Ects

Logiciel d'Économétrie
Version 3

Russell Davidson

Mars 1999

Ects, Version 3

AVANT PROPOS

Le but de ce petit manuel est de permettre aux utilisateurs d'***Ects*** de se servir des nouvelles fonctionnalités de la version 3 du logiciel. Vu que la version 3 existe depuis assez longtemps en version préliminaire, je précise ici que la version décrite dans ce manuel est la version 3.3. Il ne constitue pas un guide complet. La documentation de la version 2, parue en 1993 et encore disponible à la Faculté des Sciences Économiques de Marseille, s'applique tout aussi bien à la nouvelle qu'à l'ancienne version du logiciel, à quelques rares exceptions près. On se limite ici à noter ces exceptions, et à décrire les fonctionnalités qui n'étaient pas fournies par les versions précédentes.

Les principales nouveautés de cette version, du point de vue de l'utilisateur, concernent le graphisme, les estimations non linéaires, et la différentiation automatique. Pour le reste, on a largement étendu la gamme des fonctions connues à ***Ects***, et on a introduit une procédure d'intégration numérique. Pour les utilisateurs qui n'auront pas ce manuel sous la main à chaque instant, un système d'aide, permettant d'obtenir des descriptifs des commandes, fonctions, et variables employées par ***Ects***, est en cours d'élaboration.

Les deux premières versions d'***Ects*** se sont avérées utiles, non seulement dans un cadre purement pédagogique, mais aussi pour nombre d'applications pratiques. Dans cette nouvelle version, j'ai essayé de profiter des expériences des six dernières années pour créer un outil plus souple, plus facile à adapter aux besoins des problèmes divers que l'on rencontre dans la pratique de l'économétrie, et mieux adapté surtout à la mise en œuvre des simulations qui sont de plus en plus demandées par les techniques récentes.

L'évolution du matériel informatique exige une évolution parallèle des logiciels. Pourtant, ***Ects*** existe toujours pour les malheureux qui n'ont pas encore pu se libérer du vieux système d'exploitation qui est DOS. Son fonctionnement exige la présence sur le système d'un serveur DPMI. Il est rare qu'un tel serveur soit absent: sous Windows en particulier le serveur DPMI est fourni en standard. Mais pour bénéficier plus largement des avantages des ordinateurs modernes, il est préférable de travailler sous un système d'exploitation plus adéquat. Celui qui se recommande le plus aux utilisations scientifiques est sans doute Unix. Il existe une version entièrement gratuite d'Unix pour PC, et bientôt aussi pour les Macs, nommée Linux. C'est sous Linux que la présente version d'***Ects*** a été developpée, et je recommande vivement à toute personne, étudiant, enseignant, ou autre, pour qui un environnement informatique convivial, efficace, et propice au calcul scientifique est important, mais qui n'a pas encore fait la connaissance de Linux, de se renseigner sur ses nombreux avantages par rapport aux produits du monopoleur dont je ne peux que murmurer le nom: Microsoft.

Pour mes amis fanas de Linux, en France comme au Canada

Ce volume a été légèrement modifié par rapport à la documentation d'***Ects*** 3 de mars 1999, 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 1999, ce volume est protégé par la licence GFDL; voir la section pertinente de la documentation de la version 4.

Table des Matières

Introduction

Ce manuel est conçu comme la suite de la documentation de la version 2 d'***Ecta***, dont la lecture, au moins partielle, s'impose pour la bonne compréhension de ce qui suit. Cette documentation antérieure, qui porte la date de Mars 1993, est disponible à la Faculté des Sciences Économiques à Marseille.[1] Il est envisagé de mettre à disposition, un jour, une documentation complète en un volume. Ce jour-là, le présent manuel ne sera plus utile, mais, entre temps, il est nécessaire si on souhaite utiliser les nouvelles fonctionnalités de la version 3.

Pour le développement des versions précédentes d'***Ecta***, je m'étais appuyé sur l'ouvrage bien connu *Numerical Recipes* de Press, Flannery, Teukolsky, et Vetterling (1986). Malheureusement, la réédition de l'ouvrage, Press *et al* (1992), impose des conditions contraignantes sur l'emploi dans les logiciels des programmes qui y sont exposés. Par conséquent, je n'ai pas utilisé ces programmes dans la version actuelle d'***Ecta***. Heureusement, on trouve sans difficulté sur le Web d'autres sources des algorithmes dont on a besoin pour les calculs économétriques. J'ai profité en particulier de la Cephes Mathematical Function Library, une bibliothèque de routines écrites en C. L'auteur de la bibliothèque est Stephen L. Moshier (`moshier@world.std.com`), qui est aussi l'auteur d'un manuel, Moshier (1989). Les droits d'auteur sont réservés par M. Moshier, mais il permet l'utilisation libre de ses programmes. Je tiens à l'en remercier. L'algorithme de Décomposition par Valeurs Singulières (SVD), qui est au cœur des estimations par moindres carrés, a été fourni par Brian Collett (`bcollett@hamilton.edu`). Le code, en C, se base sur un algorithme écrit en Algol publié par Golub et Reinsch (1970).

La première version d'***Ecta*** est écrite en C, la deuxième est partiellement récrite en C++, et la version actuelle est complètement récrite en C++, à l'exception de quelques algorithmes purement numérique, où la version en C était déjà largement suffisante. Les auteurs du C++ s'étaient donné pour mission la création d'un C amélioré (« a better C ») – voir Stroustrup (1991). Je confirme pour ma part que la programmation en C++, même si elle exige une période d'apprentissage et de réorientation intellectuelle, est mille fois plus agréable que la programmation en C, et elle conduit en plus à des programmes beaucoup plus lisibles que ceux écrits en C. Une partie intégrante du C++ moderne est la bibliothèque standard, qui facilite énormément un grand

[1] Note de la version 4 : Le stock est épuisé, et on préfère maintenant diffuser le documentation sur l'Internet sous format PDF. Tous les volumes de documentation sont disponibles sur mon site à la Vieille Charité de Marseille.

nombre d'opérations courantes de la programmation. La bibliothèque standard n'est pas encore complètement réalisée dans le compilateur que j'utilise, et, en attendant la version définitive, je me suis servi largement d'une version « draft » ou provisoire, publié dans Plauger (1995). Le code de cette version est publié dans cet ouvrage : il ne peut être diffusé librement, mais il est permis de l'utiliser gratuitement dans les logiciels. On demande aussi que la phrase suivante soit imbriquée dans le logiciel et imprimée dans la documentation :

> Portions of this work are derived from The Standard C++ Library, copyright (c) 1995 by P.J. Plauger, published by Prentice-Hall, and are used with permission.

Cette phrase signifie que je me suis servi dans la création du logiciel du code de M. Plauger, publié dans l'ouvrage cité, et que l'auteur m'en donne la permission. Même si, un jour, je pourrai simplement employer une bibliothèque devenue vraiment standard, j'exprime ici ma reconnaissance à M. Plauger, dont le travail a beaucoup aidé le mien.[2]

Il reste à remercier tous les gens, réunis sous l'égide de la Free Software Foundation, qui ont créé le compilateur C++ de GNU. Il est difficile d'exprimer combien et à quel point les logiciels GNU ont transformé le monde de l'informatique scientifique. Même le système d'exploitation Linux n'existerait pas sans le support de ces logiciels. Je m'en sers tous les jours, et c'est grâce à eux que le développement d'***Ects*** a été possible. J'espère dans un avenir proche pouvoir rajouter ***Ects*** à la liste des logiciels mis gratuitement à la disposition de la communauté scientifique sous les conditions libres de GNU.[3]

Ce manuel n'est pas un manuel d'économétrie. C'est pourquoi, à plusieurs reprises, je fais référence à un vrai manuel d'économétrie quand il s'agit d'un point économétrique dont l'exposé n'aurait pas sa place ici. Le manuel en question est Davidson et MacKinnon (1993) : je me réfère désormais simplement à DM. Il est aussi nécessaire de temps en temps de se référer au manuel de la version 2 : la référence est simplement Man2.

[2] Note de la version 4 : La version 4 utilise en effet la librairie standard et n'a plus besoin de la bibliothèque de M. Plauger.

[3] Note de la version 4 : Ceci est le cas de la version 4.

Chapitre Premier
Fonctionnalités Nouvelles

Dans ce premier chapitre, nous verrons comment utiliser quelques-unes des fonctionnalités propres à la version 3 d'***Ects***. Quoique les estimations non linéaires ne soient traitées qu'au Chapitre 2, on parlera ici d'un nouvel outil qui facilite largement la mise en œuvre de ces estimations, à savoir, la différentiation automatique. Parmi les erreurs commises en écrivant des programmes ***Ects***, les plus fréquentes étaient sans aucun doute les erreurs dans la spécification des dérivées des fonctions. Or, pour lancer les commandes d'estimation non-linéaire, `nls`, `ml`, et `gmm`, il est nécessaire de préciser les dérivées, d'une fonction de régression, ou d'une fonction de logvraisemblance, ou de toute autre fonction objectif, par rapport aux paramètres qu'on cherche à estimer. Maintenant, on peut déléguer cette tâche à ***Ects***.

La plupart des logiciels d'économétrie disponibles dans le commerce se vantent de leurs capacités en matière de graphisme. Jusqu'ici, ***Ects*** était complètement défaillant à cet égard. Malgré les demandes, fréquentes et insistantes, de la part des étudiants, j'ai longtemps hésité sur la meilleure manière de réaliser une interface graphique pour ***Ects***. L'une des difficultés était simplement qu'il y a trop de possibilités. Veut-on afficher des graphiques à l'écran de l'ordinateur ? Dans ce cas, quelles sont les capacités de l'écran et de la carte vidéo ? Quelle résolution faut-il choisir ? Comment mettre l'écran en mode graphique ? Pour chaque réponse possible à ces questions, des inconvénients se manifestaient en grand nombre. Si on souhaite sortir des graphiques sur une imprimante, les questions et les inconvénients se démultiplient davantage. Finalement, il m'est venu à l'esprit que j'utilisais, chaque jour ou presque, un logiciel, nommé `gnuplot`, disponible librement et gratuitement, pour mes propres besoins en graphisme, que ce soit à l'écran ou à l'imprimante. Les auteurs de ce logiciel avaient en effet déjà résolu tous les problèmes auxquels je faisais face. Le développement du logiciel a été un effort coopératif, avec le concours de plusieurs programmeurs de talent : Ceux qui détiennent les droits de la version courante sont Thomas Williams et Colin Kelley.

Il arrive souvent dans le calcul d'un ensemble de fonctions, ou de dérivées, que l'on ait à recalculer plusieurs fois la même expression. C'est particulièrement vrai des dérivées qu'il faut fournir à `nls` et ses cousins. Refaire plusieurs fois la même chose est toujours fastidieux et peu efficace, mais, dans le cadre d'une estimation non linéaire, procédure lente de par sa nature, il y a un grand profit à tirer en éliminant des opérations répétées inutilement. Afin de faciliter

cette élimination, ***Ects*** permet maintenant de définir des **procédures**. Ces procédures sont constituées d'un bloc d'opérations qui servent à calculer, une fois pour toutes, tout ce dont on a besoin. Le calcul ne sera relancé que si les arguments de la procédure changent. On verra plus tard comment l'utilisation de la commande `procedure` permet d'accélérer certaines estimations.

Les progrès de l'informatique font que l'installation d'***Ects*** ne soit plus une longue histoire, plein d'écueils. Toutefois, il convient d'en parler brièvement, avant de nous lancer dans le vif de notre sujet.

1. Installation

La documentation de la version 2 d'***Ects*** décrit comment on installait le logiciel sur le matériel fruste d'il y a six ans. Aujourd'hui, quand tout le monde ou presque a accès à l'Internet, les choses sont différentes. De manière générale, ***Ects*** n'est plus diffusé sur disquette, même si rien ne l'interdit. Pour se procurer la version la plus récente du logiciel, connectez-vous à mon ordinateur par ftp anonyme.[4] Le nom de l'ordinateur est

```
russell.cnrs-mrs.fr
```

et, pour se connecter, on fait

```
ftp russell.cnrs-mrs.fr
```

Quand la machine répond, elle demande un identificateur de session. Vous répondez

```
ftp
```

ou bien

```
anonymous
```

Ensuite, on demande un mot de passe. À ce stade, on peut taper n'importe quoi, mais il est d'usage de donner son adresse email. Un message d'accueil bilingue s'affiche. Ensuite, vous vous mettez dans le répertoire `pub/ects3` par la commande

```
cd pub/ects3
```

Vous pouvez maintenant faire afficher la liste des fichiers contenus dans ce répertoire au moyen de la commande

```
dir
```

Parmi ces fichiers, il y aura `LISEZ.MOI` et `README`. Selon votre préférence linguistique, vous pourrez lire dans ces fichiers les informations sur la version

[4] Note de la version 4 :Les adresses de tous les ordinateurs de la Vieille Charité ont changé. En plus, il est aujourd'hui préférable de vous connecter par `http` au site

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

où vous trouverez des liens vers les répertoires d'***Ects*** 3 et d'***Ects*** 4.

la plus récente. Si vous préférez, vous pourrez vous connecter par `netscape`. L'URL qu'il faut lui donner est

`ftp://russell.cnrs-mrs.fr/pub/ects3`

Vous pourrez lire directement les fichiers `LISEZ.MOI` et/ou `README`, dans lesquels vous trouverez un mode d'emploi.

Les fichiers exécutables dont vous aurez besoin dépendent du système d'exploitation que vous utilisez et de la préférence linguistique. Pour Linux, vous trouverez `ects3`, la version anglophone, et `ects3fr`, la version francophone. Le système d'aide utilise aussi `ectshelp` et `ectshelpfr`. Le fichier `settexts` n'est nécessaire que si vous souhaitez modifier les réponses des autres fichiers exécutables, par exemple, si vous voulez rajouter des remarques aux descriptifs des commandes, ou si vous voulez changer de langue. Pour DOS/Windows, les fichiers exécutables portent les mêmes noms, plus la terminaison `.exe`: On trouve `ects3.exe`, `ects3fr.exe`, `ectshelp.exe`, `ectshelpfr.exe`, et `settexts.exe`.

Si jamais vous souhaitez utiliser ***Ects*** sous un autre système d'exploitation, il sera nécessaire de compiler le logiciel pour ce système à partir du code source. Le code source est fourni avec les exécutables, sauf la partie de la bibliothèque pour laquelle je me suis servi du code trouvé dans Plauger (1995). De toute manière, il serait plus simple de me joindre, par mail de préférence, avant de vous plonger dans une telle compilation.

Pour le graphisme, il faut le programme `gnuplot`. Ce programme est disponible librement et gratuitement, et il peut être compilé pour au moins autant d'architectures qu'***Ects***. Le fichier exécutable de `gnuplot` doit être trouvé dans le chemin d'accès (PATH) de l'utilisateur. En plus, il faut un répertoire nommé `tmp` dans le répertoire racine. Les systèmes Unix, y compris Linux, sont fournis d'office de ce répertoire, configuré de manière à donner à tout utilisateur la permission d'écriture. Sous d'autres systèmes d'exploitation, le répertoire doit être créé, s'il n'existe pas, de manière à ce que tout le monde puisse écrire (c'est-à-dire, créer des fichiers) dans ce répertoire.

Il est impossible d'éviter qu'un programme comme ***Ects*** se plante si les données qu'on lui fournit sont suffisamment bizarres pour donner lieu à des exceptions au niveau de l'unité d'arithmétique flottante. Pourtant, ceci ne devrait se produire que rarement. En revanche, tout programme donnant lieu à une faute ou erreur de segmentation est à me signaler, afin que je puisse éliminer le bogue qui l'a produite. Même si ce genre d'erreur est inoffensif sous Unix, d'autres systèmes d'exploitation, moins privilégiés, doivent être relancés dans certains cas suite à une telle erreur.

2. Les Graphiques

Avant de créer des graphiques, il serait prudent de vérifier la configuration de `gnuplot`. Tapez la commande

```
gnuplot
```

Si le système répond qu'il ne peut pas le trouver, on sait que le fichier exécutable de `gnuplot` n'est pas dans le chemin d'accès. Mais, s'il est trouvé, `gnuplot` affiche un message d'accueil, et ensuite une ligne qui aura l'aspect suivant :

```
Terminal type set to 'x11'
```

Ceci signifie que le mode graphique par défaut de `gnuplot` est `x11`. Sous Unix, le graphisme est confié au système de fenêtrage 'X', dont la version courante (en 1998) est la 11, d'où `x11`. Sous DOS/Windows, le mode graphique sera différent ; le plus souvent il sera SVGA. `gnuplot` est normalement en mesure de détecter automatiquement le mode graphique qui convient, mais en cas d'erreur on peut mettre dans la variable d'environnement `GNUTERM` la valeur qu'il faut. Pour connaître les modes graphiques reconnus par `gnuplot`, tapez

```
set term
```

après avoir lancé `gnuplot`. Il affichera une longue liste, et vous y verrez les noms de plusieurs imprimantes. On verra plus tard comment imprimer vos graphiques.

Création de graphiques

Pour créer des graphiques, on se sert de la commande `plot`. La syntaxe de cette commande est assez souple, et permet de créer et d'afficher un ou plusieurs graphiques à l'écran. Considérez à présent le fichiers de commandes `testplot.ect`, dont le contenu est

```
sample 1 100
read ols.dat y x1 x2 x3
ols y c x1 x2 x3
%set linestyle = 1
plot y fit res, (y fit), (y res) (y res fit) y fit, y res
quit
```

On fait appel à l'un de nos fichiers de données préférés, `ols.dat`. La commande `plot` dans ce programme fera afficher six graphiques, un à la fois. Quand on veut passer au graphique suivant, on appuie sur la touche Retour ou Entrée[5].

Le premier graphique serait donné par la commande

```
plot y fit res
```

qui donne lieu à trois tracés, de `y`, `fit`, et `res`.

* * * *

Après chaque commande `ols`, ***Ects*** met les valeurs ajustées de la régression dans la variable `fit`, et les résidus de la régression dans la variable `res`.

* * * *

[5] Sous DOS et/ou Windows, il se peut qu'on ait à appuyer deux fois sur la touche.

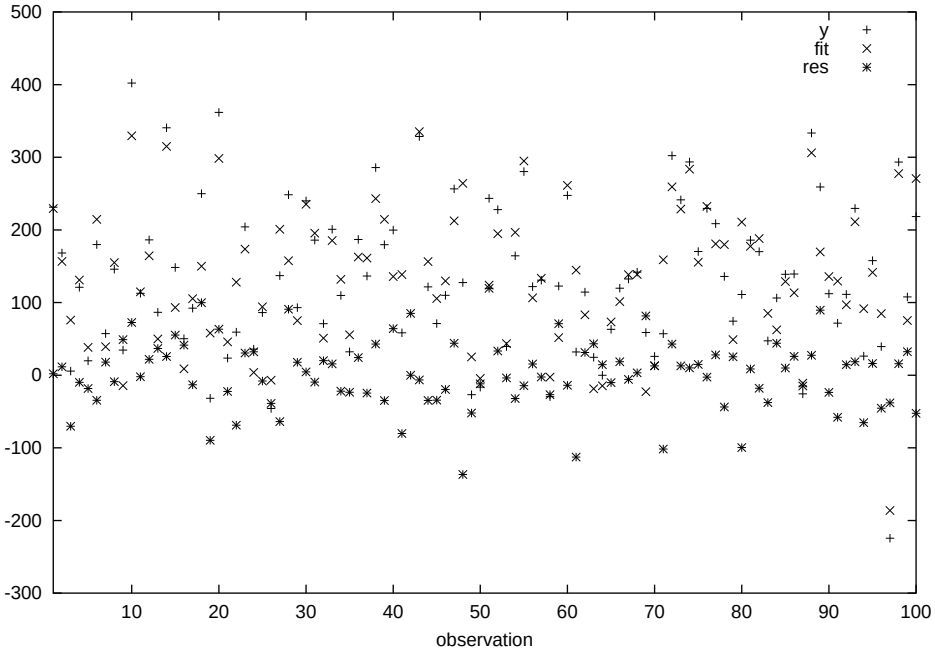

Figure 1 : Le résultat de `plot y fit res`

Pour chaque tracé, on a en abscisse l'indice de l'observation. En fait, le mot observation se trouve en dessous de l'axe horizontal, et l'on voit que les indices varient de 1 à 100, correspondant à la taille de l'échantillon. En ordonnée, on a la valeur de la variable en fonction de l'observation. Des couleurs différentes sont affectées à chaque variable, et, en haut à droite, sont affichés les noms des variables avec la couleur correspondante. Sans les couleurs, mais avec des symboles différents à la place des couleurs différentes, le graphique affiché est similaire à ce qu'on voit dans la Figure 1.

Notez que, juste au dessus de la commande `plot`, il y a une commande `set`, précédée du signe `%`. L'effet du `%` est le même que celui de la commande `rem`. C'est-à-dire que tout ce qui suit ce signe n'est pas lu par ***Ects***. On obtient le même effet d'une troisième manière, en mettant le signe `#` comme premier caractère d'une ligne de commande.

* * * *

Par « premier caractère », on entend le premier caractère autre qu'un espace blanc ou une tabulation horizontale.

* * * *

Les signes `%` et `#` sont utilisés à cette fin par plusieurs programmes, et on m'a demandé qu'il en soit autant pour ***Ects***.

Si on efface le `%`, que se passe-t-il ? Tant que la variable `linestyle` n'est pas définie, ou qu'elle vaut zéro, `gnuplot` met un gros point pour chaque observation. Mais si la valeur de `linestyle` est différente de zéro, des lignes droites sont tracées entre les observations. Si une variable varie de manière lisse d'une observation à la suivante, les droites sont plus jolies que les points.

En revanche, si une variable évolue de manière très irrégulière, les points peuvent mieux rendre compte de ce fait.

EXERCICES:

Génerez quatre ou cinq variables par la loi normale centrée réduite, en utilisant la fonction `random` :

```
gen y1 = random()
gen y2 = random()
⋮
```

et affichez-les avec la variable `linestyle` égale à 0 et à 1. De cette manière, vous verrez le comportement type d'un **bruit blanc**.

Faites la même chose avec d'autres variables que vous générerez de manière déterministe. Essayez par exemple

```
sample 1 180
gen y = sin(time(0)*PI/180)
plot y
```

Notez que la variable `PI` est automatiquement disponible dans la version 3.3 d'***Ects***. Sa valeur est, bien sûr, $\pi = 3,1415926535897932$.
Changez la taille de l'échantillon par

```
sample 90 180
```

et refaites

```
plot y
```

Vous verrez ainsi que le tracé n'est fait que pour l'échantillon en cours.

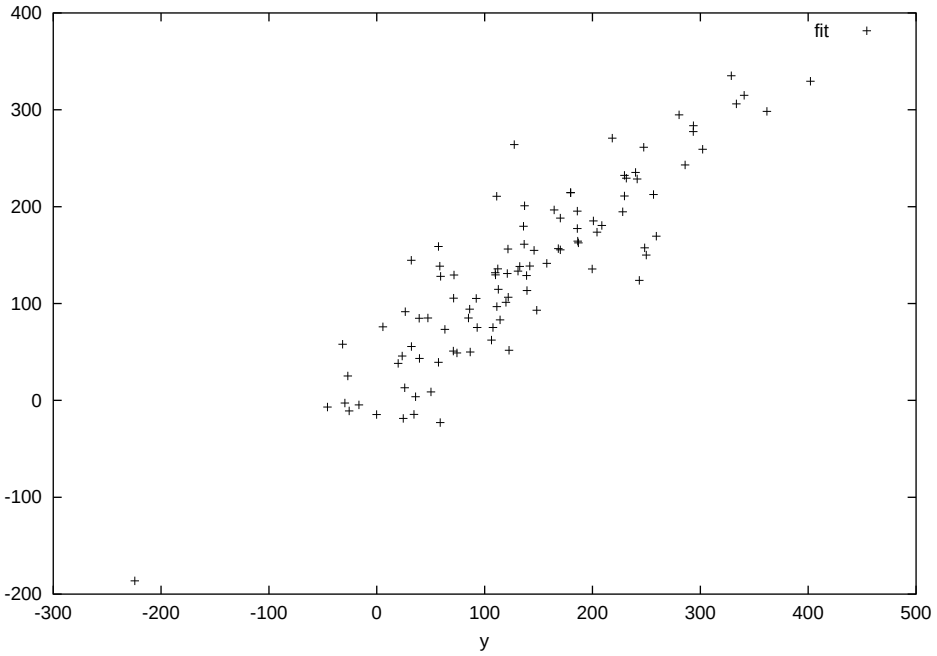

Figure 2 : `plot (y fit)` avec des points

Le deuxième graphique obtenu par la commande `plot` de `testplot.ect` serait donné, à lui seul, par la commande

```
plot (y fit)
```

Les parenthèses signifie que le graphique n'aura plus observation en abscisse, mais plutôt la première variable trouvée à l'intérieur des parenthèses, ici `y`. En ordonnée, on a `fit`, en fonction de `y`. L'échelle de variation des deux variables est calculée automatiquement par `gnuplot`. On voit le résultat de la commande dans la Figure 2 avec des points, et dans la Figure 3 avec des droites.

Le troisième graphique serait généré par la commande

```
plot (y res)
```

Ce graphique est construit suivant le même principe que le précédent.

Le quatrième graphique, qui serait généré par

```
plot (y fit res)
```

démontre qu'on peut avoir plusieurs tracés sur le même graphique, tout comme dans le cas où on a observation en abscisse. Le résultat de la commande est présenté dans la Figure 4.

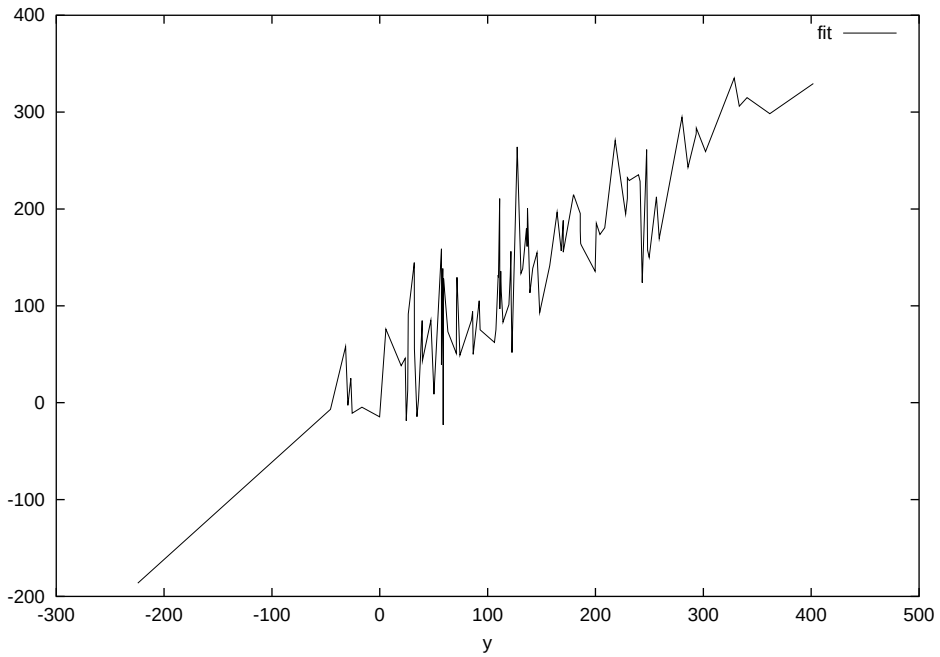

Figure 3 : `plot (y fit)` avec des droites

Il y a encore deux graphiques produits par `testplot.ect`. Ils seraient générés séparément par les commandes

```
plot y fit
plot y res
```

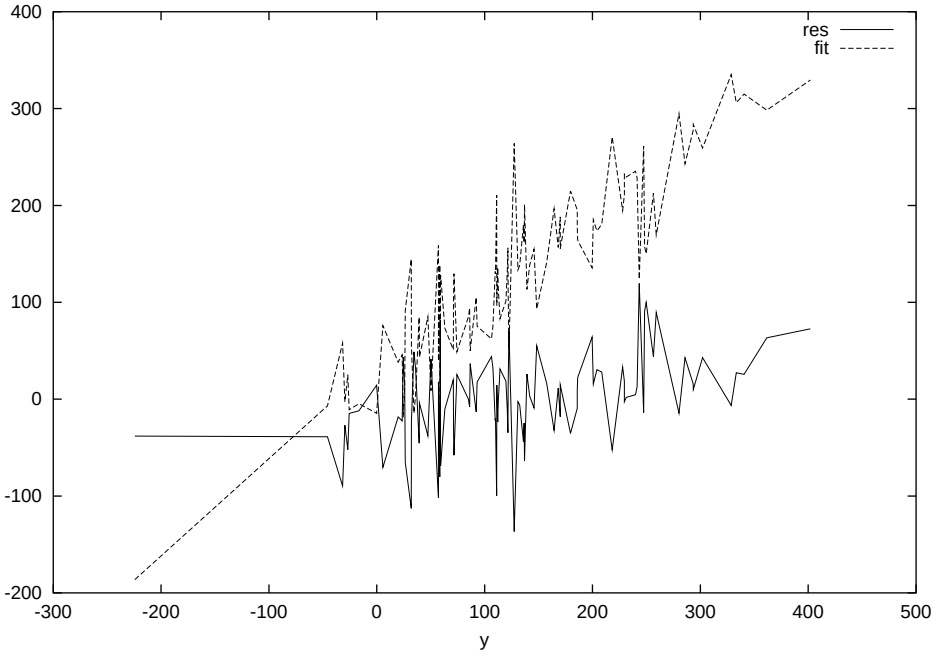

Figure 4 : `plot (y fit res)`

Pour les deux, observation est en abscisse, et en ordonnée on a `y` et soit `fit` soit `res`.
Pourquoi créer plusieurs graphiques au moyen d'une seule commande `plot`, plutôt que plusieurs commandes `plot` et un seul graphique par commande ? Le deuxième choix est évidemment entièrement possible, mais, dans ce cas, ***Ects*** fera autant d'appels à `gnuplot` que de commandes `plot` trouvés. Ceci n'est pas trop gênant, mais on remarquera que les graphiques s'affichent plus lentement.

Pour séparer les différents graphiques dans une commande `plot`, on se sert d'une virgule : `,` . Toutefois, si on ferme une parenthèse, il va sans dire que ce qui suit doit constituer un nouveau graphique, et, par conséquent, on peut, facultativement, se passer de la virgule. Ainsi, dans la commande

```
plot y fit res, (y fit), (y res) (y res fit) y fit, y res
```

on a mis une virgule apres `(y fit)`, mais pas après `(y res)` et `(y res fit)`. En revanche, si on supprime la virgule entre `y fit` et `y res` à la fin de la commande, on aura un seul graphique à la place de deux, et, dans ce graphique, la variable `y` sera tracée deux fois. Si l'on met des virgules à l'intérieur d'un ensemble de variables entre parenthèses, les conséquences sont imprévisibles, et, en toute vraisemblance, différentes de celles auxquelles vous vous attendiez ! Même si on peut se passer de la virgule après la *fermeture* d'une parenthèse, elle reste obligatoire avant l'ouverture d'une parenthèse. Si on l'oublie, en faisant, par exemple,

```
plot y fit res (y fit)
```

```
rem ERREUR !!
```

Ects sera perturbé par la non existence de la variable `(y`.

Jusqu'ici, tous les arguments qu'on a soumis à `plot` sont des vecteurs, représentant chacun une seule variable. On peut également avoir comme arguments des matrices à plus d'une colonne. Par exemple, si on fait

```
gen yfr = colcat(y,fit,res)
plot yfr
plot (yfr)
```

l'effet est identique à celui de

```
plot y fit res
plot (y fit res)
```

sauf pour les noms des variables affichés par `gnuplot`. À la place des noms explicites, `y`, `fit`, `res`, on aura le nom de la matrice, suivi de l'indice de la colonne : `yfr1`, `yfr2`, `yfr3`. On peut faire

```
plot yfr, (yfr)
```

pour éviter une interruption entre l'affichage des deux graphiques. La règle est simplement que tout argument matrice est décomposé en colonnes avant d'être traité par `gnuplot`.

EXERCICES:

Reprenez les données du fichier `ols.dat`, et programmez une boucle qui permet de faire l'estimation

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

pour tous les échantillons définis par

```
sample 1 n
```

pour `n` $= 10, \ldots, 100$. Au fûr et à mesure, sauvez les estimations paramétriques et les estimations $\hat{\sigma}^2$ dans cinq variables `a`, `b1`, `b2`, `b3`, et `s2`. Ensuite créez un graphique dans lequel vous tracez l'évolution des estimations en fonction de la taille de l'échantillon, $n = 10, \ldots, 100$.

Impression de graphiques

Les fonctionnalités graphiques d'***Ects*** ont été conçues surtout pour l'affichage à l'écran de l'ordinateur. Si on souhaite imprimer des graphiques, c'est plutôt `gnuplot` qui s'en occupe. Selon le système d'exploitation, il est plus ou moins facile de fournir à `gnuplot` les informations nécessaires.

Lors de l'exécution d'une commande `plot`, ***Ects*** crée des fichiers qu'il met dans le répertoire `tmp`. Le premier de ces fichiers est un fichier de commandes destiné à `gnuplot`. Ce fichier porte le nom de `gnuplot.gnu`. Ensuite, selon le nombre de graphiques demandés, il crée des fichiers de données, qui portent les noms `gnuplot.`n, pour $n = 0, 1, \ldots, m-1$, où on note m le nombre de

graphiques qu'il faut. Si on travaille sous un système d'exploitation multi-tâche, on peut lancer ***Ects***, et, pendant l'affichage à l'écran d'un graphique, aller dans le répertoire `tmp` pour retrouver et sauvegarder les fichiers pertinents en les copiant ailleurs. Il est important de faire ceci *pendant* l'exécution, parce que les fichiers seront effacés à la fermeture d'***Ects***. Alternativement, si on fait

```
set savegnu = 1
```

avant de lancer la commande `plot`, les fichiers ne seront pas effacés, et on pourra les retrouver dans le répertoire `tmp` après la fermeture d'***Ects***.

* * * *

Mais attention ! on ne retrouvera que les fichiers correspondant au *dernier* graphique. Les fichiers précédents, portant les mêmes noms que ceux du dernier, seront écrasés lors de la création de celui-ci.

* * * *

Si on travaille sous DOS, on n'aura pas la possibilité de faire quoi que ce soit pendant l'exécution d'***Ects*** sauf regarder le déroulement d'***Ects***, parce que DOS n'est pas un système multi-tâche. À moins d'utiliser la variable `savegnu` donc, il sera nécessaire de créer les fichiers soi-même. Même sous d'autres systèmes d'exploitation, il est utile de savoir comment procéder. Voyons ici, à titre illustratif, le fichier de commandes `gnuplot.gnu` créé par la commande

```
plot y fit res, (y fit), (y res) (y res fit) y fit, y res
```

qu'on a étudiée plus haut. Pour le premier graphique, on a

```
set xrange [1 :  100]
set xlabel "observation"
plot "/tmp/gnuplot.0" using 1:2 title 'y',\
"/tmp/gnuplot.0" using 1:3 title 'fit',\
"/tmp/gnuplot.0" using 1:4 title 'res'
pause -1
```

Le `xrange` définit les limites des valeurs de l'abscisse, ici la variable artificielle observation. En effet, l'échantillon est défini de 1 à 100. Le `xlabel` est l'étiquette (*label* en anglais) affichée en dessous de l'axe horizontal. La commande suivante mérite un peu d'attention. La syntaxe est la suivante :

```
plot "⟨nom de fichier⟩" using n:m title '⟨var⟩'
```

où à la place de ⟨*nom de fichier*⟩ on met le nom du fichier qui contient les données à afficher. Ce fichier, qui, dans notre cas, porte le nom de `/tmp/gnuplot.0`, contient plusieurs colonnes de chiffres. Chaque ligne du fichier correspond à un point du graphique. Les indices n et m sont les numéros des colonnes à utiliser pour les coordonnées horizontales (n) et les coordonnées verticales (m) des points qui constituent le graphique. Finalement, *var* est le nom ou l'étiquette à associer au tracé.

Comme on peut le constater, les informations nécessaires au tracé de la variable `y` sont contenues dans les colonnes 1 et 2 du fichier. La colonne 1 donne

la variable observation, qui est l'abscisse de tous les tracés. La colonne 2 correspond à la variable `y`. De même, le tracé de la variable `fit` est construit sur la base des colonnes 1 et 3 du fichier, et celui de `res` sur la base des colonnes 1 et 4.

Si on travaille sous DOS, on ne peut pas bénéficier de la construction automatique d'un fichier de données par ***Ects***, et on aura à construire le fichier `/tmp/gnuplot.0` directement. Ceci n'est pas du tout difficile. La variable observation peut être générée directement par

```
gen observation = time(0)
```

et le fichier de données par

```
write gnuplot.0 observation y fit res
```

Notez que, si on crée le fichier de données directement, il n'est pas nécessaire de le mettre dans le répertoire `tmp`.

La ligne

```
pause -1
```

est un signal à `gnuplot` qui le fait arrêter après l'affichage du graphique, jusqu'à ce qu'on appuie sur la touche Retour ou Entrée. Pour des raisons évidentes, cette ligne est inutile si on souhaite imprimer un graphique.

Dans le fichier `gnuplot.gnu` créé par ***Ects***, le caractère \ n'est pas utilisé, parce que toute la commande `plot`, du mot `plot` au mot `'res'`, n'occupe qu'une seule ligne. Il serait impossible d'imprimer une telle ligne dans ce manuel, et il est souvent souhaitable dans la pratique d'éviter des lignes trop longues. `gnuplot` permet d'étaler une commande sur plusieurs lignes si l'on met un \ à la fin de chaque ligne sauf la dernière de la commande, et c'est ce qu'on a fait ici pour que le programme soit plus lisible.

* * * *

La même pratique est possible avec ***Ects*** même : voir la section 2.5

* * * *

Les deux derniers graphiques, ceux qui peuvent être générés par

```
plot y fit
plot y res
```

sont créés en donnant à `gnuplot` les commandes suivantes :

```
set xrange [1 :  100]
set xlabel "observation"
plot "/tmp/gnuplot.4" using 1:2 title 'y',\
"/tmp/gnuplot.4" using 1:3 title 'fit'
pause -1
set xrange [1 :  100]
set xlabel "observation"
plot "/tmp/gnuplot.5" using 1:2 title 'y',\
"/tmp/gnuplot.5" using 1:3 title 'res'
pause -1
```

Ici le seul élément à noter est que les graphiques successifs se servent de fichiers de données différents. Même si les variables tracées dans ces graphiques sont les mêmes que celles du premier graphique, ***Ects*** crée de nouveaux fichiers.

Voyons maintenant les commandes `gnuplot` qui génèrent les autres graphiques, ceux où on a autre chose que la variable observation en abscisse.

```
set autoscale
set xlabel "y"
plot "/tmp/gnuplot.1" using 1:2 title 'fit'
pause -1
set autoscale
set xlabel "y"
plot "/tmp/gnuplot.2" using 1:2 title 'res'
pause -1
set autoscale
set xlabel "y"
plot "/tmp/gnuplot.3" using 1:2 title 'res',\
"/tmp/gnuplot.3" using 1:3 title 'fit'
pause -1
```

La commande

```
set autoscale
```

demande à `gnuplot` de déterminer lui-même les valeurs extrêmes des variables. Elle se substitue donc à la commande

```
set xrange ...
```

des graphiques précédents. Le `xlabel` n'est plus observation, mais plutôt `y`, parce que `y` est la variable en abscisse.

Afin de générer les fichiers `gnuplot.1,2,3` directement, il faut une manipulation supplémentaire qui n'est pas nécessaire quand la variable en abscisse est observation. On sait que chaque ligne d'un fichier de données correspond à un point d'un tracé. `gnuplot` construit ses tracés dans l'ordre des lignes du fichier. Par conséquent, si la première colonne du fichier de données n'est pas *triée*, par ordre croissant ou décroissant, les points du tracé seront construits d'une manière irrégulière. Si chaque point est representé par un point, ceci n'a pas d'importance, mais si les points successifs sont connectés par des lignes droites, ces lignes droites constitueront une excellente représentation du Chaos. Pour générer directement le fichier `gnuplot.1`, donc, on procède comme suit :

```
gen yf = colcat(y,fit)
gen yf = sort(yf)
write gnuplot.1 yf
```

La commande `sort` sert à trier les lignes de la matrice `yf` par ordre croissant des éléments de la première colonne, c'est-à-dire, les éléments de la variable en abscisse, `y`.

Résumons à présent la procédure qui permet d'imprimer un graphique. Un premier point est qu'il est toujours préférable de n'imprimer qu'un seul graphique à la fois. Sinon, on peut s'exposer à des difficultés de pagination difficiles à surmonter. Ici, on se limite à l'impression des graphiques créés par les deux commandes

```
plot y fit res
plot (y fit res)
```

Ces graphiques apparaissent dans les Figures 1 et 4. D'abord, pour créer les fichiers de données, on fait

```
gen observation = time(0)
gen data = colcat(observation,y,fit,res)
write gnuplot.0 data
gen data = colcat(y,fit,res)
gen data = sort(data)
write gnuplot.1 data
```

Ensuite, on se sert de n'importe quel éditeur de texte pour créer le fichier `gnuplot.gnu`, dont le contenu sera similaire à

```
set xrange [1 :  100]
set xlabel "observation"
set term postscript eps
set out "fig1.ps"
plot "gnuplot.0" using 1:2 title 'y',\
"gnuplot.0" using 1:3 title 'fit',\
"gnuplot.0" using 1:4 title 'res'
```

Notez bien les deux commandes

```
set term postscript eps
set out "fig1.ps"
```

La première est la commande qui sert à sélectionner le format du fichier qui sera envoyé par la suite à l'imprimante. Le choix que j'ai fait ici est celui qui convient pour l'impression d'une figure dans ce manuel, à savoir le PostScript encapsulé. D'autres choix possibles auraient été

```
set term hpljii 300
```

qui signifie le format requis par une imprimante HP LaserJet II, à une résolution de 300 points par pouce, ou bien

```
set term latex
```

qui donnerait un fichier en format LaTeX. Ce ne sont là que deux exemples : on trouve des centaines de possibilités dans la documentation de `gnuplot`.

La commande

```
set out "fig1.ps"
```

demande à gnuplot de créer un fichier `fig1.ps` et de l'utiliser comme fichier de sortie. Étant donné que j'avais demandé un fichier PostScript, je donne le nom d'un tel fichier, avec la terminaison `.ps`.

Ensuite on lance `gnuplot` avec pour argument le nom du fichier de commandes :

```
gnuplot gnuplot.gnu
```

Après l'exécution de cette commande, on devrait trouver le fichier de sortie dans le répertoire courant. Pour moi, ce fichier porterait le nom de `fig1.ps`. La dernière étape est bien sûr d'envoyer le fichier ainsi créé à l'imprimante. La commande pertinente dépend du système d'exploitation. Sous DOS, par exemple, elle serait

```
copy /b fig1.ps prn
```

Pour que ceci marche correctement, il faut, bien entendu, que le format du fichier corresponde à l'imprimante. Pour `fig1.ps`, par exemple, il faudrait une imprimante PostScript.

La procédure pour l'impression du graphique créé par

```
plot (y fit res)
```

est très similaire. Les données triées existent déjà dans le fichier `gnuplot.1`. Seul le contenu de `gnuplot.gnu` est à changer. On a

```
set autoscale
set xlabel "y"
set term postscript eps
set out "fig2.ps"
plot "gnuplot.1" using 1:2 title 'res',\
"gnuplot.1" using 1:3 title 'fit'
```

Après l'exécution de ces commandes par `gnuplot`, le fichier `fig2.ps` peut être envoyé à une imprimante PostScript.

EXERCICES:

Créez un fichier contenant les données nécessaires à un tracé du graphique de la fonction $\sin(x)$ pour $x \in [0, 2\pi]$. Si vous avez accès à une imprimante, écrivez un programme `gnuplot` qui permettra d'imprimer le graphique. Sinon, modifiez le programme de manière à ce qu'il affiche le graphique à l'écran de l'ordinateur.

3. La Différentiation Automatique

Les données manipulées par ***Ects*** sont des tableaux de chiffres, qui représentent des scalaires, vecteurs, et matrices. Quand on calcule une dérivée, c'est la dérivée d'une *fonction*. Or, ***Ects*** ne permet pas de représenter les fonctions. Toutefois, les **macros** créées par la commande `def` permettent de représenter des expressions algébriques susceptibles d'être évaluées par ***Ects***. C'est sur la base de cette fonctionnalité que le mécanisme de différentiation automatique de la version 3 d'***Ects*** est construit.

Prenons un exemple très simple. On sait que la dérivée de la fonction x^2 est $2x$. On pourrait espérer qu'une construction comme

```
diff(x^2,x)
```

représenterait la dérivée de x^2 par rapport à x. En effet, si l'on fait

```
set x = 4
set y = diff(x^2,x)
show y
```

la réponse affichée par ***Ects*** est

```
y = 8.000000
```

On trouve que `y` égale deux fois la valeur de `x`.

Il est important de comprendre que les dérivées calculées par ***Ects*** sont obtenues par des manipulations symboliques. Si, par exemple, on exécutait

```
set x = 4
set x2 = x^2
set y = diff(x2,x)
show y
```

la réponse serait

```
y = 0.000000
```

parce que `x2` de dépend pas de `x`. Il faut l'*expression* `x^2`, donnée en termes de `x`, pour que la dérivée soit autre chose que zéro.

On peut mettre n'importe quelle fonction de x à la place de x^2. Quelques exemples : le programme

```
set y = diff(x^4 - 3*x^3 + 2*x^2 + 3*x - 4,x)
show y
set y = diff(sin(x),x)
show y
set y = diff((sin(x))^2 + (cos(x))^2,x)
show y
```

donne les réponses consécutives :

```
y = 131.000000
y = -0.653644
y = 0.000000
```

Calculons :

$$\frac{d}{dx}(x^4 - 3x^3 + 2x^2 + 3x - 4) = 4x^3 - 9x^2 + 4x + 3.$$

Cette expression, évaluée en $x = 4$, vaut $4.64 - 9.16 + 4.4 + 3 = 131$, qui est la première réponse. Si l'on fait

```
set y = cos(x)
show y
```

on peut vérifier la deuxième réponse : en effet on obtient -0.653644. La dernière réponse est la conséquence de l'identité trigonométrique

$$\sin^2(x) + \cos^2(x) = 1.$$

La dérivée de la constante 1 égale 0, conformément à la réponse d'***Ecta***.

Dans tous ces calculs, ***Ecta*** manipule les symboles de l'expression à différentier afin de trouver une représentation symbolique de la dérivée, et, à la fin seulement, il évalue cette représentation symbolique selon les règles de la commande courante. Dans tous les exemples considérés jusqu'ici, cette commande a été `set`. On aurait pu tout aussi bien employer `gen`. Par exemple, si on exécute

```
sample 1 100
gen x = 0.1*time(0)
gen y = diff(chisq(x,2),x)
plot (x y)
```

on fera afficher un graphique de la densité de la loi du khi-deux (χ^2) à deux degrés de liberté sur l'intervalle $]0, 10]$. Le processus est le suivant. Dans un premier temps, ***Ecta*** détermine que la dérivée de la fonction de répartition du $\chi^2(2)$ s'exprime en termes de la **fonction gamma incomplète**.

* * * *

> Voir la section 7.1 de la documentation de la Version 2 d'***Ecta*** pour plus d'informations sur cette fonction. On en parlera plus loin dans la section 4.1, lors de l'exposé des nouvelles fonctions reconnues par ***Ecta***.

* * * *

Une représentation symbolique de cette fonction est créée. Dans un deuxième temps, cette représentation est passée à la commande `gen`, qui génère un vecteur `y` dont les composantes sont les valeurs de la densité du $\chi^2(2)$ (la dérivée de la fonction de répartition) en les composantes correspondantes de `x`.

On peut aussi utiliser la fonction `diff` dans une commande `mat`. Considérez le programme suivant :

```
sample 1 100
read ols.dat y x1 x2 x3
ols y c x1 x2 x3
set a = coef(1)
set b1 = coef(2)
set b2 = coef(3)
set b3 = coef(4)
def residu = y - a*c - b1*x1 - b2*x2 - b3*x3
def critere = residu'*residu
mat db1 = diff(critere,b1)
show db1
quit
```

On se sert encore une fois des données du fichier `ols.dat`. Après avoir fait tourner la régression, on sauve les paramètres estimés et on définit une macro

`residu` qui est l'expression algébrique des résidus de la régression. Notez bien qu'une *macro* constitue pour ***Ects*** une *expression*, plutôt qu'une matrice numérique. Une seconde macro, `critere`, sert à définir la fonction critère dont la minimisation donne l'estimateur des moindres carrés, c'est-à-dire la somme des carrés des résidus. L'expression

```
diff(critere,b1)
```

donne lieu à une représentation de la dérivée de la fonction critère par rapport à l'un des paramètres, `b1`. Si on explicitait cette représentation, on aurait quelque chose comme

```
-x1'*(y-a*c-b1*x1-b2*x2-b3*x3) - (y-a*c-b1*x1-b2*x2-b3*x3)'*x1
```

Si cette expression est évaluée par une commande `mat`, le résultat est une matrice 1×1, c'est-à-dire, un scalaire. La valeur du scalaire devrait être zéro, grâce aux conditions du premier ordre de la minimisation. Si on exécute le programme ci-dessus, on verra que, en effet, `db1` égale 0.

Une remarque s'impose. Dans la définition de la macro `residu`, on fait appel explicitement au vecteur constant `c`. Sous `gen`, on aurait pu écrire simplement

```
y - a - b1*x1 - b2*x2 - b3*x3
```

sans le vecteur `c`. Mais, sous `mat`, il y aurait une incohérence due au mélange des vecteurs avec le scalaire `a`. Dans le processus de différentiation automatique, la dérivée de `a` par rapport à `a` est 1, un scalaire, mais la dérivée de `a*c` est `c`, un vecteur, conformément aux règles du calcul matriciel.

EXERCICES:

Exécutez le programme ci-dessus afin de vérifier que `db1` = 0. Évaluez la macro `residu` par une commande `gen` et vérifiez que le résultat est identique au vecteur `res` créé par la commande `ols`. Vérifiez aussi que ce résultat est inchangé si la macro `residu` est évaluée par `mat`. Ensuite changez la définition de la macro `residu` en imposant la contrainte `b3` = `b1`. Évaluez la macro `critere` (par `mat`) : la valeur sera supérieure à la variable `ssr`, parce que cette variable contient la valeur minimisée de la somme des carrés des résidus. Finalement, évaluez `db1` de nouveau, et vérifiez que sa valeur est maintenant différente de zéro.

Il existe deux autres fonctions qui se servent de la différentiation automatique. Ces fonctions, `grad` et `hess`, ne sont disponibles que dans les commandes `mat`. Elles servent à calculer, respectivement, le **gradient** et la **hessienne** d'une expression matricielle, qui doit être un scalaire, même si elle est composée d'éléments non scalaires. En étendant l'exemple ci-dessus, considérez le code suivant :

```
mat gr = grad(critere,a,b1,b2,b3)
sample 1 4
show gr
mat H = hess(critere,a,b1,b2,b3)
mat H = 2*H inv
show H XtXinv
```

La macro `critere` est toujours la somme des carrés des résidus, sous forme matricielle. La valeur est pourtant scalaire, parce que `critere` définit un produit scalaire. La commande

```
mat gr = grad(critere,a,b1,b2,b3)
```

demande le calcul du vecteur, 4×1, des dérivées partielles de la fonction somme des carrés des résidus par rapport aux quatre variables, `a`, `b1`, `b2`, et `b3`. La commande

```
mat H = hess(critere,a,b1,b2,b3)
```

demande le calcul de la matrice 4×4 des dérivées secondes de la fonction par rapport aux mêmes variables.

L'évaluation du gradient de la somme des carrés des résidus en l'estimateur des moindres carrés devrait être nulle, à cause des conditions du premier ordre de la minimisation. Si on exécute les commandes ci-dessus, on peut vérifier que tel est le cas. Quant à la hessienne, en termes des notations habituelles, on a à construire la matrice des dérivées du gradient

$$-2\boldsymbol{X}^\top(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}).$$

et cette matrice s'écrit simplement $2\boldsymbol{X}^\top\boldsymbol{X}$. Deux fois l'inverse de cette matrice est $(\boldsymbol{X}^\top\boldsymbol{X})^{-1}$, et les deux matrices `H` et `XtXinv` sont donc identiques.

Exercices:

Si on dérive une expression par rapport à une variable dont elle ne dépend pas, le résultat doit être zéro. ***Ects*** permet de dériver les expressions par rapport à des variables qui n'existent même pas. Calculez les dérivées des macros `residu` et `critere` par rapport à une variable `b4`. Dans tous les cas, le résultat sera un simple scalaire, 0. Ensuite, créez la variable `b4`, par une commande `set`, par exemple. Vérifiez que les résultats de la différentiation sont inchangés. Changez la valeur de `b3`, afin que le gradient de `critere` ne s'annule plus. Ensuite, calculez le gradient et la hessienne de `critere` par rapport aux variables `a`, `b1`, et `b4`. Là, on obtiendra un vecteur 3×1 et une matrice 3×3, mais avec des éléments nuls correspondant aux dérivées par rapport à `b4`.

On peut calculer les dérivées secondes, troisièmes, *etc.*, en faisant appel à `diff` plus d'une fois. On fait, par exemple,

```
mat d2f = diff(diff(f,x),x)
```

pour calculer la dérivée seconde d'une macro `f` par rapport à `x`. Calculez les éléments de la hessienne de `critere` de cette manière, et vérifiez que les résultats sont les mêmes que ceux donnés par `hess`.

L'utilisation la plus importante des fonctions `diff`, `grad`, et `hess` est dans le contexte des estimations non linéaires. On en parlera plus longuement au chapitre suivant.

* * * *

Il existe une dernière commande associée à la différentiation automatique, `differentiate`. La syntaxe qu'elle utilise est

`differentiate` <*expression*> <*variable*>

où l'<*expression*> à différentier peut comprendre des macros, et la différentiation se fait par rapport à la <*variable*>. Cette commande ne donne rien dans le fichier de sortie, mais il affiche des choses à l'écran. Si je ne précise pas davantage, c'est parce que ces choses-là ne se lisent pas simplement, étant écrites en une représentation interne à ***Ects***. Cette commande est beaucoup plus utile pour moi que pour vous !

* * * *

4. L'Intégration Numérique

Il existe aujourd'hui des algorithmes très puissants permettant l'intégration symbolique d'un grand nombre de fonctions. Ces algorithmes sont imbriqués dans des logiciels comme Mathematica et Maple, qui effectuent des opérations symboliques sur des expressions algébriques. Jusqu'ici, au moins, la mission d'***Ects*** ne comprend pas de telles opérations. Il n'est pourtant pas très difficile de calculer *numériquement* les valeurs de certaines intégrales, et la version 3.3 d'***Ects*** est dotée d'une fonction `int` qui effectue ces calculs.

Comme la différentiation, l'intégration opère sur des *fonctions*, qui doivent être représentées pour ***Ects*** par des *expressions*, soit explicites, soit sous forme de macro(s). À la différence de `diff`, la fonction `int` ne peut être utilisée que dans une commande `set`. Ceci s'explique par le fait que le résultat d'une intégration numérique est un scalaire. Le fichier `integral.ect` contient plusieurs exemples de l'utilisation de la fonction `int`.

```
#set showint = 1
set I = int(1,0,3,x)
show I
set I = int(1,3,0,x)
show I
set I = int(x,0,3,x)
show I
set I = int(-x,3,0,x)
show I
set I = int (x^2,0,3,x)
show I
set I = int (x^2,-3,0,x)
show I
set I = int(sin(x),0,PI,x)
show I
set I = int(cos(x),0,PI,x)
show I
set I = int(asin(x),0,1,x)
set J = PI/2-1
show I J
```

```
set I = int(diff(chisq(z,8),z),0,1,z)
set J = chisq(1,8) - chisq(0,8)
show I J


set maxintiter=10
set INTTOL=1E-8
set E = 2*int(z*(1-chisq(z,5)),0,200,z) \
  - (int(1-chisq(z,5),0,100,z))^2
show E
quit
```

On voit que la fonction prend quatre arguments, qui s'interprètent comme suit :

`int(`⟨*expression*⟩, ⟨*a*⟩, ⟨*b*⟩, ⟨*symbole*⟩`)`

où ⟨*expression*⟩ est l'expression à intégrer, ⟨*a*⟩ est la limite inférieure de l'intégrale, ⟨*b*⟩ la limite supérieure, et ⟨*symbole*⟩ le nom de la variable par rapport à laquelle on intègre.

Par conséquent, on peut interpréter la commande

```
set I = int(1,0,3,x)
```

comme une demande d'évaluation de l'intégrale

$$\int_0^3 1 \, dx,$$

intégrale dont la valeur numérique est 3. En effet, après la commande

```
show I
```

la réponse d'***Ects*** est

```
I = 3.000000
```

De même, la commande

```
set I = int (x^2,0,3,x)
```

évalue l'intégrale

$$\int_0^3 x^2 \, dx$$

et donne la réponse

```
I = 9.000000
```

On vérifie aisément que cette réponse est correcte.

On sait que l'intégration et la différentiation sont des opérations inverses. Les commandes

```
set I = int(diff(chisq(z,8),z),0,1,z)
set J = chisq(1,8) - chisq(0,8)
```

servent à démontrer ce fait. Soit $F_8(\cdot)$ la fonction de répartition d'un χ^2 à 8 degrés de liberté. La valeur de cette fonction en z s'écrit en ***Ects*** comme `chisq(z,8)`. Alors,

$$\int_0^1 F_8'(z)\,dz = \Big[F_8(z)\Big]_{z=0}^{z=1} = F_8(1) - F_8(0),$$

où la dernière expression s'écrit comme `chisq(1,8) - chisq(0,8)`. En fait, $F_8(0) =$ `chisq(0,8)` $= 0$.

Les commandes

```
set maxintiter=10
set INTTOL=1E-8
set E = 2*int(z*(1-chisq(z,5)),0,100,z) \
  - (int(1-chisq(z,5),0,100,z))^2
```

illustrent comment on peut modifier le fonctionnement de la fonction `int`. On veut que la variable `E` contienne la valeur de la variance d'un χ^2 à 5 degrés de liberté.

* * * *

La variance d'un χ^2 à n degrés de liberté est $2n$. Si le calcul est correct, on devrait obtenir une valeur de 10.

* * * *

Si l'on note F_5 la fonction de répartition de cette loi, on a

$$\begin{aligned}
\mathrm{Var}(\chi^2_5) &= E\Big((\chi^2_5)^2\Big) - \Big(E(\chi^2_5)\Big)^2 \\
&= \int_0^\infty z^2\,F_5'(z)\,dz - \left(\int_0^\infty z\,F_5'(z)\,dz\right)^2 \\
&= 2\int_0^\infty z\big(1-F_5(z)\big)\,dz - \left(\int_0^\infty \big(1-F_5(z)\big)\,dz\right)^2.
\end{aligned}$$

La dernière ligne résulte d'une intégration par parties : Pour la deuxième intégrale, notez que

$$\frac{d}{dz}\Big(z\big(1-F_5(z)\big)\Big) = 1 - F_5(z) - zF_5'(z)$$

et que

$$\Big[z\big(1-F_5(z)\big]_{z=0}^{z=\infty} = 0,$$

parce que $F_5(\infty) = 1$. La première intégrale se justifie de la même manière. L'expression fournie à la commande `int` est donc correcte, sauf qu'il a fallu remplacer l'infini par une valeur finie, ici 100.

L'intégration numérique est réalisée par un algorithme itératif. Cet algorithme est contrôlé par l'action des variables `maxintiter`, qui détermine le nombre

maximal d'itérations, et `INTTOL`, qui détermine le critère de convergence employé par l'algorithme. Les itérations s'arrêtent quand la valeur calculée de l'intégrale diffère de moins de `INTTOL` d'une itération à la suivante, ou bien quand `maxintiter` itérations ont été effectuées, même si la convergence n'est pas encore acquise. Dans le cas de la variance du χ^2, on voit en faisant tourner `integral.ect` que les choix de `maxintiter` et de `INTTOL`, ainsi que de l'infini, permettent d'obtenir la bonne réponse de 10, du moins à la précision du résultat affiché.

La première ligne du fichier `integral.ect` n'a pas d'effet, parce qu'elle commence par le caractère `#`. Si on enlève ce caractère, on verra que les valeurs calculées par les itérations successives s'affichent à l'écran. Ceci est la conséquence de la valeur non nulle de la variable `showint`.

Exercices:

Jouez avec les valeurs de `maxintiter` et `INTTOL` jusqu'à ce que la réponse ne soit plus la bonne. Vous pouvez faciliter la tâche en regardant les valeurs intermédiaires affichées si la variable `showint` est différente de zéro.

Chapitre 2
Les Estimations Non-linéaires

1. Introduction

La version 3 d'***Ects*** offre un plus grand choix de procédures d'estimation non linéaire que son prédécesseur, qui n'avait que trois commandes permettant d'effectuer une telle estimation, `nls`, `ml`, et `gmm`. Ces trois commandes existent toujours, mais elles sont complétées par six nouvelles commandes, qui étendent les possibilités d'estimation non linéaire, et à facilitent des procédures dont la mise en œuvre était auparavant un peu difficile.

2. Moindres Carrés Non-linéaires

La commande `nls` n'a changée que très peu depuis la version 2 du logiciel. La principale différence du point de vue de l'utilisateur est la possibilité de se servir de la différentiation automatique, évitant ainsi les erreurs, parfois trop fréquentes, dans la formulation des dérivées de la fonction de régression sous forme ***Ects***.

Nous résumons rapidement ici la syntaxe de la commande `nls`, et quelques détails de son fonctionnement. Le modèle économétrique qui fait l'objet d'une estimation par `nls` est le **modèle de régression non linéaire**, qui s'écrit sous la forme

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

où $\boldsymbol{y}$ est la variable dépendante, $\boldsymbol{u}$ est le vecteur d'aléas, et $\boldsymbol{x}(\boldsymbol{\beta})$ est le vecteur des fonctions de régression : voir les chapitres 2 et 3 de DM pour de plus amples informations sur ce modèle, ainsi que le chapitre 4 du manuel de la version 2 d'***Ects***. Si nous reprenons à présent le modèle qui sert d'exemple dans Man2, la fonction de régression associée à l'observation t s'écrit comme

$$\alpha + \beta x_{t1} + \frac{1}{\beta} x_{t2}.$$

La commande utilisée dans Man2 pour l'estimation de ce modèle est :

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

Si l'on préfère se servir de la différentiation automatique, une autre manière de procéder est comme suit :

```
def fctreg = alpha + beta*x1 + (1/beta)*x2
nls y = fctreg
deriv alpha = diff(fctreg,alpha)
deriv beta = diff(fctreg,beta)
end
```

On définit d'abord une macro `fctreg` qui représente la fonction de régression. Ensuite, dans les lignes `deriv`, il suffit de faire appel à la fonction `diff` pour que la dérivée soit calculée automatiquement par ***Ects***. Les résultats seraient identiques si la macro n'était pas définie, et si à sa place on mettait `alpha + beta*x1 + (1/beta)*x2`. Il est clair qu'une macro est beaucoup plus pratique du moment que la fonction de régression est un peu compliquée.

Avant la version 3, il était obligatoire que les paramètres (ici `alpha` et `beta`) soient définis, par des commandes `set` par exemple, avant de lancer la commande `nls`. La raison en était qu'il fallait aussi affecter à chacun des paramètres une valeur de départ. Il est maintenant admissible de se passer de l'initialisation des paramètres, auquel cas leurs valeurs de départ seront nulles. Ceci s'applique également à toutes les estimations non linéaires, non seulement aux estimations `nls`.

Il est maintenant possible de suivre le déroulement du processus de minimisation ou maximisation de la fonction critère d'une estimation non linéaire. Si la valeur de la variable `showprogress` est différente de zéro, la valeur de la fonction critère est affichée à l'écran à la fin de chaque itération de la boucle qui met en œuvre l'optimisation de cette fonction.

* * * *

Plus précisément, la valeur est écrite sur le fichier d'annonce. Dans la plupart des cas, ceci revient à l'écran (voir le Chapitre 5 de Man2). Une redirection de ce qu'on appelle le **standard error** d'***Ects*** permettrait de faire autrement. Si cette phrase ne vous dit rien, ne vous inquiétez pas. Elle n'est destinée qu'aux friands de l'informatique.

* * * *

Le nombre maximal d'itérations est contrôlé par la variable `maxiter`. Normalement, si l'optimisation n'est pas terminée à la fin de l'itération `maxiter`, ***Ects*** s'arrête, en posant la question

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

ou, en français

n `itérations sans convergence.  On continue (y/n)?`

où n est le nombre d'itérations effectuées. Si l'on répond par `n` (non), la commande s'arrête tout de suite, même si le critère de convergence n'est pas encore satisfait.

Dans certaines circonstances, il n'est pas souhaitable que l'utilisateur doive obligatoirement donner son oui ou non. Par exemple, si on a lancé un grand nombre de simulations au moyen d'une boucle, on préférerait que le programme continue jusqu'à la fin sans interruption. Dans de tels cas, on peut se servir de la variable `noquestions`. Si une estimation non linéaire arrive à la fin de l'itération `maxiter`, et si la valeur de `noquestions` est différente de zéro, l'exécution de l'estimation s'arrête, et le logiciel passe sans question ni commentaire à la commande suivante.

Dans le fichier `nls.ect`, dont on a déjà modifié quelque peu le contenu, l'estimation par NLS est suivi par les commandes suivantes :

```
gen e = y - alpha - beta*x1 - (1/beta)*x2
gen ralpha = 1
gen rbeta = x1 - x2/(beta*beta)
ols e ralpha rbeta
```

Il s'agit de la GNR (Régression de Gauss-Newton) qui correspond au modèle estimé : voir le Chapitre 6 de DM. Il est toujours souhaitable de faire tourner la GNR après une estimation par NLS afin de s'assurer que les conditions du premier ordre soient bien vérifiées. La régressande de cette **régression artificielle** est le vecteur de résidus du modèle non linéaire, et les régresseurs sont les dérivées de la fonction de régression par rapport aux paramètres du modèle. Ceci étant, on aurait pu se servir du fait que les résidus sont disponibles dans la variable `res`. En lancant

```
ols res ralpha rbeta
```

on peut se passer de la variable `e` dans le listing ci-dessus. Pour faciliter davantage la mise en œuvre de la GNR après une estimation par `nls`, la version courante du logiciel recycle la variable `CG`. Après une commande `nls`, cette variable contient la matrice, notée $\boldsymbol{X}(\hat{\boldsymbol{\beta}})$ dans les développements algébriques, des dérivées des fonctions de régression par rapport aux paramètres, évaluées en $\hat{\boldsymbol{\beta}}$, le vecteur de paramètres estimés. Il s'ensuit que le bout de programme ci-dessus peut être remplacé par une seule commande :

```
ols res CG
```

tout de suite après l'exécution de l'estimation non linéaire. Il est rappelé que, si les conditions du premier ordre sont vérifiées, les paramètres estimés de la GNR, ainsi que les Students, sont égaux à zéro aux erreurs d'arrondi près.

Pour des raisons diverses, on s'intéresse parfois à la moyenne de la variable dépendante d'un modèle de régression. La moyenne se calcule sans peine, mais sa valeur est maintenant disponible sans calcul explicite dans la variable `ybar`. Cette nouvelle variable est aussi disponible après l'exécution des commandes `ols` et `iv`. Si l'on se sert de ces commandes pour effectuer des estimations

linéaires multivariées, avec plus d'une variable dépendante, la variable `ybar` devient un vecteur ligne, avec un élément pour chaque variable dépendante.

Une dernière remarque sur les variables mises à jour par `nls`. Pour des raisons qui relèvent plus de l'histoire de l'économétrie que du bon sens, il est de coutume d'afficher un R^2 non seulement pour les régressions linéaires, mais aussi pour les régressions non linéaires. ***Ecta*** s'incline devant les vœux de l'histoire, et la valeur affichée, et disponible dans la variable `R2`, est égale à `sse/(sse+ssr)`, où `sse` est la somme des carrés expliquées et `ssr` la somme des carrés des résidus. Une conséquence de cette définition est que $0 \leq R^2 \leq 1$. Il est rappelé que l'équation `sse+ssr = sst`, où `sst` est la somme des carrés totaux, n'est pas vérifiée par les modèles non linéaires.

3. Estimation Non Linéaire par Variables Instrumentales

Il est nécessaire de recourir aux variables instrumentales à chaque fois que les explicatives d'un modèle sont déterminées simultanément avec la variable dépendante. Cette règle s'applique aux modèles non linéaires comme aux modèles linéaires. Avec les versions précédentes d'***Ecta***, il était nécessaire d'utiliser la commande `gmm` pour effectuer une estimation non linéaire par variables instrumentales. (Voir la section 4.3 de Man2 pour les détails.) Cette procédure, quoique possible, exige des manipulations de matrices peu évidentes. La version actuelle du logiciel fournit une commande spécifique pour ce genre d'estimation.

Le fichier `nliv.ect` contient un programme ***Ecta*** qui sert à illustrer l'utilisation de la commande `nliv`. Les données nécessaires aux estimations se trouvent dans le fichier `ivnls.dat`, et on trouve dans le fichier `ivnls.ect` un programme qui tourne sous la version 2 du logiciel. On constatera que non seulement la nouvelle commande est plus facile à utiliser, mais qu'elle donne également de meilleurs résultats. Les premières commandes dans `nliv.ect` sont comme suit :

```
sample 1 50
read ivnls.dat y x1 x2 w
iv y c x1 x2 (c x1 w)
set b0 = 1
set b1 = 1
set b2 = 1
nliv y = b0*c + b1*x1 + b2*x2
instr c x1 w
deriv b0 = c
deriv b1 = x1
deriv b2 = x2
end
```

Après la lecture des données contenues dans le fichier `ivnls.dat`, on effectue une régression linéaire par variables instrumentales, au moyen de la com-

mande `iv`. Ensuite, on refait la même estimation en se servant de la commande `nliv`. La syntaxe est très similaire à celle qu'on utilise avec la commande `nls`. Tout de suite après `nliv`, on écrit l'équation, linéaire ou non linéaire, dont on souhaite estimer les paramètres. Sur la ligne suivante, on met le mot `instr`, suivi de la liste de variables à utiliser comme instruments. La seule différence par rapport à ce qu'on fait pour une estimation linéaire est que la liste des instruments figure sur une ligne à part, plutôt que dans une parenthèse tout de suite après la liste des explicatives. La suite de la commande suit les mêmes règles que la suite d'une commande `nls`. On donne les dérivées partielles de la fonction de régression par rapport aux paramètres à estimer, une ligne par dérivée. Vu que la régression est linéaire, la différentiation automatique n'est pas nécessaire, parce que les dérivées sont simplement les explicatives.

EXERCICES:

Faites tourner les commandes ***Ecta*** ci-dessus, et démontrez que les commandes `iv` et `nliv` donnent des résultats identiques.

On passe ensuite à une vraie estimation non linéaire. En imposant la contrainte $\beta_2 = 1/\beta_1$, le modèle devient

$$\boldsymbol{y} = \beta_0 + \beta_1 \boldsymbol{x}_1 + \boldsymbol{x}_2/\beta_1 + \boldsymbol{u}. \tag{1}$$

La version ***Ecta*** de ce modèle s'écrit comme :

```
set showprogress = 1
gen W = colcat(c, x1, w)
def fctreg = b0*c + b1*x1 + x2/b1
nliv y = fctreg
instr W
deriv b0 = c
deriv b1 = diff(fctreg,b1)
end
ols res CG
```

Plusieurs choses sont à remarquer ici. D'abord, comme c'est le cas pour `iv`, les instruments peuvent être regroupés dans une matrice. On peut mélanger les variables et les matrices : On aurait pu faire

```
gen W = colcat(x1,w)
```

et, pour la liste des instruments,

```
instr c W
```

sans changer les résultats.

Le programme ci-dessus démontre également qu'une macro peut être utile pour représenter la fonction de régression. Elle permet aussi la différentiation automatique, utilisée ici pour le paramètre `b1`.

Après toute estimation non linéaire, une vérification des conditions du premier ordre s'impose. Après une estimation faite par `nliv`, cette vérification se fait de la même manière qu'après une estimation `nls`. La commande

```
ols res CG
```

sert, encore une fois, à faire tourner la GNR qui correspond à l'estimation non linéaire que l'on vient d'effectuer. Notons la régression non linéaire, comme d'habitude, de la façon suivante :

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

voir DM, chapitre 7. Soit $\boldsymbol{W}$ la matrice de variables instrumentales. Les conditions du premier ordre s'écrivent alors comme

$$\boldsymbol{X}^\top(\hat{\boldsymbol{\beta}})\boldsymbol{P}_{\boldsymbol{W}}\big(\boldsymbol{y} - \boldsymbol{x}(\hat{\boldsymbol{\beta}})\big) = \boldsymbol{0}, \tag{2}$$

où la matrice $\boldsymbol{X}(\boldsymbol{\beta})$ est encore une fois la matrice des dérivées partielles des fonctions de régression, qui sont les composantes du vecteur $\boldsymbol{x}(\boldsymbol{\beta})$, et la matrice $\boldsymbol{P}_{\boldsymbol{W}}$ est la projection orthogonale sur l'espace engendré par les variables instrumentales. La variable `res` contient, comme on peut s'en douter, les résidus de la régression, c'est-à-dire, le vecteur $\boldsymbol{y} - \boldsymbol{x}(\hat{\boldsymbol{\beta}})$. La matrice `CG` est maintenant la matrice dont l'expression algébrique est $\boldsymbol{P}_{\boldsymbol{W}}\boldsymbol{X}(\hat{\boldsymbol{\beta}})$. Si les conditions (2) sont remplies, la régression de `res`, ou $\boldsymbol{y} - \boldsymbol{x}(\hat{\boldsymbol{\beta}})$ sur `CG`, ou $\boldsymbol{P}_{\boldsymbol{W}}\boldsymbol{X}(\hat{\boldsymbol{\beta}})$, donnera des estimations paramétriques nulles, parce que les résidus seront orthogonaux aux colonnes de $\boldsymbol{P}_{\boldsymbol{W}}\boldsymbol{X}(\hat{\boldsymbol{\beta}})$.

On a pu remarquer que, avant de lancer la commande `nliv`, on a affecté une valeur de 1 à la variable `showprogress`. Par conséquent, en faisant tourner le programme, on voit à l'écran le défilé des itérations :

```
crit = 22.1712; newcrit = 10.3901
crit = 10.3901; newcrit = 0.546993
crit = 0.546993; newcrit = 0.161842
crit = 0.161842; newcrit = 0.156805
crit = 0.156805; newcrit = 0.156803
crit = 0.156803; newcrit = 0.156803
```

Chaque ligne représente une itération de la procédure non linéaire qui cherche à minimiser une fonction critère. Dans le cas présent, cette fonction s'écrit comme

$$Q(\boldsymbol{\beta}) = \big(\boldsymbol{y} - \boldsymbol{x}(\boldsymbol{\beta})\big)^\top \boldsymbol{P}_{\boldsymbol{W}}\big(\boldsymbol{y} - \boldsymbol{x}(\boldsymbol{\beta})\big). \tag{3}$$

On vérifie sans peine que les conditions du premier ordre pour un minimum de (3) sont les équations (2). La valeur de la fonction $Q(\boldsymbol{\beta})$ au début d'une itération est donnée par la variable `crit`, et à la fin de l'itération, par `newcrit`. À chaque étape de la procédure, la valeur de $Q(\boldsymbol{\beta})$ décroît, jusqu'à ce que le minimum soit atteint. À la fin, la variable `crit` contient cette valeur minimum.

Voyons à présent les résultats de la commande `nliv`, tels qu'ils se trouvent dans le fichier de sortie :

```
Nonlinear Instrumental Variables Estimation :

Number of iterations = 6

Parameter  Parameter estimate  Standard error  T statistic

b0            -0.014659           0.069393      -0.211248
b1             0.328674           0.014647      22.439042

Number of observations = 50   Number of estimated parameters = 2
Number of instruments = 3
Mean of dependent variable = 2.030593
Sum of squared residuals = 4.316909
Explained sum of squares = 293.919327
Estimate of residual variance
  (with d.f. correction) = 0.089936
Mean of squared residuals = 0.086338
Standard error of regression = 0.299893
Overidentification statistic = 1.816150

Estimated covariance matrix:

0.004815  0.000805
0.000805  0.000215
```

Ce tableau comporte les éléments habituels : Les paramètres estimés, disponibles comme d'habitude dans le vecteur `coef`, ainsi que les écarts-type et les Students, disponibles respectivement dans les vecteurs `stderr` et `student`. La matrice de covariance, imprimée comme toujours tout à la fin du listing, est la matrice $\hat{\sigma}^2(\hat{\boldsymbol{X}}^\top \boldsymbol{P}_{\boldsymbol{W}} \hat{\boldsymbol{X}})^{-1}$, où $\hat{\boldsymbol{X}} \equiv \boldsymbol{X}(\hat{\boldsymbol{\beta}})$, et

$$\hat{\sigma}^2 = \frac{1}{n-k} \|\boldsymbol{y} - \boldsymbol{x}(\hat{\boldsymbol{\beta}})\|^2. \tag{4}$$

Cette matrice se trouvera sous le nom de `vcov`. La valeur de $\hat{\sigma}^2$ peut être retrouvée dans la variable `errvar`. Le choix entre un dénominateur de $n-k$ ou de n n'a pas d'importance majeure pour les estimations par variables instrumentales. La `Mean of squared residuals`, c'est-à-dire, la moyenne des résidus au carré, donne la valeur de (4) où $n-k$ est remplacé par n.

Après une estimation linéaire par variables instrumentales, on sauve sous le nom de `XtPwXinv` la matrice $(\boldsymbol{X}^\top \boldsymbol{P}_{\boldsymbol{W}} \boldsymbol{X})^{-1}$. Pour une estimation non linéaire, l'expression algébrique de la matrice sauvée sous ce nom est $(\hat{\boldsymbol{X}}^\top \boldsymbol{P}_{\boldsymbol{W}} \hat{\boldsymbol{X}})^{-1}$.

Pour le reste, les variables `res` et `fit` contiennent respectivement les résidus $\boldsymbol{y} - \boldsymbol{x}(\hat{\boldsymbol{\beta}})$ et les valeurs ajustées $\boldsymbol{x}(\hat{\boldsymbol{\beta}})$ de la régression. On trouve dans les variables scalaires `ssr`, `sse`, `sst`, et `ybar` les expressions suivantes :

$$\texttt{ssr} = \|\boldsymbol{y} - \boldsymbol{x}(\hat{\boldsymbol{\beta}})\|^2, \quad \texttt{sse} = \|\boldsymbol{x}(\hat{\boldsymbol{\beta}})\|^2, \quad \texttt{sst} = \|\boldsymbol{y}\|^2, \quad \texttt{ybar} = n^{-1} \sum_{t=1}^{n} y_t.$$

La taille de l'échantillon, n, est donnée par `nobs`, le nombre de paramètres, k, par `nreg`, le nombre d'instruments par `ninst`, et le nombre d'itérations par `niter`.

La dernière ligne du listing avant la matrice de covariance donne la `Overidentification statistic`, ou statistique de sur-identification. Cette statistique est égale à $Q(\hat{\boldsymbol{\beta}})/\hat{\sigma}^2$, c'est-à-dire, la valeur minimisée de la fonction critère (3), divisée par une estimation de la variance des aléas, celle dont le dénominateur est n plutôt que $n-k$. Sous l'hypothèse nulle selon laquelle $\mathrm{E}(u_t \mid \boldsymbol{W}_t) = 0$, la loi asymptotique de la statistique est un χ^2 à $l-k$ degrés de liberté, où l est le nombre d'instruments, de sorte que $l-k$ égale le degré de sur-identification. La statistique, disponible sous le nom de `oir`, est fournie non seulement par `nliv`, mais aussi par `iv`.

Remarquons ici que, dans la version 3 d'***Ecta***, la commande `iv` crée une nouvelle variable matricielle, qui porte le nom de `PwX`. La matrice, de la forme $n \times k$, est celle qu'on exprime algébriquement par $\boldsymbol{P}_{\boldsymbol{W}}\boldsymbol{X}$.

Exercices:

Refaites l'estimation du modèle (1) avec les données du fichier `ivnls.dat`, en utilisant `b1=1` comme point de départ. Vous verrez que l'algorithme converge vers un autre point que celui de la première estimation.

On verra que ce phénomène provient du fait que la fonction critère $Q(\boldsymbol{\beta})$ possède deux minima locaux, dont un seul est le minimum global. Pour faciliter l'étude des deux minima, démontrez que, si les variables $\boldsymbol{y}$, $\boldsymbol{x}_1$, $\boldsymbol{x}_2$ et $\boldsymbol{w}$ sont centrées, par exemple par l'action de la projection $\boldsymbol{M}_\iota$ qui annule la constante, l'estimation du modèle

$$\boldsymbol{M}_\iota \boldsymbol{y} = \beta_1 \boldsymbol{M}_\iota \boldsymbol{x}_1 + \boldsymbol{M}_\iota \boldsymbol{x}_2/\beta_1 + \boldsymbol{u}, \tag{5}$$

avec les deux instruments $\boldsymbol{M}_\iota \boldsymbol{x}_1$ et $\boldsymbol{M}_\iota \boldsymbol{w}$, donne la même estimation de β_1 et les mêmes résidus que l'estimation du modèle (1).

La fonction critère associée au modèle (5) ne dépend que d'un seul paramètre, β_1. Créez un graphique où on trace le graphe de la fonction critère sur l'intervalle $[0.2, 2.5]$ de β_1. Vous devrez voir clairement les deux minima de la fonction.

4. Le Maximum de Vraisemblance

Les fonctionnalités offertes par ***Ecta*** pour l'estimation des modèles par le maximum de vraisemblance (ML) ont été largement étendues dans la version actuelle du logiciel. Pour bien comprendre comment les utiliser, on pourra considérer l'estimation d'un **modèle logit**. La variable dépendante d'un tel modèle est une variable **binaire**, ou **dichotomique**, c'est-à-dire, une variable qui n'a que deux valeurs possibles, 0 et 1. Pour l'observation t, la probabilité que la variable dépendante y_t soit égale à 1 est donnée par

$$\Pr(y_t = 1) = F(\boldsymbol{X}_t\boldsymbol{\beta}),$$

où $F(\cdot)$ est une fonction dont les valeurs se situent dans le segment $[0, 1]$, $\boldsymbol{X}_t$ est un vecteur $1 \times k$ de variables explicatives, et $\boldsymbol{\beta}$ est un vecteur $k \times 1$ de paramètres à estimer. La théorie des **modèles à réponse binaire** est donnée dans le chapitre 15 de DM, et un exemple d'une estimation logit se trouve dans la section 5.3 de Man2. Nous reprenons ici cet exemple afin d'illustrer comment mettre en œuvre les différentes méthodes d'estimation ML qu'on peut employer.

La fonction de log-vraisemblance du modèle s'écrit comme la somme des contributions correspondant à chacune des observations :

$$\ell(\boldsymbol{y}, \boldsymbol{\beta}) = \sum_{t=1}^{n} \ell_t(y_t, \boldsymbol{\beta}),$$

où la contribution $\ell_t(\cdot)$ de l'observation t se définit comme suit :

$$\ell_t(y_t, \boldsymbol{\beta}) = y_t \log\big(F(\boldsymbol{X}_t\boldsymbol{\beta})\big) + (1 - y_t) \log\big(1 - F(\boldsymbol{X}_t\boldsymbol{\beta})\big). \tag{6}$$

On voit qu'il n'y a jamais qu'un seul terme par contribution : Si $y_t = 1$, le deuxième terme s'annule, et si $y_t = 0$, c'est le premier terme qui s'annule. Dans tous les cas, la valeur de la contribution est le logarithme de la probabilité d'avoir observé la valeur y_t.

Le modèle logit est un cas particulier d'un modèle à réponse binaire où la fonction $F(\cdot)$ est la **fonction logistique** :

$$F(x) = \frac{e^x}{1 + e^x}. \tag{7}$$

On vérifie que, avec cette définition, F est une fonction croissante de son argument. En fait, F a toutes les propriétés d'une **fonction de répartition**, à savoir :

$$\lim_{x \to -\infty} F(x) = 0, \quad \lim_{x \to \infty} F(x) = 1, \quad F'(x) \geq 0.$$

La loi de probabilité caractérisée par la fonction de répartition logistique s'appelle la **loi logistique**.

On trouve dans le fichier `newlogit.ect` les commandes nécessaires à l'estimation d'un modèle logit. Pour les données, on se sert encore une fois du contenu du fichier `ols.dat`. On génère les variables qui seront utilisées par les commandes suivantes :

```
sample 1 100
read ols.dat y x1 x2 x3
gen x1 = x1/100
gen x2 = x2/100
gen x3 = x3/100
gen Y = y - 100
gen y = 0.5*(sign(Y) + 1)
```

La variable dichotomique `y` est générée par les deux dernières lignes. La **variable qualitative** `y` est égale à 1 si la variable quantitative `Y` est supérieure à 100, et à 0 sinon. Par conséquent, la probabilité que $y_t = 1$ est égale à la probabilité que $Y_t > 100$.

* * * *

Les explicatives $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, et $\boldsymbol{x}_3$ sont toutes divisées par 100. Ceci est une mesure de précaution pour éviter des difficultés d'ordre numérique. La valeur de la fonction exponentielle e^x croît très rapidement avec x, entraînant ainsi la possibilité d'une valeur plus grande que la valeur maximale permise par l'unité d'arithmétique flottante de l'ordinateur. Même s'il n'y a pas de débordement, on risque des erreurs d'arrondi importantes.

* * * *

Outre les explicatives $\boldsymbol{x}_i$, $i = 1, 2, 3$, on aura besoin de la constante, notée `iota`. Rappelons que les macros sont toujours d'une très grande utilité pour la représentation des fonctions non linéaires. On a donc

```
gen iota = 1
def X = a*iota + x1*b1 + x2*b2 + x3*b3
def e = exp(X)
def F = e/(1+e)
def lhd = y*log(F)+(1-y)*log(1-F)
def dldF = (y-F)/(F*(1-F))
def dFde = 1/(1+e)^2
```

La macro `lhd` permet de calculer la fonction de log-vraisemblance ; `dldF` s'interprète comme la dérivée de ℓ par rapport à F, et `dFde` comme la dérivée de F par rapport à `e` $= \exp(\boldsymbol{X}_t\boldsymbol{\beta})$.

* * * *

Dans l'exécution de la commande `ml` et les autres commandes de sa famille que nous verrons tout à l'heure, toutes les expressions sont évaluées comme dans une commande `gen`. Il en découle qu'il n'est pas strictement nécessaire d'utiliser le vecteur `iota` comme on l'a utilisé ici. Mais si on a l'habitude d'expliciter la constante sous forme vectorielle, on évitera beaucoup d'ennuis dans l'exécution des commandes de la famille `gmm`, où les expressions sont évaluées comme dans une commande `mat`.

* * * *

Les commandes suivantes, qui s'appuient sur la commande `ml`, marcheraient avec la version 2 du logiciel :

```
set a = 0
set b1 = 0
set b2 = 0
set b3 = 0
ml lhd
deriv a = dldF*dFde*e
deriv b1 = dldF*dFde*e*x1
deriv b2 = dldF*dFde*e*x2
```

```
deriv b3 = dldF*dFde*e*x3
end
ols iota CG
```

La version 3 accepte les mêmes commandes. Toutefois, il n'est plus *nécessaire* d'initialiser les paramètres `a`, `b1`, *etc.*, surtout si les valeurs de départ sont nulles, comme c'est le cas ici. Mais une initialisation explicite ne fait aucun mal, et dans certains cas, affecter une valeur nulle à un paramètre peut être dangereux.

* * * *

La dernière commande, `ols iota CG`, est la vérification habituelle et nécessaire des conditions du premier ordre.

* * * *

Le calcul analytique des dérivées par rapport aux paramètres de la fonction de log-vraisemblance de notre modèle n'est pas d'une difficulté insurmontable. La forme explicite de la fonction logistique permet une simplification. Si nous notons F_t la fonction $F(\boldsymbol{X}_t\boldsymbol{\beta})$, et e_t la fonction $\exp(\boldsymbol{X}_t\boldsymbol{\beta})$, on a

$$\frac{\partial \ell_t}{\partial \beta_i} = \frac{\partial \ell_t}{\partial F_t}\frac{\partial F_t}{\partial e_t}\frac{\partial e_t}{\partial \beta_i} = \frac{y_t - F_t}{F_t(1-F_t)}\frac{1}{1+e_t^2}e_t X_{ti},$$

où X_{ti} est la valeur de l'explicative i pour l'observation t. Or,

$$F_t(1-F_t) = \frac{e_t}{1+e_t}\frac{1}{1+e_t} = \frac{e_t}{(1+e_t)^2},$$

si bien que

$$\frac{\partial \ell_t}{\partial \beta_i} = X_{ti}(y_t - F_t).$$

Les lignes `deriv` peuvent être remplacées donc par

```
deriv a = y-F
deriv b1 = x1*(y-F)
deriv b2 = x2*(y-F)
deriv b3 = x3*(y-F)
```

Mais, si l'on préfère se passer complètement des calculs analytiques, la différentiation automatique est toujours disponible. On aurait tout simplement

```
deriv a = diff(lhd,a)
deriv b1 = diff(lhd,b1)
deriv b2 = diff(lhd,b2)
deriv b3 = diff(lhd,b3)
```

EXERCICES:

Essayez les trois techniques d'estimation, avec les dérivées simplifiées, non simplifiées, et la différentiation automatique. Les résultats devraient être identiques, mais les temps de calcul seront assez différents.

Voyons maintenant le format des résultats. On a

```
Maximising a Sum of Contributions:

Number of iterations = 22

Parameter  Parameter estimate  Standard error  T statistic

a             -2.535442          1.712046      -1.480943
b1             8.625195          2.306146       3.740090
b2           -13.131199          2.791365      -4.704222
b3             4.078517          1.329014       3.068830

Number of observations = 100   Number of estimated parameters = 4
Maximised value of criterion function = -28.246752

Estimated covariance matrix:

 2.931102  -3.238547   1.800739   0.895411
-3.238547   5.318312  -4.811321   0.502940
 1.800739  -4.811321   7.791716  -1.838118
 0.895411   0.502940  -1.838118   1.766278
```

Ce tableau est à comparer à celui qui se trouve à la page 58 de Man2. La constante estimée est identique, et les autres paramètres estimés sont 100 fois plus grands que ceux de l'ancien tableau : c'est normal ; on a divisé les explicatives par 100. Les écarts-type et les Students sont quand-même un peu différents, même si on tient compte du facteur de 100 pour les écarts-type. Les matrices de covariance sont également différentes. La raison est reliée aux différentes méthodes d'estimation. L'estimation dans Man2 a été effectuée au moyen d'une **régression artificielle**, alors que l'estimation qu'on vient de faire ici utilise l'algorithme DFP (voir Man2), qui ne fournit qu'une approximation de la Hessienne de la fonction de log-vraisemblance, dont l'inverse, au signe près, sert d'estimation de la matrice de covariance des paramètres estimés. La matrice obtenue au moyen de la régression artificielle est plus fiable.

* * * *

Il faut 22 itérations ici contre seulement 5 pour la régression artificielle. Trois raisons différentes expliquent ce phénomène. D'abord, la régression artificielle, étant spécifique au problème traité, est plus efficace que la procédure polyvalente `ml`. Ensuite, notre point de départ, où tous les paramètres sont nuls, est moins favorable que celui employé par l'autre procédure. Finalement, le critère de convergence employé par `ml` est plus strict que celui dont on s'est servi avec la régression artificielle.

* * * *

La matrice de covariance estimée par `ml` est disponible après l'exécution de la commande dans la variable `invhess`. Le nom est quelque peu trompeur, pour les raisons que nous venons de voir. Les autres variables créées ou mises à jour par `ml` sont : `coef`, les paramètres estimés ; `stderr`, les écarts-type ; `student`, les Students ; `nobs`, la taille de l'échantillon ; `nreg`, le nombre de paramètres estimés ; `niter`, le nombre d'itérations ; `lt`, le vecteur de contributions à la

log-vraisemblance, évaluées en $\hat{\boldsymbol{\beta}}$; `lhat`, la log-vraisemblance maximisée, dont la valeur égale la somme des éléments de `lt` ; et, bien sûr, `CG`, la matrice CG des contributions au gradient, dont l'élément (t, i) est $\partial \ell_t / \partial \beta_i(\hat{\boldsymbol{\beta}})$.

EXERCICES:

Refaites l'estimation du modèle sans diviser les explicatives par 100. Vous éprouverez probablement des difficultés numériques. Essayez de les résoudre en divisant par 10 plutôt que par 100.

Dans la suite du fichier de commandes `newlogit.ect`, on refait les estimations avec une nouvelle commande, `mlopg`, à la place de `ml`. Sur la plupart des ordinateurs actuels, cette commande s'exécutera plus vite. Les résultats sont toutefois très similaires, mais la matrice de covariance estimée est encore une fois un peu différente :

```
Estimated covariance matrix:

 2.883893  -3.140939   2.172086   1.122439
-3.140939   5.613164  -6.148303   0.672352
 2.172086  -6.148303  10.784860  -2.342453
 1.122439   0.672352  -2.342453   2.318406
```

La théorie des régressions artificielles est exposée dans un article de Davidson et MacKinnon (1999). On y voit que, pour la quasi-totalité des modèles que l'on estime par ML, la régression artificielle OPG peut servir de base d'une procédure de maximisation de la log-vraisemblance. Cette régression artificielle, dont la mise en œuvre est très simple, est employée par `mlopg`. La polyvalence de la régression artificielle entraîne souvent un manque d'efficacité. En effet, on constate que le nombre d'itérations avec `mlopg` est plus élevé encore qu'avec `ml`. Ce défaut est compensé, dans la plupart des cas, par la simplicité de l'algorithme et des calculs qu'il nécessite.

La matrice de covariance estimée est, naturellement, celle qu'on connaît sous le nom de l'estimateur OPG ; voir le chapitre 8 de DM et Davidson et MacKinnon (1999). Elle est disponible sous le nom de `invOPG`. Malheureusement elle est très peu fiable. Même si la commande `mlopg` permet une estimation rapide, il est souhaitable de recalculer la matrice de covariance par une autre procédure si on veut faire des inférences fiables.

* * * *

La régression, `ols iota CG`, utilisée pour la vérification des conditions du premier ordre, est une régression OPG.

* * * *

On peut remarquer qu'il y a une discordance de signe entre la valeur maximisée de la log-vraisemblance et les valeurs de `crit` et `newcrit` affichées à l'écran pendant l'exécution de `ml` et `mlopg` si la valeur de `showprogress` est non nulle. D'une part, on voit

```
Maximised value of criterion function = -28.246752
```

et d'autre part, le logiciel affiche, par exemple,

```
crit = 28.2471; newcrit = 28.2468
```

La raison est banale : dans les tripes d'***Ects***, toute fonction critère est minimisée. Afin de s'accommoder à ce fait, on minimise l'opposé de la log-vraisemblance. La valeur imprimée dans le tableau de résultats est la bonne.

Plus loin encore dans le fichier `newlogit.ect`, on utilise la commande `mlhess` pour refaire l'estimation de notre modèle logit. Comme le nom le suggère, l'algorithme derrière cette commande utilise la **Hessienne** de la log-vraisemblance. Le principe de base de la plupart des algorithmes de maximisation/minimisation de fonctions est la **méthode de Newton**. Sous sa forme d'origine, cette méthode s'appuie sur les derivées premières et secondes de la fonction à maximiser. Autrement dit, il faut à la fois le gradient et la hessienne de la log-vraisemblance pour la mise en œuvre de la méthode. Ceci se voit clairement dans la première estimation par `mlhess` :

```
def f = e/(1+e)^2
mlhess lhd
deriv a = y-F
deriv b1 = x1*(y-F)
deriv b2 = x2*(y-F)
deriv b3 = x3*(y-F)
second a,a = -f
second a,b1 = -x1*f
second a,b2 = -x2*f
second a,b3 = -x3*f
second b1,b1 = -x1*x1*f
second b1,b2 = -x1*x2*f
second b1,b3 = -x1*x3*f
second b2,b2 = -x2*x2*f
second b2,b3 = -x2*x3*f
second b3,b3 = -x3*x3*f
end
```

Grâce à la macro `f`, les éléments de la hessienne sont assez facile à exprimer algébriquement. La syntaxe n'est pas compliquée : Par l'utilisation du mot-clé `second` à la place de `deriv`, on signale qu'il s'agit d'une dérivée seconde. Ensuite, il faut deux paramètres plutôt qu'un seul, parce qu'il faut deux dérivations pour une dérivée seconde. Étant donné que l'ordre des deux dérivations n'a pas d'importance, on peut se contenter de la dérivée `second a,b1`, par exemple : ***Ects*** sait que cette dérivée sert aussi pour `second b1,a`.

EXERCICES:

Démontrez analytiquement que les dérivées secondes du programme représentent bien les dérivées secondes de la fonction (6).

La différentiation automatique où est-elle dans tout ceci ? Jusqu'ici nulle part, évidemment. Mais on obtiendrait les mêmes résultats avec le programme suivant :

```
mlhess lhd
deriv a = y-F
deriv b1 = x1*(y-F)
deriv b2 = x2*(y-F)
deriv b3 = x3*(y-F)
end
```

où aucune dérivée seconde n'est spécifiée. ***Ects*** est en mesure de formuler lui-même les dérivées secondes dont il a besoin. En fait, on peut panacher : Si l'utilisateur fournit un sous-ensemble des dérivées secondes nécessaires à l'estimation, seules celles qui manquent seront fournies automatiquement.

Pourquoi ne laisse-t-on pas tout le travail des dérivées, premières et secondes, à ***Ects*** ? On a le droit de le faire : Le programme

```
mlhess lhd
deriv a = diff(lhd,a)
deriv b1 = diff(lhd,b1)
deriv b2 = diff(lhd,b2)
deriv b3 = diff(lhd,b3)
end
```

donnera exactement les mêmes résultats que nos programmes précédents avec `mlhess`. Seulement, le temps de calcul sera *beaucoup* plus long. ***Ects***, dans son avatar actuel, sait très bien dériver, mais il ne sait pas *simplifier* les expressions, parfois très longues, des dérivées qu'il calcule. La conséquence en est que, même si on arrive au même résultat, le chemin suivi par la différentiation automatique est beaucoup moins direct que celui qui se base sur des expressions courtes et simples des dérivées secondes.

La matrice de covariance estimée par `mlhess` est disponible sous le nom de `invhess`. Cette fois-ci, à la différence de ce qui se produit avec `ml`, c'est l'inverse de la vraie hessienne plutôt que d'une approximation.

EXERCICES:

Faites tourner les quatre estimations par `mlhess` que vous trouverez dans le fichier `newlogit.ect` sur votre ordinateur habituel, et comparez les temps de calcul. Si la valeur de `showprogress` est non nulle, vous pourrez observer le temps nécessaire pour chaque itération.

Si on regarde de près les résultats des régressions OPG

```
ols iota CG
```

utilisées pour la vérification des conditions du premier ordre après les estimations qu'on a faites au moyen de `ml` ou `mlopg`, on verra que les valeurs des Students sont très petites, par exemple `-1.216695e-10`, mais non nulles. En revanche, les Students obtenus après une estimation par `mlhess` sont nuls au

niveau de précision de la machine. Ceci s'explique par le fait que la vraie méthode de Newton, celle qui utilise les dérivées secondes analytiques, converge plus rapidement dans le voisinage du maximum que les méthodes approximatives employées par `ml` et `mlopg`. Dans le cas présent, la convergence est globalement plus rapide, dont témoigne le plus faible nombre d'itérations, 8 en l'occurrence, contre 22–24 pour les autres méthodes, toujours avec le même point de départ. La conclusion à tirer est que, quand il faut une grande précision, `mlhess` est la meilleure commande à utiliser. Le coût se paye en termes soit de temps de calcul, soit du calcul analytique des dérivées secondes.

Dans le fichier `logit.ect`, distribué avec la version 2 d'***Ects***, l'estimation est faite au moyen de la régression artificielle connue sous le nom de **BRMR**, pour *Binary Response Model Regression* – voir la section 15.4 de DM. La même procédure, légèrement modifiée, se trouve tout à la fin de `newlogit.ect`. Comme l'article de Davidson et MacKinnon (1999) le démontre, des régressions artificielles existent pour plusieurs classes de modèles, et elles conduisent souvent à des méthodes d'estimation efficaces. On peut profiter de ce fait au moyen de la commande `mlar`, où le `ar` signifie *Artificial Regression*.

La BRMR pour le cas général du modèle caractérisé par (6) s'écrit comme

$$\frac{y_t - F_t}{\big(F_t(1-F_t)\big)^{1/2}} = \frac{f_t \boldsymbol{X}_t}{\big(F_t(1-F_t)\big)^{1/2}}\boldsymbol{b} + \text{résidu},$$

où $F_t \equiv F(\boldsymbol{X}_t\boldsymbol{\beta})$, et $f_t \equiv f(\boldsymbol{X}_t\boldsymbol{\beta}) = F'(\boldsymbol{X}_t\boldsymbol{\beta})$. Pour le modèle logit, F est donné par (7), et

$$F'(x) = f(x) = \frac{e^x}{(1+e^x)^2}.$$

Une macro qui permet d'évaluer $f(\cdot)$ existe déjà : On s'en est servi pour les dérivées secondes utilisées par `mlhess`. La syntaxe d'une commande `mlar` est illustrée par le programme suivant.

```
def arden = sqrt(F*(1-F))
mlar lhd
lhs = (y-F)/arden
deriv a = f/arden
deriv b1 = f*x1/arden
deriv b2 = f*x2/arden
deriv b3 = f*x3/arden
end
```

La deuxième ligne définit la **régressande** de la régression artificielle. Elle est introduite par le mot-clé `lhs`. La régressande est le membre de gauche de la régression, et, en anglais, le membre de gauche est le *left hand side*. Ensuite, les lignes `deriv` donnent les **régresseurs** de la régression artificielle. La macro `arden` facilite l'expression de la régressande et des régresseurs. Il est important de noter que, en général, les régresseurs, bien qu'introduits par `deriv`, ne sont pas des dérivées.

Selon les règles des régressions artificielles, une estimation de la matrice de covariance des paramètres estimés est fournie par les régresseurs de la régression artificielle. Si on note $\boldsymbol{R}(\boldsymbol{\beta})$ la matrice de régresseurs, la matrice de covariance estimée est l'inverse de $\boldsymbol{R}^\top(\hat{\boldsymbol{\beta}})\boldsymbol{R}(\hat{\boldsymbol{\beta}})$. Cette matrice inversée est accessible après l'exécution de la commande dans la variable `XtXinv`.

Si on effectue l'estimation par `mlar`, on constatera que les résultats sont parfaitement équivalents à ceux donnés par `mlhess`. Ceci n'est pas un accident, mais plutôt la conséquence d'une propriété très spécifique de la BRMR appliquée au modèle logit.

Exercices:

Soit $\boldsymbol{R}(\boldsymbol{\beta})$ la matrice des régresseurs de la BRMR pour le modèle logit. Démontrez que $\boldsymbol{R}^\top(\boldsymbol{\beta})\boldsymbol{R}(\boldsymbol{\beta})$ est égale, au signe près, à la hessienne de la log-vraisemblance. Pourquoi ce fait explique-t-il l'équivalence des résultats de `mlhess` et `mlar` ?

La vérification des conditions du premier ordre se fait un peu différemment après l'exécution d'une commande `mlar`. En effet, les paramètres estimés et les Students de la régression artificielle elle-même doivent s'annuler après la convergence de la procédure. Voici les éléments qu'il faut :

```
gen r = (y-F)/arden
ols r CG
```

La régression artificielle s'effectue par la commande `ols r CG`, parce que la matrice `CG`, suite à une commande `mlar`, n'est plus la matrice CG, mais plutôt la matrice des régresseurs de la régression artificielle, évalués en $\hat{\boldsymbol{\beta}}$.

Exercices:

Utilisez la commande `mlar` pour estimer un modèle de régression non linéaire par la GNR. On peut essayer celui qu'on a considéré dans la section 2 de ce chapitre. Pour la fonction critère, on utilisera moins le carré du résidu (pourquoi?).

5. La Méthode des Moments Généralisée

La méthode de moments généralisée (GMM), considérée lors de la rédaction de Man2 comme une méthode « sophistiquée », est devenue maintenant un outil standard de l'économétrie. Le chapitre 17 de DM est toujours une référence incontournable dans la matière. Du point de vue d'***Ecta***, à la commande `gmm`, disponible dans la version 2 du logiciel, viennent s'ajouter deux commandes nouvelles, `gmmhess` et `gmmweight`. La première, `gmmhess`, comme `mlhess`, fait appel aux dérivées secondes analytiques de la fonction critère. La deuxième, `gmmweight`, est d'une nature différente, que nous verrons plus tard.

Par rapport à la famille de commandes `ml`, la principale différence dans l'utilisation de `gmm` et `gmmhess` est que les expressions sont évaluées comme si on était dans une commande `mat` plutôt que dans une commande `gen`.

Ceci signifie, par exemple, que les variables `smplstart` et `smplend`, dont les valeurs sont affectées par la commande `sample`, n'ont aucun effet. Une autre différence est que la fonction critère est *minimisée*.

On pourra étudier les performances de `gmm` et `gmmhess` en regardant le fichier de commandes `gmm.ect`. Dans ce fichier, on utilise les mêmes données et les mêmes modèles que dans `nliv.ect`. Bien entendu, la GMM ne se limite pas aux modèles de régression estimés à l'aide de variables instrumentales, mais la simplicité de ces modèles les rend utiles pour la clarté de l'exposé des commandes d'***Ects***.

Rappelons-nous que la fonction critère à minimiser est donnée par l'expression (3). En exprimant cette fonction et ses dérivées en termes de matrices à la manière d'***Ects***, on obtient :

```
sample 1 50
read ivnls.dat y x1 x2 w
gen iota = 1
gen W = colcat(iota, x1, w)
mat WtWinv = (W'*W)inv
def resid = y - b0*iota - b1*x1 - b2*x2
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
sample 1 3
print grad
```

* * * *

Il est indispensable dans les commandes `gmm` et `gmmhess` d'expliciter le vecteur constant `iota`. Sinon, les dimensions des matrices ne seront pas correctes pour les produits matriciels.

* * * *

On vérifie aisément que l'expression `resid'*W*WtWinv*W'*resid` représente la fonction $Q(\boldsymbol{\beta})$ de (3), parce que $\boldsymbol{P_W} = \boldsymbol{W}(\boldsymbol{W}^\top\boldsymbol{W})^{-1}\boldsymbol{W}^\top$. De même, les expressions dans les lignes `deriv` représentent les dérivées de Q.

Le tableau de résultats contient beaucoup moins d'information que les tableaux générés par `ols`, `ml`, *etc*. Le voici :

```
Minimising a Criterion Function:

Number of iterations = 13

b0 = -0.166437
b1 = 0.586427
b2 = 2.732630

Number of estimated parameters = 3
Minimised value of criterion function = 0.000000

Estimate of Inverse of Hessian of Criterion Function:
```

```
 0.100231  -0.120279   0.108585
-0.120279   0.213099  -0.265844
 0.108585  -0.265844   0.408608
```

On constate quelques différences par rapport au tableau de la page 47 de Man2. Certaines sont dues simplement à la division par 100 des variables. Les autres sont d'une importance mineure ; elles relèvent de la nature approximative de la hessienne de Q.

Il y a relativement peu de variables créées ou mises à jour par `gmm`. Comme d'habitude, les paramètres estimés se trouvent dans la variable `coef`, `nreg` contient le nombre de paramètres estimés, `niter` le nombre d'itérations, et `crit` la valeur minimisée de la fonction critère. L'inverse de la hessienne (approximative !) de la fonction critère se trouve dans `invhess`, et le gradient dans `grad`. Cette dernière variable est là afin de permettre la vérification des conditions du premier ordre. Normalement, avec `gmm`, on ne peut plus faire appel à une régression artificielle simple. Mais le gradient doit être nul si les conditions sont respectées. Vu que l'estimation non linéaire n'est jamais exacte, il faut normalement se contenter d'une nullité approximative.

Ici encore, on peut se servir de la différentiation automatique. On trouvera cette version de la commande dans `gmm.ect`, ainsi que l'estimation par `gmm` du modèle soumis à la contrainte non linéaire $\beta_2 = 1/\beta_1$.

La commande `gmmhess` a exactement la même syntaxe que `gmm`. Dans la suite de `gmm.ect`, on l'utilise pour refaire l'estimation du modèle non linéaire, avec et sans différentiation automatique. Avec `gmmhess`, bien entendu, la hessienne n'est plus approximative. Une comparaison des résultats donnés par `gmm` et `gmmhess` démontre que l'approximation utilisée par celui-là est assez grossière. On constate aussi que le nombre d'itérations nécessaires pour la convergence de `gmmhess` est inférieur à celui dont a besoin `gmm`, 11 contre 16. Les variables créées par `gmmhess` sont les mêmes que dans le cas de `gmm`.

Comme `mlhess`, `gmmhess` permet à l'utilisateur de définir ses propres dérivées secondes. Plus loin dans `gmm.ect`, on trouve

```
def dresd1 = -x1 + x2/b1^2
gmmhess residual'*W*WtWinv*W'*residual
deriv b0 = -2*iota'*W*WtWinv*W'*residual
deriv b1 = -2*(x1 - x2/(b1*b1))'*W*WtWinv*W'*residual
second b0,b0 = 2*iota'*W*WtWinv*W'*iota
second b0,b1 = -2*iota'*W*WtWinv*W'*dresd1
second b1,b1 = 2*dresd1'*W*WtWinv*W'*dresd1 - \
4*(x2/b1^3)'*W*WtWinv*W'*residual
end
```

où tout est explicite. Les résultats sont inchangés.

Dans le programme ci-dessus, on s'est servi de la possibilité de prolonger une ligne sur la ligne suivante au moyen du caractère \. Si, lors de la lecture d'une ligne d'un fichier de commandes par ***Ects***, le dernier caractère est \,

le contenu de la ligne suivante est lu et rajouté à la ligne précédente. Il est possible ainsi de faire lire à ***Ects*** les commandes les plus longues, en les étalant sur plusieurs lignes, avec un \ à la fin de chacune sauf la dernière.

La nouvelle commande `gmmweight` est très différente de `gmm` et `gmmhess`. D'abord, comme sous les commandes de la famille `ml`, les expressions sont évaluées comme dans une commande `gen`, avec prise en compte de la taille de l'échantillon, telle qu'elle est définie par `smplstart` et `smplend`. Un autre aspect important de la commande `gmmweight` est qu'elle ne s'applique qu'à des modèles d'une forme spécifique.

La fonction critère minimisée quand on effectue une estimation non linéaire par variables instrumentales est donnée par (3), dont la forme est celle requise par `gmmweight` si on explicite la matrice de projection :

$$Q(\boldsymbol{\beta}) = \big(\boldsymbol{y} - \boldsymbol{x}(\boldsymbol{\beta})\big)^{\top}\boldsymbol{W}(\boldsymbol{W}^{\top}\boldsymbol{W})^{-1}\boldsymbol{W}^{\top}\big(\boldsymbol{y} - \boldsymbol{x}(\boldsymbol{\beta})\big). \tag{8}$$

Si on se sert du langage de la GMM, le vecteur $\boldsymbol{W}^{\top}\big(\boldsymbol{y}-\boldsymbol{x}(\boldsymbol{\beta})\big)$ est un vecteur de **moments**, c'est-à-dire un vecteur dont chaque composante est une fonction des données et des paramètres du modèle. Si on utilise les vrais paramètres dans l'évaluation des moments, les espérances de ceux-ci sont nulles. La fonction critère est une forme quadratique définie par les moments et une **matrice de pondération**, ici la matrice $(\boldsymbol{W}^{\top}\boldsymbol{W})^{-1}$, qui doit être une matrice définie positive. L'estimateur GMM est efficace si la matrice de pondération est proportionnelle (asymptotiquement) à l'inverse de la matrice de covariance des moments, évalués en les vrais paramètres. On vérifie que, sous les hypothèses d'homoscédasticité et non autocorrélation des résidus $\boldsymbol{y}-\boldsymbol{x}(\boldsymbol{\beta})$, cette condition est vérifiée par la fonction (8).

Il arrive souvent dans la pratique que les moments soient définis en termes d'un vecteur $\boldsymbol{f}(\boldsymbol{y}, \boldsymbol{\beta})$, fonction des données $\boldsymbol{y}$ et des paramètres $\boldsymbol{\beta}$, dont la dimension est la taille de l'échantillon. Dans le cas présent, ce vecteur est simplement le vecteur des résidus $\boldsymbol{y}-\boldsymbol{x}(\boldsymbol{\beta})$. Les moments sont construits comme les produits scalaires de ce vecteur et de l variables, qu'on appelle assez généralement des variables instrumentales. Il est clair que, pour le cas de l'estimation d'un modèle de régression par variables instrumentales, les colonnes de la matrice $\boldsymbol{W}$ sont les instruments non seulement dans la terminologie usuelle, mais aussi dans la terminologie spécialisée de la GMM. On peut démontrer (voir DM, chapitre 17) que, si on définit les moments comme $\boldsymbol{W}^{\top}\boldsymbol{f}(\boldsymbol{y}, \boldsymbol{\beta})$, alors, avec n'importe quelle matrice de pondération définie positive $\boldsymbol{A}$ de la forme $l \times l$, la minimisation de la fonction critère

$$\boldsymbol{f}^{\top}(\boldsymbol{y}, \boldsymbol{\beta})\boldsymbol{W}\boldsymbol{A}\boldsymbol{W}^{\top}\boldsymbol{f}(\boldsymbol{y}, \boldsymbol{\beta}) \tag{9}$$

conduit à une estimation convergente, mais en général non efficace, des paramètres du modèle.

Pour utiliser la commande `gmmweight`, il faut spécifier les trois ingrédients de la structure qu'on vient de décrire, à savoir, le vecteur $\boldsymbol{f}$, les instruments $\boldsymbol{W}$, et la matrice de pondération $\boldsymbol{A}$. (En anglais, *weight* signifie

poids, ou *pondération*.) Les commandes suivantes, extraites de la fin du fichier `gmm.ect`, servent à illustrer la syntaxe :

```
sample 1 50
def residual = y - b0*iota - b1*x1 - (1/b1)*x2
gen W = colcat(iota, x1, w)
gmmweight residual
instr W
weightmatrix = (W'*W) inv
deriv b0 = -iota
deriv b1 = diff(residual,b1)
end
```

Le nom de la commande est suivi par le vecteur $\boldsymbol{f}$, qu'on représente ici par la macro `residual`. Sur la ligne suivante, exactement comme pour la commande `nliv`, on met le mot `instr` suivi d'une liste d'instruments, qui peut mélanger vecteurs et matrices. Ensuite, sur la ligne suivante, le mot clé `weightmatrix`, est suivi du signe d'égalité « = », et une expression qui représente la matrice de pondération $\boldsymbol{A}$. Cette expression, *et elle seule*, est évaluée selon les règles de `mat` plutôt que `gen`. Après, comme d'habitude, on a une suite de lignes introduites par `deriv`, une par paramètre à estimer, où on donne les dérivées partielles de $\boldsymbol{f}$ par rapport aux paramètres. Finalement, les lignes de la commande sont terminées par `end`.

Regardons à présent le tableau des résultats de la commande. Grâce à la structure spécifique du modèle estimé, les informations contenues dans le tableau sont beaucoup plus riches que dans ceux donnés par `gmm` et `gmmhess`. On constate que les valeurs numériques sont identiques à celles données par `nliv` pour le même modèle, et que le nombre d'itérations est similaire au nombre utilisé par `nliv`, et par conséquent moins que le nombre nécessaire à la convergence des algorithmes employés par `gmm` et `gmmhess`.

```
Minimising a Quadratic Form:

Number of iterations = 7

Parameter  Parameter estimate  Standard error  T statistic

b0            -0.014659           0.069393      -0.211248
b1             0.328674           0.014647      22.439041

Number of observations = 50   Number of estimated parameters = 2
Number of instruments = 3
Sum of squared residuals = 4.316909
Estimate of residual variance
  (with d.f. correction) = 0.089936
Mean of squared residuals = 0.086338
Standard error of regression = 0.299893
Minimised value of criterion function = 0.156803

Estimated covariance matrix:

0.004815  0.000805
0.000805  0.000215
```

```
Heteroskedasticity Consistent Estimate:
 (Note: Estimate above valid only with efficient weightmatrix)
0.006719  0.001158
0.001158  0.000262
```

On a remarqué que l'estimation par GMM est efficace si la matrice de pondération est proportionnelle à l'inverse de la matrice de covariance des moments. Sinon, la première des deux matrices de covariance estimées n'est plus correcte. Cette matrice, disponible sous le nom de `vcov`, est donnée par l'expression matricielle suivante :

$$\hat{\sigma}^2 \hat{\boldsymbol{V}} \equiv \hat{\sigma}^2 (\hat{\boldsymbol{X}}^\top \boldsymbol{W} \boldsymbol{A} \boldsymbol{W}^\top \hat{\boldsymbol{X}})^{-1},$$

où

$$\hat{\sigma}^2 = \frac{1}{n-k} \boldsymbol{f}^\top(\boldsymbol{y}, \hat{\boldsymbol{\beta}}) \boldsymbol{f}(\boldsymbol{y}, \hat{\boldsymbol{\beta}})$$

et $\hat{\boldsymbol{X}} = \boldsymbol{X}(\hat{\boldsymbol{\beta}})$, $\boldsymbol{X}(\cdot)$ étant la matrice des dérivées partielles des composantes de $\boldsymbol{f}$ par rapport aux k composantes de $\boldsymbol{\beta}$. La valeur de $\hat{\sigma}^2$ est sauvée dans la variable `errvar`. Il n'est pas trop difficile de démontrer que la matrice $\hat{\sigma}^2 \hat{\boldsymbol{V}}$ est une estimation convergente de la matrice de covariance de $\hat{\boldsymbol{\beta}}$ si les composantes de $\boldsymbol{f}$ sont homoscédastiques et non autocorrélées, et $\boldsymbol{A}^{-1}$ est proportionnelle à la matrice de covariance des moments.

Plus généralement, on peut supposer que les composantes de $\boldsymbol{f}$ sont hétéroscédastiques mais toujours non autocorrélées. Soit la matrice de covariance du vecteur $\boldsymbol{f}$ la matrice diagonale $\boldsymbol{\Omega}$. Dans ce cas, on démontre que la bonne forme de la matrice de covariance asymptotique de $\hat{\boldsymbol{\beta}}$ est

$$\boldsymbol{V} \boldsymbol{X}^\top \boldsymbol{W} \boldsymbol{A} \boldsymbol{W}^\top \boldsymbol{\Omega} \boldsymbol{W} \boldsymbol{A} \boldsymbol{W}^\top \boldsymbol{X} \boldsymbol{V},$$

où on a supprimé les chapeaux dans cette expression théorique. Notons d'abord que la matrice $\hat{\boldsymbol{V}}$ est disponible sous le nom de `XtWAWtXinv`. La deuxième matrice de covariance estimée dans le tableau, celle qui figure sous le nom de `Heteroskedasticity Consistent Estimate`, ou, en français, Estimateur Robuste à l'Hétéroscédasticité, est justement donnée par l'expression

$$\hat{\boldsymbol{V}} \hat{\boldsymbol{X}}^\top \boldsymbol{W} \boldsymbol{A} \boldsymbol{W}^\top \hat{\boldsymbol{\Omega}} \boldsymbol{W} \boldsymbol{A} \boldsymbol{W}^\top \hat{\boldsymbol{X}} \hat{\boldsymbol{V}}, \tag{10}$$

où $\hat{\boldsymbol{\Omega}}$ est une matrice diagonale dont l'élément diagonal type est $f_t^2(\hat{\boldsymbol{\beta}})$. Le chapitre 17 de DM donne les détails de cet estimateur. Comme l'indique l'avertissement dans le tableau de résultats précédent, lui seul est valable en présence d'hétéroscédasticité ou dans le cas où la matrice de pondération $\boldsymbol{A}$ n'est pas proportionnelle à l'inverse de la matrice de covariance des moments. La matrice (10) est disponible dans la variable `HCCME`.

Les conditions du premier ordre pour la minimisation de la fonction critère (9) s'écrivent comme

$$\boldsymbol{X}^\top(\boldsymbol{\beta})\boldsymbol{W}\boldsymbol{A}\boldsymbol{W}^\top\boldsymbol{f}(\boldsymbol{\beta}) = \boldsymbol{0}. \tag{11}$$

Dans les calculs d'***Ects***, on fait appel à une matrice $\boldsymbol{B}$, de forme triangulaire supérieure, telle que $\boldsymbol{B}^\top\boldsymbol{B} = \boldsymbol{A}$. La matrice $\boldsymbol{B}\boldsymbol{W}^\top\hat{\boldsymbol{X}}$, de dimension $l \times k$, se trouve, après l'exécution de `gmmweight`, dans la variable `CG`. De même, la matrice $\boldsymbol{B}\boldsymbol{W}^\top\hat{\boldsymbol{f}}$, de la forme $l \times 1$, se trouve dans `grad`. Les commandes suivantes, extraites de `gmm.ect`, permettent de vérifier les conditions du premier ordre.

```
sample 1 2
mat t = CG'*grad
print t
sample 1 3
ols grad CG
```

En effet, le produit matriciel `CG'*grad` est égal à $\hat{\boldsymbol{X}}\boldsymbol{W}\boldsymbol{B}^\top\boldsymbol{B}\boldsymbol{W}^\top\hat{\boldsymbol{f}}$, qui, d'après (11), doit s'annuler si l'algorithme a convergé. Si l'on tient à ce que la vérification se fasse par régression artificielle, la régression de `grad` sur `CG` peut servir.

Parmi les autres variables créées ou mises à jour par `gmmweight`, les scalaires `nobs`, `nreg`, `ninst`, et `niter` ont leurs sens habituels, c'est-à-dire, respectivement, n, k, l, et le nombre d'itérations. Les vecteurs $k \times 1$, `coef`, `stderr`, et `student` contiennent $\hat{\boldsymbol{\beta}}$, et les écarts-type et Students associés aux composantes de celui-ci en utilisant la matrice de covariance `vcov`, qui n'est pas forcément la bonne. Si on préfère `HCCME`, il faudra effectuer un calcul explicite des écarts-type et des Students. Le n–vecteur `res` est l'analogue du vecteur de résidus, c'est-à-dire, $\hat{\boldsymbol{f}} \equiv \boldsymbol{f}(\boldsymbol{y}, \hat{\boldsymbol{\beta}})$. Le scalaire `ssr` est la somme des carrés des éléments de `res`. Finalement, le scalaire `crit` est la valeur minimisée de la fonction critère.

EXERCICES:

Pour le modèle de `nliv.ect` et `gmm`, faites tourner une estimation par `gmmweight` et ensuite construisez explicitement la matrice $\hat{\boldsymbol{X}}$. La matrice $\boldsymbol{B}$ peut être obtenue par la commande `mat B = (lowtriang(A))'`. Démontrez que les définitions données ci-dessus de `CG`, `grad`, `vcov`, `XtWAWtXinv`, et `HCCME` sont correctes.

Refaites toutes les estimations dans `nliv.ect` et `gmm.ect` sur un sous-échantillon consistant en les 25 dernières observations. Pour les commandes `nliv`, `gmmweight`, et celles de la famille `ml`, il devrait suffire de faire `sample 26 50`. Pour `gmm` et `gmmhess`, il sera nécessaire de sélectionner des blocs des matrices contenant les variables. Dans tous les cas, vous saurez si les manipulations sont correctes si les résultats sont tous les mêmes.

6. Les Procédures

Si on regarde de nouveau dans le fichier `newlogit.ect`, on constatera qu'il y a beaucoup de calculs qui doivent être effectués plusieurs fois. On commence par définir des macros `dldF` et `dFde`, qui dépendent à leur tour d'autres macros, `F`, `e`, et `X`. Par la suite, dans les procédures d'estimation non linéaires, dans le calcul de chaque dérivée de la fonction de log-vraisemblance, on fait appel à l'ensemble de ces macros. Dans la première estimation par `ml`, par exemple, l'expression `dldF*dFde*e` doit être calculée quatre fois à chaque itération de l'algorithme. Si on ne fait qu'un petit nombre d'estimations, ceci n'est pas gênant. Mais, si on a beaucoup d'estimations non linéaires à faire, lors d'une expérience de simulation avec beaucoup de répétitions, par exemple, le fait de refaire plusieurs fois le même calcul peut entraîner des temps de calcul excessivement longs.

Dans la version 3 d'***Ects***, il existe un mécanisme qui permet d'éviter ces calculs répétés. Ce mécanisme est illustré dans le fichier `proclogit.ect`, où on reprend quelques-unes des estimations de `newlogit.ect`. Afin de se servir du mécanisme, on utilise la commande `procedure`, comme suit.

```
procedure logit 4
   def X = arg1*iota + x1*arg2 + x2*arg3 + x3*arg4
   gen lf = lhd
   answer lf
   gen D0 = dldF*dFde*e
   answer D0
   gen D1 = D0*x1
   answer D1
   gen D2 = D0*x2
   answer D2
   gen D3 = D0*x3
   answer D3
end
```

Le nom de la commande, `procedure`, est suivi du nom de la procédure, ici `logit`, et le nombre d'*arguments* de la procédure, ici 4. Les arguments de la procédure, qui doivent être des scalaires, vont correspondre aux paramètres à estimer. Ensuite, il y a plusieurs lignes, parmi lesquelles on voit des commandes ***Ects*** ordinaires. La fin de la définition de la procédure est signalée, comme d'habitude, par `end`.

Le but de la procédure est de remplacer la commande suivante, qu'on trouve dans `newlogit.ect`,

```
ml lhd
deriv a = dldF*dFde*e
deriv b1 = dldF*dFde*e*x1
deriv b2 = dldF*dFde*e*x2
deriv b3 = dldF*dFde*e*x3
```

par une commande où on fait appel à la procédure, comme suit :

```
ml logit(a,b1,b2,b3,1)
deriv a = logit(a,b1,b2,b3,2)
deriv b1 = logit(a,b1,b2,b3,3)
deriv b2 = logit(a,b1,b2,b3,4)
deriv b3 = logit(a,b1,b2,b3,5)
end
```

Voyons à présent comment cela marche, et comment on évite ainsi les calculs répétés.

La commande `procedure`, avec les lignes suivantes jusqu'au `end` final, n'a pour effet que la *définition* de la procédure. Aucune commande ***Ects*** trouvée à l'intérieur de cette définition ne sera exécutée à ce stade. Après, quand la commande `ml` est lancée, le contenu de la définition est exécuté au moment où l'expression

```
deriv b1 = logit(a,b1,b2,b3,3)
```

par exemple, est évaluée au cours des itérations de l'algorithme employé par `ml`. De telles expressions sont à évaluer cinq fois par itération, une fois pour la fonction critère dont la représentation ***Ects*** est donnée tout de suite après la commande `ml`, et quatre fois pour les quatre dérivées de la fonction. Normalement, la fonction elle-même sera évaluée avant les dérivées. À ce moment, on entame l'exécution des commandes trouvées à l'intérieur de la procédure. La première est la définition d'une macro, `X`, qui fait appel à quatre variables qu'on n'a définies nulle part, `arg1`, ..., `arg4`. Comme on peut s'en douter, ces variables sont les quatre arguments de la procédure. En lisant l'expression `logit(a,b1,b2,b3,1)`, ***Ects*** affecte aux variables scalaires `arg1`, ..., `arg4` les valeurs des arguments trouvés dans l'expression, ici `a,b1,b2,b3`.

Le principal intérêt d'une procédure est qu'elle sert à calculer plusieurs expressions à la fois. C'est pourquoi, après les quatre arguments, on en trouve encore un, l'indice de l'expression dont on a besoin. Donc, la fonction critère est la première expression calculée par la procédure, la dérivée par rapport à `a` est la seconde, celle par rapport à `b1` la troisième, et ainsi de suite. Dans les commandes de la procédure, après la définition de la macro, on trouve une commande `gen` qui crée une variable `lf`, dont la valeur est donnée par la macro `lhd`, qui représente la fonction critère. Sur la ligne suivante, on fait appel à la commande `answer`, qui n'est utilisée qu'à l'intérieur d'une procédure : Ailleurs elle donne lieu à une erreur de syntaxe. Elle sert à rendre réponses de la procédure. (*answer* veut dire *réponse* en anglais.) Comme le démontre la commande

```
answer lf
```

il faut un seul argument, qui doit être le nom d'une variable existante. Elle peut être un scalaire, un vecteur, une matrice, peu importe, mais elle doit exister au moment où la commande `answer` est exécutée. Une procédure

peut contenir un nombre quelconque de commandes `answer`. Chacune définit l'une des expressions calculées par la procédure. À la variable trouvée dans la première `answer`, on affecte l'indice 1, à la seconde l'indice 2, et ainsi de suite. Dans la procédure `logit`, la première variable est `lf`, qui contient l'évaluation de la fonction critère.

Logiquement, donc, la variable dans la deuxième commande `answer` doit correspondre à la dérivée par rapport à `a`, cette dérivée étant donnée par `logit(a,b1,b2,b3,2)`. En effet, la variable `D0`, qui figure dans la deuxième commande `answer`, est créée par la commande

```
gen D0 = dldF*dFde*e
```

exactement comme dans la commande du fichier `newlogit.ect`. Jusqu'ici, on a l'impression de n'avoir rien gagné. Mais, ensuite, la variable `D1`, créée par la commande

```
gen D1 = D0*x1
```

évite le recalcul de l'expression `dldF*dFde*e`, en utilisant la variable `D0` déjà évaluée. De même pour les deux autres dérivées, données par `D2` et `D3`.

Pour que la procédure serve à gagner du temps, il faut qu'elle ne soit exécutée qu'une seule fois par itération de l'algorithme de `ml`. À chaque fois qu'on demande une réponse d'une procédure, les valeurs des arguments sont comparées à celles qui étaient courantes lors de l'appel précédent. Si ces valeurs n'ont pas changé, la procédure rend simplement la réponse existante. En revanche, si au moins une des valeurs des arguments a changé, toutes les commandes de la procédure sont de nouveau exécutées. Lors d'une estimation non linéaire, quand les arguments de la procédure sont les paramètres à estimer, les valeurs des paramètres ne changent qu'à la fin d'une itération. Par conséquent, on a précisément ce qu'il faut : On recalcule quand il faut recalculer, mais pas autrement, et, quand il faut recalculer, on évite les calculs répétés.

* * * *

Les *arguments* d'une procédure sont évalués à chaque fois que l'on fait appel à la procédure. Cette évaluation se fait comme si on était dans une commande `set`, assez logiquement, parce que les arguments doivent être des *scalaires*. Le dernier argument, qui correspond à l'indice de la réponse souhaitée, est calculé de la même manière, et ensuite interprété comme un *entier*. Si cet entier n'est pas positif, il y a erreur de syntaxe. Le dernier argument une fois pris en compte, si le nombre d'arguments restants est inférieur au nombre d'arguments déclaré lors de la définition de la procédure, il y a encore une fois erreur de syntaxe. En revanche, s'il y a trop d'arguments, les derniers, après le nombre déclaré, sont simplement perdus, sans commentaire de la part d'***Ects***.

* * * *

Ce n'est qu'après l'exécution des commandes de la procédure que l'on voit si ou non il existe une réponse correspondant à l'indice demandé. Sinon, un message d'erreur s'affiche. Si une réponse existe, elle est transmise à la commande qui avait fait appel à la procédure. Si cette commande est `set` ou

une commande assimilée, comme `procedure` elle-même quand elle est en train d'évaluer ses arguments, seul le premier élément de la réponse est utilisé. Sous `gen` ou assimilé, seules les lignes de l'échantillon en cours de validité, défini par la commande `sample`, sont modifiées. Sous `mat` ou assimilé, la matrice donnée par la procédure est utilisée telle quelle.

La prochaine estimation que l'on trouve dans `proclogit.ect`, avec la procédure qui la précède, illustre quelques-uns des points cités au dernier paragraphe. Voici les commandes :

```
procedure argt 4
   sample 1 2
   answer a
   answer b1
   gen bb2 = b2
   answer bb2
   answer b3
end

mlopg logit(argt(a,b1,b2,b3,1), argt(a,b1,b2,b3,2), \
argt(a,b1,b2,b3,3), argt(a,b1,b2,b3,4),1)
deriv a = logit(a,b1,b2,b3,2)
deriv b1 = logit(a,b1,b2,b3,3)
deriv b2 = logit(a,b1,b2,b3,4)
deriv b3 = logit(a,b1,b2,b3,5)
end
```

La procédure `argt` n'a d'utilité que pour cette illustration. Elle sert simplement à rendre les quatre arguments `a`, `b1`, `b2`, et `b3`. Elle sert aussi à démontrer qu'une procédure peut faire appel à une autre procédure.

Vu que la procédure `logit`, en lisant ses arguments, fait comme la commande `set`, seul le premier élément de la troisième réponse, `bb2`, est conservé. En fait, `bb2` a deux éléments, grâce à la commande `sample 1 2` que l'on voit dans la procédure `argt`. Il est important de noter que, à l'intérieur d'une procédure, la commande `sample` n'a qu'un effet *local*. À l'issue de la procédure, la commande est oubliée, et l'échantillon qui était en cours à l'entrée reprend son effet. Mais les effets de toutes les autres commandes persistent après l'exécution de la procédure. Il y a pourtant une autre commande qui ne pourrait jamais fonctionner dans une procédure : `quit`. Si on met un `quit` dans une procédure, il n'aura pour effet que l'affichage d'un message d'erreur.

On peut s'assurer que la commande `sample 1 2` à l'intérieur de la procédure a eu son effet au moyen des commandes suivantes :

```
set i = rows(bb2)
show i
```

La fonction `rows` prend un seul argument, le nom d'une variable existante. La valeur est simplement le nombre de lignes de cette variable. (Un *row* d'une matrice signifie en anglais une ligne de la matrice.) De même, la fonction

`cols` renvoie le nombre de colonnes de son argument. Utilisées sous `set` et `mat`, la valeur de la fonction est un scalaire ; sous `gen`, on obtient un vecteur dont chaque élément compris dans l'échantillon en cours est égal à cette valeur scalaire.

Si la commande `procedure` peut conduire à des avantages non négligeables, elle peut aussi conduire à des inconvénients. Considérez la suite du fichier `proclogit.ect`, qui contient les commandes suivantes :

```
set i = 3

procedure logithess i-2
   sample 1 100
   gen lf = lhd
   answer lf
   gen D0 = dldF*dFde*e
   answer D0
   gen D1 = D0*x1
   answer D1
   gen D2 = D0*x2
   answer D2
   gen D3 = D0*x3
   answer D3
   gen D00 = -f
   answer D00
   gen D01 = x1*D00
   answer D01
   gen D02 = x2*D00
   answer D02
   gen D03 = x3*D00
   answer D03
   gen D11 = x1*D01
   answer D11
   gen D12 = x2*D01
   answer D12
   gen D13 = x3*D01
   answer D13
   gen D22 = x2*D02
   answer D22
   gen D23 = x3*D02
   answer D23
   gen D33 = x3*D03
   answer D33
end

set a = 0
set b1 = 0
set b2 = 0
set b3 = 0
def X = a*iota + x1*b1 + x2*b2 + x3*b3

mlhess logithess(a,1)
```

```
deriv a = logithess(a,2)
deriv b1 = logithess(a,3)
deriv b2 = logithess(a,4)
deriv b3 = logithess(a,5)
second a,a = logithess(a,6)
second a,b1 = logithess(a,7)
second a,b2 = logithess(a,8)
second a,b3 = logithess(a,9)
second b1,b1 = logithess(a,10)
second b1,b2 = logithess(a,11)
second b1,b3 = logithess(a,12)
second b2,b2 = logithess(a,13)
second b2,b3 = logithess(a,14)
second b3,b3 = logithess(a,15)
end
```

À la lecture des deux premières lignes de ce programme, on constate que l'argument qui donne le nombre d'arguments de la procédure n'est pas forcément une simple constante numérique. Cet argument, comme les indices des réponses, est calculé comme sous `set`, et la valeur est interprétée comme un entier. Si l'entier est négatif, un message d'erreur s'affiche. En revanche, rien n'empêche qu'une procédure n'ait pas d'arguments ; seulement, une telle procédure ne serait jamais exécutée qu'une seule fois.

L'inconvénient évoqué tout à l'heure se manifeste dans la commande `mlhess`. Si les dérivées de la fonction de log-vraisemblance sont spécifiées normalement, ***Ecta*** est en mesure de calculer les dérivées secondes analytiquement au besoin. Mais, si elles sont définies au moyen d'une procédure, la différentiation automatique n'est plus disponible. Dans ce cas, il faut spécifier explicitement toutes les dérivées secondes. Ceci peut se faire, comme c'est le cas ci-dessus, à l'intérieur de la même procédure, ici `logithess`, qui génère la fonction de log-vraisemblance et ses dérivées premières et secondes. Encore une fois, une telle pratique permet d'éviter beaucoup de calculs répétés.

Encore trois remarques sur le programme ci-dessus. *Primo*, on voit que les valeurs des paramètres sont réinitialisées avant d'entamer l'estimation non linéaire, afin que les performances de toutes les différentes commandes puissent être comparées directement. Sinon, chaque commande convergerait en une seule itération, parce que les paramètres auraient déjà les valeurs estimées par ML. *Secondo*, la procédure `logithess` n'est définie qu'avec un seul argument. En général, les valeurs de tous les paramètres changent d'une itération à la suivante, et il n'est nécessaire de tenir compte que de l'un d'eux pour déclencher le recalcul de la procédure au début de chaque itération. Ce fait permet de simplifier les écritures. Pourtant, il est important de noter que nous ne pouvons plus nous servir des variables `arg2`, *etc.*, parce que seul `arg1` est définie. Mais ceci ne nous concerne pas ici, parce que `logithess` ne fait pas appel à ces variables, préférant un emploi direct des paramètres

`a`, `b1`, *etc.* *Tertio*, il s'avère nécessaire de redéfinir la macro `X`. La définition donnée à l'intérieur de `logit` est justement en termes des variables `arg`, qui n'interviennent pas dans `logithess`.

Passons maintenant à la dernière estimation trouvée dans `proclogit.ect`, effectuée par `mlar`. Elle permet d'illustrer encore quelques points d'intérêt général. Voici la procédure :

```
procedure logitar 1
   sample 1 100
   gen D = 1/sqrt(F*(1-F))
   gen Dl = (y-F)*D
   answer Dl
   gen D0 = f*D
   answer D0
   gen D1 = x1*D0
   answer D1
   gen D2 = x2*D0
   answer D2
   gen D3 = x3*D0
   answer D3
   answer D
end
```

On voit qu'il est parfaitement possible de générer une variable, ici `D`, au début de la procédure, de l'utiliser pour la génération d'autres variables, et de la déclarer comme la toute dernière réponse. On a une liberté complète en ce qui concerne l'ordre des réponses. On pourrait faire d'abord tous les calculs, et ensuite déclarer les réponses dans l'ordre le plus logique du point de vue de l'utilisateur.

Ensuite on fait

```
equation diffb1 b1 = logitar(a,3)
equation diffb2 b2 = logitar(a,4)
mlar lhd
lhs = logitar(a,1)
deriv a = logitar(a,2)
deriv diffb1
deriv diffb2
deriv b3 = f*x3*logitar(a,6)
end
```

On fait appel ici à une nouvelle commande, `equation`. Cette commande n'introduit aucune fonctionnalité nouvelle ; elle sert simplement à faciliter les écritures et éventuellement à clarifier la structure d'un programme. Après `equation`, on met un nom que l'on pourra utiliser par la suite à la place d'une équation, c'est-à-dire, tout ce qui peut s'exprimer comme

<variable> `=` *<expression>*

Ainsi, le nom `diffb1` est associé à l'équation

```
b1 = logitar(a,3)
```

Une telle équation doit intervenir dans les commandes `set`, `gen`, `mat`, `def`, `nls`, `nliv`, et dans les lignes de toutes les commandes d'estimation non linéaire qui commence par `deriv` ou `second`. Une telle équation intervient dans la commande `equation` elle-même. Si on souhaite donner deux noms à une même équation, on peut faire

```
equation nom2 nom1
```

par exemple, où `nom1` est déjà définie.

EXERCICES:

Dans la commande `mlar` qu'on vient de considérer, on trouve la ligne

```
lhs = logitar(a,1)
```

Cette ligne a la forme d'une `equation`. Définissez une équation appropriée, et utilisez-la dans la commande `mlar` à la place de la ligne `lhs`.
Dans le chapitre 14 de DM, on étudie une régression artificielle à longueur double, c'est-à-dire, une régression qui a deux fois le nombre d'observations du vrai échantillon. Mettez en œuvre l'une des procédures à longueur double que vous trouverez dans ce chapitre. Vous constaterez pourquoi il est important de permettre une taille d'échantillon différente à l'intérieur et à l'extérieur d'une procédure.

Finalement, la toute dernière commande de `proclogit.ect` avant `quit` est tout simplement `showall`. Au moyen de cette commande, on peut savoir l'état des tables internes d'***Ects***. D'abord on a une liste de toutes les variables connues au moment de lancer la commande `showall`. Les éléments de la liste apparaissent comme suit (on ne donne qu'une petite sélection) :

```
Variables currently defined are
Dl:  dimensions 100 x 1
PI: dimensions 1 x 1
R2:  dimensions 1 x 1
TOL: dimensions 1 x 1
XtXinv:  dimensions 4 x 4
```

On y voit les noms des variables avec leurs dimensions matricielles.

* * * *

Toutes les variables d'***Ects*** sont des matrices. Un scalaire est une matrice 1×1, et une variable est un vecteur $n \times 1$, pour un entier $n > 1$.

* * * *

Après, on trouve la liste des macros, définies par `def` (on ne donne qu'une seule macro) :

```
Macros currently defined are
F:
e 1 e +(2) /(2)
```

La première ligne est le nom de la macro, et sur la ligne suivante on a sa représentation interne ; comme celle donnée par `differentiate`. Si vous

aimez déchiffrer les rébus, vous pouvez essayer de deviner le sens de cette représentation.

Ensuite on a la liste des procédures, dont voici un extrait :

```
Procedures currently defined are
argt
logit
```

et, finalement, les équations :

```
Equations currently defined are
diffb1
diffb2
```

* * * *

Si vous voulez étudier la représentation interne des expressions, il existe une commande `expand` qui affichera la représentation d'une seule macro sur demande. Si vous faites

```
def X = a + b1*x1 + b2*x2
expand X
```

Ects affichera

```
X is:
a b1 x1 *(2) +(2) b2 x2 *(2) +(2)
```

Cette commande n'a pas beaucoup d'intérêt pour la plupart des utilisateurs.

* * * *

Tout ce qui figure dans les listes affichées par `showall` peut être détruit, c'est-à-dire, éliminé de la mémoire de l'ordinateur, par la commande `del`. Les ordinateurs modernes ont très souvent beaucoup de mémoire vive, mais, sur une machine moins moderne, il est parfois nécessaire de faire un peu de nettoyage au moyen de `del`.

EXERCICES:

Rajoutez une commande `del` à la fin du fichier `proclogit.ect`, par exemple

```
del diffb1 lhd logit
```

et refaites `showall`. Vous verrez que les objets qui figurent dans la liste qui suit `del` ne sont plus reconnus par ***Ects***.

Chapitre 3
Aide à la Simulation

1. Simulation Récursive

L'énorme puissance des ordinateurs modernes a fait que la simulation est devenue un outil de la toute première importance dans beaucoup de disciplines, dont l'économétrie n'est pas la moindre. Le chapitre 6 de Man2 est consacré aux expériences Monte Carlo, qui sont, bien entendu, basées sur les méthodes de simulation. Dans ce chapitre, je fais état des nombreuses fonctionnalités nouvelles de la version 3 d'***Ects***, qui peuvent aider à la mise en œuvre de simulations de toutes sortes.

La quasi-totalité des simulations font appel à une boucle, qui sert à faire un grand nombre de fois une suite d'opérations, avec des réalisations différentes des éléments aléatoires des opérations à chaque fois. Le mode d'opération d'***Ects*** est de lire chaque ligne du fichier de commandes et ensuite d'exécuter la commande trouvée sur cette ligne. Avec un tel mode d'opération, l'exécution des boucles est fatalement beaucoup plus lente que si on écrivait un programme directement en C++ ou tout autre langage de programmation approprié. Dans les versions antérieures du logiciel, les commandes d'une boucle sont lues et relues pour chaque itération de la boucle, démultipliant ainsi les lenteurs intrinsèques entraînées par la boucle. Les boucles de la version 3 sont déjà beaucoup plus rapides, parce que maintenant les commandes ne sont lues qu'une seule fois. Mais, malgré cette amélioration, il reste que les boucles donnent lieu à des lenteurs.

Même en dehors du contexte des simulations, il faut parfois utiliser des boucles pour générer des séries de manière *récursive*. L'exemple le plus simple est une série AR(1), c'est-à-dire, une série qui constitue un processus autorégressif à l'ordre 1. Ce genre de processus est traité dans la section 6.4 de Man2. L'équation définissante d'une série AR(1) y_t est

$$y_t = \rho y_{t-1} + u_t,$$

où u_t est un bruit blanc et ρ est un paramètre dont la valeur absolue est inférieure à 1.

Considérez le fichier `argen.ect`. On y trouve quatre manières de générer une série AR(1). La première est celle proposée dans Man2 :

```
set N = 1000
set NREP = 100
set rho = 0.5
sample 1 N
gen u = random()
set i = 0

while i < NREP
   set i = i+1
   gen t = time(-1)
   gen rhot = rho^t
   gen y = conv(rhot, u)
end
```

Ce bout de programme sert à créer une série AR(1) de 1.000 observations, avec $\rho = 0,5$. La boucle, qui fait la même chose 100 fois, n'a d'autre but que de mesurer le temps de calcul. Sur un Pentium 166, l'exécution de cette boucle prend 5 secondes environ avec la version 3, contre plus de 15 secondes avec la version 2. On voit clairement l'accélération due au fait que la boucle n'est lue qu'une seule fois.

La méthode que nous venons d'utiliser n'est pas complètement transparente, grâce à l'emploi de la fonction `conv`. Une méthode plus naïve se servirait d'une boucle explicite plutôt que de cette fonction :

```
set NREP = 20
set i = 0
while i < NREP
   set i = i+1
   set j = 1
   set y(1) = u(1)
   while j < N
      set j = j+1
      set y(j) = rho*y(j-1) + u(j)
   end
end
```

Ici, on n'a refait la génération de la série que 20 fois, mais le temps de calcul est déjà proche de 30 secondes. Par rapport à la première méthode, le calcul est presque 30 fois plus long !

EXERCICES:

Vérifiez que les deux méthodes génèrent exactement la même série. Pour éviter une attente trop longue, il est conseillé de faire `set NREP = 1` avant de lancer le programme.

On n'a pas toujours la possibilité de trouver une astuce comme celle qu'on exploite dans la première méthode pour faire une simulation récursive. Ce qu'on aimerait faire est écrire une commande comme

```
gen y = rho*lag(1,y)+u
```

Mais l'opération de cette commande n'est pas récursive. Si la variable `y` existe avant que la commande ne soit lancée, elle est d'abord retardée par la fonction `lag`. Le résultat est ensuite multiplié par `rho` et rajouté à `u`. Sinon, il y a tout simplement erreur de syntaxe. Dans l'un, comme dans l'autre cas, on n'a pas d'opération récursive.

Le résultat souhaité peut être obtenu si la commande est exécutée à l'intérieur d'une commande `recursion`. Dans le cas le plus simple, on ferait

```
set NREP = 100
set i = 0
while i < NREP
   set i = i+1
   recursion
   gen yy = rho*lag(1,yy)+u
   end
end
```

Le temps de calcul est maintenant d'environ 80 secondes, soit deux fois plus rapide que la boucle explicite, mais toujours beaucoup plus lent que la méthode avec `conv`. La conclusion à tirer est bien claire : Quand on trouve des astuces qui permettent d'éviter les boucles explicites ou même la commande `recursion`, il faut en profiter.

Comme les autres commandes qui s'étalent sur plusieurs lignes, `recursion` doit être terminée explicitement par `end`, parce que – on le verra plus tard – on peut avoir plusieurs commandes `gen` dans une `recursion`. Ceci est très utile dans les cas où il n'existe pas d'astuce pour éviter une récursion explicite.

2. Processus ARMA

Dans l'analyse des **séries temporelles**, ou **chronologiques**, les processus les plus couramment rencontrés sont les **processus ARMA**. Ces processus sont traités brièvement dans la chapitre 10 de DM, et de manière beaucoup plus complète dans Hamilton (1994). Le processus ARMA(p, q) s'écrit comme suit :

$$y_t = \sum_{i=1}^{p} \rho_i y_{t-i} + u_t + \sum_{j=1}^{q} \alpha_j u_{t-j}, \tag{12}$$

où la série u_t est un bruit blanc. Les ρ_i et α_j, avec la variance σ^2 de u_t, sont les paramètres du processus.

Il est pratique de noter un processus ARMA(p, q) en termes de deux polynômes, A et B, de la manière suivante :

$$A(\mathrm{L})y_t = B(\mathrm{L})u_t, \tag{13}$$

où A et B se définissent comme

$$A(x) = 1 - \rho_1 x - \rho_2 x^2 - \ldots - \rho_p x^p = 1 - \sum_{i=1}^{p} \rho_i x^i, \text{ et}$$

$$B(x) = 1 + \alpha_1 x + \alpha_2 x^2 + \ldots + \alpha_q x^q = 1 + \sum_{j=1}^{q} \alpha_j x^j.$$

On note L l'**opérateur retard**, dont la propriété définissante s'écrit comme $\mathrm{L}y_t = y_{t-1}$. De même, $\mathrm{L}^i y_t = y_{t-i}$. On vérifie aisément que (12) et (13) sont deux formulations équivalentes.

Le processus AR(p) est une spécialisation du processus ARMA(p, q), avec $q = 0$. L'équation définissante est donc $A(\mathrm{L})y_t = u_t$. Pareillement, le processus MA(q) est un ARMA avec $p = 0$, défini par l'équation $y_t = B(\mathrm{L})u_t$. Simuler un MA(q) ne présente aucune difficulté, parce qu'aucune récursion n'est nécessaire. Il suffit de faire

```
gen u = random()
gen y = u + a1*lag(1,u) + ... + aq*lag(q,u)
```

Il existe un autre moyen, plus compliqué, de faire la même chose. On le décrit ici non pas parce qu'il est préférable dans le cas que nous traitons, mais parce que, dans d'autres cas, il permet de faire des choses intéressantes.

Le programme suivant, dans lequel on génère un processus MA(3) à partir d'un bruit blanc `u` à variance 1, illustre les deux approches.

```
sample 1 20
set a1 = 0.2
set a2 = 0.3
set a3 = -0.1
gen u = random()
gen y = u + a1*lag(1,u) + a2*lag(2,u) + a3*lag(3,u)
lagpoly 1 a1 a2 a3
gen yy = polylag(u)
```

La deuxième approche fait appel à la commande `lagpoly`.[6] Le nom de la commande s'inspire de *polynôme en l'opérateur retard*, où *retard* = *lag*. Le polynôme $B(\mathrm{L})$ du processus MA(3) est ici $B(\mathrm{L}) = 1 + \mathtt{a1}\mathrm{L} + \mathtt{a2}\mathrm{L}^2 + \mathtt{a3}\mathrm{L}^3$. La commande

```
lagpoly 1 a1 a2 a3
```

met en mémoire précisément ce polynôme.

* * * *

Il y a un petit danger dans la formulation des polynômes par `lagpoly`. Par exemple, si on faisait

[6] Note de la version 4 : Cette commande n'existe pas dans la version 4. On a trouvé un bien meilleur moyen de générer les séries ARMA ; voir la documentation de la version 4 pour les détails.

```
lagpoly 1 0.2 0.3 -0.1
```

cette commande serait interprétée par ***Ecta*** comme

```
lagpoly 1 0.2 0.2
```

parce que, en effet, $0,3 - 0,1 = 0,2$. On peut éviter cet inconvénient soit en utilisant des variables, ou des macros, à la place d'une expression arithmétique, soit en séparant les arguments par des virgules. Donc, la commande

```
lagpoly 1, 0.2, 0.3, -0.1
```

ou même

```
lagpoly 1 0.2 0.3, -0.1
```

marcherait correctement.

* * * *

Pour se servir du polynôme ainsi mis en mémoire, on utilise la fonction `polylag`. Cette fonction est disponible dans une commande `gen`, mais pas dans `set` ou `mat`, où elle donne lieu à une erreur de syntaxe. `polylag` prend un seul argument, ici la série `u`. La valeur de la fonction s'écrit comme

$$\texttt{polylag}(u_t) = \texttt{a0}u_t + \texttt{a1}u_{t-1} + \texttt{a2}u_{t-2} + \texttt{a3}u_{t-3},$$

qui est précisément la série qu'on a notée $B(\mathrm{L})u$.

* * * *

La lecture des arguments de `lagpoly` se fait comme dans une commande `mat`, pour une raison que l'on verra plus tard.

* * * *

Pour une série AR(p), on a $A(\mathrm{L})y_t = u_t$. Formellement, ceci devient $y_t = \big(A(\mathrm{L})\big)^{-1}u_t$. Le terme constant du polynôme A est toujours 1 ; nous pouvons donc écrire $A(\mathrm{L}) = 1 + a(\mathrm{L})$. Le théorème du binomial permet d'obtenir l'expression explicite de $\big(A(\mathrm{L})\big)^{-1}$:

$$\big(A(\mathrm{L})\big)^{-1} = \sum_{k=0}^{\infty}\big(a(\mathrm{L})\big)^k. \tag{14}$$

Dans cette écriture, il est entendu que $\big(a(\mathrm{L})\big)^0 = 1$. Il est donc possible d'évaluer explicitement le membre de droite de (14), afin d'obtenir un résultat de la forme

$$\big(A(\mathrm{L})\big)^{-1} = 1 + \sum_{i=1}^{\infty}\alpha_i \mathrm{L}^i. \tag{15}$$

La somme apparemment infinie dans cette expression ne l'est pas dans la pratique, parce que, pour $i > n$, n étant la taille de l'échantillon, $\mathrm{L}^i y_t = 0$ pour tout $t = 1, \ldots, n$ et pour tout vecteur y_t. Le calcul du membre de droite de (15) n'est qu'un problème d'algèbre simple, que l'on peut résoudre de plusieurs manières.

Pour l'utilisateur d'***Ects***, la manière la plus simple est de confier la tâche à ***Ects*** même. Pour ce faire, on utilise la commande `invertlagpoly`. La syntaxe est identique à celle reconnue par `lagpoly`. En faisant

```
invertlagpoly 1 0.2 0.3, -0.1
```

on met en mémoire l'inverse du polynôme $A(\mathrm{L}) \equiv 1 + 0,2\mathrm{L} + 0,3\mathrm{L}^2 - 0.1\mathrm{L}^3$. Bien entendu, la somme infinie qui figure dans l'équation (15) est tronquée après les `smplend` premiers termes. Si par la suite on fait

```
gen y = polylag(u)
```

la série `y` vérifie l'autorégression

$$y_t = -0,2y_{t-1} - 0,3y_{t-2} + 0,1y_{t-3} + u_t.$$

Pour démarrer une telle autorégression, il faut les valeurs des trois premiers éléments du vecteur `y`. Si on fait

```
sample 1 20
invertlagpoly 1 0.2 0.3, -0.1
gen u = random()
gen y = polylag(u)
```

on a implicitement posé $y_0 = y_{-1} = y_{-2} = 0$. En revanche, en faisant

```
sample 1 20
gen y = 0
set y(1) = 3
set y(2) = 4
set y(3) = 5
lagpoly 1 0.2 0.3, -0.1
gen u = polylag(y)
sample 4 20
gen u = random()
sample 1 20
invertlagpoly 1 0.2 0.3, -0.1
gen y = polylag(u)
```

on spécifie explicitement les trois premiers éléments de `y`, et le processus de génération aléatoire ne commence qu'à partir du quatrième élément. En spécifiant les trois premières composantes de `y`, on définit implicitement les trois premières composantes de `u`. Pour expliciter cette définition implicite, il suffit d'utiliser une commande `lagpoly` et ensuite de générer `u` en fonction de `y`. Après, on met dans `u` des éléments aléatoires à partir de la quatrième composante. On peut vérifier le bon fonctionnement du programme par les commandes

```
gen uu = y + 0.2*lag(1,y) + 0.3*lag(2,y) - 0.1*lag(3,y)
print u uu
```

On constatera que les vecteurs `u` et `uu` sont identiques.

Exercices:

Servez-vous de l'équation (14) pour l'évaluation directe de l'inverse du polynôme $A(\mathrm{L})$ aux coefficients $1, 0, 2, 0, 3, -0, 1$. Au besoin, ***Ects*** peut vous aider à faire les multiplications de polynômes. Le polynôme inversé peut être représenté par une série de 20 éléments, qui seront les 20 premiers coefficients. Notons cette série `r`. Démontrez que la commande

```
gen y = conv(r,u)
```

génère la même série `y` que celle obtenue par la commande `polylag`.

La génération d'une série ARMA(p, q) peut se faire en deux étapes. D'abord, on construit la partie MA, c'est-à-dire, la série $B(\mathrm{L})u_t$. Soient `b1`, ..., `bq` les coefficients du polynôme $B(\mathrm{L})$. Après avoir créé la partie MA, on obtient la série ARMA en appliquant l'inverse du polynôme $A(\mathrm{L})$ à la série MA. Soient `a1`, ..., `ap` les coefficients de $A(\mathrm{L})$. On a donc

```
gen u = random()
lagpoly 1 b1 ... bq
gen u = polylag(u)
invertlagpoly 1 a1 ... ap
gen u = polylag(u)
```

Alternativement, on peut générer la partie MA directement, sans faire appel à la commande `lagpoly`. On aurait

```
gen u = u + b1*lag(1,u) + ... + bp*lag(p,u)
```

à la place de la commande `lagpoly` et la commande `gen` qui la suit immédiatement.

Bien entendu, on peut générer une simple série AR(1) au moyen de la commande `invertlagpoly`. À la fin du fichier `argen.ect`, on trouve les deux commandes

```
invertlagpoly 1, -rho
gen y = polylag(u)
```

La première mise en œuvre de la fonction `invertlagpoly` a été excessivement inefficace, de sorte que le temps de calcul du programme ci-dessus était très long, 15 secondes environ. Maintenant, la méthode qui se sert de `invertlagpoly` est presque aussi rapide que celle qui se sert de la fonction `conv`.

3. Processus ARMAX et VAR

En économétrie, il est rare qu'un modèle univarié[7] n'ait que les retards de la variable dépendante comme explicatives. Un cas de figure plus courant est

[7] Un modèle est dit **univarié** si la variable dépendante est un scalaire. Les modèles où la variable dépendante est un vecteur se disent **multivariés**, ou encore **bivariés** si le vecteur a deux composantes.

fourni par un modèle de la forme

$$y_t = \boldsymbol{X}_t\boldsymbol{\beta} + \sum_{i=1}^{p} \rho_i y_{t-i} + u_t + \sum_{j=1}^{q} \alpha_q u_{t-q},$$

où le vecteur ligne $\boldsymbol{X}_t$ contient des explicatives exogènes. Ce modèle fait appel à un **processus ARMAX**, ou plus précisément, un ARMAX(p, q). Le 'X' correspond à la notation $\boldsymbol{X}$ pour la matrice des explicatives exogènes.

Numériquement, il n'y a pas de raison de distinguer le terme exogène $\boldsymbol{X}_t\boldsymbol{\beta}$ des terms aléatoires $u_t + \ldots$. La génération du processus peut se faire donc de la manière suivante :

```
gen u = random()
lagpoly 1 a1 ... aq
gen u = polylag(u)
gen Xu = x1*beta1 + ... + xk*betak + u
invertlagpoly 1 rho1 ... rho2
gen y = polylag(Xu)
```

Si $p = 0$, le processus peut être généré directement sans l'utilisation des commandes `lagpoly` et `invertlagpoly`. Si $p = 1$, on obtiendra un temps de calcul beaucoup plus court en utilisant `conv` :

```
gen u = random()
gen u = u + a1*lag(1,u) + ... + aq*lag(q,u)
gen Xu = x1*beta1 + ... + xk*betak + u
gen t = time(-1)
gen rhot = rho^t
gen y = conv(rhot,Xu)
```

EXERCICES:

Générez un processus ARMAX$(p, 0)$ selon la procédure décrite ci-dessus, et vérifiez le bon fonctionnement de la procédure en comparant le vecteur aléatoire u_t à l'expression $y_t - \sum_{i=1}^{p} \rho_i y_{t-i} - \boldsymbol{X}_t\boldsymbol{\beta}$.

Une technique d'estimation très couramment utilisée en économétrie repose sur l'idée d'une **autorégression vectorielle**, ou **VAR**, pour *Vector Auto-Regression*. Une VAR s'écrit comme suit :

$$\boldsymbol{Y}_t = \boldsymbol{\alpha} + \sum_{i=1}^{p} \boldsymbol{A}_i \boldsymbol{Y}_{t-i} + \boldsymbol{U}_t, \tag{16}$$

où le vecteur $\boldsymbol{Y}_t$ est de la forme $m \times 1$, ainsi que le vecteur constant $\boldsymbol{\alpha}$, et le vecteur $\boldsymbol{U}_t$ des aléas. Les matrices $\boldsymbol{A}_i$ sont carrées, de la forme $m \times m$. On pourrait rajouter un terme $\boldsymbol{B}\boldsymbol{X}_t$, avec un vecteur $\boldsymbol{X}_t$, de la forme $k \times 1$, constituée d'explicatives exogènes, et une matrice de paramètres $\boldsymbol{B}$ de la forme $m \times k$, mais ceci est plutôt rare dans la pratique, sauf si $\boldsymbol{X}_t$ ne contient que

des variables purement déterministes, telles des variables saisonnières ou des tendances temporelles.

L'estimation des paramètres d'un modèle VAR est très simple, parce qu'elle peut être effectuée par un simple OLS, comme l'illustre le fichier `var.ect`. On utilise les données du fichier `ols.dat`, soit 100 observations de 4 variables. L'estimation d'un modèle VAR avec ces 4 variables et avec 3 retards ($p = 3$) est effectuée par les commandes suivantes :

```
sample 1 100
read ols.dat y1 y2 y3 y4
gen Y = colcat(y1,y2,y3,y4)
gen Y1 = lag(1,Y)
gen Y2 = lag(2,Y)
gen Y3 = lag(3,Y)
ols Y c Y1 Y2 Y3
```

La commande `ols` accepte une matrice à plusieurs colonnes comme variable dépendante ; ici la matrice `Y` dont les 4 colonnes sont les 4 variables du modèle. Chacune des colonnes de `Y` est régressée sur la constante et les trois premiers retards des quatre variables, pour un total de 13 régresseurs.

Ensuite le vecteur $\hat{\boldsymbol{\alpha}}$ et les trois matrices $\hat{\boldsymbol{A}}_i$, $i = 1, 2, 3$, sont extraits de la grande matrice `coef` :

```
mat alpha = coef(1,1,1,4)
mat A1 = coef(2,5,1,4)
mat A2 = coef(6,9,1,4)
mat A3 = coef(10,13,1,4)
```

Un modèle VAR peut s'exprimer en termes d'un **polynôme matriciel** en l'opérateur retard. Le polynôme qui correspond modèle (16) s'écrit comme

$$\boldsymbol{A}(\mathrm{L}) \equiv \mathbf{I} - \sum_{i=1}^{p} \boldsymbol{A}_i \mathrm{L}^i,$$

d'où on trouve que

$$\boldsymbol{A}(\mathrm{L})\boldsymbol{Y}_t = \boldsymbol{\alpha} + \boldsymbol{U}_t. \tag{17}$$

Pour simuler un processus VAR, on peut encore une fois se servir de la commande `lagpoly`. Dans `var.ect` on a les commandes suivantes :

```
gen iota = 1
mat Alpha = iota*alpha
mat U = Alpha+res
mat I = A1^0
lagpoly I, -A1, -A2, -A3
gen UU = polylag(Y)
mat tt = UU-U
mat tt = tt'*tt
show tt
showlagpoly
```

Les lignes de la matrice `U`, transposées, représentent le membre de droite de (17). Notez que chaque ligne du produit matriciel `Alpha`, dont la forme est 100×4, est égale au vecteur $\hat{\boldsymbol{\alpha}}$, lui aussi transposé. Pour créer rapidement une matrice identité 4×4, on utilise le fait que toute matrice 4×4 à la puissance 0 est cette matrice identité. Si on fait tourner `var.ect`, on verra que tous les éléments de la matrice `tt` sont nuls. Ceci démontre que la matrice `UU`, créée par `polylag`, et la matrice `U` sont identiques.

* * * *

On voit maintenant que la raison pour laquelle les commandes `lagpoly` et `invertlagpoly` lisent leurs arguments à la manière de la commande `mat` est que ceci permet d'utiliser des polynômes matriciels.

* * * *

L'opération inverse, qui permet de simuler la matrice $\boldsymbol{Y}$, est plus intéressante. Pour l'effectuer, il suffit d'utiliser `invertlagpoly` à la place de `lagpoly`.

```
invertlagpoly I, -A1, -A2, -A3
gen YY = polylag(U)
mat tt = YY-Y
mat tt = tt'*tt
show tt
showlagpoly
```

Encore une fois, tous les éléments de `tt` sont nuls. On peut donc reproduire `Y` à l'identique à partir de la matrice `U` qui contient les constantes et les aléas. Il en découle que le processus VAR défini par les paramètres $\hat{\boldsymbol{\alpha}}$ et $\hat{\boldsymbol{A}}_i$ peut être simulé en remplaçant la matrice `res` par une matrice aléatoire, qui pourrait être générée par `random`, par exemple.

Après chacune des commandes `show tt`, on trouve la commande `showlagpoly`. Cette commande existe surtout pour les fins du programmeur plutôt que pour celles de l'utilisateur d'***Ects***, mais elle permet ici de voir la nature des polynômes en l'opérateur retard créés par `lagpoly` et `invertlagpoly`, qu'ils soient scalaires ou matriciels. Le polynôme mémorisé grâce à la commande `lagpoly` est décrit dans le fichier de sortie de la manière suivante :

```
The number of terms in the lag polynomial is 4:

At lag 0:
  1.000000    0.000000    0.000000    0.000000
  0.000000    1.000000    0.000000    0.000000
  0.000000    0.000000    1.000000    0.000000
  0.000000    0.000000    0.000000    1.000000
At lag 1:
  0.227360   -0.038240   -0.070657   -0.064302
 -1.327870   -0.526401   -0.049297   -0.211917
 -1.272313   -0.227526    0.263922   -0.369891
 -1.273761    0.022772    0.190848   -0.654683
At lag 2:
  0.348915    0.046788   -0.049145    0.016212
```

```
  1.066620    0.277487    0.028368    0.168480
  0.254965    0.213882    0.012838   -0.116361
 -0.127309    0.013068    0.123233    0.542982
At lag 3:
 -0.015650    0.032638    0.017103   -0.014088
 -0.969398   -0.278550   -0.038755   -0.093459
 -0.592363    0.085799    0.173043   -0.164176
  0.174623    0.068046   -0.000404   -0.256209
```

On voit que, comme il se doit, le polynôme a quatre termes, dont le premier est la matrice identité, les trois autres étant les matrices $\boldsymbol{A}_i$, $i = 1, 2, 3$. Le polynôme créé par `invertlagpoly` est beaucoup plus compliqué, parce que, mathématiquement, il a un nombre infini de termes. Heureusement, on peut se passer de tous les termes qui n'aurait d'effet qu'après la fin de l'échantillon, dont la taille est ici égale à 100. Ce fait est annoncé dans le fichier de sortie :

```
The number of terms in the lag polynomial is 100:

At lag 0:
  1.000000    0.000000    0.000000    0.000000
  0.000000    1.000000    0.000000    0.000000
  0.000000    0.000000    1.000000    0.000000
  0.000000    0.000000    0.000000    1.000000
At lag 1:
 -0.227360    0.038240    0.070657    0.064302
  1.327870    0.526401    0.049297    0.211917
  1.272313    0.227526   -0.263922    0.369891
  1.273761   -0.022772   -0.190848    0.654683
```

La liste continue inexorablement jusqu'à ce que 100 matrices aient été imprimées. Mais on peut constater que les matrices tendent finalement vers zéro. En fait, on voit

```
At lag 43:
 -0.000000   -0.000000   -0.000000   -0.000000
 -0.000000    0.000000    0.000000   -0.000000
  0.000000   -0.000000   -0.000000    0.000000
  0.000000    0.000000    0.000000    0.000000
```

et, du $43^{\text{ième}}$ retard jusqu'au $100^{\text{ième}}$, toutes les matrices sont nulles. Ceci est dû au fait que le polynôme est *stationnaire* : Dans d'autres cas, les termes successifs pourraient tendre vers l'infini.

EXERCICES:

Inversez des polynômes scalaires simples au moyen de la commande

```
invertlagpoly 1 rho
```

pour des valeurs différentes de `rho`. Pouvez-vous caractériser les valeurs qui produisent des polynômes stationnaires, dont les termes tendent vers 0, et celles qui donnent lieu à une explosion des termes du polynôme ?

On a démontré que les commandes `lagpoly` et `invertlagpoly` marchent correctement en vérifiant que les séries contenues dans la matrice `Y` peuvent être reproduites à partir des résidus. Dans une vraie simulation, il y a un autre aspect du problème à prendre en compte. Même dans l'absence d'une autocorrélation des résidus, on peut s'attendre à ce que les m résidus d'une ligne $\boldsymbol{U}_t$ soient corrélés entre eux : On parlerait alors d'une **corrélation contemporaine** des résidus.

* * * *

Le phénomène de corrélation contemporaine est prise en compte par les estimations SUR : voir la section 3.3 de Man2 et le chapitre 9 de DM.

* * * *

La matrice de covariance contemporaine peut être estimée par la matrice $\hat{\boldsymbol{\Sigma}}$, de la forme $m \times m$, donnée par

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

où $\hat{\boldsymbol{U}}$ est la matrice $n \times m$ des résidus de la régression vectorielle (16). Afin d'effectuer une simulation de (16), il faut générer une matrice $\boldsymbol{U}^*$ d'aléas simulés, dont chaque ligne $\boldsymbol{U}_t^*$ est un tirage de la loi normale multivariée $\mathrm{N}(\mathbf{0}, \hat{\boldsymbol{\Sigma}})$.

* * * *

Ici, et plus loin, l'étoile (*) signifie qu'il s'agit d'une grandeur simulée.

* * * *

Comment faire ? Soit $\boldsymbol{A}$ une matrice $m \times m$ telle que

$$\boldsymbol{A}\boldsymbol{A}^\top = \boldsymbol{\Sigma}. \tag{18}$$

Si le vecteur $\boldsymbol{z}$ à m composantes est un tirage de la loi $\mathrm{N}(\mathbf{0}, \mathbf{I})$, on calcule que

$$\mathrm{Var}(\boldsymbol{A}\boldsymbol{z}) = \mathrm{E}(\boldsymbol{A}\boldsymbol{z}\boldsymbol{z}^\top\boldsymbol{A}^\top) = \boldsymbol{A}\mathrm{E}(\boldsymbol{z}\boldsymbol{z}^\top)\boldsymbol{A}^\top = \boldsymbol{A}\mathbf{I}\boldsymbol{A}^\top = \boldsymbol{A}\boldsymbol{A}^\top = \boldsymbol{\Sigma}.$$

Il en découle que $\boldsymbol{A}\boldsymbol{z}$ est un tirage de la loi $\mathrm{N}(\mathbf{0}, \boldsymbol{\Sigma})$.

La première étape consiste donc à trouver la matrice $\boldsymbol{A}$. On explique dans Man2 que la fonction `uptriang` fait le nécessaire : la commande

```
mat A = uptriang(Sigma)
```

génère une matrice `A`, de forme triangulaire supérieure, qui vérifie la relation (18). Dans la version 3.3 d'***Ects***, on a rajouté la fonction `lowtriang`, identique à `uptriang` sauf que la matrice générée est triangulaire inférieure. Les commandes suivantes, trouvées dans `var.ect`, illustrent l'utilisation de ces deux fonctions :

```
mat Sigma = (res'*res)/100
mat B = uptriang(Sigma)
mat A = lowtriang(Sigma)
show A B
mat M1 = B*B'
mat M2 = A*A'
show Sigma M1 M2
```

En faisant tourner `var.ect`, on voit que `A` et `B` ont les formes souhaitées, et que les matrices `M1` et `M2` sont égales à la matrice de covariance estimée `Sigma`.

La matrice $\boldsymbol{A}$ une fois en main, la simulation se fait en trois lignes :

```
gen Us = colcat(random(), random(), random(), random())
mat Us = Us*A'
gen Ys = polylag(Alpha+Us)
```

La matrice `Us`, de la forme 100×4, est générée en faisant appel 4 fois à la fonction `random`. Ensuite la matrice est multipliée à droite par `A'`, la transposée de $\boldsymbol{A}$. Finalement la matrice simulée, `Ys`, est créée par la fonction `polylag`.

EXERCICES:

Pourquoi faut-il utiliser la transposée de la matrice $\boldsymbol{A}$ dans la simulation ?

Comment peut-on créer des matrices $\boldsymbol{A}$ et $\boldsymbol{B}$ triangulaires, supérieure et inférieure, telle que $\boldsymbol{A}^\top\boldsymbol{A} = \boldsymbol{B}^\top\boldsymbol{B} = \boldsymbol{\Sigma}$?

Créez des graphiques permettant de comparer les 4 séries simulées aux 4 séries de base dans le fichier `ols.dat`. Quelle est la principale différence entre les séries simulées et les séries de base ? Pouvez-vous expliquer cette différence ?

4. Processus ARCH et GARCH

On ne peut pas faire ici un exposé utile sur les processus ARCH et GARCH, actuellement beaucoup utilisés en économétrie, surtout en économétrie financière. Le lecteur peut se référer au chapitre 16 de DM, et à la littérature abondante qui y est citée. La définition formelle d'un processus ARCH(p) s'écrit comme

$$u_t = h_t^{1/2} v_t, \quad v_t \sim \text{IID}(0,1), \quad h_t = \sigma_t^2 = \alpha + \sum_{i=1}^{p} \gamma_p u_{t-i}^2. \tag{19}$$

Sur la base d'un bruit blanc v_t, on construit la série u_t de manière à ce que la variance σ_t^2 de u_t soit une fonction des p premiers retards de u_t. Cette construction inspire la terminologie **ARCH**, qui signifie *Auto-Regressive Conditional Heteroskedasticity*, ou, en français, hétéroscédasticité conditionnelle autorégressive.

* * * *

Un **bruit blanc** est un processus dont les éléments sont indépendants, d'espérance nulle, et homoscédastiques, c'est-à-dire, ayant tous la même variance σ^2. Un bruit blanc est souvent normal, surtout dans le contexte des simulations, mais pas forcément.

* * * *

Dans le fichier `archdemo.ect` on trouve plusieurs manières de générer un processus ARCH(1). La première fait appel à la commande `recursion`, qu'on a déjà vue dans le contexte des séries AR(p). Après des commandes préliminaires,

```
set n = 100
sample 1 n
set gamma = 0.4
set alpha = 1
```

qui servent à définir la taille de l'échantillon et les paramètres α et γ du processus, on commence par l'initialisation de la série h_t, du bruit blanc v_t, et du processus u_t qui sera un ARCH(1) :

```
gen h = 1
set h(1) = alpha/(1-gamma)
gen v = random()
gen u = 0
set u(1) = sqrt(h(1))*v(1)
```

La commande `set h(1) = alpha/(1-gamma)` n'est pas nécessaire à la mise en œuvre. Toutefois, elle n'est pas entièrement dépourvue de sens, parce qu'elle affecte à `h(1)` l'espérance de la série h_t, si celle-ci est stationnaire.

* * * *

Pour de plus amples renseignements, voir le chapitre 16 de DM.

* * * *

Ensuite, on a la récursion qui correspond à (19), de façon tout à fait explicite :

```
sample 2 n
recursion
   gen h = alpha + gamma*(lag(1,u))^2
   gen u = sqrt(h)*v
end
```

On voit ici l'un des avantages majeurs de la commande `recursion` : Elle peut contenir un nombre arbitrairement grand de commandes `gen`, dont chacune sera appliquée récursivement, dans l'ordre. Il est important de noter que *seule* la commande `gen` peut paraître à l'intérieur d'une `recursion`. Toute autre commande donne lieu à un message d'erreur. Il est aussi important de noter l'effet de la déclaration de la taille de l'échantillon par `sample`. Pour éviter de défaire l'initialisation de `h` et `u`, il faut commencer la récursion à la deuxième observation.

Le fonctionnement de la commande `gen` à l'intérieur d'une `recursion` est assez différent de son fonctionnement habituel. En commençant par l'observation `smplstart`, c'est-à-dire la première observation de l'échantillon déclaré par `sample`, chaque commande `gen` à l'intérieur de la récursion est appliquée à une seule observation, en utilisant le mode de calcul employé par la commande

`set`. Après avoir traversé toutes les commandes de la récursion, on passe à l'observation suivante, jusqu'à ce qu'on arrive à l'observation `smplend`, la dernière observation de l'échantillon déclaré. C'est pour cette raison que la fonction `lag`, dont l'utilisation est cruciale à la récursion, trouve à chaque fois la valeur appropriée. Si on essayait de faire une récursion en marche arrière, avec des valeurs avancées plutôt que retardées, la démarche serait vouée à l'échec, parce que les évaluations procèdent toujours de `smplstart` à `smplend`.

La définition du processus GARCH(p, q) est donnée par l'équation

$$u_t = h_t^{1/2} v_t, \quad v_t \sim \text{IID}(0,1), \quad h_t = \sigma_t^2 = \alpha + \sum_{i=1}^{p} \gamma_i u_{t-i}^2 + \sum_{j=1}^{q} \delta_j h_{t-j}.$$

Comme pour les processus ARCH, on part d'un bruit blanc v_t, que l'on multiplie par un écart-type $h_t^{1/2}$, défini cette fois-ci d'une manière plus complexe : h_t dépend non seulement du passé du processus u_t, mais aussi de son propre passé. On remarque qu'un ARCH(p) est un GARCH$(p, 0)$. En effet, le 'G' de GARCH signifie *Generalized*, ou généralisé.

EXERCICES:

Dans le programme ci-dessus, la série `u` a été initialisée de manière à avoir exactement 100 éléments. Quelles sont les conséquences si la série `u` est initialisée avec un plus petit ou un plus grand nombre d'éléments ? Démontrez que, si d'abord il y en a trop peu, les éléments manquants seront créés, et que, s'il y en a trop, ceux qui ne sont pas compris dans l'échantillon déclaré (de 2 à 100) restent inchangés.

Écrivez un programme permettant de générer un processus ARCH(p) pour $p = 3$, et un processus GARCH(p, q) pour $p = 3$, $q = 4$. Prenez soin d'initialiser correctement, et de commencer la récursion à la bonne observation après l'initialisation.

Il est entièrement possible de générer un (G)ARCH au moyen d'une boucle explicite. Une telle boucle est même nécessaire avec les anciennes versions du logiciel, où la commande `recursion` n'existe pas. Pour l'exemple d'un processus ARCH(1), on trouve dans `archdemo.ect` un bout de code similaire à ce qui suit :

```
sample 1 n
gen v = random()
gen h = 0
gen u = 0
set h(1) = alpha/(1-gamma)
set u(1) = sqrt(h(1))*v(1)
set j = 2
while j < n
     set h(j) = alpha + gamma*(u(j-1))^2
     set u(j) = sqrt(h(j))*v(j)
     set j = j+1
end
```

La structure logique est identique à celle du code qui utilise `recursion`. Cette fois-ci, il faut que les séries `h` et `u` soient définies pour tout l'échantillon avant de lancer la boucle ; sinon, les commandes `set` n'auraient pas d'effet en dehors de l'échantillon déclaré au moment de la création de `h` et `u`.

EXERCICES:

Comme on l'a fait pour les différentes méthodes de génération des processus AR(1), construisez des boucles qui génèrent plusieurs fois un processus ARCH(1), d'abord au moyen d'une `recursion`, après par une boucle explicite, afin de comparer les temps de calcul des deux méthodes. L'avantage de la récursion par rapport à la boucle explicite est plus important pour le processus ARCH(1) que pour le processus AR(1).

Pour les processus ARCH(1), il existe une astuce qui permet de les générer très rapidement. Malheureusement, l'astuce ne s'applique qu'aux processus ARCH(1) et ne se généralise pas. On la trouve dans `archdemo.ect`, où elle est utilisée pour la troisième méthode de génération dans ce fichier. Le code pertinent est comme suit :

```
sample 1 n
gen v = random()
gen x = gamma*v*v
set x(1) = x(1)/(1-gamma)
gen b = lag(1,x)
set b(1) = 1
gen tmp = product(1/b)
gen h = alpha*sum(tmp)/tmp
set h(1) = alpha/(1-gamma)
gen u = sqrt(h)*v
```

On peut remarquer l'emploi de la fonction `product`, disponible pour la première fois dans la version 3.3 d'***Ects***. La syntaxe est la même que celle utilisée par `sum`, mais un produit est calculé plutôt qu'une somme.

* * * *

Si on n'a pas la version la plus récente d'***Ects***, on peut obtenir le résultat de la commande

```
gen tmp = product(1/b)
```

par la commande

```
gen tmp = exp(sum(-log(b)))
```

Vu que tous les éléments de la série `b` sont positifs, les logarithmes existent. Mais, dans d'autres applications, il faut veiller à ce qu'il n'y ait pas d'éléments négatifs.

* * * *

EXERCICES:

Utilisez la fonction `product` pour calculer les factoriels des entiers de 1 à 10.

Le programme qui génère un ARCH(1) peut être utilisé comme une recette de cuisine. Dans la suite, je donne les calculs précis, mais uniquement pour ceux et celles qui souhaitent comprendre son fonctionnement, qui n'est pas tout à fait évident. De la définition (19), pour le cas où $p = 1$, on voit que

$$h_t = \alpha + \gamma u_{t-1}^2 = \alpha + \gamma h_{t-1} v_{t-1}^2.$$

En résolvant récursivement, on trouve que

$$\begin{aligned} h_2 &= \alpha + \gamma h_1 v_1^2, \\ h_3 &= \alpha + \alpha\gamma v_2^2 + \gamma^2 v_2^2 v_1^2 h_1, \\ h_4 &= \alpha + \alpha\gamma v_3^2 + \alpha\gamma^2 v_3^2 v_2^2 + \gamma^3 v_3^2 v_2^2 v_1^2 h_1, \ldots \end{aligned}$$

Pour $t > 2$, soit $x_t = \gamma v_t^2$. Pour $t = 1$, $x_1 = \gamma v_1^2 h_1/\alpha$. On trouve alors que, pour $t > 1$,

$$h_t = \alpha \left(1 + \sum_{s=1}^{t-1} \prod_{u=0}^{s-1} x_{t-1-u} \right).$$

On peut simplifier cette expression si on adopte la règle selon laquelle tout produit qui n'a aucun facteur est égal à 1. On obtient ainsi, pour $t > 1$,

$$h_t = \alpha \sum_{s=0}^{t-1} \prod_{u=t-s}^{t-1} x_u,$$

qui, après quelques manipulations algébriques, s'écrit également comme

$$h_t = \alpha \left(\prod_{u=1}^{t-1} \frac{1}{x_u} \right)^{-1} \sum_{s=1}^{t} \prod_{u=1}^{s-1} \frac{1}{x_u}.$$

Dans le programme ***Ects***, la série `x` correspond à x_t. Il en découle que la série `tmp` représente $\prod_{u=1}^{t-1}(1/x_u)$, au sens où l'élément t de `tmp` est égal à ce produit, le premier élément étant simplement égal à 1. Il est important que le premier élément de la série `b`, qui est la série `x` retardée, soit égal à 1, afin d'éviter les divisions par zéro qui se produiraient si on gardait la valeur par défaut 0. La série `tmp` une fois définie, on voit que l'élément t de `sum(tmp)` est

$$\sum_{s=1}^{t} \texttt{tmp}_s = \sum_{s=1}^{t} \prod_{u=1}^{s-1} \frac{1}{x_u}.$$

Maintenant, on voit sans trop de peine que `alpha*sum(tmp)/tmp` représente h_t, sauf pour le premier élément. Mais parce qu'on sait que cet élément est `alpha/(1-gamma)`, il suffit de préciser ce fait.

* * * *

Dans le fichier `archdemo.ect`, on trouve encore une méthode qui permet de générer un processus ARCH(1), mais elle est déconseillée. Elle s'y trouve pour servir d'avertissement. Pas toutes les méthodes possibles ne sont d'une grande utilité !

* * * *

EXERCICES:

Comparez le temps de calcul de la méthode que nous venons de décrire avec celui de la procédure par récursion. L'astuce n'est peut-être pas très élégante, mais elle s'avère très efficace.

5. Rééchantillonnage et le Bootstrap

Le **bootstrap** est une méthode très générale qui permet de faire des inférences statistiques sur la base de simulations. L'idée au cœur de la méthode est que, si l'on ne connaît qu'approximativement la distribution, sous l'hypothèse nulle qu'on teste, d'une statistique de test, on peut souvent en obtenir une meilleure approximation en faisant des simulations sous l'hypothèse nulle. Une introduction aux méthodes du bootstrap se trouve dans Efron et Tibshirani (1993). On va maintenant considérer des exemples qui devraient éclairer la méthode et qui devraient en même temps illustrer des fonctionnalités d'***Ects*** qui facilitent sa mise en œuvre.

On sait que les Students calculés lors d'une estimation par moindres carrés non linéaires (NLS) ne suivent la loi de Student qu'asymptotiquement sous les hypothèses nulles qu'ils testent. Par conséquent, les inférences basées sur ces Students ne sont qu'approximatives. Afin de voir comment le bootstrap peut améliorer cette circonstance, prenons un cas concret. Soit la régression non-linéaire

$$\boldsymbol{y} = \alpha\boldsymbol{\iota} + \beta\boldsymbol{x}_1 + (1/\beta)\boldsymbol{x}_2 + \boldsymbol{u}. \tag{20}$$

où on note $\boldsymbol{\iota}$ le vecteur constant dont chaque élément égale 1. Le fichier `gv.dat` contient 10 observations des trois variables $\boldsymbol{y}$, $\boldsymbol{x}_1$, et $\boldsymbol{x}_2$. Sur la base de ce très petit échantillon, on souhaite tester l'hypothèse de la bonne spécification du modèle (20), contre l'alternative linéaire

$$\boldsymbol{y} = \alpha\boldsymbol{\iota} + \beta\boldsymbol{x}_1 + \gamma\boldsymbol{x}_2 + \boldsymbol{u}, \tag{21}$$

avec un γ non contraint.

Le modèle (20) peut être estimé par les NLS, d'où on obtient des estimations $\tilde{\alpha}$ et $\tilde{\beta}$ des paramètres. La régression de Gauss-Newton (GNR) qui correspond au modèle (20) s'écrit comme

$$\boldsymbol{y} - \alpha - \beta\boldsymbol{x}_1 - (1/\beta)\boldsymbol{x}_2 = b_\alpha\boldsymbol{\iota} + b_\beta\big(\boldsymbol{x}_1 - (1/\beta^2)\boldsymbol{x}_2\big) + \text{résidus}\,; \tag{22}$$

voir le chapitre 6 de DM. Celle qui correspond au modèle (21) s'écrit naturellement comme

$$\boldsymbol{y} - \alpha - \beta\boldsymbol{x}_1 - \gamma\boldsymbol{x}_2 = b_\alpha\boldsymbol{\iota} + b_\beta\boldsymbol{x}_1 + b_\gamma\boldsymbol{x}_2 + \text{résidus}. \tag{23}$$

Afin d'obtenir une statistique de test, on évalue toutes les variables des deux GNR, (22) et (23), en les valeurs obtenues par l'estimation de l'hypothèse

nulle, à savoir, $\alpha = \tilde{\alpha}$, $\beta = \tilde{\beta}$, $\gamma = 1/\tilde{\beta}$. Les deux GNR ont ainsi la même régressande. Ensuite on exprime la GNR (23) sous la forme d'une régression augmentée par rapport à (22). On vérifie sans peine que les régresseurs de la régression

$$\boldsymbol{y} - \tilde{\alpha} - \tilde{\beta}\boldsymbol{x}_1 - (1/\tilde{\beta})\boldsymbol{x}_2 = b_\alpha \boldsymbol{\iota} + b_\beta\big(\boldsymbol{x}_1 - (1/\tilde{\beta}^2)\boldsymbol{x}_2\big) + b_\delta \boldsymbol{x}_2 + \text{résidus} \quad (24)$$

engendrent le même espace linéaire que ceux de (23), et que (24) est simplement la GNR (22), évaluée en $\tilde{\alpha}$ et $\tilde{\beta}$, avec un régresseur supplémentaire, $\boldsymbol{x}_2$. On a changé le nom du paramètre fictif associé à ce régresseur, parce que le paramètre fictif ne correspond plus au paramètre γ.

Si on fait tourner les deux régressions artificielles ainsi obtenues, on peut utiliser les deux sommes des carrés des résidus afin de calculer un Fisher. Mais, vu que le test n'a qu'un seul degré de liberté, on peut utiliser le Student associé au paramètre fictif b_δ donné par (24), et se passer ainsi de la GNR (22) correspondant à l'hypothèse nulle. Il existe une autre possibilité, due au fait que la GNR (24) est évaluée en les valeurs $\tilde{\alpha}$ et $\tilde{\beta}$ estimées sous l'hypothèse nulle. Cette deuxième possibilité est le nR^2 de (24), où $n = 10$ est la taille de l'échantillon, et le R^2 est non centré.

Les premières commandes du fichier `gv.ect` servent à effectuer les opérations décrites ci-dessus. Les voici :

```
set n = 10
sample 1 n
read gv.dat y x1 x2
set beta = 1
nls y = alpha + beta*x1 + x2/beta
deriv alpha = 1
deriv beta = x1 - x2/(beta^2)
end
set sigma = sqrt(errvar)
gen Rb = x1 - x2/(beta^2)
ols res c Rb x2
set t = student(3)
set nR2 = n*R2
set Pt = 2*(1 - tstudent(abs(t),n-3))
set PnR2 = 1 - chisq(nR2,1)
show Pt PnR2
```

Pour éviter des ennuis, il est prudent d'affecter au paramètre β une valeur non nulle avant de lancer la commande `nls`. Sinon, on aura des problèmes numériques reliés à des dénominateurs nuls. La commande qui fait tourner la GNR est la suivante :

```
ols res c Rb x2
```

et voici les résultats pertinents, extraits du fichier de sortie :

```
ols res c Rb x2
```

```
Ordinary Least Squares:

Variable  Parameter estimate  Standard error  T statistic

constant    -31.165705          17.686371      -1.762131
Rb            5.511348           3.566145       1.545464
x2            6.505508           3.586173       1.814053

Number of observations = 10   Number of regressors = 3
R squared (uncentred) = 0.319780  (centred) = 0.319780
```

Les deux dernières commandes servent à calculer les P values asymptotiques associées aux deux tests, c'est-à-dire, les niveaux de significativité marginaux. Les valeurs affichées à l'écran sont :

```
Pt = 0.112545
PnR2 = 0.073737
```

Ces valeurs sont assez différentes. La différence témoigne du fait qu'une approximation asymptotique peut fonctionner assez mal si la taille de l'échantillon n'est que de 10.

Avant d'aller plus loin, une remarque s'impose concernant le R^2 d'une régression linéaire. Le coefficient de détermination, qu'on appelle souvent simplement R^2, peut être défini de plusieurs manières, dont les versions antérieures d'***Ects*** ne reconnaissent qu'une seule, le R^2 dit **non centré**, qui est le rapport de la somme des carrés expliqués (en ***Ects***, `sse`) à la somme des carrés totaux (en ***Ects***, `sst`). On accède au R^2 non centré par la variable `R2`. Quoique le R^2 n'ait pas d'interprétation statistique formelle en général, il est utile, comme dans l'exemple que nous traitons, pour le calcul de certaines statistiques de test. Si la constante figure parmi les régresseurs d'un modèle estimé par les OLS, on pourrait s'intéresser au R^2 dit **centré**. Cette deuxième version du R^2 est le R^2 fourni par une régression où toutes les variables non constantes, y compris la variable dépendante, sont remplacées par les écarts par rapport à leurs moyennes respectives, et la constante est supprimée. Dans la version 3.3 d'***Ects***, si la constante `c` figure parmi les régresseurs d'une commande `ols`, le tableau de résultats contient deux R^2, le premier non centré, le second centré, comme on les voit dans le listing ci-dessus. Le R^2 centré est disponible sous le nom de `R2c`.

* * * *

Pour que le R^2 centré soit disponible, il faut que la constante soit désignée par le nom `c`. Si on définit la constante explicitement sous un autre nom, par exemple par `gen iota = 1`, alors la commande `ols y iota x` ne fournit pas le R^2 centré.

* * * *

EXERCICES:

Expliquez pourquoi les deux R^2 sont identiques pour la GNR que nous venons d'estimer.

Asymptotiquement, et sous l'hypothèse nulle, le Student que nous venons de calculer est un tirage de la loi normale centrée réduite, et le nR^2 un tirage de la loi du χ^2 à un degré de liberté. En échantillon fini, les statistiques sont des tirages d'autres lois, généralement inconnues analytiquement. Mais on peut les étudier par simulation. La démarche est comme suit : On établit d'abord un processus générateur de données (PGD) dit **PGD bootstrap**. Ce PGD bootstrap doit impérativement vérifier l'hypothèse nulle. Ensuite on utilise le PGD bootstrap pour tirer des échantillons artificiels, simulés, de la même taille que l'échantillon de données réelles. Pour chacun de ces échantillons simulés, on calcule une ou plusieurs statistiques de test, exactement comme on les a calculées pour les données réelles. Finalement, on construit ce qu'on appelle la **fonction de répartition empirique** des statistiques simulées. La fonction de répartition empirique est la version simulée de la vraie fonction de répartition, inconnue, des statistiques en échantillon fini sous l'hypothèse nulle. Quand le nombre de simulations tend vers l'infini, l'écart entre la fonction empirique et la vraie fonction inconnue tend vers zéro.

Une difficulté pratique résulte du fait que l'hypothèse nulle n'est pas en général limitée à un seul PGD. Dans notre exemple, sous l'hypothèse nulle, il y a trois paramètres inconnus, α, β, et σ^2, la variance des aléas. Afin de spécifier le PGD bootstrap, il faut affecter des valeurs précises à ces paramètres. La solution la plus simple et la plus efficace est d'utiliser les estimations des paramètres sous l'hypothèse nulle. Un autre aspect que l'hypothèse nulle ne précise pas complètement est la loi de probabilité suivie par les aléas. On résoudra ce dernier problème de deux manières totalement différentes, mais qui conduisent à des résultats très similaires. La première approche est de faire l'hypothèse que les aléas sont générés par la loi normale. La seconde est de pratiquer un **rééchantillonnage** des résidus de l'estimation de l'hypothèse nulle. La première approche constitue le **bootstrap paramétrique**, la seconde le **bootstrap non paramétrique**, ou **semi-paramétrique**.

Le bootstrap paramétrique

On jette un coup d'oeil d'abord sur le bootstrap paramétrique, dont la mise en œuvre est un peu plus facile. Le PGD bootstrap peut s'écrire de la manière suivante :

$$\boldsymbol{y}^* = \tilde{\alpha}\boldsymbol{\iota} + \tilde{\beta}\boldsymbol{x}_1 + (1/\tilde{\beta})\boldsymbol{x}_2 + \hat{\sigma}\boldsymbol{v}^*, \quad \boldsymbol{v}^* \sim N(\mathbf{0}, \mathbf{I}).$$

Encore une fois, les étoiles signifient des grandeurs simulées. Les explicatives $\boldsymbol{x}_1$ et $\boldsymbol{x}_2$, supposées exogènes, sont les mêmes pour chaque simulation. Il est clair que ce PGD vérifie l'hypothèse nulle.

Dans la suite de `gv.ect`, on trouve :

```
set B = 999
sample 1 B
gen tt = 0
gen nRR2 = 0
```

```
def fnreg = a + b*x1 + x2/b
def dbeta = x1 - x2/(b{}2)

sample 1 n
gen rfn = alpha + beta*x1 + x2/beta
set i = 0
silent
noecho
while i < B
   set i = i+1
   gen ys = rfn + sigma*random()
   set a = alpha
   set b = beta
   nls ys = fnreg
   deriv a = 1
   deriv b = dbeta
   end
   ols res CG x2
   set tt(i) = student(3)
   set nRR2(i) = n*R2
end
echo
restore
```

Ici, on note `B` le nombre de simulations. Pour des raisons théoriques, on préfère choisir `B` tel que $B+1$ soit un chiffre bien rond, plutôt que `B` lui-même, d'où le choix `B = 999`. Les vecteurs `tt` et `nRR2` sont crées pour contenir les `B` statistiques simulées. Étant donné que l'expression $\tilde{\alpha} + \tilde{\beta}\boldsymbol{x}_1 + (1/\tilde{\beta})\boldsymbol{x}_2$ est la même pour toutes les simulations, on la calcule une fois pour toutes dans le vecteur `rfn`. En plus, on définit les macros `fnreg` et `dbeta` afin d'accélérer les calculs à l'intérieur de la boucle. On fait ainsi une fois pour toutes le travail de l'interprétation des expressions qui représentent la fonction de régression et sa dérivée par rapport à β.

Les simulations proprement dites se trouvent à l'intérieur de la boucle. Le vecteur `ys` représente $\boldsymbol{y}^*$. Dans le contexte des simulations que nous faisons ici, les « vraies » valeurs des paramètres, c'est-à-dire, les valeurs du PGD, sont connues. Vu que l'algorithme d'estimation non linéaire converge d'autant plus vite que le point de départ est proche de l'estimation, il est prudent d'initialiser les paramètres des régressions simulées (`a` et `b` à la place de `alpha` et `beta`) par les vraies valeurs. Pour la GNR, on utilise le fait que la commande NLS crée les variables `c` et `Rb` automatiquement comme les deux colonnes de la variable `CG`. La commande `ols res CG Rb` sert donc à faire tourner la GNR, après quoi on sauve les deux statistiques simulées dans les vecteurs prévus à cet effet.

À ce stade, on est prêt à construire les fonctions de répartition empirique des deux statistiques. Ceci se fait aisément si l'on se sert de la fonction

`cdf` disponible pour la première fois dans la version 3.3 d'***Ects***. Le nom de la fonction provient de l'expression anglaise pour signifier une fonction de répartition, la *Cumulative Distribution Function*. La suite du programme montre comment l'utiliser.

```
sample 1 101
set linestyle = 1
gen x = -4 + 8*time(-1)/100
mat edft = cdf(tt,x)
gen lstud = tstudent(x,n-3)
plot (x edft lstud)
gen x = 10*time(-1)/100
mat edfR2 = cdf(nRR2,x)
gen lchi2 = chisq(x,1)
plot (x edfR2 lchi2)
```

On cherche à construire une représentation graphique des fonctions de répartition empirique des 999 réalisations de chacune des deux statistiques. Ces réalisations sont contenues dans les vecteurs `tt` et `nRR2`. Afin de tracer les graphiques des deux fonctions de répartition, il faut d'abord choisir les points en lesquels les fonctions seront évaluées. Les choix doivent être différents pour les deux statistiques, parce que le Student peut prendre des valeurs négatives, mais non le nR^2. Pour le Student, on choisit une grille de 101 points, espacés régulièrement sur l'intervalle $[-4, 4]$. Pour le nR^2, la grille s'étend de 0 à 10. Le meilleur moyen de créer les grilles est d'employer la fonction `time`, parce que, formellement, les grilles ne sont que des tendances temporelles.

La fonction `cdf`, utilisée dans une commande `mat`, prend deux arguments.

* * * *

Ce fait signifie que la règle, énoncée dans Man2, selon laquelle toute fonction utilisée dans une commande `mat` ne prend qu'un seul argument, sous peine d'erreur de syntaxe, a dû être assouplie dans la version actuelle du logiciel. Dans le chapitre suivant, on trouvera d'autres fonctions nouvelles, qui elles aussi enfreignent la vieille règle.

* * * *

L'effet de la commande

```
mat fre = cdf(stat,abscisses)
```

où le premier argument, `stat`, est une matrice $B \times m$, et le deuxième, `abscisses`, une matrice $n \times 1$, est de créer une matrice `fre` de la forme $n \times m$, avec autant de lignes que `abscisse`, et autant de colonnes que `stat`. Notons x_i, $i = 1, \ldots, n$, l'élément i de `abscisses`. Alors l'élément (i, j) de `fre`, pour $i = 1, \ldots, n$, $j = 1, \ldots, m$, est la proportion des éléments de la colonne j de `stat` qui sont inférieurs ou égaux à x_i.

Notons b_k, $k = 1, \ldots, B$, l'élément k de la première colonne de `stat`. La définition formelle de la fonction de répartition $\hat{F}$ des B éléments de cette

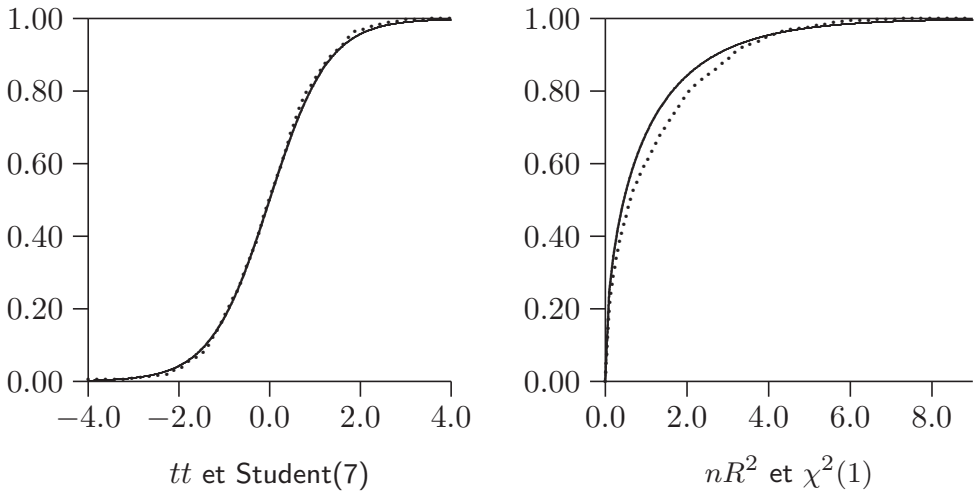

Figure 5 Fonctions de répartitions empiriques et théoriques

colonne est comme suit :

$$\hat{F}(x) = \frac{1}{B}\sum_{k=1}^{B} I(b_k \leq x), \tag{25}$$

où la valeur de la **fonction indicatrice** $I(b_k \leq x)$ égale 1 si l'inégalité qui sert d'argument à la fonction est vérifiée, et 0 sinon. La somme dans la définition (25) est donc le nombre d'éléments b_k inférieurs ou égaux à x. En divisant par B, on obtient la proportion. La conséquence de la définition (25) est que l'élément (i, j) de `fre` est la valeur de la fonction de répartition des éléments de la colonne j de `stat` en x_i, l'élément i de `abscisses`.
Il est intéressant de comparer les répartitions empiriques avec les lois de probabilité données par la théorie asymptotique. Ces lois sont celle de Student, à $n - 3 = 7$ degrés de liberté, et celle du χ^2, à 1 degré de liberté. Dans le programme, on évalue les fonctions de répartition correspondantes en les mêmes points que les répartitions empiriques, et on affiche les résultats par la commande `plot`. On voit ces résultats dans la Figure 5, où les tracés en pointillé sont les fonctions empiriques et les tracés pleins les fonctions théoriques. Il paraît que la loi de Student à 7 degrés de liberté reste une assez bonne approximation, mais que celle du $\chi^2(1)$ l'est moins. Ceci peut expliquer la différence entre les P values asymptotiques données par les deux statistiques.

Le plus souvent, on ne s'intéresse pas à tous les détails de la fonction de répartition empirique d'une statistique. Pour faire des inférences, ce qu'il faut est la P value bootstrap. Toute P value est une mesure de la masse de probabilité dans la ou les queues d'une distribution, au delà de la valeur réalisée de la statistique. Pour le test en χ^2, cette masse de probabilité est 1 moins la valeur de la fonction de répartition du χ^2 en la statistique réalisée. Pour le Student, elle est 2 fois 1 moins la valeur de la fonction de répartition de la loi de Student en la valeur absolue de la statistique réalisée, du moins

pour un test bilatéral. Le calcul des P values bootstrap pour l'exemple dans `gv.ect` se fait comme suit :

```
sample 1 B
gen Pt = abs(tt) > abs(t)
gen PnR2 = nRR2 > nR2
gen iota = 1
mat Pt = iota'*Pt/B
mat PnR2 = iota'*PnR2/B
show Pt PnR2
```

Les vecteurs `Pt` et `PnR2` ont des éléments 1 si la statistique bootstrap est plus loin dans la queue de la distribution que les statistiques calculées avec les données réelles, et 0 sinon. Ensuite, on additionne les éléments de ces vecteurs, et on divise le résultat par `B` pour avoir les P values bootstrap.

* * * *

Rappel : L'utilisation des signes d'égalité (`=`) ou d'inégalité (`<` et `>`) dans une commande **`gen`** crée des valeurs **booléennes**, c'est-à-dire, 1 si l'égalité ou l'inégalité est vérifiée, et 0 sinon.

* * * *

Dans la pratique, si on ne s'intéresse qu'à la P value bootstrap, il est inutile de stocker les valeurs de l'ensemble des statistiques bootstrap, comme on l'a fait dans le vecteurs `tt` et `nRR2`. Il suffit de définir des variables scalaires, initialisées à 0, et de les incrémenter à chaque fois que la statistique bootstrap est plus loin dans la queue de la distribution que la vraie statistique. Quoiqu'il en soit, les valeurs calculées pour notre exemple sont

```
Pt = 0.090090
PnR2 = 0.090090
```

On a eu de la chance en trouvant deux valeurs identiques, mais ce fait indique clairement qu'une inférence basée sur l'une ou l'autre est plus fiable qu'une inférence asymptotique.

Le bootstrap non paramétrique

Considérons maintenant le bootstrap non paramétrique. Cette version du bootstrap ne fait pas d'hypothèse précise sur la loi de probabilité des aléas. À la place d'une telle hypothèse, on effectue un rééchantillonage des résidus donnés par l'estimation de l'hypothèse nulle. Le terme « rééchantillonnage » peut s'expliquer en termes de la fonction de répartition empirique de ces résidus. Alors que le bootstrap paramétrique tire les aléas simulés d'une loi normale, un rééchantillonnage tire les aléas de la loi caractérisée par la fonction de répartition empirique des résidus. On dit « rééchantillonnage » parce que chaque aléa bootstrap est égal à un des résidus de l'estimation d'origine. Ce qui change est que l'ordre des résidus n'est plus respecté, et un même résidu peut être tiré zéro, une, ou plusieurs fois. Afin que chaque aléa bootstrap soit

un tirage de la même loi empirique, le tirage se fait *avec remise*. On entend par là que, si le tirage se faisait en tirant un morceau de papier d'une urne, il faudrait remettre le morceau après chaque tirage.

Notons $\hat{F}(\hat{\boldsymbol{u}})$ la répartition empirique des résidus $\hat{\boldsymbol{u}}$ donnés par l'estimation de la régression (20). Alors, le PGD du bootstrap non paramétrique s'écrit comme

$$\boldsymbol{y}^* = \tilde{\alpha}\boldsymbol{\iota} + \tilde{\beta}\boldsymbol{x}_1 + (1/\tilde{\beta})\boldsymbol{x}_2 + \boldsymbol{u}^*, \quad \boldsymbol{u}^* \sim \hat{F}(\hat{\boldsymbol{u}}).$$

La pratique moderne préfère à ce PGD bootstrap classique une version améliorée, où les résidus sont multipliés par $n/(n-k) = 10/7$ avant de faire l'objet du rééchantillonnage. La raison en est que la variance de la loi empirique des résidus ainsi modifiés est égale à $\hat{\sigma}^2 = \text{SSR}/(n-k)$, l'estimateur sans biais de la variance aléatoire.

Le bootstrap non paramétrique par rééchantillonnage s'effectue au moyen des commandes suivantes, trouvées dans `gv.ect` :

```
gen us = res*sqrt(n/(n-3))
while i < B
   set i = i+1
   gen ys = rfn + us(random(0.9,n+0.9))
   set a = alpha
   set b = beta
   nls ys = fnreg
   deriv a = 1
   deriv b = dbeta
   end
   ols res CG x2
   set tt(i) = student(3)
   set nRR2(i) = n*R2
end
```

Le vecteur `us` contient les résidus modifiés, et le rééchantillonnage se fait par la commande

```
gen ys = rfn + us(random(0.9,n+0.9))
```

le reste de la procédure étant identique au bootstrap paramétrique. Voyons à présent comment et pourquoi l'expression

```
us(random(0.9,n+0.9))
```

constitue un tirage avec remise dans « l'urne » des résidus modifiés.

Le résultat de la commande

`gen B = A(`⟨*expn*⟩`)`

où `A` est une variable déjà présente dans la mémoire de l'ordinateur, est une matrice dont le nombre de lignes correspond à l'échantillon en cours et dont le nombre de colonnes égale celle de `A`. Chaque ligne de `B` est une des lignes de `A`, choisie de la manière suivante. L'argument ⟨*expn*⟩ est évalué, selon les règles de la commande `gen`, avec pour résultat une matrice de la forme `smplend` $\times m$,

$m \geq 0$, dont seule la première colonne sera prise en compte. Correspondant à chaque élément de cette colonne, on calcule un entier selon la règle déjà énoncée dans Man2 pour l'évaluation des indices : Soit x_i l'élément i de la colonne ; l'indice est alors le plus grand entier n_i tel que $n_i \leq (x_i + 0,1)$. Ensuite on affecte à la ligne i de `B` la ligne n_i de `A`.

Si on utilise deux indices, comme par exemple dans la commande

```
gen C = A(⟨expn1⟩,⟨expn2⟩)
```

la variable `C` sera un vecteur colonne, dont l'élément i est l'élément $\texttt{A}(n_i, m_i)$, où n_i et m_i sont respectivement les indices calculés à partir des éléments i des premières colonnes des matrices obtenues par l'évaluation de $\langle expn_1 \rangle$ et $\langle expn_2 \rangle$.

* * * *

À chaque fois que l'on demande un élément inexistant, au moyen d'un indice nul ou négatif par exemple, les éléments de la ligne correspondante du résultat sont nuls. La syntaxe décrite dans les deux paragraphes ci-dessus n'est pas comprise par les versions précédentes d'***Ects***.

* * * *

Quand l'expression

```
random(0.9,n+0.9)
```

est évaluée, le résultat est un vecteur dont chaque élément est un tirage de la loi uniforme sur l'intervalle qui s'étend de 0,9 à `n` + 0,9. On voit que toute réalisation de la loi uniforme comprise dans le segment [0,9, 1,9[donne lieu à un indice égal à 1, que toute réalisation dans [1,9, 2,9[donne un indice égal à 2, et ainsi de suite. Le dernier segment, [`n` − 1 + 0,9, `n` + 0,9[, donne un indice égal à `n`. La loi uniforme affecte à chacun des segments, de longueur 1, la même probabilité, égale à 1/`n`. Il s'en suit que la probabilité que chacun des indices $1, 2, \ldots,$ `n` soit sélectionné est égale à 1/`n`. Ce fait, en combinaison avec la manière dont la commande `gen` interprète les indices, signifie que chaque élément de

```
us(random(0.9,n+0.9))
```

est un tirage, indépendant de tous les autres, de la loi empirique définie par les éléments du vecteur `us`, comme on l'avait souhaité.

Les P values bootstrap donnés par le bootstrap non paramétrique sont encore une fois identiques pour les deux statistiques considérées, et très similaires à celles données par le bootstrap paramétrique :

```
Pt = 0,100100
PnR2 = 0.100100
```

Sur la base de 999 simulations bootstrap, les valeurs de 0,090090 et 0,100100 ne diffèrent pas de manière significative.

Vu que toutes les simulations font appel au générateur de nombres aléatoires, il convient ici d'en parler un peu plus longuement. Comment un appareil

déterministe comme un ordinateur peut-il générer des nombres *aléatoires* ? La réponse directe à la question est que l'ordinateur se sert de ce qu'on appelle en mathématiques le **chaos déterministe**, qui génère des nombres qui, selon les apparences, sont aléatoires. Pour les fins des simulations, les apparences suffisent largement. Il reste que le processus de génération est déterministe, et, par conséquent, reproductible. Si on veut générer deux fois de suite les mêmes nombres « aléatoires », il suffit de préciser le même point de départ. Le point de départ du générateur d'***Ects*** consiste en deux chiffres, contenus dans la variable `seed`. Les deux éléments de cette variable sont mises à jour après chaque appel à la fonction `random` ; on peut les examiner à tout instant en faisant

```
show seed
```

Au moment où ***Ects*** est lancé, la variable `seed` contient les valeurs par défaut : `20000` et `987654321`. Pour changer le point de départ du générateur de nombres aléatoires, on utilise la commande `setseed`. La syntaxe est simple :

`setseed` ⟨$expn_1$⟩ ⟨$expn_2$⟩

Ects affecte au premier élément la valeur de ⟨$expn_1$⟩, calculée selon les règles de la commande `set`, au deuxième la valeur de ⟨$expn_2$⟩. En même temps, la variable `seed` est mise à jour. Il est important de noter qu'il ne suffit pas de changer les éléments de `seed`, parce que le changement n'est pas répercuté à l'intérieur du générateur lui-même. Seul `setseed` permet de changer les valeurs du générateur.

EXERCICES:

Le but de cet exercice est de démontrer les deux facettes du chaos déterministe. D'abord l'aspect déterministe. Générez une série de nombres aléatoires par la commande `random` après avoir pris note des valeurs dans la variable `seed`. Ensuite remettez les mêmes valeurs au moyen de `setseed` et générez une deuxième suite de nombres aléatoires. Vérifiez que les deux suites sont identiques. Ensuite l'aspect chaotique. Refaites l'exercice, mais changez la valeur de `seed(1)` en y rajoutant 1 avant de générer la deuxième suite. Calculez la corrélation entre les deux suites : Elle sera assez proche de zéro.

Chapitre 4
Autres Aspects Nouveaux

1. Fonctions Mathématiques

Le nombre de fonctions mathématiques connues par ***Ects*** a été largement étendu. Maintenant, ***Ects*** reconnaît la quasi-totalité des fonctions trouvées dans la bibliothèque standard du langage C. Dans ce domaine, le C++ repose entièrement sur le C, de sorte que l'on ne perd ni ne gagne rien en passant d'un langage à l'autre.

Dans Man2, on donne une liste de fonctions d'un argument scalaire renvoyant une valeur scalaire. Cette liste comporte les fonctions `log`, `sqrt`, `sign`, `abs`, `exp`, `sin`, `cos`, et `tan`. En plus, on répertorie la fonction `phi`, la fonction de répartition de la loi normale centrée réduite. Attention ! Cette fonction est maintenant disponible sous le nom de `Phi`, avec un 'P' majuscule. La fonction `phi` existe toujours, mais elle donne la *densité* de la loi $N(0,1)$. Si on préfère se passer complètement d'une fonction dont le sens dépend de la version d'***Ects*** qu'on utilise, la densité est aussi disponible sous le nom de `stdnorm`, dont la définition formelle est

$$\phi(x) = \frac{1}{2\pi} \exp\left(-\frac{1}{2}x^2\right).$$

Le changement au sein d'***Ects*** s'inspire du fait que la notation standard veut que $\Phi(x)$ soit la fonction de *répartition* et $\phi(x)$ la *densité* de la loi $N(0,1)$.

Aux fonctions trigonométriques `sin`, `cos`, et `tan` viennent se rajouter les **fonctions hyperboliques**, sous les noms de `sinh`, `cosh`, et `tanh`. Ces fonctions peuvent se définir en termes de la fonction exponentielle :

$$\sinh x = \frac{1}{2}(e^x - e^{-x}), \quad \cosh x = \frac{1}{2}(e^x + e^{-x}), \quad \tanh x = \sinh x / \cosh x.$$

Les inverses de toutes ces fonctions sont disponibles sous les mêmes noms précédés d'un `a`. On a donc les six fonctions `asin`, `acos`, `atan`, `asinh`, `acosh`, et `atanh`. Il existe en mathématiques deux sortes de notations standard pour ces fonctions : On écrit par exemple soit $\sin^{-1}(x)$ soit $\arcsin(x)$. La deuxième notation est celle qui inspire les noms employés par ***Ects***, qui sont identiques à ceux employés par la bibliothèque du C.

On trouve des définitions de la plupart des autres fonctions de cette bibliothèque dans plusieurs références, dont celle que je préfère est Abramowitz et Stegun (1964). On y trouve les fonctions `erf` et `erfc`, qui sont fortement reliées à la distribution $\Phi(\cdot)$ de la loi $N(0,1)$. Les définitions formelles sont

$$\texttt{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2)\, dt \text{ et}$$
$$\texttt{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty \exp(-t^2)\, dt = 1 - \texttt{erf}(x).$$

Ces fonctions s'appellent la **fonction d'erreur** et la **fonction d'erreur complémentaire** respectivement.

D'autres fonctions sont reliées à la loi du khi-deux (χ^2). La fonction de répartition de cette loi est fournie par la fonction `chisq`, qui exige un deuxième argument, le nombre de degrés de liberté. La fonction inverse, que l'on peut utiliser pour évaluer les seuils critiques, est notée `chicrit`. Elle aussi prend les degrés de liberté comme deuxième argument. On rappelle ici que la distribution du χ^2 est définie en termes de la fonction gamma, notée $\Gamma(x)$, dont le logarithme est disponible sous le nom de `gln`. La fonction gamma incomplète, qui prend deux arguments, porte en ***Ects*** le nom de `gammp`, tandis que la fonction bêta incomplète, qui a besoin de trois arguments, porte le nom de `betafn`.

* * * *

Ces fonctions sont définies aux pages 74-75 de Man2. Une erreur s'est glissée dans l'ancienne documentation : La fonction bêta incomplète porte le nom de `betafn`, et non pas de `betacf`.

* * * *

Dans la version 3.3 du logiciel, la fonction `gammp` est aussi disponible sous le nom de `igam`. En plus la fonction `digamma`, qu'on appelle **fonction digamma** ou aussi **fonction psi**, renvoie la dérivée du logarithme de la fonction gamma :

$$\psi(x) = \frac{d \log \Gamma(x)}{dx} = \frac{\Gamma'(x)}{\Gamma(x)}.$$

Cette fonction mérite une remarque concernant la différentiation automatique. La dérivée de $\psi(x)$ ne s'exprime pas sous forme analytique sans appel à des suites infinies. Pour cette raison, la valeur d'une expression comme

```
diff(digamma(x),x)
```

est toujours nulle, quelle que soit la valeur de `x`.[8]

Les lois de Student et de Fisher sont servies par les fonctions `tstudent` et `fisher`, qui fournissent les fonctions de répartition, et les fonctions inverses

[8] Note de la version 4 : Ceci n'est plus le cas. Voir la documentation.

`studcrit` et `fishcrit`. Pour la loi de Student, il faut les degrés de liberté en deuxième argument, pour celle de Fisher, il faut les degrés de liberté du numérateur et du dénominateur, en deuxième et troisième arguments respectivement. La différentiation automatique permet de calculer les dérivées des fonctions de répartition et de leurs inverses, mais uniquement par rapport à l'argument principal. La dérivée par rapport à un argument degrés de liberté est nulle.

Grâce à la bibliothèque standard du C, ***Ects*** est en mesure de calculer les valeurs des **fonctions de Bessel** d'ordre 0 ou 1. Ces fonctions s'écrivent en mathématiques comme $J_0(x)$ et $J_1(x)$ pour les fonctions dites de première espèce, et comme $Y_0(x)$ et $Y_1(x)$ pour les fonctions dites de deuxième espèce.

* * * *

Voir Abramowitz et Stegun (1964), chapitre 9.

* * * *

Ects, ainsi que la bibliothèque du C, utilisent pour ces fonctions les noms `j0`, `j1`, `y0`, et `y1` respectivement. Leurs dérivées, qui sont elles aussi des fonctions de Bessel, peuvent être obtenues par la différentiation automatique.

2. Autres Fonctions

Outre les fonctions mathématiques, ***Ects*** fournit plusieurs fonctions qui permettent de manipuler les matrices de différentes façons. Les utilisations de `colcat`, `diag`, `lag`, `rowcat`, `seasonal`, `sort`, `sum`, `time`, et `uptriang` sont décrites dans Man2 : Elles sont inchangées depuis la version 2 du logiciel.

Dans la section 2.6, on a vu les fonctions `rows` et `cols`, qui prennent un seul argument, dont elles renvoient le nombre de lignes ou de colonnes. Le comportement de la fonction `det`, qui renvoie le déterminant de son argument, a été légèrement modifié pour être exactement conforme à celui de `rows` et `cols` : Sous `set` et `mat`, la valeur du déterminant est scalaire ; sous `gen`, cette valeur est affectée à chaque élément de l'échantillon en cours d'un vecteur colonne.

Deux autres fonctions qui se comportent différemment maintenant sont `max` et `min`. À l'origine, ces fonctions prenaient exactement deux arguments, mais maintenant elles en admettent un nombre arbitraire. S'il n'y en a pas, le résultat est un zéro scalaire sous `set` et, sous `gen`, un vecteur colonne dont chaque élément de l'échantillon en cours est nul. Ces fonctions ne sont toujours pas disponible sous `mat`. S'il y a un seul argument, seul son premier élément est conservé sous `set`, et sa première colonne sous `gen`. Avec plusieurs arguments, chaque élément du résultat, scalaire ou vecteur, est la plus grande ou la plus petite (dans le sens algébrique) des éléments correspondants des premières colonnes des arguments.

Les deux fonctions `greatest` et `smallest` (« le plus grand » et « le plus petit »), comme `max` et `min`, servent à trouver le maximum ou le minimum d'un

ensemble de valeurs, mais leur syntaxe est assez différente. Un seul argument est admis. Sous `set`, la valeur scalaire renvoyée est le maximum ou le minimum, selon le cas, des éléments compris dans l'échantillon courant de la première colonne de l'argument. Sous `gen`, la même valeur est affectée à tous les éléments d'un vecteur colonne qui sont compris dans cet échantillon. Sous `mat`, on obtient une valeur scalaire, qui est le maximum ou le minimum de l'ensemble des éléments de l'argument.

Dans certains cas, on souhaite sélectionner des lignes ou des colonnes d'une matrice donnée. Trois fonctions existent à cette fin, `rowselect`, `colselect`, et `select`. Toutes les trois sont disponibles uniquement sous `mat`. Soit `A` une matrice $n \times m$. Alors le résultat de la commande

```
mat B = rowselect(r,A)
```

est une matrice `B` de la forme $k \times m$, où k est le nombre d'éléments non nuls de la première colonne de `r`. Les k lignes de `B` sont précisément celles qui correspondent à une ligne de `r` dont le premier élément est non nul. Dans l'exemple suivant, on extrait d'une matrice `X` de la forme 100×4, contenant des données trimestrielles, une observation par ligne, les observations qui correspondent à la première saison.

```
sample 1 100
read ols.dat x1 x2 x3 x4
mat A = colcat(x1,x2,x3,x4)
gen r = seasonal(4,1)
mat B = rowselect(r,A)
```

La matrice `B` a exactement 25 lignes, qui sont les lignes $1, 5, 9, \ldots, 97$ de `A`. Si on sélectionne des lignes après la ligne n de la matrice `A`, la ligne correspondante dans `B` sera créée avec des éléments nuls.

La sélection des colonnes se fait de manière similaire, mais avec une syntaxe qui est en quelque sorte la transposée de celle employée par `rowselect`. La commande

```
mat B = colselect(A,col)
```

où `A` a la forme $m \times n$, crée une matrice `B` de la forme $m \times k$, les k colonnes de `B` étant les colonnes de `A` correspondant à un élément non nul de la première ligne de `col`. Voici un exemple où on sélectionne la première colonne d'une matrice et la dernière :

```
sample 1 6
gen a = time(0)
mat A = a*a'
gen col = seasonal(5)
mat col = col'
mat B = colselect(A,col)
```

Cet exemple n'a pas d'interprétation économétrique : Il sert uniquement à illustrer quelques propriétés d'***Ects***. On crée d'abord une matrice `A` de la forme 6×6, dont on souhaite extraire les colonnes 1 et 6. Afin d'effectuer la sélection,

on fait appel à une utilisation nouvelle de la fonction `seasonal`. Dans la version 3.3 d'***Ects***, cette fonction est surchargée. L'utilisation habituelle se voit dans l'exemple de sélection de lignes : Le vecteur `seasonal(4,1)` est la variable saisonnière qui sélectionne la première de 4 saisons. Mais si on ne donne qu'un seul argument à la fonction :

```
gen S = seasonal(m)
```

on crée une matrice de la forme `smplend` × `m` dont les `m` colonnes sont les variables « saisonnières » avec une périodicité de `m`. Ici, si on fait

```
show col
```

on verra (on n'imprime que 4 chiffres par élément)

```
col =
1.000  0.000  0.000  0.000  0.000
0.000  1.000  0.000  0.000  0.000
0.000  0.000  1.000  0.000  0.000
0.000  0.000  0.000  1.000  0.000
0.000  0.000  0.000  0.000  1.000
1.000  0.000  0.000  0.000  0.000
```

Pour les 6 observations de l'échantillon déclaré, on a 5 colonnes, chacune étant une variable indicatrice dont un élément sur 5 est égal à 1, les autres éléments étant nuls.

* * * *

À ce point, il convient de signaler que, à la différence des versions antérieures d'***Ects***, la version 3.3 ne permet plus d'utiliser la fonction `seasonal` sous `mat`.

* * * *

Après la transposition de la matrice `col`, la première ligne devient

```
1.000 0.000 0.000 0.000 0.000 1.000
```

d'où on voit clairement que l'on demande de sélectionner les colonnes 1 et 6.

La fonction `select` permet de sélectionner simultanément des lignes et des colonnes. La syntaxe est illustrée par la commande

```
mat B = select(r, A, col)
```

qui crée une matrice `B` qui ne retient des éléments de `A` que ceux dont la ligne est sélectionnée par un élément non nul de `r` et la colonne par un élément non nul de `col`. Un exemple avec la matrice `A` de l'exemple précédent :

```
gen col = seasonal(5)
mat col = col'
gen r = seasonal(4,1)
mat B = select(r,A,col)
```

On sélectionne les éléments des lignes 1 et 5 et des colonnes 1 et 6 de `A`.

EXERCICES:

Vérifiez que tous les exemples de sélection marchent correctement. Comment pourrait-on créer une matrice `B` dont les k lignes sont des tirages au hasard, sans remise, des n lignes ($n > k$) d'une autre matrice `A` ?

Une opération dont on n'a que rarement besoin, mais qui s'avère parfois indispensable, consiste à inverser l'ordre des lignes d'une matrice. La fonction `reverse` sert à effectuer cette opération. On voit dans le programme suivant comment l'utiliser.

```
sample 1 100
read ols.dat x1 x2 x3 x4
gen X = colcat(x1,x2,x3,x4)
gen XX = reverse(X)
mat Y = reverse(X)
sample 51 90
gen XXX = reverse(X)
```

Il apparaît que la fonction est disponible sous `gen` et sous `mat`. En effet, les matrices `XX` et `Y` seront identiques. La première ligne des deux sera la $100^{\text{ième}}$ de `X`, la deuxième la $99^{\text{ième}}$ de `X`, et ainsi de suite. Mais si on déclare un échantillon qui ne coïncide pas avec les dimensions de l'argument de la fonction, alors, uniquement sous `gen`, seules les lignes de l'échantillon déclaré sont prises en compte. Dans l'exemple, la matrice `XXX` a la forme 90×4. Les 50 premières lignes ont des éléments nuls, la $51^{\text{ième}}$ ligne est la ligne 90 de `X`, et la $90^{\text{ième}}$ est la ligne 51 de `X`.

* * * *

Pour inverser l'ordre des colonnes d'une matrice, il suffit de transposer, d'inverser les lignes de la transposée, et de retransposer.

* * * *

Dans Man2, on parle du **produit Schur** de deux vecteurs. Si on a deux vecteurs $\boldsymbol{x}$ et $\boldsymbol{y}$, leur produit Schur s'écrit comme $\boldsymbol{x} * \boldsymbol{y}$, et son élément t est simplement $x_t y_t$. Ce genre de produit Schur est très facilement obtenu par `gen`: La commande

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

fait le nécessaire. Si le vecteur $\boldsymbol{x}$ est remplacé par une matrice $\boldsymbol{X}$ de la forme $n \times k$, la commande

```
gen Z = X*y
```

donne lieu à une matrice également de la forme $n \times k$ dont les colonnes sont les produits Schur des colonnes successives de $\boldsymbol{X}$ avec le vecteur $\boldsymbol{y}$.

* * * *

Pour la clarté de l'exposé, on suppose que `smplstart` $= 1$ et que `smplend` $= n$.

* * * *

Si maintenant on remplace $\boldsymbol{y}$ par une matrice $\boldsymbol{Y}$ de la forme $n \times l$, la commande

```
gen Z = X*Y
```

crée une matrice de la forme $n \times k$ dont les colonnes sont les produits Schur des colonnes de $\boldsymbol{X}$ avec la *première* colonne de $\boldsymbol{Y}$. Les autres colonnes de $\boldsymbol{Y}$

ne jouent aucun rôle. Il se peut que l'on souhaite créer le produit Schur des deux matrices $\boldsymbol{X}$ et $\boldsymbol{Y}$ dans le sens habituel, où l'élément (i, j) du produit est le produit des éléments (i, j) de $\boldsymbol{X}$ et de $\boldsymbol{Y}$. Dans ce cas, on utilise, sous `mat` uniquement, la fonction `schur`. Si on fait

```
mat Z = schur(X,Y)
```

où `X` a la forme $n \times k$, et `Y` a la forme $m \times l$, la matrice `Z` aura la forme $\min(n, m) \times \min(k, l)$ et son élément (i, j), $1 \leq i \leq \min(n, m)$, $1 \leq j \leq \min(k, l)$, sera égal à $x_{ij}y_{ij}$, dans une notation évidente.

EXERCICES:

Comparez les résultats des commandes suivantes :

```
read ols.dat x1 x2 x3 x4
gen X = colcat(x1,x2)
gen Y = colcat(x3,x4)
mat Z = schur(X,Y)
gen ZZ = X*Y
```

Refaites l'exercice avec des sous-matrices de `X` et `Z` de dimensions différentes.

L'économétrie des modèles multivariés fait appel parfois au **produit Kronecker**. Soit $\boldsymbol{A}$ une matrice $n \times k$ et $\boldsymbol{B}$ une matrice $m \times l$. Le produit Kronecker se note $\boldsymbol{A} \otimes \boldsymbol{B}$. Ce produit est une matrice $nm \times kl$, que l'on peut exhiber sous forme partitionnée comme suit :

$$\boldsymbol{A} \otimes \boldsymbol{B} = \begin{bmatrix} a_{11}\boldsymbol{B} & \dots & a_{1k}\boldsymbol{B} \\ \vdots & \ddots & \vdots \\ a_{n1}\boldsymbol{B} & \dots & a_{nk}\boldsymbol{B} \end{bmatrix}.$$

La matrice est donc constituée de nk blocs de la forme $m \times l$, chacun proportionnel à la matrice $\boldsymbol{B}$. Le bloc en position (i, j) est multiplié par l'élément (i, j) de $\boldsymbol{A}$.

Le produit Kronecker s'obtient, encore une fois sous `mat` uniquement, au moyen de la fonction `kron`. On fait simplement

```
mat C = kron(A,B)
```

pour obtenir le résultat souhaité.

EXERCICES:

Le produit Kronecker n'est pas un produit commutatif. Créez une matrice identité $n \times n$ $\mathbf{I}$ et une colonne quelconque $\boldsymbol{y}$ de la forme $n \times 1$. Calculez les deux produits Kronecker, de la matrice avec le vecteur, et du vecteur avec la matrice, et expliquez les différences que vous observez.

La fonction `diag` sert à extraire d'une matrice ses éléments diagonaux et de les ranger dans un vecteur colonne : Voir la section 3.3 de Man2. Il peut s'avérer utile de ranger les éléments d'un vecteur sur la diagonale principale d'une

matrice carrée, dont les autres éléments seraient nuls. La fonction `makediag` permet de faire cette opération. Le résultat de

```
mat A = makediag(v)
```

où la matrice `v` a n lignes, est une matrice diagonale `A`, de la forme $n \times n$, dont l'élément diagonal i est l'élément i de la première colonne de `v`. Sous `gen`, soit $n =$ `smplend` $-$ `smplstart` $+ 1$. `A` est alors une matrice diagonale $n \times n$ dont les éléments diagonaux sont les éléments de `v` à partir de `smplstart` jusqu'à `smplend`.

Il reste à décrire une fonction matricielle qui est utile quand on veut définir une matrice qui ne contient rien. Considérez le programme schématique suivant :

```
mat A = emptymatrix()
set i = 0
while i < n
     set i = i+1
     gen x = ⟨expn⟩
     mat A = colcat(A,x)
end
```

On souhaite rajouter des colonnes à une matrice `A` à chaque itération d'une boucle. Au début de la boucle, il faut que la matrice `A` soit définie, mais vide. On voit que la fonction `emptymatrix`, qui ne nécessite aucun argument, permet une telle définition.

3. Entrées et Sorties

Les modalités d'entrée et de sortie de données et de texte sont largement étendues et assouplies dans la version 3.3 d'***Ecta***. Nous commençons la discussion par un point mineur, avant d'aborder les changements plus importants. Dans Man2, il est expliqué comment on peut créer un fichier de sortie au moyen de la commande `out`. Malgré la grande utilité de cette commande, il avait un défaut dans les versions antérieures d'***Ecta*** : Si on crée deux fois dans un même fichier de commandes un fichier de sortie dont le nom est le même chaque fois, le premier fichier est écrasé par le deuxième lors de la création de celui-ci. Or, on pourrait très bien vouloir interrompre momentanément l'utilisation d'un fichier de sortie, afin d'en utiliser un autre, ou pour passer en mode interactif, par exemple. Ou bien on pourrait vouloir rajouter des résultats supplémentaires à un fichier déjà existant. Pour toutes ces raisons, la commande `out` n'écrase plus un fichier existant. Les résultats nouveaux sont simplement rajoutés à la fin de l'ancien contenu du fichier. Si on préfère écraser un fichier qui pourrait exister, on peut employer la commande `outnew`, dont le comportment est identique à celui de `out` dans les versions précédentes.

Normalement, ***Ecta*** ne manipule que des fichiers texte. Ce système a des avantages et des inconvénients. Le principal avantage provient du fait que les

fichiers texte existent sous tout système d'exploitation que l'on peut rencontrer, et sont facilement convertibles entre les différents formats qui existent. Un second avantage est tout simplement qu'un fichier texte est lisible non seulement par les ordinateurs mais aussi par les utilisateurs d'ordinateurs.
Le principal inconvénient est illustré par le programme suivant :

```
sample 1 20
gen x = random()
gen y = 2 + x + 3*random()
ols y c x
write tmp.dat y x
read tmp.dat y x
ols y c x
```

Parmi les résultats de la première commande `ols`, on trouve

```
Sum of squared residuals = 232.275424
Explained sum of squares = 40.106680
```

mais la seconde donne

```
Sum of squared residuals = 232.275408
Explained sum of squares = 40.106692
```

Les chiffres sont très similaires, mais ils ne sont pas identiques. La raison en est que, quand les données sont écrites dans le fichier `tmp.dat`, ce fichier texte ne conserve que les 6 premières décimales après la virgule. La petite perte de précision se manifeste par les petites différences que nous venons de constater.

Le problème est d'autant plus grave que les valeurs des éléments des vecteurs sont proches de zéro. Par exemple, si on divise `y` et `x` par 10.000, on trouve

```
Sum of squared residuals = 0.000002
Explained sum of squares = 4.010668e-07
```

avec les données elles-mêmes, et

```
Sum of squared residuals = 0.000002
Explained sum of squares = 4.008570e-07
```

après le filtrage effectué par `write` suivi de `read`. Un coup d'œil dans le fichier `tmp.dat` explique pourquoi : Le premier observation enregistrée dans le fichier est

```
-0.000036 0.000062
```

où seuls deux chiffres ont été conservés. Si l'on divise par 1.000.000, le problème s'aggrave davantage.

Données Binaires

Pour éviter cet inconvénient, on se sert des deux commandes `writematrix` et `readmatrix`. Si on remplace les commandes `write` et `read` par

```
mat X = colcat(y,x)
```

```
writematrix binary tmp.bin X
readmatrix binary tmp.bin X
mat y = XX(1,20,1,1)
mat x = XX(1,20,2,2)
```

les résultats de la régression seront identiques avant et après cette opération.

La commande `writematrix` admet deux syntaxes différentes. Outre celle que nous venons de voir, on a aussi

`writematrix ascii` ⟨*nom de fichier*⟩ `X`

où `X` est la matrice que l'on souhaite sauver dans un fichier qui porte le nom de ⟨*nom de fichier*⟩.

En mode `binary` (binaire), le fichier créé est un fichier **binaire**. Un tel fichier n'est lisible que par un ordinateur. Qui pis est, le format des fichiers binaires dépend à la fois du matériel (PC, Mac, Station Sun, *etc*), et du système d'exploitation. Voilà précisément l'inconvénient que l'on évite en n'utilisant que des fichiers texte. La contrepartie est que, si on se limite à un même matériel et un même système d'exploitation, la précision numérique n'est pas perdue quand on stocke des données dans un fichier binaire. En mode `ascii`, le fichier est toujours un fichier texte. La commande `writematrix` en mode `ascii` est tout à fait similaire à la commande `write`, sauf qu'il ne sauve qu'une seule matrice à la fois.

On a vu que, pour lire une matrice d'un fichier binaire, il suffit de remplacer `writematrix` par `readmatrix`. Il faut, bien entendu, que le fichier qu'on lit ait été créé précédemment par une commande `writematrix` sur le même genre d'ordinateur tournant sous le même système d'exploitation. Sinon, les résultats sont imprévisibles et probablement désastreux. La commande `readmatrix`, comme `writematrix`, peut être employée aussi en mode `ascii`. Dans ce cas, on a accès à des possibilités qui ne sont pas disponibles avec la simple commande `read`. Pour `read`, chaque colonne doit constituer une seule variable, à laquelle il faut affecter un nom. En revanche, `readmatrix` peut lire une matrice de dimensions quelconques, qu'il faut fournir à la fin de la commande. Par exemple,

```
readmatrix ascii tmp.dat X 20 2
```

permet de créer une matrice `X`, de la forme 20×2, dont les éléments sont trouvés, sous format texte, dans `tmp.dat`. Le même fichier de données peut être utilisé pour créer des matrices de dimensions différentes. La commande

```
readmatrix ascii tmp.dat X 8 5
```

rangerait les 40 chiffres trouvés dans `tmp.dat` dans une matrice 8×5. Ces chiffres sont affectés successivement, d'abord aux éléments de la première ligne, ensuite à ceux de la deuxième, et ainsi de suite. Si on arrive à la fin du fichier avant de remplir la matrice, les éléments restants sont nuls. S'il reste des chiffres dans le fichier après la constitution de la matrice complète, ils sont perdus.

EXERCICES:

Faites tourner un programme où les variables `x` et `y` sont divisées par 1.000.000, et sauvées par `write` et relues par `read`. Les résultats de la régression de `y` sur `x` avant et après seront assez différents. Refaites le même exercice en utilisant `writematrix` et `readmatrix` en mode binaire. Les résultats des deux régressions seront maintenant identiques.

La difficulté évoquée ci-dessus est due au fait que, quand on sauve des données dans un fichier texte, on ne conserve que 6 chiffres après la virgule. Avec la version 3.3 du logiciel, on a la possibilité de choisir le nombre de chiffres après la virgule, non seulement dans les opérations `write` et `writematrix` en mode `ascii`, mais aussi dans les opérations effectuées par les commandes `print`, `show`, `put`, ainsi que par une commande nouvelle `putnum`, et dans les affichages des résultats d'une estimation. ***Ecta*** utilise la variable `precision` pour déterminer le nombre de chiffres après la virgule. Sa valeur par défaut égale 6. On peut la changer soit directement par une commande `set`, soit par l'utilisation d'une commande spécifique, `setprecision`, dont la syntaxe est simplement

`setprecision` ⟨*expn*⟩

où la valeur de ⟨*expn*⟩ est calculée selon les règles d'évaluation des indices ; voir la section 2.1 de Man2 et la section 3.5 du présent manuel.

Contrôle de l'impression de chiffres et de texte

La discussion qui suit, sur les possibilités offertes par ***Ecta*** pour l'impression de chiffres et de texte, est forcément assez technique. Ceux et celles pour qui l'impression n'a pas un grand intérêt sont invités à sauter à la section suivante.

Outre la variable `precision`, il y a encore deux variables prises en compte quand ***Ecta*** veut imprimer une valeur numérique. Elles sont `numdigits` et `TOL`, dont les valeurs par défaut sont respectivement 6 et 1.0E-8, c'est-à-dire, 10^{-8}. Voici les règles de leur utilisation. Supposons que l'on veuille imprimer le réel x. Soit n le nombre de chiffres non nuls *avant* la virgule décimale. Si $n >$ `numdigits`, la valeur de x sera imprimée en **notation scientifique**. Afin d'illustrer cette notation, prenons le cas où $x = 1234567890$. Si on fait

```
set x = 1234567890
show a
```

la ligne suivante s'affiche :

```
x = 1.234568e+09
```

ce qui signifie que $x = 1,234568 \times 10^9$. C'est à peu près juste. Si, avant de faire afficher la valeur de `x`, on fait

```
setprecision 9
```

on obtient

```
x = 1.234567890e+09
```

avec précisément 9 chiffres après la virgule. Si on se fiche de la précision, on pourrait faire

```
setprecision 1
```

avec pour résultat

```
x = 1.2e+09
```

* * * *

Si l'on essaie d'affecter à `precision` une valeur de 0, la valeur par défaut de 6 sera utilisée, à moins que `x` soit un entier que l'on peut exprimer en moins de `numdigits` chiffres, sans virgule décimale.

* * * *

Dans les exemples que nous venons de voir, la valeur de x donne lieu à 10 chiffres avant la virgule. Si maintenant on fait

```
set numdigits = 10
```

on obtient

```
x = 1234567890.000000
```

toujours avec 6 chiffres (inutiles !) après la virgule. La notation scientifique est abandonnée parce qu'on s'est autorisé d'avoir jusqu'à 10 chiffres avant la virgule. Si la valeur de `precision` égale 0, on peut finalement obtenir

```
x = 1234567890
```

comme on le voudrait !

La notation scientifique est utilisée non seulement pour les grandes valeurs mais aussi pour certaines petites valeurs, avec un exposant négatif. Toutefois, toute valeur inférieure ou égale à `TOL` sera imprimée comme si elle était strictement nulle. Par exemple, le résultat de

```
show TOL
```

est toujours

```
TOL = 0.000000
```

où le nombre de zéros après la virgule dépend de la valeur de `precision`. Si `TOL` et `precision` ont leurs valeurs par défaut de 10^{-8} et 6, et si on fait

```
set x = 1.01E-8
show x
```

on aura

```
x = 1.010000e-08
```

Une valeur non nulle est imprimée parce que `x` est plus grand que `TOL`.

* * * *

Dans toute cette section, « plus grand » et « plus petit » signifient plus grand ou plus petit en valeur absolue.

* * * *

La notation scientifique n'est employée que pour les valeurs plus petites qu'un seuil défini en termes de `numdigits`. Si on note d la valeur de `numdigits`, le seuil est égal à 10^{-d}. La règle générale s'exprime donc de la manière suivante : Toute valeur plus grande que 10^d est exprimée en notation scientifique, ainsi que toute valeur plus petite que 10^{-d} et plus grande que `TOL`. Toute valeur plus petite que 10^{-d} s'exprime comme zéro. Entre les seuils de 10^{-d} et 10^d, les valeurs sont imprimées normalement. La valeur de `precision` sert à déterminer le nombre de chiffres après la virgule, que ce soit en notation normale ou scientifique.

La souplesse permise par la version 3.3 d'***Ects*** dans l'impression des chiffres a nécessité une reprogrammation totale des méthodes d'impression des tableaux et des matrices. Les tableaux imprimés dans un fichier de sortie suite à l'exécution d'une commande qui lance une procédure d'estimation, `ols`, `iv` `nls`, ou autre, sont quelque peu modifiés depuis les versions antérieures du logiciel. Personnellement, je les trouve plus beaux maintenant ; j'espère que d'autres utilisateurs partageront mon avis. Quoi qu'il en soit, et à toutes fins utiles, le mécanisme qui est utilisé dans les tripes du logiciel est aussi mis à la disposition de l'utilisateur.

Le mécanisme est très largement inspiré du mécanisme d'**alignement horizontal** employé par le logiciel TEX, création du célèbre informaticien D. E. Knuth, et qui m'a permis de créer ce manuel, ainsi que son prédécesseur, et dont l'utilisation est largement répandue dans le monde des scientifiques. Le nom de la commande qui permet de créer des tableaux, `halign`, est emprunté à une commande de TEX. Le programme suivant n'a pas une grande utilité scientifique : Il sert toutefois à illustrer le mode d'emploi de `halign`.

```
set alignnums = 0
mat alignnums = alignnums(1,8,1,1)
set alignnums(1) = 34.76
set alignnums(2) = 8.76543e+9
set alignnums(3) = 0.00045678
set alignnums(4) = 8009.654321
set alignnums(5) = 234.7689
set alignnums(6) = 8.76543e-9
set alignnums(7) = 20.00045678
set alignnums(8) = 9.654321

halign
\hfil#&\hskip2\hfil#&\hskip2\hfil#&\hskip2\hfil#\cr
Label 1\hfil&Label 2\hfil& Label 3\hfil&Label 4\hfil\cr
\#/&\#/&\#/&\#/\cr
Next 1&Next 2\cr
\#/&\#/&\#/&\#/\cr
end

text halign
```

Dans ce programme, un rôle crucial est joué par la matrice spéciale `alignnums`. Avant d'employer la commande `halign`, on met toutes les valeurs numériques du tableau dans cette matrice, dans l'ordre d'impression.

Le nom de la commande `halign` occupe à lui seul une ligne du fichier de commandes. La ligne suivant fournit le **modèle** des lignes qui la suivent jusqu'à la fin de la commande, signalée comme d'habitude par `end`. Dans le modèle, un dièze (`#`) signifie l'endroit où quelque chose, un bout de texte ou un chiffre, sera mis dans les lignes du tableau. Entre chaque paire de `#`, on doit trouver quelque part le signe `&`, qui sépare les colonnes du tableau. À la fin du modèle, et aussi à la fin de chaque ligne du tableau proprement dit, on trouve obligatoirement le symbole `\cr`. Dans les cases du modèle, séparées par `&`, on trouve, outre le `#`, des indications de l'espacement souhaité. La notation `\hskip`, suivi d'un entier n, signifie qu'il faute insérer n espaces blancs à cet endroit. La notation `\hfil`, dans la terminologie de Knuth, signifie qu'il faut mettre de la « colle » à cet endroit. Pour que les colonnes d'un tableau soient correctement alignées, il faut mettre des nombres variables d'espaces blancs dans certaines lignes. Ces espaces blancs seront insérés là où on met de la colle. Si une case contient plus d'une goutte de colle, les espaces blancs seront répartis entre les plusieurs gouttes de manière aussi équitable que possible.

Le modèle de notre programme s'interprète donc comme suit. Chaque ligne du tableau comportera quatre colonnes. La première, terminée par le premier `&`, commencera par le nombre d'espaces blancs qu'il faut pour que la fin du vrai contenu de la colonne (le premier `#`) soit alignée avec les fins des premières colonnes des autres lignes du tableau. La deuxième colonne commencera dans tous les cas par deux espaces blancs (le `\hskip2` qui suit le premier `&`), éventuellement suivis d'autres espaces nécessaires à l'alignement, et, en dernier lieu, le vrai contenu, texte ou chiffre, de la colonne. Les deux dernières colonnes sont identiques à la deuxième.

Les lignes qui suivent le modèle représentent les lignes du tableau. Le format de ces représentations est le même que celui du modèle, à la différence près qu'il faut maintenant mettre le vrai contenu des cases du tableau. Tout ce qui est texte apparaît textuellement ; tout ce qui est chiffre est noté `\#/`, et sera remplacé par le prochain élément de `alignnums`. On voit qu'il est admissible d'incorporer des `\hskip` et des `\hfil` supplémentaires dans ces lignes. La première ligne et la troisième contiennent du texte exclusivement, la deuxième et la quatrième des chiffres exclusivement, mais on peut tout aussi bien panacher. La deuxième ligne montre aussi qu'il est possible de terminer une ligne prématurément, avant de remplir toutes les cases. Dans tous les cas, la notation `\cr` signifie la fin de la ligne. Une ligne vide entre deux lignes du tableau s'obtient si la ligne ne contient que `\cr`.

La commande `halign` ne sert qu'à *mémoriser* le tableau. Il faut demander son impression explicitement. Ceci se fait par la commande

```
text halign
```

si l'on veut que le tableau soit imprimée dans le fichier de sortie, ou par

```
message halign
```

pour que le tableau soit affiché à l'écran. Bien entendu, les commandes `text` et `message` peuvent être employées, comme dans les versions antérieures, pour imprimer ou afficher du texte explicite. Mais les deux utilisations différentes des deux commandes s'excluent mutuellement : Si on met `halign`, rien d'autre n'est admissible, et même le `end` que l'on trouve habituellement à la fin de la commande est supprimé.

Le tableau créé par notre exemple est imprimé dans le fichier de sortie. Voici son aspect final :

```
 Label 1      Label 2      Label 3     Label 4
 34.760000  8.765430e+09   0.000457  8009.654321
    Next 1         Next 2
234.768900       0.000000  20.000457     9.654321
```

On peut remarquer qu'une conséquence de l'alignement est que, pour une `precision` donnée, les chiffres sont alignés de sorte que les virgules (ou les points en notation informatique) sont alignées, sauf pour les chiffres exprimés en notation scientifique.

Il y a un autre aspect du format des résultats. Pour illustrer cet aspect, prenons un cas simple, les résultats d'une commande `ols`. En anglais, on a, par exemple,

```
Variable  Parameter estimate  Standard error  T statistic

constant      0.116872           0.104444       1.118992
x1           -0.106970           0.128790      -0.830577
```

mais en français, on aurait plutôt

```
Variable   Paramètre estimé   Écart type  T de Student

constante    0.116872        0.104444        1.118992
x1          -0.106970        0.128790       -0.830577
```

À mon sens, le tableau suivant

```
Variable   Paramètre estimé  Écart type  T de Student

constante      0.116872       0.104444     1.118992
x1            -0.106970       0.128790    -0.830577
```

serait plus joli. Même en anglais, le tableau serait nettement moins beau si la valeur de `precision` n'était pas la valeur par défaut de 6. L'alignement du tableau est contrôlé par la variable `tabskips`. Seuls les quatre premiers éléments de cette variable sont pris en compte. Ils déterminent le nombre d'espaces blancs à mettre à la fin des quatre colonnes des lignes du tableau après la première. Les valeurs par défaut sont, dans l'ordre, 0, 6, 3, 1 ; elles ont été choisies pour la beauté du tableau anglais avec une `precision` de 6. On vérifie que seuls les 2 espaces prévus par le modèle séparent la fin du mot `constant` (fin de la première colonne) du début du mot `Parameter` (début

de la deuxième colonne), avec 0 espaces supplémentaires ; de la fin du chiffre `0.116872` (deuxième colonne) à la fin du mot `estimate` il y a 6 espaces ; de la fin de `0.104444` (troisième colonne) à la fin de `error`, 3 espaces ; et de la fin de `1.118992` (quatrième colonne) à la fin de la ligne (le `c` final de `Statistic`), 1 espace. Le premier tableau français s'organise de manière identique. Le second tableau français, en revanche, utilise les valeurs 0, 4, 1, 2. Pour obtenir le second tableau, il suffit de faire

```
sample 1 4
gen tabskips = 0
set tabskips(2) = 4
set tabskips(3) = 1
set tabskips(4) = 2
```

avant de lancer la commande `ols`.

EXERCICES:

À la fin du fichier de commandes `newlogit.ect`, on trouve une utilisation non triviale de la commande `halign`, dans une reprise de l'exercice de la section 5.3 de Man2. Si vous faites tourner la totalité des commandes du fichier `newlogit.ect`, vous verrez que le résultat est bien plus joli que celui obtenu par le programme de Man2. Une étude attentive des commandes nécessaires à l'obtention de ce résultat permettra de maîtriser les subtilités de `halign`. On y voit également des emplois de la commande `putnum`, qui sert tout simplement à imprimer dans le fichier de sortie un chiffre – donné comme l'argument de la commande – *sans* que le chiffre soit suivi d'un saut de ligne. La commande `putspace`, sans argument, est encore plus simple. Elle insère un espace blanc dans le fichier de sortie.

4. Le Système d'Aide

Au moment de la rédaction de ce manuel, le système d'aide d'***Ects*** est encore largement défaillant. Le système même est en place, mais il ne contient que peu de choses. Le principe est le suivant. Il y a un fichier exécutable, nommé `ectshelp`, ou `ectshelp.exe` sous DOS/Windows, qui se charge de tout. On peut s'en servir directement, mais, le plus souvent, on fera appel à la commande `help` d'***Ects***. Si on lance ***Ects*** en mode interactif, on voit l'instruction suivant :

```
Type quit to exit, help for help
```

ou, en français

```
Tapez "quit" pour quitter, "help" pour l'aide
```

Si justement on tape `help`, le message suivant s'affiche :

```
Tapez "help" suivi de "commands", "functions", ou "variables",
pour afficher des listes de celles-ci, ou bien tapez "help" suivi
du nom d'une commande, fonction, ou variable afin d'être
renseigné(e) sur un thème précis
```

* * * *

Je donne le message en français afin d'éviter des traductions détaillées du message anglais.

* * * *

Voyons maintenant les quatre possibilités. Si on tape

```
help commands
```

on voit

```
Les commandes disponibles dans cette version d'Ects sont :

batch beep def del differentiate echo else end equation
expand gen gmm gmmhess gmmweight groupname halign help if
input interact invertlagpoly iv lagpoly mat mem message ml
mlar mlhess mlopg nliv nls noecho ols out outnew pause plot
print procedure put putnum putspace quit read readmatrix
recursion rem restore run sample set setprecision setseed
show showall showlagpoly silent svdcmp system text while
write writematrix
```

dont le sens est évident. Les commandes `help functions` et `help variables` donnent des résultats similaires : Dans tous les cas, on a une liste des « thèmes précis » sur lesquels on peut être renseigné.

La quatrième possibilité est illustrée par la commande

```
help beep
```

La commande `beep` est l'une des rares commandes pour lesquelles une documentation est actuellement disponible dans le système d'aide. On obtient l'aide suivante :

```
"beep" sans argument produit un bip sonore.

Avec deux arguments, "beep" peut éventuellement, selon le
matériel, émettre des sons musicaux, d'une qualité parfois
discutable.  Le premier argument est la fréquence, en Hz
(cycles/seconde), du son souhaité, et le second est la
durée, en millisecondes.

Les silences peuvent se produire en spécifiant une durée
non nulle et une fréquence nulle.

Un exemple complet se trouve dans le Manuel, pp 55-56.
```

Le « Manuel » en question est, bien entendu, Man2, de Mars 1993. Si on essaie d'aller plus loin, avec la commande `help set`, par exemple, on n'aura que

```
Aucune aide disponible pour set
```

réponse bien décevante. Au fil du temps, cette réponse deviendra de plus en plus rare, et les messages d'aide de plus en plus fréquents.

* * * *

Un site Web a été créé, grâce aux efforts de Christian Raguet, où le système d'aide est bien mieux développé. Il se trouve en ce moment à

`http://russell.cnrs-mrs.fr/pub/ects3/aide/`

mais il pourra être déplacé dans un avenir proche.

* * * *

Dans la section 1.1, on a mentionné les versions francisées des fichiers exécutables. En effet, c'est la version française de `ectshelp` qui a produit les messages reproduits ci-dessus. On verra dans la section suivante comment les versions françaises ont été créées, et comment créer une version italienne, par exemple, ou allemande. Une version arabe, en revanche, qui utiliserait un alphabet tout à fait différent, devra attendre la finalisation de l'internationalisation du C++ lui-même.

5. Internationalisation du Logiciel

Le statut linguistique d'***Ects*** est paradoxal depuis la création de sa toute première version en 1991. La documentation est rédigée en français, mais le logiciel lui-même ne parle qu'en anglais. Le paradoxe s'explique par des raisons pédagogiques : Il fallait absolument que la documentation soit en français pour que les étudiants marseillais à qui elle était destinée puissent s'en servir. En même temps, un logiciel qui ne parlait qu'anglais – comme la quasi-totalité des autres logiciels d'économétrie, d'ailleurs – permettait aux étudiants d'apprendre la terminologie anglaise, ou mieux, internationale, sans trop de peine.

À l'époque, la décision que le logiciel serait anglophone une fois prise, il m'aurait été difficile de créer une version francophone. Entre temps, le développement, aussi maigre soit-il, du système d'aide m'a suggéré des astuces de programmation qui m'ont permis de franciser non seulement le système d'aide mais aussi le logiciel lui-même sans difficulté. Si vous regardez le contenu du fichier `errors.txt`, vous trouverez au début du fichier

```
Ects, Version 3.3, February 1999.
%%%
Type quit to exit, help for help
%%%
Ects terminated.
%%%
File
%%%
not found
%%%
⋮
```

Tous les textes utilisés par ***Ects***, et non seulement les messages d'erreur comme le nom du fichier pourrait laisser entendre, se trouvent dans ce fichier.

* * * *

On y trouve non seulement les textes, mais aussi les modèles utilisés pour les tableaux de résultats créés par `ols`, *etc.*

* * * *

Il aurait été possible, mais peu efficace, que, à chaque lancement d'***Ects***, ce fichier soit ouvert et son contenu mis dans la mémoire de l'ordinateur. Parmi d'autres difficultés, si j'avais adopté cette solution, il aurait fallu que le fichier soit toujours présent dans le répertoire courant. Le chemin d'accès (PATH) d'un utilisateur permet de trouver les fichiers exécutables (comme `gnuplot`, par exemple), mais pas les fichiers texte. D'où la solution adoptée : les textes doivent être mis dans des fichiers exécutables. Pour le système d'aide, le fichier exécutable est `ectshelp` ou un membre de sa famille ; pour le logiciel lui-même, inutile de chercher plus loin que le principal fichier exécutable `ects`.

L'internationalisation est possible du moment que l'on peut facilement changer la partie texte d'un fichier exécutable. L'outil qui effectue cette opération est le fichier exécutable `settexts`, ou `settexts.exe`. Si vous lancez ce programme sans argument, il affichera un bref descriptif de sa syntaxe :

```
Usage is :
     settexts <exefile> <idchar> <infile>
where <exefile> is the file into which text is put from <infile>
and <idchar> is the character marking the insertion point
```

* * * *

On voit que `settexts` lui-même n'est pas encore internationalisé !

* * * *

La commande qu'il faut pour insérer le contenu de `errors.txt` dans `ects` est donc

```
settexts ects @ errors.txt
```

où le caractère `@` indique à `settexts` qu'il faut chercher une chaine de caractères comprenant au moins 30 `@`. C'est tout de suite après cette chaine de caractères que `settexts` insérera le contenu du fichier `errors.txt`.

Tout le monde peut se servir de `settexts` pour insérer le texte qui lui plaît. L'opération le plus simple est la francisation. Il suffit de faire

```
settexts ects $ errorsfr.txt
```

pour que les textes français remplacent les textes anglais. On peut inverser l'opération tout aussi facilement en relançant `settexts` avec `errors.txt` à la place de `errorsfr.txt`. Si on se donne la peine de traduire les textes en italien, on peut créer un fichier `errorsit.txt`, en respectant l'ordre et le format du fichier original `errors.txt` (attention aux `%%%` !) et insérer son contenu dans `ects` ou, mieux, dans une copie de `ects`.

* * * *

Malheureusement, les langues autres que l'anglais pose un petit problème concernant les accents. Les codes utilisés pour les caractères avec accent dépendent du système d'exploitation. Sous Unix et Linux on utilise le standard nommé **isolatin** mais sous DOS/Windows les caractères sont encodés différemment. Pour les textes anglais, ce fait n'a

aucune importance. Pour les textes français, deux fichiers sont fournis, `errorsfr.iso` et `errorsfr.dos` L'utilisateur peut choisir entre les deux selon son système d'exploitation préféré. Si on se trompe, ce n'est pas grave : Il suffit de relancer `settexts` avec l'autre choix.

* * * *

Pour les fichiers exécutables du système d'aide, les fichiers contenant les messages d'aide sont `help.txt` et `helpfr.txt`. En ce moment, ces fichiers sont très courts ! Des mises à jours devront être bientôt disponibles. Si on le souhaite, on peut facilement rajouter des messages d'aide à ces fichiers, et les insérer dans les fichiers exécutables de la famille de `ectshelp` à l'aide de `settexts`. Le caractère spécial qu'il faut donner comme deuxième argument à `settexts` n'est plus `@`, mais plutôt `$`.

6. Derniers Détails, Dernières Remarques

Il reste une commande dont on n'a pas encore discuté : `system`. Cette commande ne répond à aucun besoin précis, mais elle a été très facile à programmer et elle peut s'avérer utile. La syntaxe

`system` ⟨*commande*⟩

sert à transmettre la ⟨*commande*⟩ au système d'exploitation. Par exemple, si on fait

```
system dir
```

le contenu du répertoire courant s'affichera. Les commandes exécutées par le système d'exploitation peuvent renvoyer un code, dit **code retour**, au système. Ce code, si on en a besoin, est retransmis à ***Ects***, qui le stocke dans la variable scalaire `retcode`.

Dans la section 1.2, il a été remarqué que les lignes qui commencent par un `#` ou un `%` ne sont pas lues par ***Ects***. Cette règle s'étend à la lecture des fichiers de données. On a donc la possibilité d'intercaler des remarques ou des commentaires dans ces fichiers, sur la nature ou la provenance des données, par exemple.

Si on précède le nom d'une commande par le caractère `@`, la commande ne sera ni affichée à l'écran ni imprimée dans le fichier de sortie. L'effet est similaire à celui de

```
noecho
silent
⟨commande⟩
restore
noecho
```

mais avec quelques différences importantes. La première est due au fait que l'effet des commandes `noecho`, `silent`, `echo`, et `restore` est *rétroactif*. Par conséquent, avec le schéma ci-dessus, on verrait

```
noecho
```

affiché à l'écran, et

```
noecho
silent
```

dans le fichier de sortie. La seconde différence importante est que `@` ne s'applique qu'à la commande elle-même, et non aux résultats éventuellement produits par l'opération de la commande. Ainsi, les résultats de la commande

```
@ols y c x1 x2
```

se trouvent comme d'habitude dans le fichier de sortie, alors que, si on fait

```
silent
ols y c x1 x2
```

tout les résultats sont perdus.

* * * *

À la limite, on pourrait faire

```
@noecho
@silent
```

afin d'éviter l'effet rétroactif. Mais pourquoi ?

* * * *

Ensuite, une petite correction à apporter à une chose incorrecte dans Man2, à la page 35, concernant la commande `svdcmp`. Cette commande permet d'effectuer la décomposition par valeurs singulières (SVD) d'une matrice $\boldsymbol{X}$. Le résultat de la décomposition consiste en trois matrices, dont une, notée $\boldsymbol{W}$, contient les valeurs singulières elles-même. Dans Man2, j'ai dit, à tort, que les valeurs singulières étaient les valeurs propres de la matrice $\boldsymbol{X}^\top\boldsymbol{X}$. En réalité, elles sont les racines carrées positives des valeurs propres de $\boldsymbol{X}^\top\boldsymbol{X}$.

En arrivant à la fin de cette nouvelle documentation, force est de constater qu'elle est plus longue que l'ancienne documentation prévue pour la version 2. Il me semble que la raison en est que, dans le courant des six dernières années, le nombre de fonctionnalités nouvelles greffées sur le logiciel de 1993 est plus important que le nombre total de fonctionnalités de la version antérieure ! Je ne peux pas terminer mon récit sans remercier les nombreux étudiant(e)s qui m'ont proposé les modifications qui voient le jour dans la version 3.3. Deux d'entre eux méritent une mention explicite, pour l'attention soutenue qu'ils ont portée au développement et même à la programmation du logiciel. Que Emmanuel Flachaire et Stéphane Luchini trouvent ici le témoignage de ma reconnaissance.

Et maintenant, cher lecteur, chère lectrice, il ne me reste qu'à vous souhaiter bonne utilisation du logiciel, et de vous inciter à me communiquer toute difficulté que vous éprouveriez dans son fonctionnement. Mes coordonnées email :

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

Notez, s'il vous plaît, que l'adresse canadienne n'est plus la même que celle que vous trouverez dans Man2.[9]

[9] Note de la version 4 : Et l'adresse française a aussi changé !

Bibliographie

Abramowitz, M., et I.A. Stegun (1964). *Handbook of Mathematical Functions*, National Bureau of Standards, Washington.

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

Davidson, R., et J.G. MacKinnon (1999). "Artificial Regressions," DT numéro 99A04, GREQAM.

Efron, B., et R. J. Tibshirani (1993). *An Introduction to the Bootstrap*, New-York, Chapman and Hall.

Golub et Reinsch (1970). *Numerische Mathematik*, 14, pp 403-470.

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

Hamilton, J.D. (1994). *Time Series Analysis*, Princeton University Press.

Moshier, S.L. (1989). *Methods and Programs for Mathematical Functions*, Prentice-Hall.

Plauger, P.J. (1995). *The Draft Standard* C++ *Library*, Prentice-Hall, New Jersey.

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

Press, W.H., B.P. Flannery, S.A. Teukolsky, et W.T. Vetterling (1992). *Numerical Recipes in C*, Cambridge University Press, Cambridge.[10]

Stroustrup, B. (1991). *The* C++ *Programming Language*, Second Edition, Addison-Wesley, New-York.

[10] Il existe d'autres versions de cet ouvrage, où les programmes sont donnés en Fortran ou en Pascal.

Index Général

Index *Ects*