée séparément. La conséquence en est que l'ensemble de commandes

```
mat A = colcat(a1, a2, a3)
gen Z = A^n
```

équivaut à l'ensemble de commandes

```
gen z1 = a1^n
gen z2 = a2^n
gen z3 = a3^n
mat Z = colcat(z1, z2, z3)
```

* * * *

De même, `set Z = A^B` a toujours un sens. Seulement, toute série, toute matrice, sera considérée comme n'ayant qu'une seule ligne.

* * * *

EXERCICES:

Créez une matrice de variables saisonnières :

```
sample 1 50
gen s1 = seasonal(4,1)
gen s2 = seasonal(4,2)
gen s3 = seasonal(4,3)
gen s4 = seasonal(4,4)
mat S = colcat(s1, s2, s3, s4)
```

Quelles seront les résultats des opérations suivantes :

```
mat Z = S^3
gen Z = S^3
gen Z = S^(-3)
gen Z = S^0
gen Z = S^s1
gen Z = S^s2
gen Z = S^S
```

et pourquoi ?

2. Éléments et Blocs d'une Matrice

Il est souvent nécessaire d'affecter des valeurs numériques aux élements d'une matrice un à un. Ou bien d'extraire un élément d'une matrice ou d'une série. Ou même d'extraire un bloc, c'est-à-dire une sous-matrice, d'une matrice plus grande. Tout cela se fait au moyen des commandes `set` et `mat`.

D'abord, si l'on souhaite changer la valeur d'un élément d'une série, `y` par exemple, la commande appropriée est

```
set y(i) = x
```

L'indice `i`, ainsi que la valeur `x`, sera calculé sous les règles de toute commande `set`, c'est-à-dire, les variables `smplstart` et `smplend` sont toutes deux égales à 1. Normalement, le résultat du calcul de l'indice `i` doit être un entier positif. Sinon, le résultat sera remplacé par $\max(1, [\mathtt{i}+0.1])$, où la notation $[x]$ signifie

le plus grand entier plus petit que l'argument en crochets. Attention ! À la différence de la commande `set y = x`, qui peut très bien servir à créer la variable scalaire `y`, la série `y` doit exister avant que l'on puisse affecter des valeurs à ses éléments. Sinon, un message d'erreur sera affiché. En revanche, si l'on essaie d'affecter une valeur à un élément qui n'existe pas, la commande sera tout simplement sans effet.

Une syntaxe similaire est utilisée pour les éléments de matrice. Pour changer la valeur de l'élément (i, j) d'une matrice, il suffit d'employer la commande

```
set A(i,j) = x
```

Les règles énoncées ci-dessus s'appliquent ici également quant à l'évaluation des indices `i` et `j`. Si l'on oublie le deuxième indice, c'est l'élément i de la première colonne qui sera changé. Si l'on met trop d'indices, comme par exemple dans

```
set A(i,j,k) = x
```

l'effet est identique à celui de

```
set A(i) = x
```

ou bien de

```
set A(i,1) = x
```

Les éléments de séries et de matrices peuvent être utilisés exactement comme les variables scalaires dans les expressions algébriques. Par exemple, dans une commande `set`,

```
set a = y(i)
```

une variable scalaire `a` est créée, après avoir été effacée si elle existe déjà, et reçoit la valeur de `y(i)`. Encore une fois, le calcul de l'indice `i` se fait selon les règles énoncées plus haut. Évidemment, pour éviter une erreur de syntaxe, il faut que la variable `y` existe. Si `y` a plusieurs colonnes, c'est la première qui sera sélectionnée par la commande, et si l'élément n'existe pas, la valeur affectée sera nulle. Pour accéder à un élément d'une colonne autre que la première, la commande appropriée est

```
set a = A(i,j)
```

Les notations `y(i)` et `A(i,j)` peuvent être utilisées également dans les commandes `gen` et `mat`, à chaque endroit où une variable scalaire peut être attendue.

* * * *

On peut avoir des résultats inattendus si l'on n'utilise qu'un seul indice pour accéder à une variable qui a plusieurs colonnes. Il se peut que de tels résultats soient utiles, mais normalement il existe une meilleure façon d'obtenir les mêmes résultats : voir le paragraphe suivant.

* * * *

Ainsi, les commandes

```
sample 1 20
gen y = A(2,3)
```

servent à créer un vecteur `y`, à 20 composantes, chacune égale à la valeur de l'élément `A(2,3)` s'il existe, et à zéro sinon.

Il est possible d'extraire des sous-matrices d'une matrice pré-existante. Par exemple, si l'on souhaite éliminer la première colonne d'une matrice `A` de la forme $n \times 3$, on peut utiliser la commande

```
mat A = A(1,n,2,3)
```

Le sens de cette écriture est que la matrice `A` sera remplacée par le bloc de la matrice pré-existante défini par les lignes de la première à la $n^{\text{ième}}$, et les colonnes de la deuxième à la troisième.

* * * *

Bien sûr, il n'est pas nécessaire d'écraser la matrice pré-existante : on pourrait tout aussi bien avoir

```
mat B = A(1,n,2,3)
```

dont le résultat serait la création d'une nouvelle matrice.

* * * *

Les quatre indices de ce genre d'expression sont évalués selon les règles habituelles. Le premier correspond à la première ligne de la sous-matrice à extraire, le second à la dernière ligne, le troisième à la première colonne, et le quatrième à la dernière colonne. On aurait pu choisir un autre système, mais il faut parfois faire des choix plus ou moins arbitraires !

Si l'indice de la dernière ligne est inférieur à celui de la première, une seule ligne, celle qui correspond à l'indice de la première ligne, sera sélectionnée. De même pour les colonnes. Comme d'habitude, les lignes et les colonnes inexistantes seront remplacées par des zéros.

Les expressions de la forme `A(i,j,k,l)` ne peuvent être utilisées que dans les *membres de droite* d'une commande `mat`. Elles ne sont reconnues ni par les commandes `set` et `gen` ni dans les membres de gauche. Ainsi, la commande

```
mat A(i,j,k,l) = ...
```

donnera lieu à une erreur de syntaxe. Pour insérer des sous-matrices à l'intérieur d'une matrice existante, il faut utiliser les fonctions `colcat` et `rowcat`. Ce n'est pas toujours très commode ; dans une version ultérieure on aura peut-être d'autres possibilités ...

EXERCICES:

Le fichier `sur.dat` contient 50 observations sur les cinq variables $\boldsymbol{y}_1$, $\boldsymbol{y}_2$, $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, et $\boldsymbol{x}_3$. On souhaite faire une estimation du système d'équations

$$\begin{aligned} \boldsymbol{y}_1 &= \alpha_1 + \beta_{11}\boldsymbol{x}_1 + \beta_{12}\boldsymbol{x}_2 + \boldsymbol{u}_1 \\ \boldsymbol{y}_2 &= \alpha_2 + \beta_{22}\boldsymbol{x}_2 + \beta_{23}\boldsymbol{x}_3 + \boldsymbol{u}_2. \end{aligned} \tag{9}$$

Dans la section 9.8 de DM, on voit qu'on peut procéder par une **régression empilée**, qui s'écrit sous la forme matricielle suivante :

$$\begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{\iota} & \boldsymbol{x}_1 & \boldsymbol{x}_2 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{\iota} & \boldsymbol{x}_2 & \boldsymbol{x}_3 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \beta_{11} \\ \beta_{12} \\ \alpha_2 \\ \beta_{22} \\ \beta_{23} \end{bmatrix} + \begin{bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{u}_2 \end{bmatrix}. \tag{10}$$

(Le vecteur $\boldsymbol{\iota}$, dont chaque composante égale 1, correspond à la constante.) Selon la théorie de la régression empilée, les estimations données par les moindres carrés ordinaires appliqués à (10) sont identiques à celles données par les MCO appliqués aux deux équations (9) séparément. Construisez les matrices de l'écriture (10) afin de vérifier ce résultat. Il est utile de savoir que, pour créer un vecteur 50×1 dont toutes les composantes sont nulles, il suffit de faire

```
sample 1 50
gen zero = 0
```

Dans la pratique, il peut arriver que l'on ait à régresser plusieurs variables sur un même ensemble de régresseurs. Par exemple, dans l'exercice précédent, on aurait pu régresser les deux variables $\boldsymbol{y}_1$ et $\boldsymbol{y}_2$ sur l'ensemble des régresseurs, à savoir, la constante, $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, et $\boldsymbol{x}_3$. De même, si l'on souhaite effectuer une estimation par les doubles moindres carrés, il est nécessaire au préalable de régresser toutes les explicatives sur l'ensemble des instruments. Pour alléger le travail de la programmation de tout cela, les commandes `ols` et `iv` acceptent des matrices à la place de la variable dépendante. Pour chaque colonne de la matrice, une estimation sera effectuée, soit par OLS, soit par IV.

EXERCICES:

Faites tourner le programme suivant :

```
sample 1 50
read sur.dat y1 y2 x1 x2 x3
mat Y = colcat(y1, y2)
gen c = 1
mat colcat(c, x1, x2, x3)
ols Y X
```

pour voir le résultat d'un `ols` avec une matrice de variables dépendantes. Notez qu'il est nécessaire de générer la constante explicitement dans cet exercice. La raison en est que la constante n'est générée automatiquement que si une *régression*, par `ols` ou par `iv`, est effectuée.

Si plusieurs estimations sont effectuées en une seule fois,[8] comme dans l'exercice précédent, certaines des variables créées ou mises à jour par ***Ects*** auront des dimensions différentes de celles du cas ordinaire. Considérons

[8] On parle alors d'une estimation ou d'une régression **multivariée**.

d'abord le cas des régressions par `ols`. Soit n la taille de l'échantillon, k le nombre de régresseurs, et m le nombre de variables dépendantes. La variable `coef` devient maintenant une matrice de la forme $k \times m$, les m colonnes étant les paramètres estimés des m régressions. Les variables `fit` et `res` deviennent des matrices $n \times m$: encore une fois, les m colonnes correspondent aux m régressions.

* * * *

Cette remarque est à relativiser si `smplstart` est différent de 1. Le nombre de lignes de `fit` et `res` est égal à `smplend`, mais les `smplstart`-1 premières lignes sont nulles.

* * * *

Les trois variables `ssr`, `sse`, et `sst`, normalement scalaires, sont maintenant des matrices $m \times m$. Si nous notons $\boldsymbol{Y}$, $\hat{\boldsymbol{U}}$, et $\hat{\boldsymbol{Y}}$ les matrices de variables dépendantes, de résidus, et de valeurs ajustées, en notation ***Ects*** `Y`, `res`, et `fit`, toutes de la forme $n \times m$, alors les définitions sont comme suit :

$$\begin{aligned} \texttt{ssr} &= \hat{\boldsymbol{U}}^\top \hat{\boldsymbol{U}}; \\ \texttt{sse} &= \hat{\boldsymbol{Y}}^\top \hat{\boldsymbol{Y}}; \quad \text{et} \\ \texttt{sst} &= \boldsymbol{Y}^\top \boldsymbol{Y}. \end{aligned}$$

La variable `R2` est transformée en vecteur $m \times 1$, les éléments successifs étant les R^2 successifs des m régressions. `errvar`, qui dans le cas univarié est simplement l'estimation de $\hat{\sigma}^2$, devient une matrice $m \times m$, définie exactement comme dans le cas univarié :

$$\texttt{errvar} = \frac{\texttt{ssr}}{n-k}.$$

Finalement, `stderr` et `student` sont, exactement comme `coef`, des matrices $k \times m$, définies de façon complètement analogue.

Quelques-unes des variables ***Ects*** gardent leur aspect ordinaire, même dans le cas multivarié. Par exemple, la matrice `XtXinv`, car les régresseurs sont justement les mêmes dans toutes les régressions. Pour la même raison, le vecteur `hat` reste inchangé par rapport à une régression simple.

Le piège ! Dans un souci de compatibilité avec la première version du logiciel, la matrice `vcov` correspond dans le cas univarié à l'expression $\hat{\sigma}^2(\boldsymbol{X}^\top \boldsymbol{X})^{-1}$. Dans le cas multivarié, $\boldsymbol{X}^\top \boldsymbol{X}$ est la même matrice pour toutes les régressions, mais $\hat{\sigma}^2$ est différent. Parce qu'il serait difficile de définir tout un vecteur de matrices, la variable `vcov`, dans le cas multivarié, correspond à la *première* régression uniquement. Pour obtenir la matrice correspondante pour la deuxième régression, par exemple, on devrait calculer comme suit :

```
mat vcov2 = errvar(2,2)*XtXinv
```

Si l'estimation multivariée est effectuée au moyen de `iv`, les choses sont très similaires. La variable `R2` n'est pas créée, parce que le R^2 n'a aucun sens dans ce contexte. Le rôle de la matrice `XtXinv` est joué par `XtPwXinv` : encore une fois cette matrice est indépendante du nombre de variables dépendantes. Et le piège y est toujours : `vcov` ne contient que la matrice de covariance estimée des paramètres estimés de la *première* régression.

EXERCICES:

Refaites les deux régressions de l'exercice précédent séparément et faites tous les calculs nécessaires à obtenir les vecteurs et matrices calculés par la procédure multivariée.

3. Autres Opérations

Il est parfois nécessaire d'effectuer des opérations matricielles plus sophistiquées que celles que nous avons considérées jusqu'ici. Dans cette section, on décrit les fonctionnalités qui sont à votre disposition dans la version 2 d'***Ects***.

* * * *

La lecture de cette section n'est pas indispensable. On peut sauter au chapitre suivant.

* * * *

D'abord, une opération assez simple. Dans le contexte d'une expression algébrique, si `A` est une matrice carrée $n \times n$, `diag(A)` est le vecteur $n \times 1$ constitué des éléments diagonaux de la matrice `A`. Même si `A` n'est pas une matrice carrée, alors `diag(A)` sera un vecteur dont le nombre de lignes égale le nombre de lignes de `A`. Les éléments éventuellement inexistants seront, comme d'habitude, remplacés par des zéros.

EXERCICES:

Soit $\boldsymbol{X}$ une matrice de régresseurs pour un échantillon de taille n pas très grande. Alors la matrice de projection orthogonale $\boldsymbol{P}_X$, de la forme $n \times n$, peut être générée par la commande

```
mat Px = X*X inv
```

en utilisant l'inverse généralisée de $\boldsymbol{X}$. Faites tourner une régression avec $\boldsymbol{X}$ comme matrice de régresseurs, et vérifiez que la série `hat` est la même que celle qui est donnée par `diag(Px)`.

On a parfois besoin du déterminant d'une matrice. Un déterminant est, bien sûr, une variable scalaire. On peut utiliser l'expression `det(A)` dans toute expression algébrique là où une variable scalaire est admissible, et l'expression `det(A)` peut également faire l'objet d'une commande `set`. Pour que le déterminant d'une matrice soit défini, il faut que la matrice soit carrée. Si on écrit `det(A)` pour une matrice `A` non carrée, le résultat est une matrice nulle.

Pour éviter des ennuis éventuels, il est préférable d'utiliser les déterminants à l'intérieur d'une commande `mat`, plutôt que les commandes `gen` et `set`. Bien qu'il existe des règles qui déterminent les résultats d'un `det` sous `gen` et `set`, elles ne sont pas simples. (Et elles ne seront pas exposées ici !)

Dans un exercice précédent, on a évoqué la notion d'une régression empilée. Normalement on utilise une telle régression pour effectuer une estimation d'un système d'équations **sans lien apparent** ; en usage courant, un **système SUR**. (De l'anglais *seemingly unrelated regressions*.) La théorie de l'estimation des systèmes SUR est exposée dans les sections 9.7 – 9.9 de DM. Selon cette théorie, on a besoin d'une matrice $\boldsymbol{\psi}$, $m \times m$ si m est le nombre d'équations comprises dans le système, telle que

$$\boldsymbol{\psi}\boldsymbol{\psi}^\top = \boldsymbol{\Sigma}^{-1},$$

où $\boldsymbol{\Sigma}$ est la matrice de covariance $m \times m$ des aléas associés aux m équations. Soit $\hat{\boldsymbol{U}}$ la matrice $n \times m$ des résidus des estimations OLS des équations prises une à une. (n = taille de l'échantillon.) L'estimation de $\boldsymbol{\Sigma}$ est alors

$$\hat{\boldsymbol{\Sigma}} = n^{-1}\hat{\boldsymbol{U}}^\top\hat{\boldsymbol{U}},$$

que l'on calcule aisément, ainsi que son inverse, $\hat{\boldsymbol{\Sigma}}^{-1}$. Normalement, il est commode d'exiger que $\boldsymbol{\psi}$ soit une matrice **triangulaire**. La fonction `uptriang` permet de calculer $\boldsymbol{\psi}$ sous la forme triangulaire supérieure, c'est-à-dire la forme suivante :

$$\boldsymbol{\psi} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ 0 & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & a_{mm} \end{bmatrix}.$$

Soit `Siginv` la notation de la matrice $\boldsymbol{\Sigma}^{-1}$. La commande qui génère $\boldsymbol{\psi}$ est alors

```
mat psi = uptriang(Siginv)
```

Exercices:

Reprenez les données du fichier `sur.dat` et faites une estimation du système (9) par la méthode SUR.

* * * *

La réponse à cette question est fournie dans le fichier `sur.ect`. Bien sûr, il est préférable de n'utiliser ce fichier qu'après avoir terminé l'exercice.

* * * *

La dernière opération que nous allons considérer ici est assez compliquée. En fait, elle est l'opération centrale dans le calcul d'une estimation OLS, et dans le calcul d'une inverse généralisée. Il s'agit de la **décomposition par valeurs singulières**, ou la **SVD**, d'après l'expression anglaise *singular value decomposition*. Il est rare que l'on doive utiliser cette opération directement, mais à l'occasion, elle est indispensable. Voici la théorie (abrégée) de cette décomposition. (L'algorithme qui effectue la SVD, et quelques éléments de la théorie, sont donnés dans la section pertinente de l'ouvrage de Press, Teukolsky, Vetterling, et Flannery (1986).)

Soit $\boldsymbol{X}$ une matrice $n \times k$, où $k \leq n$. On peut démontrer qu'il existe les matrices $\boldsymbol{U}$, $n \times k$, $\boldsymbol{W}$, $k \times k$, $\boldsymbol{V}$, $k \times k$, telles que

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{W}\boldsymbol{V}^\top, \tag{11}$$

où, en outre,

$$\boldsymbol{U}^\top\boldsymbol{U} = \mathbf{I}_k; \qquad \boldsymbol{V}^\top\boldsymbol{V} = \boldsymbol{V}\boldsymbol{V}^\top = \mathbf{I}_k,$$

et $\boldsymbol{W}$ est une matrice diagonale, dont les éléments diagonaux sont non-négatifs. En effet, ces éléments sont les racines carrées des valeurs propres de la matrice $\boldsymbol{X}^\top\boldsymbol{X}$, qui est symétrique et semi-définie positive. L'inverse généralisée de $\boldsymbol{X}$ se calcule comme suit :

$$\boldsymbol{X}^+ = \boldsymbol{V}\boldsymbol{W}^+\boldsymbol{U}^\top,$$

d'où on voit que l'intérêt de la SVD est que l'inversion (généralisée) de $\boldsymbol{W}$ est très simple : il suffit d'inverser les éléments diagonaux un à un. S'il y a des éléments diagonaux de $\boldsymbol{W}$ qui sont nuls, on posera les éléments correspondants de $\boldsymbol{W}^+$ égaux à zéro.

* * * *

Étant donné les limitations du calcul numérique sur ordinateur, une valeur « nulle » est une valeur inférieure à un seuil proche de zéro. Pour la version actuelle d'***Ects***, ce seuil est 10^{-8}.

* * * *

Soit `X` une matrice $n \times k$. Pour effectuer la décomposition (11), la commande pertinente est

```
svdcmp X
```

Les résultats de la commande ne sont pas imprimés dans le fichier de sortie. En revanche, trois matrices sont créées, ou mises à jour si elles existent déjà, nommées `SVDU`, `SVDW`, et `SVDV`. Les dimensions de ces matrices sont $n \times k$, $k \times 1$, et $k \times k$. Attention ! La matrice diagonale $\boldsymbol{W}$ est stockée sous la forme d'un *vecteur* `SVDW`, dont les composantes sont les éléments diagonaux de $\boldsymbol{W}$. Si la matrice $\boldsymbol{X}$ a plus de colonnes que de lignes, c'est-à-dire si $n < k$, elle sera transposée avant le calcul de $\boldsymbol{U}$, $\boldsymbol{W}$, et $\boldsymbol{V}$, et les résultats vérifieront

$$\boldsymbol{X}^\top = \boldsymbol{U}\boldsymbol{W}\boldsymbol{V}^\top.$$

Exercices:

Prenez les variables $\boldsymbol{y}$, $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, et $\boldsymbol{x}_3$ du fichier `ols.dat` et créez une matrice $\boldsymbol{X}$, de la forme 100×4, constituée de la constante et les trois explicatives. Recalculez les estimations paramétriques, ainsi que la matrice de covariance estimée des paramètres, sans faire appel à la commande `ols`. Recalculez aussi le vecteur `hat`, qui contient les éléments diagonaux de la matrice de projection $\boldsymbol{P}_{\boldsymbol{X}}$.

Chapitre 4

Les Estimations Non Linéaires

1. Les Régressions Non Linéaires

Ects est capable de faire des estimations non-linéaires de plusieurs sortes. Dans cette section nous verrons comment faire les estimations non-linéaires les plus simples, au moyen de la procédure des **moindres carrés non-linéaires**, ou **NLS**, acronyme qui provient de l'anglais *nonlinear least squares*. Le modèle de régression non-linéaire s'écrit de la manière suivante :

$$\boldsymbol{y} = \boldsymbol{x}(\boldsymbol{\beta}) + \boldsymbol{u}.$$

(Voir les chapitres 2 et 3 de DM.) La variable dépendante $\boldsymbol{y}$ est représentée, comme dans le cas des OLS, par un vecteur $n \times 1$, ainsi que le vecteur aléatoire $\boldsymbol{u}$. Mais la **fonction de régression**, $\boldsymbol{x}(\boldsymbol{\beta})$, n'est plus en général une fonction linéaire des paramètres $\boldsymbol{\beta}$. Dans la théorie de la régression artificielle de Gauss-Newton, la GNR, on utilise une matrice de la forme de la matrice de régresseurs dans le cadre de la régression linéaire. Cette matrice, notée $\boldsymbol{X}(\boldsymbol{\beta})$, est définie par la relation suivante :

$$\boldsymbol{X}_{ti}(\boldsymbol{\beta}) \equiv \frac{\partial x_t}{\partial \beta_i}(\boldsymbol{\beta}). \tag{12}$$

En effet, dans le cas de la régression linéaire, on a $x_t(\boldsymbol{\beta}) = \boldsymbol{X}_t\boldsymbol{\beta}$, et la matrice $\boldsymbol{X}$ de régresseurs est égale à $\boldsymbol{X}(\boldsymbol{\beta})$, quel que soit $\boldsymbol{\beta}$.

L'algorithme employé par ***Ects*** pour la minimisation de la somme des carrés des résidus d'une régression non-linéaire est exposé dans l'ouvrage de Press *et al* (1986), sous le nom de la **méthode de Marquardt**, ou de Levenberg-Marquardt. Pour sa mise en œuvre, on a besoin des éléments de la matrice $\boldsymbol{X}(\boldsymbol{\beta})$. Pour éviter des complications inutiles, la commande `nls` exige que l'utilisateur les fournisse, en même temps que la fonction de régression elle-même, bien sûr. Afin de comprendre comment on spécifie un modèle non-linéaire, regardons le fichier `nls.ect`. Il est court ; voici son contenu :

```
sample 1 50
read nls.dat y x1 x2
ols y c x1 x2
set alpha = coef(1)
set beta = coef(2)
nls y = alpha + beta*x1 + (1/beta)*x2
deriv alpha = 1
deriv beta = x1 - x2/(beta*beta)
end
gen e = y - alpha - beta*x1 - (1/beta)*x2
gen ralpha = 1
gen rbeta = x1 - x2/(beta*beta)
ols e ralpha rbeta
quit
```

Le fichier `nls.dat` doit contenir au moins 50 observations sur au moins 3 variables. (C'est le cas du fichier fourni avec ***Ects***.) Le modèle que l'on souhaite estimer par les moindres carrés non-linéaires est :

$$\boldsymbol{y} = \alpha + \beta \boldsymbol{x}_1 + \frac{1}{\beta}\boldsymbol{x}_2 + \boldsymbol{u}. \tag{13}$$

La fonction de régression est donc $\alpha + \beta \boldsymbol{x}_1 + (1/\beta)\boldsymbol{x}_2$, qui dépend de deux paramètres, α et β. La régression (13) peut être considérée comme une régression *linéaire* soumise à une contrainte non-linéaire : si dans la régression linéaire

$$\boldsymbol{y} = \alpha + \gamma_1 \boldsymbol{x}_1 + \gamma_2 \boldsymbol{x}_2 + \boldsymbol{u} \tag{14}$$

on pose $\gamma_1\gamma_2 = 1$, on retrouve (13). Il convient alors au préalable de faire une estimation de (14) : c'est le sens de la commande

```
ols y c x1 x2
```

Ensuite l'estimation non-linéaire. Avant de pouvoir spécifier le modèle (13) à ***Ects***, il faut *obligatoirement* donner à l'algorithme qui effectue la minimisation de la somme des carrés des résidus un point de départ. Autrement dit, les paramètres α et β de la fonction de régression doivent être définis, par `set`, et on doit leur affecter des valeurs appropriées. Ici, on a choisi un point de départ tout à fait naturel : on a affecté à α et à β les estimations fournies par la régression linéaire.

* * * *

Rappel : `coef` est l'une des séries mises à jour par ***Ects*** après chaque régression linéaire : voir les sections 1.2, 2.3, et 3.2.

* * * *

C'est maintenant le moment d'utiliser `nls`. La commande comporte plusieurs lignes ; la voici :

```
nls y = alpha + beta*x1 + (1/beta)*x2
```

```
deriv alpha = 1
deriv beta = x1 - x2/(beta*beta)
end
```

La première ligne sert à spécifier la variable dépendante et la fonction de régression. À la différence de la commande `ols`, il faut séparer ces deux éléments par un =. Le reste de la ligne obéit à la même syntaxe que celle employée par `gen`. En effet l'appareil digestif de ***Ects*** n'a qu'un algorithme pour l'analyse des expressions algébriques, bien que cet algorithme puisse être adapté aux besoins des commandes `set` et `mat` aussi bien qu'à ceux de `gen`. La raison pour laquelle c'est la version adaptée à `gen` que l'on utilise ici est simple : l'expression qui suit la première partie de la ligne de commande, c'est-à-dire l'expression qui suit `nls y =` , sera interprétée comme une *série*, parce qu'à chaque observation il correspond une fonction de régression.

Après, pour chacun des paramètres du modèle, il faut une ligne qui commence par `deriv`, suivi d'abord par le nom d'un des paramètres du modèle, ensuite d'un signe d'égalité =, et, finalement, d'une expression algébrique égale à la dérivée partielle de la fonction de régression par rapport au paramètre en question. Toutes ces dérivées partielles, comme la fonction de régression, seront évaluées à la manière de `gen`, comme des séries. L'ordre des paramètres n'a pas d'importance essentielle ; il sert toutefois à établir l'ordre des paramètres dans le listing. Après la liste des expressions algébriques des dérivées, il faut encore une ligne, qui contient le mot `end` (la *fin* en anglais). Pourquoi cette ligne terminale ? Parce que, à part le fait qu'elle enlève quelques difficultés auxquelles le programmeur devrait autrement faire face, cette convention permet d'incorporer dans la fonction de régression des variables scalaires dont les valeurs resteront inchangées. Autrement dit, la minimisation de la somme des carrés des résidus ne sera effectuée que par rapport aux paramètres qui figurent dans la liste de dérivées.

Le dernier groupe de commandes implémente la GNR[9] linéaire associée au modèle estimé par les NLS. Si tout a été fait correctement, les paramètres estimés seront nuls, aux erreurs d'arrondi près. À noter qu'après une estimation par `nls`, les paramètres estimés, ici α et β, ont les valeurs données par l'estimation ; les anciennes valeurs, celles du point de départ, sont perdues.

EXERCICES:

Le fichier `nlsols.ect` reprend les commandes de `nls.ect`, mais avec sauvegarde des paramètres du point de départ. À la fin de la procédure non-linéaire, une nouvelle régression est spécifiée par `nls`. Elle est parfaitement équivalente à la régression linéaire du début du programme. On voit qu'il est possible d'effectuer des régressions linéaires par `nls`. Modifiez vers le haut la valeur de la variable `TOL` et notez les conséquences à l'égard de la GNR et de la régression linéaire effectuée par `nls`.

Étudiez le fichier `arnls.ect`. On utilise les mêmes données que celles qui ont servi pour `ar.ect` (voir les exercices de la section 2.1). Cette fois-ci, l'estimation d'un

[9] Régression de Gauss-Newton

modèle à erreurs AR(1) est effectuée directement par une procédure non-linéaire. Essayez de bien comprendre le lien entre les différentes procédures des deux fichiers.

Reprenez le modèle (1) et les données de `ols.dat`. Imposez et ensuite testez la restriction non-linéaire $2\beta_1\beta_3 + \beta_2 = 0$.

Les variables créées ou mises à jour par `nls` sont presque identiques à celles qui sont produites par `ols`. Encore une fois, `coef` est un vecteur qui contient les paramètres estimés, dans l'ordre des dérivées partielles données par les lignes `deriv`. Il est à noter que la commande `nls` ne peut accepter qu'une seule variable dépendante, à la différence d'`ols` et d'`iv`. La conséquence en est que les variables `ssr`, `sse`, `sst`, `R2`, et `errvar` sont des scalaires, tandis que `fit`, `res`, `stderr`, et `student` sont des séries. La matrice `vcov` est sans ambiguïté la matrice de covariance estimée des paramètres estimés, c'est-à-dire, la matrice $\hat{\sigma}^2(\hat{\boldsymbol{X}}^\top\hat{\boldsymbol{X}})^{-1}$, où $\hat{\sigma}^2 =$ `errvar`, et $(\hat{\boldsymbol{X}}^\top\hat{\boldsymbol{X}})^{-1} =$ `XtXinv`.

* * * *

Rappel : $\hat{\boldsymbol{X}} \equiv \boldsymbol{X}(\hat{\boldsymbol{\beta}})$.

* * * *

Il est clair à partir de toutes ces définitions que les estimations des variances, écarts-type, et Students, sont *asymptotiques*. Les scalaires `nobs` et `nreg` sont définies comme pour un `ols`, mais dans le contexte non-linéaire `nreg` correspond au nombre de paramètres plutôt qu'au nombre d'explicatives.

Finalement, il y a un nouveau scalaire, `niter`, qui n'est défini que pour les estimations non-linéaires. Sa valeur est le nombre d'itérations qui ont été nécessaires à la convergence de l'algorithme de minimisation.

Si le nombre de paramètres est important, il se peut qu'on ait besoin d'un nombre d'itérations assez élevé. Le temps de calcul de tout algorithme de minimisation non-linéaire est déjà élevé par rapport à une estimation linéaire. En conséquence, ***Ects*** s'arrête après avoir effectué un nombre d'itérations spécifié par la variable interne `maxiter`. La valeur par défaut de cette variable est 20, mais on a la possibilité de changer cette valeur au moyen d'un simple `set`. Par exemple, si on veut permettre à ***Ects*** de tourner jusqu'à 100 itérations, on fait

```
set maxiter = 100
```

avant d'entamer la procédure d'estimation non-linéaire.

Après avoir tourné `maxiter` fois, ***Ects*** demande à l'utilisateur s'il veut continuer. En effet, ***Ects*** pose la question suivante :

```
n iterations without convergence.  Continue (y/n) ?
```

où n est le nombre d'itérations effectuées au moment où la question est posée. Si on répond `y` (oui = *yes*), les itérations continuent, soit jusqu'à la convergence de l'algorithme, soit jusqu'à ce que encore `maxiter` itérations soient achevées.

EXERCICES:

Choisissez l'une des estimations non-linéaires que vous avez faites précédemment, et posez `maxiter` égal à une valeur inférieure au nombre d'itérations annoncé par ***Ects***. Vous verrez ainsi le déroulement du processus décrit ci-dessus.

2. Le Maximum de Vraisemblance

Les principales différences entre l'estimation par la **méthode du maximum de vraisemblance** et l'estimation par les NLS sont cachées à l'intérieur d'***Ects***. Les commandes qui sont données par l'utilisateur pour demander les deux sortes d'estimation sont très similaires. Le nom de la commande est quand-même (forcément !) différent : pour entamer une estimation par maximum de vraisemblance, on utilise la commande `ml`. Malgré son nom (`ml` = *maximum likelihood* = maximum de vraisemblance), `ml` peut servir pour d'autres sortes d'estimations non-linéaires. En effet, la commande donne accès à l'algorithme utilisé par ***Ects*** pour la maximisation de fonctions de plusieurs variables. L'algorithme se trouve encore une fois dans l'ouvrage de Press *et al* (1986), sous le nom de l'algorithme de **Davidon-Fletcher-Powell**[10] ou simplement l'algorithme **DFP**.

Pour pouvoir employer `ml`, il faut qu'un estimateur soit défini par la maximisation ou la minimisation d'une **fonction critère**, qui s'exprime comme la somme d'un ensemble de **contributions**. Il est démontré dans le chapitre 8 de DM que l'estimateur ML vérifie cette condition. Pour la théorie générale des ***M*-estimateurs**, voir le chapitre 17 de DM.

Comme d'habitude, le meilleur moyen de comprendre comment `ml` fonctionne est de regarder attentivement un exemple particulier. Le fichier `ml.ect` contient un tel exemple. Il est banal : il s'agit encore une fois d'une régression linéaire, pour qu'il n'y ait pas de difficultés dans la formulation du modèle :

$$\boldsymbol{y} = \alpha + \beta_1 \boldsymbol{x}_1 + \beta_2 \boldsymbol{x}_2 + \boldsymbol{u}.$$

Comme pour `nls`, il est obligatoire d'affecter des valeurs de départ à tous les paramètres du modèle par la commande `set`. Ensuite, la commande `ml` commence. La première ligne contient la formulation de la contribution de chaque observation à la **log-vraisemblance**. Cette ligne est assez longue : on verra après comment on pourrait la raccourcir.

```
ml - log(sig) - (1/(2*sig*sig))*(y - alpha -
beta1*x1 - beta2*x2)^2
```

[10] Attention à l'orthographe du premier nom !

La suite de la commande est tout à fait analogue à ce qu'il faut faire pour `nls`. On doit préciser les dérivées partielles des contributions par rapport à chacun des paramètres du modèle. La fin de la liste de dérivées est encore une fois signalée par `end`. Dans le contexte d'une estimation ML, le paramètre `sig`, ou σ, l'écart-type des aléas, joue le même rôle que tout autre paramètre.

Il est connu que la méthode du maximum de vraisemblance est équivalente aux moindres carrés pour les modèles de régression à aléas normaux. On peut faire une deuxième estimation en tenant compte de ce fait. La deuxième commande `ml` du fichier :

```
ml -(y - alpha - beta1*x1 - beta2*x2)^2
```

met en œuvre cette estimation – observez que σ n'intervient plus ni dans la contribution ni dans les dérivées.

Entre les deux ensembles de commandes qui définissent les deux procédures d'estimation, vous trouverez une nouvelle commande, `beep`. Parce qu'une estimation non-linéaire peut durer longtemps, il est parfois utile de pouvoir signaler à l'utilisateur qu'on est arrivé à la fin. L'effet de la commande `beep` est de faire un signe audible.

Mais la deuxième estimation prendra moins de temps que la première, à cause justement de la suppression de la variable `sig`. Si on regarde bien, on voit que la fonction qu'***Ects*** a eu à maximiser cette fois-ci est simplement, au signe près, la somme des carrés des résidus de la régression. En principe, on devrait trouver les mêmes estimations paramétriques par l'utilisation de la commande `nls`, et, parce que la régression est linéaire, de la commande `ols`. La suite du fichier procède à la vérification de ce fait.

Pour comprendre les résultats d'une estimation faite par `ml`, nous pouvons maintenant analyser le contenu (partiel) du fichier de sortie. Voici ce qui est produit par la première commande :

```
Maximising a Sum of Contributions:

Number of iterations = 10

Parameter    Parameter estimate   Standard error    T statistic

alpha        -22.328306           22.822253         -0.978357
beta1        3.195281             0.324747          9.839279
beta2        0.604578             0.474059          1.275323
sig          56.183999            5.519618          10.178965

Number of observations = 50
Number of estimated parameters = 4
Maximised value of criterion function = -226.431605
```

```
Estimated covariance matrix (from numerical Hessian):

520.855210   -5.421844    3.753240   -0.494462
-5.421844    0.105461    -0.133966   0.007337
3.753240     -0.133966   0.224732    -0.007864
-0.494462    0.007337    -0.007864   30.466181

Estimate from Outer Product of the Gradient:

850.095786   -9.996039    9.532800    34.664047
-9.996039    0.167459     -0.208499   -0.268544
9.532800     -0.208499    0.306705    0.368863
34.664047    -0.268544    0.368863    43.032181
```

Le premier tableau ressemble fort à ce qui est donné par `nls` ou même `ols`. On a les paramètres estimés, les écarts-type estimés, et les Students, où, comme c'est toujours le cas dans le contexte d'une estimation non-linéaire, les deux derniers n'ont qu'une justification asymptotique.

Ensuite, à part les scalaires à interprétation immédiate, on trouve

```
Maximised value of criterion function = -226.4316049
```

La grandeur qui est imprimée ici est la fonction critère (ici la log-vraisemblance) évaluée en $\hat{\boldsymbol{\theta}}$, l'estimation paramétrique qu'on vient d'obtenir. Autrement dit, c'est la valeur maximale de la fonction critère.

* * * *

La fonction qu'on a maximisée n'est pas exactement la log-vraisemblance du modèle. Pour être tout à fait correct, il faudrait rajouter un terme $-(1/2)\log(2\pi)$ à chaque contribution. Pour le calcul des statistiques LR, ce n'est pas grave, car ces termes sont identiques dans les deux fonctions de log-vraisemblance.

* * * *

Viennent ensuite deux estimations différentes de la matrice de covariance des paramètres estimés.. La seconde est la plus facile à comprendre. Soit $\boldsymbol{G}(\boldsymbol{\theta})$ la **matrice CG** associée à un modèle (voir la section 8.2 de DM pour les détails), c'est-à-dire, soit

$$G_{ti}(\boldsymbol{\theta}) = \frac{\partial \ell_t}{\partial \theta_i}(\boldsymbol{\theta}),$$

où ℓ_t est la contribution à la log-vraisemblance faite par l'observation t, et où l'on note $\boldsymbol{\theta}$ les paramètres du modèle. **L'estimateur OPG** de la matrice de covariance de $\hat{\boldsymbol{\theta}}$ se définit comme suit :

$$\mathrm{Var}_{\mathrm{OPG}}(\hat{\boldsymbol{\theta}}) = (\hat{\boldsymbol{G}}^\top \hat{\boldsymbol{G}})^{-1},$$

où $\hat{\boldsymbol{G}} \equiv \boldsymbol{G}(\hat{\boldsymbol{\theta}})$. Cet estimateur converge vers l'espérance du **produit extérieur du gradient** (en anglais, on a *outer product of the gradient*) mais sa vitesse de

convergence laisse à désirer. En effet, dans le cas actuel, on peut voir que tous les éléments de l'estimateur OPG sont plus grands que ceux de l'estimateur OLS.

L'autre estimation de la matrice de covariance s'annonce de la manière suivante :

```
Estimated covariance matrix (from numerical Hessian)
```

Malheureusement, ce n'est pas tout à fait vrai. La Hessienne empirique est définie par la relation

$$H_{ij}(\boldsymbol{\theta}) = \frac{\partial^2 \ell}{\partial \theta_i \partial \theta_j}(\boldsymbol{\theta}).$$

Le calcul du membre de droite de cette équation exige que les dérivées secondes de la log-vraisemblance soit connues. ***Ects*** est très faible en calcul différentiel : il a été nécessaire de lui fournir même les dérivées premières. L'estimateur « Hessienne » donné par ***Ects*** n'est donc qu'une approximation numérique à la vraie Hessienne empirique. Pour les détails de l'approximation, voir Press *et al* (1986). On laisse à l'utilisateur la tâche du calcul correct de la Hessienne.

* * * *

Ceci dit, il serait encore mieux de calculer la vraie **matrice d'information**. À cette fin, on peut se servir de l'espérance soit du produit extérieur du gradient, soit de la Hessienne. Voir la section 8.7 de DM.

* * * *

La commande `ml` crée ou met à jour des variables tout comme les autres commandes qui effectuent une estimation. Comme d'habitude, `coef` est le vecteur de paramètres estimés, et comme c'est fait par `nls`, les variables qui représentent les paramètres du modèle sont changées par `ml` – les valeurs qui ont servi de point de départ sont perdues. `stderr` et `student` ont leur interprétation habituelle, ainsi que `nobs`, `nreg`, et `niter`. Les deux matrices de covariance estimées se trouvent dans les variables `invhess` et `invOPG`. La matrice CG aussi peut être récupérée dans `CG`. Finalement, la série `lt` contient le vecteur de contributions à la fonction critère, évaluées en $\hat{\boldsymbol{\theta}}$, et la variable scalaire `lhat` est la somme de ces contributions, identique à la `Maximised value of criterion function`.

* * * *

La variable `CG` est également mise à jour par la commande `nls`. Son contenu est la matrice $\boldsymbol{X}(\boldsymbol{\beta})$ définie dans (12).

* * * *

La longueur des expressions des contributions et de leurs dérivées peut poser un problème. On a vu dans l'exemple ci-dessus que même un simple modèle de régression engendre des expressions assez lourdes. Une application plus importante peut devenir rapidement ingérable. Pour éviter de telles difficultés, il existe une commande `def`, dont l'effet est très similaire à ce qu'on appelle souvent en informatique une **macro**. Pour voir clairement comment ça marche,

reformulons notre modèle de régression. Les commandes suivantes ont le même effet que celles qui ont été utilisées pour la première estimation ML.

```
def u = y - alpha - beta1*x1 - beta2*x2
def sig2 = sig*sig
ml - log(sig) - (1/(2*sig2))*u^2
deriv alpha = u/sig2
deriv beta1 = x1*u/sig2
deriv beta2 = x2*u/sig2
deriv sig = -(1/sig)*(1 - u^2/sig2)
end
```

On voit qu'il est possible de remplacer une longue expression par un seul symbole. Mais il est déjà possible de faire autant au moyen d'une commande `gen`, non ? Pas de la même manière. Si, à la place des commandes `def` on utilisait des commandes `gen`, la fonction critère ne dépendrait des paramètres du modèle qu'au travers du `log(sig)` du premier terme. Les dérivées par rapport à `alpha`, `beta1`, et `beta2` ne dépendrait plus du tout des paramètres, et la dérivée par rapport à `sig` dépendrait uniquement du `sig` au dénominateur. L'algorithme de maximisation a beau changer les valeurs des paramètres, la fonction critère ne bougera qu'à cause des changements de `sig`.

Qu'est-ce qui fait la différence si on utilise `def` ? Cette commande ne crée aucune variable nouvelle ; elle sert simplement à mettre dans la mémoire de l'ordinateur le *texte* qui suit le signe d'égalité, et d'affecter le nom (ici, `u` ou `sig2`) à ce bout de texte. Mais ensuite, chaque fois que l'ordinateur a à évaluer une expression qui contient le nom, il cherche le texte correspondant, et l'expression sera évaluée en conséquence. Ceci fait que la fonction critère, ainsi que les dérivées partielles, sont évaluées chaque fois correctement : les valeurs courantes des paramètres sont utilisées, telles qu'elles sont mises à jour par l'algorithme de maximisation.

Même si les expressions ne sont pas trop longues, il est souvent souhaitable d'utiliser des expressions définies par `def` pour la clarté de l'écriture, et, par conséquent, la lecture, du programme.

* * * *

Bien sûr, un utilisateur « petit malin » peut affecter le même nom par un `gen` et par un `def`. Dans ce cas, les résultats des deux commandes restent dans la mémoire, d'une part une série ou une matrice, d'autre part, une chaîne de texte. Mais c'est la variable, série ou matrice, qui sera utilisée dans les expressions ultérieures.

* * * *

Que se passe-t-il si la mémoire de l'ordinateur est insuffisante ? La réponse dépend de la version du fichier exécutable que l'on utilise. Si c'est la version de base, `ects86.exe`, il reste possible que la machine se plante.[11] Normalement,

[11] Très peu souhaitable ! J'ai fait mon possible, et ça arrive moins souvent que dans le temps.

un message d'erreur s'affiche, et le programme s'arrête avec retour à DOS. Si une telle chose risque de se produire après une suite d'estimations et de calculs précieux, il y a une précaution à prendre. Si on lance la commande `del`, suivi d'une liste de variables (scalaires, séries, et matrices) ou de macros (définis par `def`), la place en mémoire vive occupée par ces variables ou ces macros sera libérée, et les noms qui leur appartenait seront effacés du tableau des symboles.

Malgré la possibilité de faire un `del`, il est souvent souhaitable de voir directement combien de mémoire vive reste à la disposition du logiciel. Pour obtenir cette information, on utilise la commande `mem`. *Uniquement dans le cas de la version* `ects86.exe`, cette commande fait afficher à l'écran le nombre d'octets encore disponibles.[12] L'information sera également inscrite dans le fichier de sortie.

* * * *

La mémoire conventionnelle d'un PC est de 640 Ko (kilo-octets), soit $640 \times 1024 = 655.360$ octets. De ce total, il faut soustraire la place occupée par le système d'exploitation (DOS), par tous les utilitaires « résidents » en mémoire vive, et par le logiciel ***Ects*** lui-même. On peut lancer un `mem` au début du programme, avant de faire autre chose, pour connaître le nombre d'octets disponibles pour contenir les variables et les macros.

* * * *

La version « grande » du logiciel, `ects.exe`, répond tout gentiment à une commande `mem` par la phrase suivante :

```
Unused command with Big version of Ects
```

en français, cette commande est inutile dans la version grande d'***Ects***. En effet, parce que cette version a accès à la **mémoire étendue** de l'ordinateur, les insuffisances de mémoire sont beaucoup moins fréquentes. En plus, le **DOS-extender** qui fait partie de la grande version peut, en certains cas, économiser la mémoire vive par un processus de stockage de données temporaire sur le disque dur.

3. La Méthode des Moments Généralisée

La **méthode des moments généralisée** est considérée comme une méthode sophistiquée. Les détails de la méthode, qui sont assez techniques, sont exposés dans le chapitre 17 de DM. Toutefois, il y a une sorte d'estimation, dont les principes sont relativement élémentaires, qui ne peut être effectuée (en informatique) que par cette méthode.

Dans le chapitre 7 de DM, l'on considère le modèle de régression non-linéaire estimé par variables instrumentales. Le modèle de départ s'écrit, comme

[12] Note de la version 4 : La commande `mem` est maintenant totalement supprimée.

d'habitude,

$$\boldsymbol{y} = \boldsymbol{x}(\boldsymbol{\beta}) + \boldsymbol{u},$$

et l'on note $\boldsymbol{W}$ la matrice d'instruments utilisés. L'estimateur de ce modèle est défini par la minimisation de la fonction critère

$$\big(\boldsymbol{y} - \boldsymbol{x}(\boldsymbol{\beta})\big)^{\top} \boldsymbol{P}_W \big(\boldsymbol{y} - \boldsymbol{x}(\boldsymbol{\beta})\big).$$

* * * *

On vérifie tout de suite que, dans le cas linéaire, le résultat de cette minimisation est l'estimateur IV habituel.

* * * *

Or, cette fonction critère ne s'exprime pas comme une somme de contributions, à la différence de la log-vraisemblance ; elle est plutôt une forme quadratique. La méthode des moments généralisée passe justement par la minimisation d'une forme quadratique.

Pour illustrer le fonctionnement de la commande `gmm`, voyons le fichier `ivnls.ect`. (`gmm` selon l'expression anglaise *Generalised Method of Moments*.) On effectue d'abord une estimation non-linéaire par `nls` du modèle suivant :

$$\boldsymbol{y} = \alpha + \beta \boldsymbol{x}_1 + \frac{1}{\beta} \boldsymbol{x}_2 + \boldsymbol{u}. \tag{15}$$

Ensuite, le modèle linéaire dont (15) est une version contrainte, à savoir

$$\boldsymbol{y} = \alpha + \beta \boldsymbol{x}_1 + \gamma \boldsymbol{x}_2 + \boldsymbol{u},$$

fait l'objet d'une estimation par variables instrumentales (c'est-à-dire, par `iv`), où les instruments sont la constante, $\boldsymbol{x}_1$, et une nouvelle variable, $\boldsymbol{w}$.

L'idée de la suite du fichier est, d'abord, de refaire l'estimation IV utilisant `gmm` à la place de `iv`, et ensuite d'imposer la contrainte $\gamma = 1/\beta$, ce qui donne un modèle non-linéaire, que l'on ne peut estimer autrement que par `gmm`. Avant de passer aux commandes `gmm`, il y a quelques manipulations à effectuer :

```
gen iota = 1
gen W = colcat(iota, x1, w)
def resid = y - b0*iota - b1*x1 - b2*x2
set b0 = alpha
set b1 = beta
set b2 = 1/beta
mat WtWinv = (W'*W)inv
```

On utilise comme point de départ de la première estimation GMM les paramètres estimés de la régression non-linéaire sans variables instrumentales. En plus, on crée une macro `resid`, qui facilitera l'écriture de la fonction critère. Finalement, on crée la matrice $(\boldsymbol{W}^{\top}\boldsymbol{W})^{-1}$, où $\boldsymbol{W}$ est la matrice de tous les instruments.

Ensuite, l'estimation proprement dite :

```
gmm resid'*W*WtWinv*W'*resid
deriv b0 = -2*iota'*W*WtWinv*W'*resid
deriv b1 = -2*x1'*W*WtWinv*W'*resid
deriv b2 = -2*x2'*W*WtWinv*W'*resid
end
```

On voit que la syntaxe est encore une fois identique à celle des commandes `nls` et `ml`. Après la définition de la fonction critère, il faut donner les dérivées partielles de cette fonction par rapport à l'ensemble des paramètres à estimer, et ensuite `end`.

La différence cruciale par rapport à une commande `ml` est que les expressions algébriques sont lues de la même manière que les expressions dans une commande `mat`, plutôt qu'une commande `gen`. Une autre petite différence, sans importance majeure, mais capable de provoquer des ennuis considérables si on l'oublie, c'est que la fonction critère est *minimisée*. À part ces différences, l'algorithme qu'utilise `gmm` est le même qu'utilise `ml`.

On voit que les estimations paramétriques données par cette première estimation GMM sont identiques à celles fournies par la commande `iv` précédente. En revanche, outre ces estimations, `gmm` n'a pas grand-chose à dire. La raison en est que, dans le cas général, il n'y aucun lien précis entre la fonction critère et la matrice de covariance des paramètres estimés. Il est donc impossible de calculer les écarts-type estimés, Students, *etc.* Toutefois, l'approximation à la Hessienne de la fonction critère est imprimée, car elle peut servir dans des calculs ultérieurs aboutissant à une estimation de la matrice de covariance. Une autre information souvent utile est la valeur minimale de la fonction critère : cette valeur est également imprimée dans le fichier de sortie.

Voici les résultats de la première commande `gmm` dans `ivnls.ect` :

```
Minimising a Criterion Function:

Number of iterations = 5

b0 = -16.6437115
b1 = 0.5864273
b2 = 2.7326305

Number of estimated parameters = 3
Minimised value of criterion function = 0.0000000

Inverse of Numerical Hessian of Criterion function:

 0.1002287      -0.0012027      0.0010858
-0.0012027       0.0000213     -0.0000266
 0.0010858      -0.0000266      0.0000408
```

EXERCICES:

La valeur minimale de la fonction critère, que l'on peut lire dans le tableau à la ligne

```
Minimised value of criterion function = 0.0000000
```

est nulle. Pourquoi ?

On passe après à l'estimation du modèle non-linéaire par variables instrumentales. Il reste très peu de choses à faire ; en fait, il suffit de redéfinir les résidus et la dérivée de la fonction critère par rapport à β :

```
def resid = y - b0 - b1*x1 - (1/b1)*x2
gmm resid'*W*WtWinv*W'*resid
deriv b0 = -2*iota'*W*WtWinv*W'*resid
deriv b1 = -2*(x1 - x2/(b1^2))'*W*WtWinv*W'*resid
end
```

EXERCICES:

Calculez une estimation correcte de la matrice de covariance des paramètres `b0` et `b1` du modèle ci-dessus.

Comme toute autre commande qui effectue une estimation, `gmm` crée ou met à jour des variables. Il y en a beaucoup moins que d'habitude, à cause de l'impossibilité d'avoir une estimation directe de la matrice de covariance. `coef` contient les paramètres estimés, et `invhess` contient l'approximation à la Hessienne. Les variables scalaires `nreg` et `niter` sont définies par `gmm` comme par `ml`. Il n'y a qu'une seule autre variable créée par `gmm`. Elle contient la valeur minimale de la fonction critère, sous le nom `crit`.

Chapitre 5

Le Mode Interactif; Gestion de Fichiers

1. Les Fichiers de Commande et de Sortie

Tous les programmes que nous avons considérés jusqu'ici ont été transmis à ***Ects*** au moyen d'un fichier de commandes. Cette façon de travailler est souvent très pratique. Les gens peuvent échanger de tels fichiers ; un fichier ASCII est facile à modifier s'il contient des erreurs,.... Mais il se peut que l'on souhaite faire un petit calcul rapide, sans devoir créer un fichier de commandes. Ou bien on pourrait vouloir arrêter le déroulement d'un programme à mi-chemin, pour contrôler, et éventuellement modifier, les valeurs des variables. C'est pour ce genre de raison qu'il existe le **mode interactif**.

On a vu précédemment (à la section 1.1) que, si on lance ***Ects*** sans le nom d'un fichier de commandes, on se trouve dans le mode interactif. Un signe s'affiche, comme suit

```
>
```

et le curseur se positionne juste après. Si on se trompe, il suffit de taper `quit`, et on se trouve de nouveau sous DOS. Mais on peut taper n'importe quelle autre commande ***Ects***, et elle sera exécutée.

EXERCICES:

Lancez ***Ects*** en mode interactif et tapez les quatre commandes du fichier `ols.ect` pour en voir le résultat.

En mode interactif, il n'y a normalement pas de fichier de sortie. En effet, les résultats de toutes les commandes, qui, en mode non-interactif, feraient imprimer ces résultats dans le fichier de sortie, sont affichés à l'écran. Pour certaines utilisations, c'est idéal. Si on a laissé sa calculatrice à la maison, par exemple, et si on doit obligatoirement faire des calculs numériques trop lourds à faire à la main, ***Ects*** peut être d'une très grande utilité. Mais pour d'autres opérations, et en particulier, les estimations économétriques, pour lesquelles les résultats sont souvent volumineux, il est préférable de créer un fichier de sortie que l'on peut soit visualiser, soit imprimer, par la suite, même si on veut entrer les commandes directement du clavier.

La commande `out` sert à créer un fichier de sortie. En mode interactif ou ailleurs, une commande de la forme

```
out resultat.out
```

crée un fichier DOS dont le nom est `resultat.out` (si un fichier de ce nom existe déjà, il sera écrasé), et, jusqu'à nouvel ordre, c'est `resultat.out` qui sera utilisé comme fichier de sortie.

Au moment où ***Ects*** est lancé, le choix entre le mode interactif et le mode non-interactif, que l'on appelle parfois le mode « batch » (un *batch* en anglais est un *lot* en français – on fait allusion au *lot* de commandes à exécuter dans un fichier de commandes) se fait selon si ou non le mot `ects` est suivi sur la ligne de commande DOS du nom d'un fichier de commandes

* * * *

Rappel : si l'extension du fichier de commandes est `.ect`, il suffit de préciser le nom du fichier, sans l'extension – voir la Section 1.1.

* * * *

Si le mode interactif est choisi, il n'y pas de fichier de sortie si la commande `out` n'est pas utilisée, et les résultats s'affichent à l'écran. Si le nom d'un fichier de commandes est fourni, le fichier de sortie porte le même nom que le fichier de commandes, et l'extension `.out`.

Si le nom du fichier de sortie choisi automatiquement par ***Ects*** ne convient pas, on a la possibilité de le préciser ainsi que le nom du fichier de commandes sur la ligne de commande DOS. Cette ligne de commande prend alors la forme

`ects` ⟨*fichier de commandes*⟩ ⟨*fichier de sortie*⟩

L'effet est comme si la première commande du ⟨*fichier de commandes*⟩ était

`out` ⟨*fichier de sortie*⟩

On peut maintenant formuler la règle générale : si la ligne de commande DOS ne contient que le mot `ects`, on sera en mode interactif, sans fichier de sortie ; si le nom d'un fichier est fourni, ce fichier sert de fichier de commandes ; si les noms de deux fichiers sont fournis, le premier nom est celui du fichier de commandes, le second celui du fichier de sortie. Si l'un ou l'autre nom ne correspond pas à un fichier accessible au logiciel, un message d'erreur sera affiché et ***Ects*** s'arrêtera.

2. Les Contextes de Travail

Pour bien comprendre la nature des autres commandes ***Ects*** qui permettent de gérer les entrées et les sorties, il convient de savoir qu'à l'intérieur d'***Ects*** il y a en chaque instant un **contexte** de travail, qui comporte quatre éléments. Ces éléments sont 1°, le fichier de commandes, autrement dit le fichier d'entrée, 2°, le fichier de sortie, 3°, le fichier d'annonce, et 4°, une variable binaire dont la valeur est 1 si on est en mode interactif, et 0 sinon.

Par défaut, le **fichier d'annonce** est l'écran.

* * * *

DOS peut gérer plusieurs sortes de fichiers, non seulement les fichiers sur disque. L'écran, le clavier, l'imprimante, tous peuvent être considérés par DOS comme des fichiers.

* * * *

Quel que soit le fichier de sortie, c'est sur le fichier d'annonce que seront imprimé les rappels des commandes en cours et les messages d'erreur. C'est pourquoi, chaque fois que l'on fait tourner un programme ***Ects***, on voit à l'écran le déroulement des commandes successives. Si le fichier d'annonce n'est pas l'écran, il n'existe qu'une autre possibilité, à savoir le fichier « nul ». Pour DOS, le fichier `nul` est un fichier qui sert de poubelle. Si on écrit sur ce fichier, rien ne se passe, si on lit de ce fichier, ce que l'on lit est justement nul. L'existence d'un tel fichier est souvent très utile. Par exemple, on peut ne pas vouloir que le déroulement des commandes s'affiche à l'écran. Si un programme est très long, et s'il comporte beaucoup de commandes, le temps de calcul sera sensiblement diminué si on supprime cet affichage.

La commande pertinente est la commande `noecho`. Il suffit d'insérer cette commande dans un fichier de commandes pour que le déroulement cesse. Seuls seront affichés les vrais messages d'erreur. L'effet de `noecho` est annulé par la commande inverse, `echo`. En mode interactif, l'« écho » est supprimé. S'il faut de toute manière taper les commandes à la main, il est inutile de les réafficher tout de suite après. Donc, les commandes `echo` et `noecho` n'ont aucun effet visible en mode interactif. Toutefois, si on fait `noecho` en mode interactif et si on revient par la suite en mode batch, l'effet de la commande persiste.

Pendant l'exécution par ***Ects*** d'un fichier de commandes, il y a écriture sur deux fichiers, le fichier de sortie et le fichier d'annonce. Il est possible de supprimer les écritures sur le fichier de sortie par la commande `silent`. L'effet de cette commande est similaire à celui de la commande

```
out nul
```

En effet, tout ce qui serait normalement destiné au fichier de sortie est jeté à la poubelle. Pour annuler une commande `silent` on peut utiliser la commande `restore`, dont l'effet est similaire à celui de la commande

`out` ⟨*fichier de sortie*⟩

où ⟨*fichier de sortie*⟩ est le nom d'un fichier de sortie autre que `nul`.

Il n'est que rarement que les différences entre `silent` suivi de `restore`, d'une part, et `out nul` suivi d'une autre commande `out`, d'autre part, ont une importance quelconque. Mais il est à remarquer que pendant la durée de l'effet d'un `silent`, le nom de l'ancien fichier de sortie est sauvé, pour être restauré après un `restore` éventuel, ce qui n'est pas le cas si l'on utilise `out`. En particulier, on devrait éviter de faire un `out nul` suivi d'un `restore`: le fichier restauré serait simplement `nul`!

On a évoqué plus haut la possibilité d'interrompre le déroulement d'un fichier de commandes pour effectuer des contrôles ou des modifications. Ceci se

fait par un passage en mode interactif. Pour obtenir une telle interruption d'un fichier de commandes, il suffit d'insérer dans le fichier de commandes, à l'endroit où l'on veut qu'il s'arrête, la commande `interact`. Le contexte existant est sauvé, et on passe en mode interactif. Même si, dans ce contexte, le fichier de sortie était le fichier `nul`, le mode interactif est lancé normalement. Si jamais on voulait détourner les résultats obtenus en mode interactif vers la poubelle, il serait nécessaire de lancer un nouveau `silent`.

EXERCICES:

Insérez la commande `interact` dans le fichier `ols.ect` après la commande `ols`. Une fois en mode interactif, visualisez les variables créées par `ols`. À cet effet, vous pouvez utiliser soit `print` soit `show` ; les effets sont identiques, pour des raisons évidentes.

Si vous avez effectué l'exercice précédent, si vous n'êtes pas toujours en mode interactif, vous aurez aperçu que, après avoir tapé `quit` pour quitter le mode interactif, un nouveau `quit` a été affiché avant la fin du programme. Ce deuxième `quit` est celui qui est contenu dans `ols.ect`.

Pourquoi est-il nécessaire de faire `quit` deux fois avant de quitter ***Ects*** ? Parce que l'effet de `quit` n'est pas, ou n'est pas toujours, de quitter ***Ects***. Les contextes successifs créés pendant le déroulement d'une session sont empilés. On a remarqué plus haut que, si on passe en mode interactif par `interact`, le contexte existant est sauvé. Il serait plus exact de dire que le contexte est empilé, sur une pile dans le sens de ce mot en informatique, de sorte qu'il sera retrouvé au moment où, dans le déroulement qui suit, le contexte qui était empilé juste en dessus est enlevé. Et voici finalement l'aboutissement de l'histoire : c'est la commande `quit` qui enlève les contextes.

Quand on termine le mode interactif, on tape `quit`. Le contexte qui avait été créé pour constituer le mode interactif est enlevé, et on se retrouve dans le contexte qui après l'enlèvement se trouve en haut de la pile. Ceci fait que si, par exemple, on tape `interact` quand on est déjà en mode interactif, un nouveau contexte interactif sera établi, l'ancien étant empilé. Ensuite, on tape `quit`, mais on ne quitte pas le mode interactif, parce que, quand le contexte en dessous est retrouvé, il est lui aussi un contexte interactif. On a quitté le deuxième contexte interactif pour retrouver le premier.

Et s'il n'y a plus de contextes ? Si la pile est vide ? Alors, on quitte le programme, et voilà pourquoi, si on ne crée qu'un seul contexte, l'effet de `quit` est toujours de terminer l'exécution d'***Ects***. Dans la section 1.2, on a remarqué que l'usage de `quit` n'est pas obligatoire. En effet, si ***Ects*** arrive à la fin d'un fichier de commandes, un `quit` sera généré automatiquement : le contexte courant sera enlevé de la pile, et s'il n'y a rien en dessous le programme est terminé.

Parce que, à part la terminaison d'un programme, l'usage le plus fréquent de `quit` est de quitter le mode interactif, il existe une commande `batch`, analogue

à la commande `interact`, qui sert à terminer le mode interactif. Mais, en réalité, cette commande n'est qu'un synonyme de `quit`.

EXERCICES:

Lancez des contextes interactifs emboîtés pour voir les effets que l'on vient de décrire. En sortant de ces contextes, utilisez la commande `batch` pour vous assurer qu'elle fonctionne exactement comme `quit`, jusqu'à la terminaison du programme.

Comment emboîter les contextes s'ils ne sont pas des contextes interactifs ? Et pourquoi ? Pour répondre d'abord à la deuxième question, pour qu'un fichier de commandes puisse en appeler un autre. Si, par exemple, la lecture de vos données se fait en plusieurs étapes, avec d'éventuelles transformations des données ainsi lues, il peut être commode de mettre les commandes pertinentes dans un fichier autre que celui qui contient les commandes relevant des estimations que l'on souhaite faire sur les données. Dans un tel cas, on peut mettre dans le fichier qui effectue les estimations une commande qui appelle le fichier contenant les commandes de lecture, avant de passer ensuite aux commandes qui lancent les estimations.

La commande en question est la commande `run`. La syntaxe est très simple :

`run` ⟨*fichier de commandes*⟩

Ce qui se passe est aussi très simple. Le contexte courant, interactif ou non, est sauvé sur la pile de contextes, et un nouveau contexte, non-interactif, est créé pour l'exécution des commandes contenues dans le ⟨*fichier de commandes*⟩. L'exécution de ces commandes une fois terminée, par un `quit` explicite ou non, on retrouve l'ancien contexte en haut de la pile. Un fichier ainsi appeleé peut en appeler encore un autre, et ainsi de suite – la seule limitation est la mémoire vive de l'ordinateur, qui doit contenir la pile de contextes.

* * * *

La section suivante sur la musique n'est retenue que pour le folklore. La version 4 d'***Ects*** n'a plus cette fonctionnalité largement éloignée des préoccupations de l'économétrie.

* * * *

Un exemple de l'utilisation de la commande `run` est fourni par le fichier `mrs.ect`. Si vous regardez le contenu de ce fichier, il sera largement incompréhensible. Mais, au début du fichier, vous verrez qu'un fichier nommé `pitch.ect` est appelé. Ce deuxième fichier contient toute une série de définitions des variables `C`, `D`, `G`, *etc*, qui apparaissent dans `mrs.ect`. Les deux fichiers utilisent les commandes `noecho` et `silent`, pour cacher leurs opérations.

EXERCICES:

Avant de lire plus loin, faites tourner le fichier `mrs.ect`. Il serait préférable, dans un premier temps, d'utiliser la version ects86.exe du logiciel, pour une raison qui sera exposée ultérieurement.

L'exercice que vous venez d'effectuer démontre les capacités de la commande `beep`. On a vu comment cette commande sert à activer le bip de l'ordinateur. Ce qu'on vient d'entendre démontre que le signal sonore peut prendre plusieurs formes. Regardons les premières commandes de `mrs.ect` :

```
set t = 1.5
set b = 400
beep .5*G*t .25*b
beep .5*G*t .5*b
beep 0 .25*b
beep .5*G*t .25*b
beep C*t .9*b
beep 0 .1*b
beep C*t .9*b
beep 0 .1*b
```

La commande `beep` est apparemment capable de prendre deux arguments. Le premier sert à établir le ton, le deuxième la durée du son émis par l'ordinateur. Pour que les commandes soient au moins en partie reliées à la notation musicale, les définitions contenues dans le fichier `pitch.ect` affectent aux variables `A`, `B`, `Bf`, `Cs`, *etc*, des fréquences appropriées. Il faut savoir que, dans les mondes anglo-saxon et allemand, les notes musicales sont identifiées par les lettres de l'alphabète, plutôt que par les syllabes du solfège. Les correspondances sont données dans le Tableau 1.

Tableau 1 Les Notes Musicales

Solfège	Lettre
ut (do)	C
ré	D
mi	E
fa	F
sol	G
la	A
si	B

Les notations `s` et `f` correspondent à **dièze** (= *sharp*) et **bémol** (= *flat*). On peut donc accéder facilement à toutes les notes de la gamme. La variable `t`, utilisée dans `mrs.ect`, permet de transposer les notes, en les multipliant par des facteurs appropriés. De même, la variable `b` permet de choisir un rythme, ou plus correctement un tempo.

* * * *

Pour de plus amples renseignements, addressez-vous à votre professeur de piano, ou au Conservatoire !

* * * *

La version « grande » du logiciel, `ects.exe`, requiert une information supplémentaire avant de savoir jouer correctement. Il faut que la variable `speed` existe et que sa valeur soit correcte pour la machine que vous utilisez. Ainsi, sur un 486 à 50 MHz la valeur appropriée est autour de 8.000 ; sur un 386 à 33 MHz elle est autour de 2.000. Plus la machine tourne vite, plus `speed` doit être élevé. La bonne valeur pour une machine donnée ne peut être établie que par des expériences.

Les versions d'***Ects*** destinées à tourner sous Unix ne peuvent émettre qu'un bip ordinaire. À moins de connaître précisément les capacités de l'ordinateur utilisé, il est impossible de programmer les commandes nécessaires à l'émission d'un son musical.

* * * *

On revient maintenant aux choses sérieuses !

* * * *

La vraie mission d'***Ects*** n'est certainement pas de jouer de la musique. Mais la fonctionnalité décrite ci-dessus fut demandée par un étudiant qui préférait un signal sonore intéressant. Si on a des goûts plus austères, on peut se contenter de la commande `pause`. Le signal sonore émis par cette commande est privé de tout élément de sensualité. Si dans un programme ***Ects*** on met la commande `pause`, sans argument, le déroulement du programme s'arrête, un bip est émis, et le message suivant s'affiche :

```
Press a key to continue
```

c'est-à-dire, appuyez sur une touche pour continuer. Cette commande permet donc d'examiner le contenu de l'écran ; chose qui n'est pas toujours possible si l'ordinateur tourne assez vite.

Si la commande `pause` est suivie d'un argument, cet argument sera évalué à la manière des arguments qui doivent être des entiers positifs (voir les sections 2.2 et 2.3). Soit n la valeur de l'argument, alors, après un bip préliminaire, la machine reste inactive pendant n secondes, après quoi il y a un nouveau bip, et l'exécution des commandes qui suivent `pause` est reprise.

3. Le Contrôle du Contenu du Fichier de Sortie

Jusqu'ici, la seule manière d'influencer le contenu du fichier de sortie est au travers des commandes `silent` et `restore`. Mais il existe aussi des moyens plus fins, qui passent par l'utilisation des commandes `text` et `put`.

Avant de regarder en détail l'effet de ces commandes, remarquons qu'une méthode traditionnelle est disponible, qui peut répondre très bien à certains types de besoins. Si une ligne dans un fichier de commandes ***Ects*** commence par `rem`, le programme saute tout de suite à la ligne suivante, quelle que soit la suite de la commande. En effet, `rem` permet de glisser des *remarques*, ou des *commentaires*, dans le programme. Mais la ligne entière sera, comme toute

autre ligne, affichée à l'écran et imprimée dans le fichier de sortie, à condition, bien sûr, que ce ne soit pas empêché par un `silent` ou un `noecho`.

* * * *

En mode interactif, il n'y a pas d'écho, et il n'y a normalement pas de fichier de sortie. Mais en mode interactif, les commentaires ne servent pas à grand-chose.

* * * *

On peut se servir des commentaires introduits par `rem` pour ponctuer et pour documenter un fichier de sortie.

L'opération des commandes `text` et `put` peut être appréhendée plus facilement si nous considérons un exemple. Un tel exemple est contenu dans le fichier `logit.ect`. La première partie de ce fichier met en œuvre une estimation d'un **modèle logit**. L'estimation n'utilise pas les commandes `ml` et `gmm`, qui auraient pu servir, mais passe plutôt par une **régression artificielle**. Cette méthode est exposée dans le chapitre 15 de DM, section 15.4. Pour le moment, on laisse de côté les détails de la méthode, pour voir comment les résultats peuvent être imprimés dans le fichier de sortie comme si on avait utilisé une méthode standard comme, par exemple `nls`.

Avant que la procédure d'estimation soit entamée, on supprime l'écho et toute impression dans le fichier de sortie par

```
silent
noecho
```

Après, on voit les commandes suivantes :

```
mat parms = rowcat(a, b1, b2, b3)
sample 1 4
gen T = parms/stderr
mat block = colcat(parms, stderr, T)
mat line1 = block(1,1,1,3)
mat line2 = block(2,2,1,3)
mat line3 = block(3,3,1,3)
mat line4 = block(4,4,1,3)
text
Estimation of Logit Model by BRM Artificial Regression

Number of iterations =
end
put i
text
Parameter       Estimate        Std error       T statistic

end
text a
end
put line1
text b1
```

```
end
put line2
text b2
end
put line3
text b3
end
put line4
text
Estimated covariance matrix:

end
put XtXinv
text

end
```

Les variables `a`, `b1`, `b2`, et `b3` sont les paramètres du modèle logit. Après l'estimation, ils sont regroupés dans une ligne `parms` par `rowcat`. Ensuite, une matrice 4×3, nommé `block` est créée par `colcat` pour contenir les paramètres estimés, les écarts-type estimés, et les Students. Finalement, les quatre lignes successives de `block` sont extraites et sauvées sous les noms `line`i, $i = 1, \ldots, 4$.

Les commandes qui suivent ces opérations servent à imprimer dans le fichier de sortie les tableaux suivants :

```
Estimation of Logit Model by BRM Artificial Regression

Number of iterations =  5.000000

Parameter    Estimate     Std error   T statistic

a            -2.535442    1.427220    -1.776490
b1           0.086252     0.019314    4.465755
b2           -0.131312    0.023472    -5.594512
b3           0.040785     0.011006    3.705706

Estimated covariance matrix:

2.993058     -0.033219    0.018429    0.008998
-0.033219    0.000548     -0.000497   0.000052
0.018429     -0.000497    0.000810    -0.000189
0.008998     0.000052     -0.000189   0.000178
```

Que se passe-t-il ? Seules deux commandes sont utilisées pour la construction de ces tableaux : `text` et `put`. L'effet de `text` est d'imprimer dans le fichier de sortie tout ce qui suit le mot `text` jusqu'à ce qu'une ligne commence par `end`. Le tout premier `text`, celui qui précède la ligne

```
Estimation of Logit Model by BRM Artificial Regression
```

est suivi justement par un saut de ligne. Dans le fichier de sortie également, on sautera à la ligne suivante avant d'imprimer le texte. Plus loin, on voit des lignes comme

```
text a
```

Dans un tel cas, le `a` est imprimé tout de suite, sans saut de ligne.

Après la première ligne de texte, celle citée dans le paragraphe précédent, il y a dans le fichier de commandes un nouveau saut de ligne, reproduit dans le fichier de sortie. Ensuite, on a

```
Number of iterations =
end
```

Comme on peut le constater dans le fichier de sortie, le tout dernier saut de ligne, celui qui précède `end`, n'est pas reproduit. La raison en est que l'on souhaite souvent imprimer, sur la même ligne du fichier de sortie, quelque chose autre qu'un texte que l'on peut fournir litéralement. En effet, c'est ici la valeur de la variable `i` qui doit être imprimée. Mais le saut de ligne est nécessaire dans le fichier de commandes, pour que le `end` soit lu comme une commande plutôt que la suite du texte.

* * * *

Ceci a pour conséquence que l'on ne peut pas commencer une ligne de `text` par le mot « end ». Attention ! On peut en revanche commencer la ligne par un ou plusieurs espaces blancs, ou par une tabulation horizontale, suivi du mot « end ». Pour que ce mot signale la fin d'un `text`, il faut qu'il soit placé au tout début de la ligne.

* * * *

La commande `put` sert à mettre les *valeurs* dans le fichier de sortie. On a vu dans la section 1.3 que cette commande écrit dans le fichier de sortie, et qu'elle ne fait pas précéder de leurs noms les scalaires, vecteurs, et matrices qu'elle imprime. On voit maintenant pourquoi une telle commande est utile. La commande a deux autres propriétés qu'il faut connaître pour l'utiliser correctement. Primo, l'impression sera toujours suivie d'un saut de ligne. Ceci explique pourquoi les paramètres estimés, écarts-type estimés, et Students ont été regroupés dans des lignes avant d'être imprimés. Sans passer par ce stade intermédiaire, seul un rangement par colonnes aurait été possible. Secondo, la commande `put` écrit dans le fichier de sortie même si on a préalablement fait un `silent`. Cette propriété est absolument nécessaire. Si l'on ne fait pas `silent` avant d'utiliser `put` et `text`, alors les lignes de commandes seront imprimées. Par exemple, si on faisait

```
text a
end
put line1
```

sans faire `silent` au préalable, on obtiendrait

```
text a
a               put line1
```

```
-2.5354420      1.4272198      -1.7764901
```

dans le fichier de sortie !

Il y a des aspects de l'extrait du fichier de commandes qui sont invisibles, mais qui ont des conséquences importantes pour l'impression dans le fichier de sortie. En fait, les lignes comme

```
text a
```

sont suivies de deux caractères de tabulation horizontale. (Le caractère 9 en ASCII.) Ceci sert à aligner les tableaux imprimés. ***Ects*** lui-même s'occupe de l'insertion de ces caractères entre les éléments des vecteurs et matrices.

EXERCICES:

Essayez de recréer les tableaux imprimés dans les fichiers de sortie par la commande `ols` par l'utilisation de `text` et `put`. N'oubliez pas de travailler sous `silent`.
Dans la section 4.3, on a considéré l'estimation du modèle (15) par la commande `gmm`, qui ne fournit pas beaucoup d'informations. Toutefois il est complètement possible de trouver une estimation de la matrice de covariance des paramètres estimés : voir les sections 7.6 et 7.7 de DM. Construisez un tableau qui présente toutes les informations habituelles, comme si l'estimation avait été faite par `iv`.

Chapitre 6
Les Expériences Monte Carlo

1. La Programmation des Boucles

Depuis longtemps, plus exactement depuis l'aube de l'ère des ordinateurs, les expériences numériques font partie de la recherche scientifique. L'économétrie participe pleinement à l'utilisation de ces expériences, qu'on appelle souvent les **expériences Monte Carlo**. Un exposé portant sur ces expériences se trouve dans le chapitre 21 de DM.

Il y a deux ingrédients essentiels à la cuisine d'une expérience Monte Carlo, à savoir, la génération de nombres pseudo-aléatoires, que nous considérerons plus loin, et la répétition un grand nombre de fois d'un ensemble de commandes, ensemble qui constitue ce qu'on appelle une **simulation**.

La répétition d'un ensemble de commandes s'effectue en informatique par les **boucles**, c'est-à-dire, des blocs de commandes qui sont exécutées plusieurs fois, jusqu'à ce qu'une condition donnée ne soit plus remplie. Pour implémenter une boucle en ***Ecta***, on emploie la commande `while`.[13] Voici un schéma qui représente une boucle :

```
while ⟨expression⟩
        ⟨bloc de commandes⟩
end
```

Après avoir lu la commande `while`, ***Ecta*** passe à l'évaluation de l'⟨*expression*⟩ qui suit la commande. Si la valeur de l'expression est différente de zéro, le ⟨*bloc de commandes*⟩ sera exécuté. Sinon, ***Ecta*** saute à la fin de la boucle, signalée par la commande `end`.

Regardons d'un peu plus près la nature des expressions que l'on peut utiliser après la commande `while`. Par exemple, vous souhaitez faire tourner la boucle dix fois. Dans ce cas, une façon très simple de procéder est d'employer une variable scalaire pour compter les **itérations** de la boucle. Avant `while`, on fait

```
set i = 0
```

[13] Le mot anglais *while* correspond à la locution française *tant que*.

par exemple, et la première commande du bloc qui constitue la boucle sera

```
set i = i+1
```

* * * *

La variable `i` peut servir à plusieurs fins. Si les dix itérations permettent de remplir les dix éléments d'un vecteur `x`, il suffit de mettre la commande

`set x(i) =` ⟨*valeur calculée*⟩

dans le bloc.

* * * *

On arrive à la fin de l'exécution de la boucle quand `i` = 10. Autrement dit, la boucle est à répéter tant que `i` < 10. La commande appropriée est donc :

```
while i < 10
```

L'expression `i < 10` est un exemple d'une expression **Booléenne**. Une telle expression ne prend que deux valeurs possibles, 0 et 1. En ***Ects***, sont considérées comme expressions Booléennes toutes les expressions qui s'interprètent comme des **relations**, d'égalité ou d'inégalité. Ainsi les expressions suivantes sont des expressions Booléennes :

```
a < b
a = b
a > b
```

Les variables `a` et `b` peuvent être des scalaires, vecteurs, ou matrices, comme d'habitude. Les expressions Booléennes peuvent faire l'objet des commandes `gen`, `set`, et `mat`. Si par exemple on fait

```
gen x = a > b
```

la composante i du vecteur `x` (le nombre de ses composantes est déterminé par l'état courant de `smplstart` et `smplend`) égale 1 si $\mathtt{a}_i > \mathtt{b}_i$, et 0 sinon.

La **priorité** des relations `=`, `>`, et `<` est plus basse que celle de toutes les autres opérations arithmétiques. Ceci veut dire que si, par exemple, on fait

```
gen z = x1*(x2 + x3) = x1*x2 + x2*x3
```

toutes les composantes du vecteur `z` seront égales à 1. Toutes les opérations d'addition et de multiplication seront effectuées avant la comparaison des deux membres $\mathtt{x1} * (\mathtt{x2} + \mathtt{x3})$ et $\mathtt{x1} * \mathtt{x2} + \mathtt{x2} * \mathtt{x3}$. Les deux membres étant égaux, composante par composante, chaque composante de `z` reçoit la valeur VRAI, ou 1.

EXERCICES:

Générez une variable comme le `z` de l'exemple ci-dessus, mais mettez `<` ou `>` à la place du `=`. Comment expliquer les résultats ?

L'évaluation de l'expression qui suit une commande `while` est faite comme si on était sous l'influence d'un `mat`.

* * * *

Ce choix a été fait pour éviter les longs calculs qui se produiraient si la taille de l'échantillon était important. En effet, la relation `i < 10` serait évaluée `smplend` fois si le calcul était fait comme sous `gen`.

* * * *

Ensuite, à l'élément $(1,1)$ de la matrice évaluée, ***Ects*** ajoute 10^{-10}, pour éviter des difficultés éventuelles associées aux erreurs d'arrondi. Finalement, si le plus grand entier qui est plus petit que la valeur ainsi obtenue est zéro, la condition est considérée comme fausse, et la boucle s'arrête, sinon comme vrai, et la boucle continue.

Un excellent exemple d'une boucle qui n'a rien à voir avec les expériences Monte Carlo est fourni par la procédure d'estimation contenue dans le fichier `logit.ect`. On a remarqué dans la section 5.3 que cette estimation passe par une régression artificielle. L'estimation est obtenue par l'itération de cette régression, jusqu'à ce que les paramètres estimés ne changent plus d'une itération à la suivante. Voici la boucle en question :

```
set tol = 1
set i = 0
while tol > 0.000001
       gen den = 1/sqrt(F*(1-F))
       gen r = (y - F)*den
       gen r0 = f*den
       gen r1 = r0*x1
       gen r2 = r0*x2
       gen r3 = r0*x3
       ols r r0 r1 r2 r3
       set a = a + coef(1)
       set b1 = b1 + coef(2)
       set b2 = b2 + coef(3)
       set b3 = b3 + coef(4)
       mat tol = coef'*coef
       set i = i+1
end
```

Avant le commencement de la boucle, on initialise les scalaires `i` et `tol`.[14] `i` sert simplement à compter les itérations. Le rôle de `tol` est de déterminer le moment où l'algorithme converge. En effet, on voit que les itérations continuent tant que `tol` est supérieur à 0.000001. Pour interpréter cette variable, il faut savoir que les paramètres estimés de la régression effectuée par `ols` au milieu de la boucle sont les *corrections* qui sont apportées aux valeurs existantes des paramètres. Ainsi, `tol` mesure le carré du déplacement, dans l'espace paramétrique, engendré par l'itération qui vient d'être effectuée. Quand ce déplacement est suffisamment petit, l'algorithme a convergé.

[14] `tol` en minuscules pour éviter une confusion éventuelle avec la variable `TOL` qui détermine la précision des estimations.

2. Exécution Conditionnelle d'un Bloc de Commandes

Pour des raisons diverses, il est parfois souhaitable d'exécuter un bloc de commandes uniquement si une condition est vérifiée. Dans l'exemple de la régression artificielle qu'on vient de considérer, il n'est pas exclu *a priori* que l'algorithme ne converge pas, même après un grand nombre d'itérations. Dans ce cas, la valeur de `tol` sera toujours supérieur à 0.000001 quand le compteur `i` atteint une valeur pré-spécifiée. Si, mais seulement si, cela arrive, on aimerait faire afficher à l'écran un message à cet effet. La commande à utiliser est `if`.

Considérons les conséquences de l'insertion des commandes suivantes à la fin de la boucle ci-dessus.

```
if i = 20
   message
   20 itérations sans convergence.  Je continue (1=Oui/0=Non)?
   end
   input a
   if a = 0
       tol = 0
   end
end
```

La commande `message` est identique à la commande `text`, au détail près que le texte n'est pas imprimé dans le fichier de sortie : il est plutôt affiché à l'écran, même si on a préalablement fait un `noecho`. La commande `input` permet d'arrêter le programme le temps de taper un chiffre, la valeur duquel sera affectée à la variable scalaire donnée en argument à la commande.

* * * *

L'effet de la commande

```
input a
```

est très similaire à celui des commandes

```
set t1 = smplstart
set t2 = smplend
sample 1 1
read con a
sample t1 t2
```

Il a paru utile d'avoir une seule commande pour effectuer un si petit travail !

* * * *

On peut maintenant décrire ce qui se passe si l'algorithme n'a toujours pas convergé après 20 itérations. ***Ects*** lit la commande

```
if i = 20
```

et il constate que la condition est vérifiée. Ensuite les commandes dans le bloc entre le `if` et le `end` sont exécutées. D'abord le `message`, terminé par le premier `end`, est affiché, et la machine se met en attente de la réponse de l'utilisateur, qui doit maintenant taper un chiffre. Le chiffre tapé est saisi

dans la variable `a`. La commande suivante est encore une commande `if`. La variable `a` est examinée, et si elle vaut zéro, le bloc entre le nouveau `if` et le deuxième `end` est exécuté. Ce bloc n'a qu'une seule commande, qui sert à annuler la variable `tol`. Et si `tol` est nul, la boucle extérieure, celle qui est terminée par le dernier `end`, s'arrête, car la condition `tol` > 0.000001 n'est plus vérifiée. Si l'utilisateur tape autre chose que 0, la boucle continue jusqu'à la convergence de l'algorithme.

EXERCICES:

Écrivez un programme ***Ects*** qui invite l'utilisateur à taper un chiffre, et qui affiche ensuite le chiffre tapé à l'écran. Insérez ce programme dans une boucle, qui se répète infiniment, ou jusqu'à ce que le chiffre tapé égale zéro.

On voit dans l'exemple ci-dessus qu'il y a plusieurs commandes qui attendent un `end` pour signaler qu'un bloc de commandes, ou de dérivées partielles, ou de texte, est terminé. Voici la liste complète de ces commandes :

```
nls
ml
gmm
text
message
while
if
else
```

Les cinq premières commandes de cette liste n'admettent pas la possibilité qu'une autre commande de la liste puisse intervenir entre la ligne de la commande elle-même et son propre `end`. Mais les trois dernières peuvent être emboîtées, c'est-à-dire que, comme on l'a vu dans l'exemple, un bloc commencé par un `if` peut très bien se trouver à l'intérieur d'un bloc commencé par un `while`. Pour que ceci soit possible, ***Ects*** se sert d'une pile, comme la pile de contextes que nous avons vue à la section 5.2. Ainsi, tout `end` est correctement associé à la commande dont il est la terminaison.

Il faut finalement considérer la commande `else`. Cette commande s'utilise à l'intérieur d'un `if`. Si la condition de l'`if` est vérifiée, les commandes du bloc sont exécutées. Si parmi ces commandes ***Ects*** rencontre un `else`, il saute à l'`end` qui correspond à l'`if`. En revanche, si la condition d'un `if` n'est pas vérifiée, ***Ects*** se met à chercher, soit l'`end` qui lui appartient, soit un `else`. Si c'est le dernier qu'il trouve, il commence à exécuter le bloc de commandes entre l'`else` qu'il a trouvé et l'`end` de l'`if`.

Autrement dit, la structure

```
if ⟨condition⟩
    ⟨bloc1⟩
else
    ⟨bloc2⟩
end
```

fait exécuter ⟨$bloc_1$⟩ si la ⟨*condition*⟩ est vérifiée, et ⟨$bloc_2$⟩ sinon. Les deux ⟨*bloc*⟩s peuvent, eux aussi, contenir d'autres blocs, associés soit à un `while` soit à un `if`, qui à leur tour peuvent encore contenir des blocs, et ainsi de suite jusqu'aux limites de la mémoire de l'ordinateur.

Une dernière remarque sur l'utilisation de la commande `end`. On a vu à la section 5.3 que, pour signaler la fin d'un `text` ou d'un `message`, il faut que la commande `end` se trouve au début de la ligne, sans espace blanc. Ceci ne s'applique qu'aux deux commandes `text` et `message`, qui doivent transférer fidèlement le contenu du fichier de commandes au fichier de sortie ou à l'écran. Pour les autres commandes qui utilise `end`, on peut très bien, comme cela a été fait dans les exemples de cette section, mettre de l'espace blanc aux débuts des lignes pour indiquer la structure du programme. Cette pratique est même à conseiller.

3. Les Nombres (Pseudo-)Aléatoires

La parenthèse dans l'intitulé de la section indique clairement qu'un ordinateur est un appareil déterministe. Sinon, l'informatique serait très différente de ce qu'elle est, et sans doute beaucoup moins utile. Cependant, on a besoin du hasard si on veut faire des simulations.

* * * *

> Et aussi pour les jeux. Mais, comme je l'ai dit à plusieurs reprises, les jeux sont stupides.

* * * *

Le mieux que l'on puisse faire si l'on ne dispose que d'un appareil déterministe est de l'employer pour calculer des **nombres pseudo-aléatoires**, c'est-à-dire, des nombres qui partagent toutes les propriétés pertinentes des nombres réellement aléatoires.

Le générateur de nombres aléatoires[15] qu'utilise ***Ects*** est un générateur du type dit **congruentiel**. Bien que ce type de générateur ne soit pas forcément le meilleur, c'est le plus courant. On trouvera un exposé élémentaire dans la section 21.2 de DM, et un exposé plus détaillé dans le Chapitre 7 de Press *et al.* (1986). Le générateur d'***Ects*** est un générateur double, c'est-à-dire que deux congruences indépendantes sont utilisées. Ceci permet de générer des suites de nombres aléatoires très longues avant de revenir au point de départ. Pour certaines expériences Monte Carlo, cet aspect est important.

La fonction que l'on utilise pour générer les nombres aléatoires est `random`. Selon le jargon informatique, cette fonction est **surchargée**. On entend par là que le même nom, `random`, sert à définir plusieurs fonctions légèrement

[15] Ce n'est plus la peine de faire la distinction. Il est entendu désormais que les nombres dits aléatoires ne sont que pseudo-aléatoires.

différentes. L'utilisateur a en effet deux possibilités. Si la fonction est utilisée sans argument, comme suit :

```
gen u = random()
```

les éléments du vecteur `u` sont des nombres aléatoires, indépendants les uns des autres, suivant tous la loi $N(0,1)$, c'est-à-dire, la loi normale centrée réduite.

* * * *

L'algorithme de Box-Müller est employé à cette fin

* * * *

Si à la place de `gen` on utilise `set`, un seul nombre aléatoire est généré. Si on utilise `mat`, l'effet est identique à celui de `gen`.

L'autre possibilité est d'employer deux arguments. La seule différence par rapport à l'emploi sans argument est que les nombres générés suivent la loi uniforme $\mathrm{U}(a,b)$, où les scalaires a et b sont respectivement le premier et le deuxième arguments fournis à `random`. Comme d'habitude, si les arguments sont des vecteurs ou des matrices, l'élément $(1,1)$ est sélectionné.

4. Monte Carlo

Maintenant tous les ingrédients sont en place. Nous pouvons essayer une petite expérience, ..., Monte Carlo, bien sûr. L'expérience que nous allons essayer se trouve dans le fichier `nearunit.ect`. Il n'est pas indispensable de comprendre tous les aspects théoriques de l'expérience, quoique l'on donne une esquisse de la théorie ci-après.

D'abord, le fichier de commandes :

```
set NOBS = 100
set NREPS = 1000

silent
noecho

sample 1 NREPS
gen zero = 0
gen tau = colcat(zero,zero,zero,zero,zero,zero)
mat z = tau

sample 1 NOBS
set rho = 0.9
gen t = time(-1)
set j = 0

while rho < 1.01
     set j = j+1
     gen rhot = rho^t
     set i = 0
     while i < NREPS
         set i = i+1
         gen eps = random()
```

```
            gen y = conv(rhot, eps)
            gen ylag = lag(1,y)
            gen dely = y - ylag
            sample 2 NOBS
            ols dely ylag
            sample 1 NOBS
            set tau(i,j) = student
            set z(i,j) = NOBS*coef
        end
        message Fin de l'expérience pour
    end
        show rho
        set rho = rho + 0.02
    end

    message Fin des expériences.

    end

    sample 1 NREPS
    echo
    restore

    ols tau c
    ols z c

    quit
```

Les deux premières commandes définissent deux variables, `NOBS`, la taille d'échantillon utilisée, et `NREPS`, le nombre de répétitions, ou simulations, à effectuer. Il est toujours souhaitable de définir de telles variables plutôt qu'utiliser des chiffres précis, pour qu'on puisse aisément changer les valeurs, une fois pour toutes, en changeant la définition au début du fichier. Ensuite, parce que le déroulement d'une expérience Monte Carlo peut durer un certain temps, qui risque d'être augmenté si tout ce qui se passe est soigneusement enregistré à l'écran et dans le fichier de sortie, on fait `silent` et `noecho`. Après, on crée deux matrices, `z` et `tau`, de la forme `NREPS` $\times$ 6, pour contenir les résultats de l'expérience.

Vient ensuite l'initialisation des variables utilisées au cours de l'expérience. `rho` est un paramètre, en l'occurrence un paramètre d'autocorrélation. `j` est le compteur des itérations de la boucle extérieure, et `t` est une tendance temporelle, dont la première composante égale 0.

La variable qui contrôle la boucle extérieure est `rho`. De sa valeur initiale de 0,9 il est augmenté par incréments de 0,02 jusqu'à une valeur finale de 1,0. Après la dernière itération, la valeur de `rho` sera 1,02, supérieure à 1,01, et la boucle s'arrête. On aurait pu employer le compteur `j` d'une façon similaire pour contrôler la boucle. À son intérieur se trouve une autre boucle, controlée par la variable `i`, qui sert à faire les `NREPS` simulations pour chacune des six valeurs possibles de `rho`, d'où la nécessité de prévoir des matrices `NREPS` $\times$ 6 pour contenir tous les résultats.

Pour chaque simulation, un vecteur aléatoire `eps` est généré, qui fait ensuite l'objet de la commande

```
gen y = conv(rhot, eps)
```

Le vecteur `rhot` est défini une seule fois pour chacune des valeurs de `rho` : la composante t du vecteur égale ρ^{t-1}. Le nom de la nouvelle fonction `conv` provient du terme technique **convolution**. Soient $\boldsymbol{x}$ et $\boldsymbol{w}$ deux vecteurs $n \times 1$, la convolution $\boldsymbol{z}$ de $\boldsymbol{x}$ et $\boldsymbol{w}$ est définie par la formule suivante :

$$z_t = \sum_{s=1}^{t} x_s w_{t-s+1}. \tag{16}$$

On vérifie rapidement que la définition est symétrique par rapport à $\boldsymbol{x}$ et $\boldsymbol{w}$. Selon cette définition, la variable `y` définie au moyen de la fonction `conv` est telle que

$$y_t = \sum_{s=1}^{t} \rho^{s-1} \varepsilon_{t-s+1} = \sum_{s=1}^{t} \rho^{t-s} \varepsilon_s.$$

Le sens de cette définition provient du fait que

$$y_t = \rho y_{t-1} + \varepsilon_t.$$

Cette équation est une auto-régression d'ordre 1, à paramètre auto-régressif ρ.

La série `y` ainsi obtenue est ensuite retardée et différentiée. La taille de l'échantillon est ajustée afin de supprimer la première observation, comme d'habitude quand il a des variables retardées, et les différences premières `dely` sont régressées sur le retard `ylag`. Le paramètre estimé, multiplié par la taille de l'échantillon, et le Student, sont alors stockés dans les grandes matrices `z` et `tau`.

Il s'agit de deux statistiques utilisées pour détecter la présence d'une **racine unitaire**. La théorie peut être trouvée dans le Chapitre 20 de DM. Les notations z et τ sont habituelles dans les exposés de cette théorie : voir la section 20.2 de DM.

Après chaque itération de la boucle extérieure, `NREPS` simulations ont été effectuées. Outre la génération des variables, chaque simulation exige une estimation OLS. Dans le cas actuel, chaque grande itération comporte mille régressions. Sur un 486/50MHz, le temps de calcul pour ces mille régressions est de 50 secondes environ. Après ce temps, un message est affiché à l'écran, dans lequel apparaît la valeur courante de `rho`. Après les six itérations de la grande boucle, un message terminal est affiché, après quoi on passe à l'analyse des résultats. Les deux commandes

```
ols tau c
ols z c
```

servent à faire tourner 12 régressions, car les deux variables `tau` et `z` ont 6 colonnes chacune. Les 12 colonnes sont régressées sur la constante, ce qui sert à calculer la moyenne de chaque colonne, ainsi que son écart-type.

Et alors ? La théorie des tests de racines unitaires est assez difficile, et il y a très peu de résultats exacts. Il est connu que les lois de probabilité suivies par les statistiques z et τ, même asymptotiquement, ne sont pas les lois standard que l'on trouve couramment en économétrie. L'expérience que l'on vient de faire nous a donné des estimations des espérances de ces lois de probabilité, en fonction du paramètre d'autocorrélation de la variable $\boldsymbol{y}$. En plus, les écarts-type estimés nous fournissent une mesure de la fiabilité de ces estimations. Prenons un cas, à titre illustratif. Quand $\rho = 1$, ce qui est le cas de la dernière grande itération, l'hypothèse nulle du test est vrai : il y a en effet une racine unitaire. On aurait pu espérer que, dans ce cas, les espérances des statistiques soient nulles. Voyons le résultat de l'expérience sur la statistique z, pour $\rho = 1$. On obtient

```
Variable    Parameter estimate   Standard error T statistic

c           -1.7138185           0.0954023      -17.9641271
```

On voit très clairement que l'hypothèse de la nullité de l'espérance de z est à rejeter fortement. L'estimation ponctuelle de cette espérance est -1.71[16], et son écart-type estimé est 0.095. Un intervalle de confiance raisonnable serait donc l'intervalle $[-1.90, -1.52]$. Pour obtenir un intervalle plus étroit, il serait nécessaire, soit tout simplement d'augmenter le nombre de simulations, soit d'utiliser une technique de réduction de variance ; voir les sections 21.5 et 21.6 de DM.

* * * *

L'expérience complète a duré moins de 5 minutes sur un 486/50. Si on est prêt à laisser tourner la machine une nuit entière, on peut obtenir un résultat beaucoup plus précis.

* * * *

Exercices:

Limitez-vous au cas où $\rho = 1$, et obtenez une estimation plus précise de l'espérance de z au moyen d'une variable de contrôle (section 21.6 de DM).

[16] Il est inutile de garder le chiffre tel qu'il est imprimé : l'erreur d'estimation est trop importante.

Chapitre 7
Tout le Reste

1. Fonctions Diverses ; Règles Diverses

Il n'a pas été possible de considérer *toutes* les fonctions auxquelles on peut accéder en ***Ects*** dans l'un des contextes économétriques considérés dans ce manuel. Pourtant, les fonctions qu'on n'a pas encore vues peuvent être d'une grande utilité dans d'autres contextes. Ce chapitre a pour but de donner un descriptif de ces fonctions, et de préciser quelques règles suivies par ***Ects*** dans l'évaluation des fonctions en général.

La première fonction à considérer est `round`, qui sert à arrondir les réels. Le résultat de `round(x)` pour un *scalaire* `x` est l'entier le plus proche de `x`. Le résultat de `round(y)` pour un *vecteur colonne* `y` est un vecteur colonne, dont la dimension est la même que celle de `y`. La fonction s'applique composante par composante. Le résultat de `round(A)` pour une *matrice* `A` quelconque est un vecteur colonne, qui a le même nombre de lignes que `A`, et qui résulte de l'application de la fonction à la *première colonne* de `A`, *si* le calcul se fait sous `gen`, mais si le calcul se fait sous `mat`, le résultat est une matrice, de la même forme que `A`, et qui résulte de l'application de la fonction élément par élément.

Les règles énoncées ci-dessus s'appliquent généralement à toutes les fonctions ***Ects***, sauf exception.

* * * *

Rappel : `set` est identique à `gen` précédé de

```
sample 1 1
```

et suivi de la remise en place des anciennes valeurs de `smplstart` et `smplend`.

* * * *

Pour éviter des difficultés ultérieures, voici une liste des exceptions, qui sont considérées plus loin.

```
colcat
rowcat
random
uptriang
conv
```

```
diag
det
sum
time
seasonal
lag
sort
```

Voici maintenant deux fonctions qui obéissent aux règles, mais qui, à la différence de `round`, prennent deux arguments. `max(a,b)` donne le maximum (algébrique, avec prise en compte du signe) des variables `a` et `b`. De même `min(a,b)` donne le minimum algébrique. On peut maintenant énoncer quelques règles encore, qui s'appliquent aux fonctions, qui, comme `max` et `min`, prennent plus d'un argument.[17] Sous `gen`, seules les premières colonnes de *l'ensemble des arguments* sont prises en compte. Sous `mat`, toute fonction à laquelle plus d'un argument est donné donne lieu à une erreur de syntaxe. Et voici une règle qui s'applique même aux fonctions exceptionnelles : Si le nombre d'arguments est inférieur à l'attente de la fonction, il y a erreur de syntaxe. Si le nombre d'arguments est supérieur à l'attente de la fonction, les arguments qui suivent sont perdus.

Il existe un ensemble de fonctions (non exceptionnelles) qui permettent de calculer les seuils critiques associés aux distributions de probabilité les plus courantes en économétrie. Ces distributions sont la normale (centrée, réduite), le χ^2, le Student, et le Fisher. La syntaxe

```
normcrit(x)
```

calcule le réel z qui vérifie l'équation

$$\Phi(z) = x,$$

où Φ est la fonction de répartition de la loi $N(0,1)$, donnée à l'équation (4). Ainsi, pour obtenir le seuil critique d'un test bilatéral à un niveau de 5%, on utilise `normcrit(0.975)`

EXERCICES:

Pourquoi ?

Il est clair que l'argument x de `normcrit`, qui a l'interprétation d'une probabilité, doit appartenir au segment $[0,1]$. Sinon, la valeur donnée par `normcrit` est une approximation soit à $-\infty$ soit à ∞.

La fonction Φ elle-même, disponible sous ***Ects*** à travers la fonction `phi`, permet de calculer le **niveau de significativité marginale** d'une statistique de test. On préfère souvent l'expression anglaise de ***P*−value**, ne serait-ce que pour sa compacité. La définition du concept se trouve dans la section 3.4 de DM. Par exemple, si on a obtenu une statistique dont la valeur numérique

[17] Toujours sauf les exceptions de la liste.

est 2,2, et qui, sous l'hypothèse nulle, est un tirage de la loi $N(0,1)$, la $P-$value associée à la statistique se calcule comme `1 - phi(2.2)`. C'est la masse de probabilité dans la queue de la distribution $N(0,1)$ au delà de la valeur 2,2.

EXERCICES:

La $P-$value que l'on calcule ainsi correspond à un test bilatéral ou unilatéral ?

Les fonctions qui correspondent à `phi` et `normcrit` pour la loi de Student sont `tstudent` et `studcrit`. La syntaxe est comme suit : `tstudent(x,n)` est la masse de probabilité à la gauche du réel `x` pour un Student à `n` degrés de liberté. `studcrit(x,n)` est le réel z qui vérifie l'équation

$$T_n(z) = x,$$

où l'on note T_n la fonction de répartition de la loi de Student à n degrés de liberté. Si l'argument x est en dehors du segment $[0,1]$, la fonction donne une valeur proche de l'infini, tout comme `normcrit`.

* * * *

Attention au nom de la fonction `tstudent`. Le `t` initial est nécessaire pour distinguer la fonction de la variable `student` qui contient les Students d'une régression.

* * * *

La loi du χ^2 est servie par les fonctions `chisq` et `chicrit`. Comme dans le cas de la loi de Student, deux arguments sont nécessaires, le premier un réel, le deuxième les degrés de liberté.

* * * *

Il n'est pas nécessaire que le nombre de degrés de liberté soit entier. En fait, la densité de probabilité associée à la loi du χ^2 existe pour n'importe quelle valeur réelle positive des degrés de liberté. Depuis l'invention des fractals on sait qu'une dimension n'est pas forcément un nombre entier. En revanche, si le nombre de degrés de liberté est négatif, la fonction `chisq` donne une valeur de 0.

* * * *

Pour la loi de Fisher, les fonctions sont `fisher` et `fishcrit`. Cette fois-ci, trois arguments sont nécessaires, les deux derniers étant les degrés de liberté. Des deux, le premier concerne le numérateur, le deuxième le dénominateur.

Trois autres fonctions sont disponibles. Elles sont utilisées par ***Ects*** pour calculer les fonctions de répartition que nous venons de considérer, et, à toutes fins utiles, elles sont mises à la disposition du grand public. La fonction `gln` correspond au logarithme de la **fonction gamma**. La définition de la fonction gamma elle-même est comme suit :

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t}\, dt.$$

Si l'argument z est un entier, la fonction est reliée au factoriel par la formule

$$z! = \Gamma(z+1).$$

C'est parce que le factoriel est une fonction dont les valeurs deviennent rapidement énormes que l'on préfère utiliser son log. La syntaxe est très simple : `gln(x)` égale $\log \Gamma(x)$.

Une autre fonction nécessaire au calcul des fonctions de répartition est la **fonction gamma incomplète**. La définition est comme suit :

$$P(a,x) = \frac{\gamma(a,x)}{\Gamma(a)} = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1}\, dt \qquad (a > 0).$$

Les fonctions $P(a,x)$ et $\gamma(a,x)$ sont connues toutes deux sous l'appellation de fonction gamma incomplète. C'est la première, $P(a,x)$, qui est disponible au moyen de l'expression `gammp(a,x)`. Les deux arguments doivent être positifs ; sinon une valeur nulle est obtenue.

Finalement, la **fonction bêta incomplète** est définie par la formule

$$I_x(a,b) = \frac{1}{B(a,b)} \int_0^x t^{a-1}(1-t)^{b-1}\, dt,$$

où la **fonction bêta** est définie en termes de la fonction gamma par la relation

$$B(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}.$$

La fonction $I_x(a,b)$ est évaluée par l'expression ***Ects*** `betafn(a,b,x)`. Attention : l'argument x doit être élément du segment $[0,1]$; sinon la valeur de la fonction est nulle.

Pour de plus amples renseignements sur les fonctions décrites ci-dessus, et sur d'autres fonctions utiles en mathématiques appliquées, voir le chapitre 6 de l'ouvrage de Press *et al* (1986).

2. Les Opérations Arithmétiques : Quelques Règles

Étant donné toutes les possibilités offertes par ***Ects*** en matière de calcul vectoriel et matriciel, mais aussi en manipulation des séries économiques, il y a obligatoirement des règles plus ou moins subtiles qui gèrent l'interprétation des expressions lues par le logiciel. On a vu à plusieurs reprises que ces expressions ont des sens différents dans les commandes `gen` et `mat`. Il n'est heureusement pas nécessaire de considérer `set` séparément, car son opération est *toujours* équivalente aux commandes suivantes :

```
set t1 = smplstart
set t2 = smplend
sample 1 1
gen ...
sample t1 t2
```

où le `gen` apparaît à la place de `set`, à une exception près : seule la commande `set` peut être utilisée pour affecter des valeurs scalaires aux *éléments* des vecteurs et matrices. Ainsi, `set A(3,4) =` ⟨*expression*⟩ est correcte, mais si l'on met `gen` ou `mat` à la place du `set`, il y a erreur de syntaxe.

En ce qui concerne les *fonctions* ***Ects***, les règles pertinentes ont été énoncées dans la section précédente. Nous allons considérer maintenant les règles qui s'appliquent aux opérations élémentaires d'addition, soustraction, multiplication, et division, notées `+`, `-`, `*`, et `/`. L'opération exponentielle, notée `^`, a été traitée dans la section 3.1.

Les opérations élémentaires sont toutes des **opérations binaires**. Autrement dit, elles ont deux arguments. Ces arguments sont chacun une matrice ; la première celle qui précède et la deuxième celle qui suit le symbole, `+`, `-`, `*`, ou `/`, de l'opération. À ce stade, il est inutile de distinguer les matrices qui sont des scalaires, des vecteurs, ou des matrices à plus d'une ligne et plus d'une colonne.

Les opérations sont effectuées selon les règles de **priorité** que nous avons évoquées dans la section 2.1. Les opérations d'inversion ou de transposition de matrices sont effectuées avant les opérations arithmétiques, de même pour l'opération exponentielle. Après, les multiplications et les divisions sont effectuées avant les additions et les soustractions. S'il y a des opérations Booléennes, symbolisées par `=`, `>`, ou `<`, elles sont effectuées en dernier. À chaque étape, il y a deux matrices, l'une de chaque côté du symbole de l'opération à effectuer.

Avant même que les opérations puissent être entamées, les matrices qui font l'objet des opérations doivent être construites par ***Ects***. La construction se fait à partir, soit d'une valeur numérique explicite, soit d'un symbole qu'***Ects*** peut reconnaître. Considérons d'abord les valeurs numériques. Imaginons qu'***Ects*** vient de lire le chiffre 4. Quelle sera la matrice construite pour représenter ce chiffre ? Dans une commande `mat`, la réponse est simple : ce sera une matrice 1×1, dont l'unique élément égale 4. Dans une commande `gen`, un vecteur colonne sera créé, à `smplend` composantes, chacune égale à 4, et ce, quelle que soit la valeur de `smplstart`.

Il y a deux sortes de symboles qui peuvent intervenir dans les expressions, outre les noms des fonctions ***Ects*** : les matrices définies antérieurement, et les macros. S'il s'agit d'une matrice, sous `mat` la matrice existante est reproduite telle quelle pour les fins de l'opération arithmétique. Sous `gen`, si la matrice existante a la forme 1×1, un vecteur colonne à `smplend` composantes est créé, et à chaque composante est affectée la valeur de l'unique élément de la matrice. Mais si la matrice existante n'est pas simplement un scalaire, une matrice est créée avec le même nombre de colonnes, et `smplend` lignes. Au début, tous les éléments de la matrice nouvelle sont nuls. Ensuite tous les éléments de l'ancienne matrice qui ne débordent pas les dimensions de la nouvelle sont transférés.

Si ***Ects*** rencontre un symbole qui représente une macro, le texte correspondant est trouvé, et une matrice est calculée selon les règles de la commande en vigueur. Ce calcul se fait tout à fait séparément, comme si le texte de la macro était entre parenthèses.

À ce stade, on peut raisonnablement se demander pourquoi la valeur de `smplstart` n'a pas été prise en compte. La raison est simplement qu'il est plus facile de la prendre en compte à la dernière minute, c'est-à-dire, au moment où l'expression entière qui fait l'objet de la commande est évaluée. Après cette évaluation, la matrice qui en résulte est utilisée soit pour créer soit pour mettre à jour la variable du membre de gauche de la commande. Ainsi, si l'on fait

```
gen z = x + y
```

le résultat de l'évaluation de l'expression `x + y` est un vecteur ou une matrice de `smplend` lignes. Notons $\boldsymbol{Z}$ cette matrice, de la forme `smplend` $\times k$.

Si la variable `z` n'existe pas, une matrice est maintenant créée, de la forme `smplend` $\times k$, chaque élément étant nul. Mais s'il existe déjà dans la mémoire de l'ordinateur une variable `z`, deux choses peuvent se produire. Si le nombre de lignes de `z` est inférieur à `smplend`, ou si le nombre de colonnes est inférieur à k, la variable existante est écrasée et une matrice nulle `smplend` $\times k$ est créée. Sinon, `z` garde ses anciennes dimensions et ses anciens éléments. Finalement, le bloc de la nouvelle matrice `z` qui va de la ligne dont l'indice est `smplstart` jusqu'à la ligne `smplend`, et de la première colonne jusqu'à la colonne k, est remplacé par le bloc correspondant de $\boldsymbol{Z}$.

Sous `mat`, les choses sont bien plus simples. Ni `smplstart` ni `smplend` ne sont prises en compte, et la matrice qui est le résultat de l'évaluation de l'expression n'est pas du tout modifiée.

Maintenant, on a vu comment ***Ects*** construit les matrices qui feront par la suite l'objet des opérations arithmétiques. Soit `A` une matrice de la forme $n \times m$, et `B` une matrice $p \times q$. Alors, si ***Ects*** doit évaluer `A + B` dans une commande `mat`, le résultat est une matrice $n \times m$. De même pour `A - B`. S'il manque des éléments de `B`, ils sont remplacés par des zéros ; s'il y en a de trop, ils sont perdus. Dans une commande `gen`, le résultat est une matrice `smplend` $\times m$. Encore une fois, tout élément manquant est remplacé par zéro, tout élément de trop est perdu. Mais voici une très grande différence par rapport à `mat`. Sous `gen`, *seule la première colonne* de `B` est prise en compte. Si l'on note $\boldsymbol{b}_1$ cette première colonne, et $\boldsymbol{a}_i$ les colonnes de `A`, $i = 1, \ldots, m$, alors `A + B` est une matrice dont les colonnes successives sont $\boldsymbol{a}_i + \boldsymbol{b}_1$, $i = 1, \ldots, m$.

Gardons les mêmes notations, et voyons `A*B`, d'abord sous `mat`. En calcul matriciel, il y a deux opérations de multiplication, la multiplication matricielle proprement dite, et la multiplication scalaire, où l'une des deux « matrices » est simplement un scalaire, et le « produit » est une matrice dont les éléments sont les produits ordinaires du scalaire et les éléments de la matrice non-scalaire. Pour faire face à ce double emploi de l'opération « multiplication »,

Ects examine d'abord les deux matrices `A` et `B` pour voir si l'une des deux est un scalaire. Dans ce cas, si c'est `A` le scalaire, la matrice `A*B` a la forme $p \times q$ de `B`. Si c'est `B`, la forme est celle de `A`. De la même façon, si dans l'expression `A/B B` est scalaire, le résultat a la forme de `A`. Par contre, si `A` est scalaire et `B` ne l'est pas, il y a erreur de syntaxe.

Si ni l'une ni l'autre matrice ne sont scalaires, l'évaluation de `A*B` est une matrice $n \times q$, calculée selon les règles de la multiplication matricielle. Comme dans le cas de l'addition, tout élément manquant est remplacé par zéro et tout élément de trop est perdu.

L'expression `A*B` dans une commande `gen` est interprétée assez différemment. Il ne s'agit plus de la multiplication matricielle, mais d'une multiplication, élément par élément, de deux séries. En fait, l'opération est tout à fait analogue à une opération d'addition. Le résultat a `smplend` lignes et m colonnes, les colonnes successives étant le résultat de la multiplication, élément par élément, de la colonne correspondante de `A` et la *première* colonne de `B`. Les colonnes de `B` après la première sont encore une fois perdues. Sous `gen`, la division obéit aux mêmes règles que la multiplication.

Finalement, les opérations Booléennes sont à considérer. Les règles sont les mêmes pour toutes ces opérations ; il suffit de traiter un seul cas, celui de `<`. L'évaluation de l'expression `A < B` sous `mat` est toujours un scalaire. Les éléments $(1, 1)$ *uniquement* des deux matrices sont pris en compte. Ainsi, si $a_{11} < b_{11}$, la valeur de `A < B` est 1, sinon 0. Sous `gen`, le résultat est une matrice `smplend` $\times\, m$, dont les éléments sont calculés à partir des colonnes successives de `A` et de la première colonne de `B`, exactement comme pour les autres opérations arithmétiques.

3. Les Fonctions Exceptionnelles

À la première section de ce chapitre, nous avons dressé la liste des fonctions ***Ects*** dont le comportement vis-à-vis de leurs arguments ne suit pas les règles générales énoncées plus haut. Il faut maintenant considérer ces fonctions exceptionnelles, sauf `random`, que l'on a traité dans la section 6.3.

La fonction `colcat` prend en général plusieurs arguments. (Voir la section 3.1.) Considérons l'expression

```
colcat(A1,...,Ak)
```

où la matrice `Ai`, pour $i = 1, \ldots, k$, a la forme $n_i \times m_i$. Le résultat est une matrice dont le nombre de lignes égale $\max_i n_i$ sous `mat` et $\min(\texttt{smplend}, \max_i n_i)$ sous `gen` ; et le nombre de colonnes égale $\sum_{i=1}^{k} m_i$ pour les deux commandes. Tout élément manquant devient zéro, et dans le cas de `gen` les lignes après `smplend` sont perdues. La syntaxe de `rowcat` est similaire. L'évaluation de

```
rowcat(A1,...,Ak)
```

est une matrice dont le nombre de lignes égale $\sum_{i=1}^{k} n_i$ et le nombre de colonnes égale $\max_i m_i$. L'effet de la fonction ne dépend pas du choix de la commande `mat` ou `gen`.

L'utilisation de la fonction `sum` sous `mat` n'a pour effet qu'une erreur de syntaxe. Sous `gen`, elle peut prendre plusieurs arguments. Le résultat de

```
sum(A1,...,Ak)
```

est une matrice dont la forme est identique à celle qui résulte de

```
colcat(A1,...,Ak)
```

Exceptionnellement, la valeur de `smplstart` est prise en compte par `sum`. Les lignes du résultat éventuellement comprises entre la première et celle dont l'indice est `smplstart` -1 sont nulles ; les lignes suivantes résultent de l'addition des lignes précédentes à partir de l'indice `smplstart`. Comme `sum`, la fonction `time` n'agit qu'à partir de la ligne `smplstart`. Cette fonction ne prend qu'un seul argument. Dans le résultat, tous les éléments de la ligne t sont majorés de t.

Le fonctionnement de `seasonal` a été exposé dans la section 2.2. Cette fonction, qui prend deux arguments, ne distingue pas entre `gen` et `mat`. La fonction `lag` a été traitée dans la section 2.1. Cette fonction donne lieu à une erreur de syntaxe si elle est utilisée sous `mat`. Le résultat de l'évaluation de `lag(k,A)`, où `A` est $n \times m$, est une matrice de la forme `smplend` $\times m$, dont les lignes de la première jusqu'à celle dont l'indice est `smplstart` -1 sont nulles, ainsi que celles qui correspondraient à une ligne inexistante de `A`.

Les fonctions `det`, `diag`, et `uptriang` ont été exposées dans la section 3.3. Leur fonctionnement est identique sous `gen` et `mat` ; elles prennent un seul argument chacune.

L'opération de convolution a été définie dans la section 6.4, où la fonction `conv` a été introduite. Quand l'expression `conv(A,B)` est évaluée, pour une matrice `A` de la forme $n \times m$, le résultat est une matrice `smplend` $\times m$ sous `gen`, et $n \times m$ sous `mat`. Seule la première colonne de `B` est utilisée. Dans la définition (16) de la convolution, la « première » observation est en fait l'observation `smplstart` sous `gen`.

La dernière fonction à regarder est `sort`, que nous n'avons pas vue jusqu'ici. C'est la fonction qui sert à **trier** (*sort* en anglais veut dire *trier*) les éléments d'un vecteur ou d'une matrice. Elle prend un seul argument, la matrice à trier. Le tri se fait par lignes, et par ordre croissant des éléments de la première colonne de la matrice. Ainsi, si `A` est une matrice $n \times m$, le résultat de `sort(A)` est une matrice $n \times m$ sous `mat`, et `smplend` $\times m$ sous `gen`. Dans le deuxième cas, les `smplstart` -1 premières lignes sont nulles, et le tri se limite aux lignes à partir de `smplstart`.

EXERCICES:

Comment trier les éléments d'un vecteur par ordre décroissant ?

4. Unix et le Code Source

Ects peut tourner non seulement sur les PC, mais aussi sur les stations de travail et autres ordinateurs sur lesquels le système d'exploitation Unix est implanté. Pour autant que je sache, il peut tourner sous d'autres systèmes d'exploitation encore : la condition nécessaire et suffisante est qu'un programme écrit en C++ puisse être compilé correctement.

La méthode traditionnelle de diffuser les logiciels destinés à une utilisation Unix est de mettre à la disposition des utilisateurs éventuels le **code source** du logiciel, accompagné d'un **Makefile**. Le code source est un programme, qui peut être très long, qui se sert d'un langage de programmation. Le langage C et le système d'exploitation Unix sont comme des jumeaux : ils ont vu le jour ensemble, et depuis ils ont travaillé ensemble en forte liaison. Seul un programme écrit en C, comme c'était le cas de la première version d'***Ects***, peut être sûr d'une compilation correcte sur toutes les machines Unix. Mais le langage C++ est la vague de l'avenir, à laquelle je n'ai pas pu résister. En fait, la programmation de la version actuelle du logiciel a été beaucoup facilitée par ce langage plus évolué. La plupart des stations de travail modernes peuvent accéder à un compilateur C++, ne serait-ce que parce qu'un tel compilateur est disponible gratuitement du Free Software Foundation du Massachusetts.[18]

Le code source d'***Ects*** est disponible sur demande sur disquette PC. Mais il est préférable de l'obtenir, si on le souhaite, par `ftp` anonyme. On se connecte au site `russell.cnrs-mrs.fr` par `ftp` (File Transfer Protocol). Un identificateur est demandé : on répond `ftp`. Ensuite un mot de passe est demandé, et il convient de taper son adresse email, ou, à défaut, son nom. La connexion une fois établie, on change de répertoire :

```
cd pub/ects
```

et dans ce répertoire se trouvent tous les fichiers pertinents. Le fichier `LISEZ.MOI` contient les descriptifs de ces fichiers.[19]

Outre les fichiers répertoriés au chapitre premier du manuel, les fichiers comprimés contiennent le code source d'***Ects***, dans un sous-répertoire `src`. Ce sous-répertoire contient également un `Makefile` qui permet d'automatiser la compilation. Sous Unix, il faudra se mettre dans le sous-répertoire `src`, et de taper

```
make Unix
```

[18] Note de la version 4 : Il s'agit du compilateur `gcc`, ou `g++` pour le C++, toujours gratuit et toujours très bon.

[19] Note de la version 4 : L'adresse donnée n'est plus correcte. Il est d'ailleurs préférable de se connecter par `http` à l'adresse suivante :

```
http://russell.vcharite.univ-mrs.fr/
```

Si tout se déroule normalement, après deux ou trois minutes, selon la vitesse de l'ordinateur, la compilation sera achevée, et le fichier exécutable `ects` sera créé. Ensuite, ce fichier exécutable peut éventuellement être déplacé dans un endroit public sur le disque de la station de travail, pour que tous les utilisateurs puissent y avoir accès. Si vous êtes l'un des heureux qui utilisent Linux, vous pouvez utiliser directement le fichier binaire `ects-2.2`.
Dans le `Makefile`, il y a d'autres cibles que `Unix`. En effet, on peut faire, sur PC,

```
make borland
```

pour obtenir la version `ects86.exe`. Ou bien

```
make dos
```

pour obtenir la version `ects.exe`. La première compilation requiert le compilateur C++ de Borland ; la seconde le compilateur DJGPP, qui est la version DOS du compilateur C++ du Free Software Foundation. Finalement, si l'on a accès au code source, on peut modifier le programme lui-même, si on le souhaite.

* * * *

Note de la version 4 : Ces détails n'ont probablement qu'un intérêt médiocre aujourd'hui. Le code source de la version 4 sera accompagné des instructions pertinentes à sa compilation sur plusieurs plateformes.

* * * *

Les difficultés de compilation sous Unix, et autres problèmes relevant du code source, sont à me signaler par courrier électronique, aux adresses suivantes :[20]

`russell@ehess.vcharite.univ-mrs.fr` (en France),
`Russell.Davidson@mcgill.ca` (au Canada).

Je vous souhaite bonne utilisation du logiciel.

[20] Note de la version 4 : J'ai changé les adresses qui figurent dans la version originale de ce volume, parce qu'elles ne sont plus actuelles. Celles données ici sont correctes en juillet 2004.

Bibliographie

Davidson, R., et J.G. MacKinnon (1993), (DM). *Estimation and Inference in Econometrics*, Oxford University Press, New-York.

Gouriéroux, Chr., et A. Monfort (1989). *Statistique et Modèles Économétriques*, Economica, Paris.

Press, W.H., B.P. Flannery, S.A. Teukolsky, et W.T. Vetterling (1986). *Numerical Recipes*, Cambridge University Press, Cambridge.[21]

[21] Une réédition de cet ouvrage a paru en 1992, en plusieurs versions, selon le langage de programmation utilisé. Les algorithmes de la première édition ont servi pour la version actuelle d'***Ects***, mais les intéressés sont vivement conseillés de se procurer la deuxième édition.

Index Général

Index *Ects*