Ects

Logiciel d'Économétrie
Version 4

Russell Davidson

Juillet 2004

James McGill

AVANT PROPOS

Depuis la parution de la version 3.3 d'***Ects*** en 1999, le code du logiciel a été entièrement refait. Les fonctionnalités ont été étendues, et elles le seront encore au fil du temps. Toutefois, j'ai essayé de maintenir inchangée l'interface utilisateur, afin que les programmes ***Ects*** puissent encore tourner sous la nouvelle version. Dans ce manuel, conçu comme un simple supplément aux deux volumes antérieurs, pour les versions 2 et 3 respectivement, je fais état des changements de l'interface les plus susceptibles de donner lieu à des désagréments à un utilisateur habitué aux versions précédentes.

Quand le temps le permettra, il y aura un seul volume (volumineux!) qui servira de documentation complète du logiciel. Il sera disponible en français et aussi en anglais, car il y a maintenant un nombre croissant d'utilisateurs canadiens, d'expression anglaise pour la plupart. En fait, le premier volume de documentation concernant la version 2 est déjà traduit en anglais, et le second est partiellement traduit.

Un projet plus ambitieux encore est la documentation du code source. Je voulais, en refaisant du début à la fin ce code, pouvoir mettre à la disposition des utilisateurs des librairies de code utiles à plusieurs fins, et qui ont servi de base au code d'***Ects*** proprement dit. Ce code documenté devra permettre à ceux et celles qui s'intéressent au C++ de voir comment on peut construire des programmes utiles pour l'conomtrie, et éventuellement de contribuer au développement ultérieur du logiciel.

Pour tous les utilisateurs fidèles

Table des Matières

Introduction

1. Le Logiciel Libre

Le monde de l'informatique et du logiciel est de plus en plus menacé par les activités de certaines grandes sociétés américaines. Vous êtes invité à consulter l'exposé qui se trouvent à

`http://www.lebars.org/sec/tcpa-faq.fr.html`

pour en être convaincu. Depuis ses origines, ***Ects*** se situe résolument dans le camp du logiciel libre ("Free Software"), où le mot « libre » a son sens associé à la liberté plutôt qu'à la gratuité, même si ***Ects*** est et sera gratuit. C'est pour assurer cette liberté dans un avenir assez incertain que j'ai pris la décision de ne plus mettre ***Ects*** dans le domaine public, mais de le protéger contre des tentatives de commercialisation en me servant de la licence GPL pour le logiciel même, et la licence GFDL pour la documentation.

Les deux licences sont gérées par le projet GNU et la Free Software Foundation. Leur site, à

`http://www.gnu.org/licenses/licenses.fr.html`

vous donnera des explications détaillées des buts des licences. La GPL (GNU General Public License) est conçue pour la protection des logiciels et leur code source; la GFDL (GNU Free Documentation License) est destinée aux manuels et autres documentations.

Le but de ces licences n'est pas du tout de contraindre l'utilisation du logiciel, bien au contraire. Un logiciel qui est carrément dans le domaine public peut faire l'objet d'une commercialisation par qui que ce soit. Le fait est que, si quelqu'un le souhaite, on peut s'emparer de la version 3 d'***Ects***, justement dans le domaine public, et en faire un logiciel commercial, ni libre ni gratuit, et, plus grave encore, interdire la distribution et l'utilisation du logiciel en dehors da la version commerciale (j'entends payante). Jusqu'ici nous n'avons pas été soumis à un tel sort, probablement parce que l'conomtrie n'intéresse que médiocrement les sociétés en question.

Le fait d'avoir fait d'***Ects*** 4 un logiciel tout neuf, complètement indépendant des versions précédentes en ce qui concerne le code source, et de protéger le code, le logiciel, et sa documentation par les licences réglant leur utilisation, nous mettra à l'abri d'une éventualité telle que je viens de la décrire.

Un inconvénient relié à la licence GFDL est qu'il faut qu'elle soit incorporée dans la documentation qu'elle protège. Vous trouverez dans le dernier chapitre

du manuel le texte complet de la licence, ainsi qu'exigé pour qu'elle s'applique légalement. Le texte est en anglais, encore une fois pour des raisons juridiques, mais, si les détails légaux vous intéressent, vous trouverez le texte traduit en français sur un certain nombre de sites web. Pour la GFDL, j'ai trouvé le site suivant:

`http://cesarx.free.fr/gfdlf.html`

ou bien

`http://www.idealx.org/dossier/oss/gfdl.fr.html`

Pour la GPL, le site

`http://www.linux-france.org/article/these/gpl.html`

fournit le texte de la licence en français.

Attention! La traduction étant ce qu'elle est, seule la version anglaise de la licence a une valeur juridique.

Je ne veux pas appliquer la licence GFDL aux deux premiers volumes de documentation tels qu'ils existent pour les versions 2 et 3 du logiciel. En revanche, j'ai créé des versions légèrement modifiées de ces deux volumes pour accompagner celui-ci. La licence s'applique à ces versions modifiées.

2. Remerciements

Même si, depuis le début, c'est moi seul qui ai fait toute la programmation d'***Ects***, il y a d'autre gens qui m'ont aidé de plusieurs façons. Emmanuel Flachaire et Stéphane Luchini m'ont souvent demandé de rajouter une fonctionnalité à laquelle je n'aurais pas pensé. De même, mon co-auteur James MacKinnon m'a fait beaucoup de suggestions précieuses. Christian Raguet a assumé la tâche de créer une page web où les utilisateurs peuvent trouver des exemples de code ***Ects***, et d'autres aides à l'emploi du logiciel. Plus récemment, Pierre-Henri Bono m'a propulsé au développement de la version 4, en faisant plusieurs suggestions utiles pour des extensions de la fonctionnalité. Il a été fortement impliqué dans la phase « test bêta » du logiciel.

Comme dans la version précédente, je me suis servi de code source qui n'est pas le mien et que j'ai trouvé sur des sites consacrés à la programmation, surtout numérique. En particulier, la librairie Cephes, la Cephes Mathematical Library, m'a fourni le code nécessaire à la programmation de la plupart des fonctions reliées aux distributions de probabilité. Le code est disponible au site

`http://www.netlib.org/cephes/`

où on trouvera aussi des explications utiles. Cette librairie contient beaucoup de fonctions qui ne sont pas incorporées dans ***Ects***, mais qui peuvent s'avérer utiles pour la programmation scientifique. L'auteur de la librairie est Stephen L. Moshier.

Le code qui effectue la décomposition par valeurs singulières est basé sur du code C écrit par Brian Collett. Il nous informe que son code s'appuie sur un algorithme de Golub et Reinsch écrit à l'origine en Algol 60, un langage de programmation depuis assez longtemps tombé dans l'oubli. Un lien permettant d'accéder au code de M. Collett se trouve au site

`http://cliodhna.cop.uop.edu/~hetrick/c-sources.html`

avec beaucoup d'autres liens utiles pour le calcul numérique.

On peut me demander pourquoi, avec tout le code que l'on trouve sur différents sites, j'ai fait tant de programmation indépendamment. Eh bien, c'est pour ma culture générale ! Le premier but d'***Ects*** a été un but pédagogique, celui d'aider les étudiants à comprendre les aspects pratiques de l'estimation économétrique. J'ai donc le droit d'être pédagogue pour moi-même, et j'avoue ouvertement que j'ai beaucoup appris en faisant la programmation d'***Ects*** dans ses versions successives. Mais, à partir de la version 4, j'ouvre les portes du logiciel, avec la possibilité d'ajouter des « modules » compilés séparément. Il en existe déjà, encore de ma propre facture, et d'autres vont suivre, non seulement de moi, mais aussi d'autres programmeurs. Je n'en parle plus pour le moment; le concept n'a pas encore fait ses preuves.

Lors de la programmation de la version 3, le C++ n'était pas encore standardisé. Le compilateur GNU avait tendance à changer souvent de syntaxe, de sorte que le code d'***Ects*** a dû changer en même temps. La librairie standard n'existait que sous une forme inaccessible à la plupart des compilateurs de l'époque, et, en attendant, je me suis servi d'une librairie « provisoire » créée par P. J. Plauger. Heureusement, tout cela n'est plus nécessaire. Le langage est standardisé, le compilateur GNU est presque complètement conforme au standard, la librairie standard marche à souhait, de sorte que je n'ai pas eu à me soucier des problèmes qui ont compliqué la programmation d'***Ects*** 3.

Je me suis pourtant servi de la librairie Loki, dont l'auteur est Andrei Alexandrescu, disponible à plusieurs endroits, dont le plus utile est probablement

`http://sourceforge.net/projects/loki-lib/`

où on voit que la librairie est toujours en cours de développement. La version utilisée dans ***Ects*** est celle qui accompagne son livre *Modern C++ Design*, Addison-Wesley, 2001. La lecture de ce livre a révolutionné mes idées des possibilités du C++. À la fin du chapitre contenant la Licence GPL, vous trouverez l'enoncé des droits d'auteur de M. Alexandrescu ainsi que la permission d'utiliser son code dans d'autres logiciels.

Les librairies Boost, disponibles à

`http://www.boost.org`

ont pour mission d'étendre les capacités de la librairie standard. L'équipe de Boost aura certainement une grande influence sur l'évolution future de la librairie standard, et du langage même. Les fonctionnalités offertes par ces

librairies m'ont énormément facilité beaucoup de tâches de programmation. Vous trouverez encore une fois la notice de Boost à la fin du chapitre pertinent.

Chapitre Premier
Les Commandes

En général, le code ***Ecta*** qui tourne sous la version 3.3 du logiciel tournera aussi sous la version 4. En particulier, les fichiers de code fournis avec les versions 2 et 3 tournent bien sous ***Ecta*** 4, à quelques exceptions près, qui seront expliquées par la suite. Les résultats sont souvent présentés sous une forme légèrement différente, et, je l'espère, plus agréable. Dans un ou deux cas, les résultats d'une estimation non linéaire sont numériquement différents. C'est normalement ***Ecta*** 4 qui a raison. Il arrive parfois à détecter un meilleur minimum d'une somme de carrés, ou un meilleur maximum de vraisemblance. Ceci ne signifie pas que ***Ecta*** 3 a tort. Chaque fois, il y a des optima multiples, et c'est en quelque sorte le hasard qui fait que ***Ecta*** 4 est amené à un optimum meilleur que celui trouvé par ***Ecta*** 3. J'espère que les nouveaux algorithmes mis en œuvre par ***Ecta*** 4 y sont pour quelque chose, mais l'expérience que j'ai en ce moment ne me permet pas de conclure définitivement.

Dans le reste de ce chapitre, je donne quelques détails sur les changements les plus importants de la nouvelle version. Les différences de formatage ne sont pas commentés ; ce serait trop fastidieux et trop peu utile. En revanche, j'essaie d'expliciter tous les changements qui pourraient piéger un utilisateur non averti.

1. Commandes Supprimées

Sous ***Ecta*** 3, on faisait

```
help commands
```

afin de faire afficher une liste des commandes reconnues par cette version du logiciel. La commande `help` n'est pas encore fonctionnelle avec ***Ecta*** 4 ; il ne l'était jamais d'ailleurs avec ***Ecta*** 3, grâce à ma propre paresse ! Mais un affichage de la liste de commandes est toujours possible. On utilise la nouvelle commande `listcommands` à cette fin.

La dernière version d'Ecta 3, la version 3.3, donne la liste suivante:

```
Commands available in this version of Ecta are:
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
settoler show showall showlagpoly silent svdcmp system text
while write writematrix
```

alors que la liste d'***Ects*** 4 est la suivante:

```
Commands currently defined are
appendnewline appendnum appendspace appendtext beep
def defdiff deftext del differentiate echo equation
exec expand function gen gens gmm gmmhess gmmweight halign
if input interact iv licence listcommands load loadlibrary
mat mats message minhess minimise ml mlar mlhess mlopg nliv
nls noecho ols out outnew pause plot print
printmatrix procedure put putnewline putnum putspace
puttext qquit read readbinvariables readmatrix
readvariables recurrence recursion rem restore run sample
set setmaxiter setprecision setrng sets setseed settext
settoler show showall showinternal showrng showtext silent
svdcmp system text while write writebinvariables
writematrix writevariables
```

On voit que les deux ensembles sont loin d'être identiques.

* * * *

Il est plus que probable que votre version d'***Ects*** 4 ajoute d'autres fonctions à cette liste. C'est parce qu'il est souvent utile d'insérer des fonctionnalités nouvelles avant qu'elles ne soient documentées.

* * * *

Les commandes qui se trouvent dans la liste d'***Ects*** 3 mais qui ne sont plus dans celle d'***Ects*** 4 sont:

```
batch else end groupname invertlagpoly lagpoly mem quit showlagpoly
```

La commande `batch` n'a jamais été qu'un synonyme de `quit`. Que je sache, personne ne l'a utilisée. En la supprimant, on ne perd aucune fonctionnalité.

Les mots clé `else` et `end` ne sont pas de vraies commandes, mais plutôt l'indication qui sépare les deux blocs d'une commande `if` et le signal de la fin d'un bloc. La programmation d'***Ects*** 4 est telle que ces mots clé ne sont plus répertoriés comme des commandes ; la fonctionnalité reste toutefois inchangée. De même, `quit` n'est plus une commande, mais si on met `quit` dans un fichier de code ***Ects***, le résultat est toujours le même. La programmation d'***Ects*** 4 est telle que les mots clé ne sont pas affichés avec les noms des vraies commandes.

La commande `mem` d'***Ects*** 3, si on l'utilise sous Linux, affiche un résumé de l'état actuel de la mémoire de l'ordinateur. Dans le temps où les ordinateurs n'avaient que peu de mémoire par rapport à la norme d'aujourd'hui, cette commande pouvait servir à quelque chose. Plus maintenant. Si j'ai absolument besoin de faire afficher ces informations sous ***Ects*** 4, je n'ai qu'à faire la commande

```
system cat /proc/meminfo
```

et j'obtiens exactement ce que j'aurais eu avec `mem`. La commande précise est spécifique aux systèmes Linux ou assimilés ; l'avantage de la nouvelle approche est qu'on peut se servir de la commande appropriée en fonction du système d'exploitation.

Les trois commandes `lagpoly`, `invertlagpoly`, et `showlagpoly` sont employées sous ***Ects*** 3 pour la manipulation des polynômes en l'opérateur retard. Ces manipulations sont parfois très utiles. Mais la structure des commandes s'est avérée inadéquate pour certaines opérations sur les séries temporelles. Ces commandes sont en effet rendues inutiles pour les variables scalaires par la nouvelle fonction `invconv`, dont l'opération est décrite plus tard. Pour les vecteurs, deux fonctions, `matconv` et `matinvconv`, sont fournies dans un **module**. Les modules permettent de rajouter aisément des fonctionnalités à ***Ects***. Selon la politique d'***Ects*** 4, les choses vraiment spécifiques, comme les polynômes vectoriels en l'opérateur retard, sont exclues du noyau du logiciel, mais sont disponibles sur demande si on charge le module pertinent.

Finalement, la commande `groupname`, jamais documentée, et donc jamais utilisée, est supprimée. Il reste à trouver un meilleur moyen de faire ce que cette commande était censée faire.

2. Commandes Nouvelles

Dans la liste des commandes affichée par ***Ects*** 4, celles que l'on ne trouve pas dans la liste d'***Ects*** 3 sont les suivantes.

```
appendnewline appendnum appendspace appendtext deftext exec
function gens instr lhs listcommands load loadlibrary mats
minhess minimise printmatrix putnewline puttext qquit
readbinvariables readvariables recurrence second setrng sets
showinternal showrng showtext weightmatrix writebinvariables
writevariables
```

Malgré l'apparence, la commande `minhess` n'est pas un rajout d'***Ects*** 4. Étant donné que la commande `gmmhess` permet de minimiser une fonction donnée, et ce sans qu'il y ait forcément un lien avec une estimation économétrique, il m'a semblé plus sensé de le dire explicitement. Mais `minhess` est un simple synonyme de `gmmhess`. De la même façon, `minimise` est identique à `gmm`.

Le dictionnaire m'apprend que le mot `recursion` est considéré comme un anglicisme en français. Le mot correct est `récurrence`. On a maintenant le choix entre l'anglicisme et le bon usage, avec la commande `recurrence`. Le fonctionnement est exactement le même que celui de `recursion`.

* * * *

La programmation en C++ de cette commande est à mon sens un peu brutale, et j'espère pouvoir faire mieux dans un avenir proche. Ce qu'il

> y a maintenant ne manque pas d'élégance, mais je ne suis pas du tout persuadé de son efficacité. De toute manière, l'interface utilisateur est inchangée.
>
> * * * *

Les commandes `sets`, `gens`, et `mats` ont le même effet que les bonnes vieilles commandes `set`, `gen` et `mat`. La différence est simplement que, avec le '`s`' à la fin, le résultat de la commande est affiché à l'écran. Avec ***Ects*** 3, si on veut voir le résultat d'un calcul, il faut quelque chose comme

```
set P = 1-fisher(F,r,n-k)
show P
```

Maintenant il suffit de faire

```
sets P = 1-fisher(F,r,n-k)
```

Le fonctionnement est similaire avec les deux autres commandes. Avec `gens`, on risque de remplir l'écran plusieurs fois si la taille de l'échantillon en cours est importante. Avec `mats`, si la matrice a beaucoup de colonnes, l'affichage débordera dans l'autre sens. Il n'y a pas de mal, mais ce ne serait pas non plus très utile.

Générateurs de Nombres Aléatoires

Ects 4 bénéficie des développements récents dans la théorie des générateurs de nombres aléatoires. On aurait pu penser qu'un outil aussi essentiel aux simulations aurait été complètement mis au point il y a longtemps. Même si des générateurs existent depuis au moins trente ans, le fait est que la plupart de ceux qu'on utilisait à l'époque étaient très mauvais. (Il y a eu des exceptions tout à fait honorables.) Plus récemment, un sérieux effort de recherche a permis de développer des générateurs à la fois plus fiables et plus performants.

Ceux qui sont disponibles avec ***Ects*** 4 portent les noms `250`, `congruential`, `default`, `jgm`, `kiss`, `mt`, et `old`. La commande `setrng` sert à sélectionner celui que l'on souhaite employer. Si la commande est lancée sans argument, la liste des générateurs disponibles est affichée. On aura quelque chose comme

```
Available generators are identified by:
250 congruential default jgm kiss mt old
```

où on voit les identificateurs qui seraient admissibles comme arguments à la commande.

En fait, et malgré les apparences, les générateurs disponibles sont au nombre de 5, et non 7. Ceci est dû au fait que les mots clé `jgm`, `default`, et `old` font référence tous à un seul générateur, à savoir celui dont ***Ects*** s'est servi depuis ses débuts. Le code de ce générateur, bien que modifié par la suite, m'a été donné à l'origine par mon collègue et co-auteur James MacKinnon.

Il existe un ensemble de tests, nommé **DIEHARD**, permettant de juger des qualités d'un générateur de nombres aléatoires (GNA). Ces tests ont été

développés par quelqu'un qui porte le nom de la ville natale d'***Ects***, mais sous une forme plus ancienne, George Marsaglia. Son site, auquel je donnais une référence dans la première version de ce manuel, a été supprimé depuis. Pourtant, le code source (en C) des tests est disponible à un autre site :

`http://stat.fsu.edu/pub/diehard/`

Il va sans dire que les GNA d'***Ects*** ont fait leurs preuves auprès de ces tests, à l'exception du générateur `congruential`. Ce générateur est de l'ancien type, un générateur congruentiel, et il peut encore servir si on n'a besoin que de peu de nombres aléatoires. Son usage est déconseillé pour les simulations sérieuses.

Le générateur `kiss` est dû à Marsaglia même. Ses performances sont attribuées à une combinaison de quatre générateurs différents, tous de type assez simple. Le mot `kiss` n'a pas sa signification habituelle en anglais ; il signifie plutôt Keep It Simple Stupid, et il témoigne de la préférence de M. Marsaglia pour la simplicité.

Le générateur `mt` est ce qu'on appelle un **Mersenne twister**, pour des raisons que je n'exposerai pas ici. Si vous souhaitez comprendre l'appellation ou même la théorie mathématique qui sous-tend ce générateur, veuillez consulter le site

`http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`

Le générateur nommé `250` est l'une de ces exceptions honorables auxquelles je faisais référence tout à l'heure. Il se base sur ce qu'on appelle un *shift register sequence* (hélas, je n'ai pas d'idée sur la version française de ce terme). Le travail fondamental est exposé dans le livre *Shift Register Sequences*, par Solomon W. Golomb, Holden-Day, 1967. Une introduction relativement simple, avec des exemples, se trouve au site

`http://www.geocities.com/CapitolHill/Congress/8327/sr.htm`

du moins au moment où je rédige ce manuel.

La commande `showrng` ne prend pas d'argument. Son effet est de faire afficher le mot clé correspondant au GNA dont ***Ects*** se sert au moment du lancement de la commande.

EXERCICES:

Vous pouvez voir lequel des GNA travaillent le plus vite en faisant tourner plusieurs fois une boucle où vous générez un vecteur d'un million de nombres aléatoires. Les résultats dépendront non seulement de l'efficacité des différents algorithmes, mais aussi de votre matériel et de votre système d'exploitation. Toutefois, il est probable que vous constaterez que le vieux générateur est le plus lent.

Fonctions Définies par l'Utilisateur

La commande `function` est utilisée pour définir soi-même des fonctions. Un exemple trivial servira à illustrer la syntaxe de la commande.

```
function z deuxfois(x)
   gen z = 2*x
end

sample 1 4
gen y = time(0)
gens y2 = colcat(y,deuxfois(y))
show deuxfois
```

Après le nom (`function`) de la commande, il faut le nom de la variable qui est calculée par la fonction et qui constitue la **valeur rendue** par la fonction. Ici, cette variable est `z`. La variable `x` est l'**argument** de la fonction. Les calculs qui génèrent la valeur peuvent être une suite quelconque de commandes ***Ecta***, terminée par le mot clé `end`. Comme on le voit, la valeur de la fonction est simplement deux fois l'argument. Après la définition de la fonction, on peut s'en servir comme on se sert de toute autre fonction ***Ecta***. L'exemple affiche une petite table de multiplication par 2:

```
function z deuxfois(x)
sample 1 4
gen y = time(0)
gens y2 = colcat(y,deuxfois(y))
y2 =
1.000000  2.000000
2.000000  4.000000
3.000000  6.000000
4.000000  8.000000
show deuxfois
deuxfois:  signature z deuxfois(x)
```

Apres la table, on demande que la fonction elle-même soit affichée. Dans le listing, on voit comment ***Ecta*** caractérise une telle fonction par sa **signature**. On entend par là une expression qui commence par le nom de la variable rendue, suivi du nom de la fonction avec, après, la liste des arguments en parenthèses. Les adeptes du C reconnaîtront tout de suite ce type d'expression.

Les variables `z` et `x` n'ont qu'une existence locale, c'est--dire, qu'elles ne sont reconnues qu'à l'intérieur de la fonction. Si après le bout de code de l'exemple on fait `show x z`, on aura la réponse suivante:

```
Symbol(s) z x not found in tables
```

Si on avait déjà défini des variables `x` et/ou `z`, leurs valeurs sont conservées pendant l'exécution de la fonction. De même, si à l'intérieur de la définition d'une fonction on redéfinit la taille de l'échantillon par une commande `sample`, cette redéfinition est oubliée à l'extérieur.

Une fonction peut prendre plus d'un argument, ou pas d'argument du tout. Les parenthèses autour des arguments ne sont pas nécessaires, la séparation des arguments par des virgules est facultative. ***Ecta*** ne fait pas de distinction entre les fonctions qui seront utilisées dans les commandes `set`, `gen` ou `mat`.

C'est l'utilisateur qui doit déterminer comment définir la fonction en vue de ses utilisations ultérieures.

EXERCICES:

Mettez `set` ou `mat` à la place de `gen` dans la définition de `deuxfois`, et observez les conséquences pour les résultats. Essayez aussi d'utiliser `set` et `mat` à la place de `gens`. Un bon exercice est de deviner les résultats avant de faire tourner vos programmes.

Le factoriel n'est pas donné directement par ***Ects***. La fonction `gln` rend le logarithme de la fonction Gamma, qui est reliée au factoriel par la relation $n! = \Gamma(n+1)$ pour tout entier non négatif n. Programmez une fonction factoriel qui travaille correctement sous `gen`, et servez-vous-en pour construire une table des factoriels des entiers de 0 à 9.

La fonction gamma incomplète est définie comme suit:

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

La fonction est disponible sous le nom de `gammp`. Programmez une fonction des deux arguments a et x qui rend la valeur $P(a,x)$, mais indirectement, en vous servant de la fonction `int`, dont les capacités son décrites plus loin.

Dans les manuels d'informatique, le factoriel est souvent l'exemple type d'une fonction qui se définit par une récurrence. On peut la définir de la même manière avec ***Ects***. Une définition de ce type est la suivante.

```
function f factorial n
   if (n=0)+(n=1)
      set f = 1
   else
      set f = n*factorial(n-1)
  end
end
```

C'est amusant, mais ce n'est pas très efficace par rapport à une définition se servant de la fonction `gln`. En plus, la définition avec la récurrence ne peut marcher que dans une commande `set`, et donne lieu à une boucle infinie si l'argument n'est pas un entier non négatif.

Autres Commandes Nouvelles

On a remarqué que `quit` n'a plus le statut d'une commande ***Ects***, mais plutôt celui d'un mot clé. On a parfois besoin d'une *commande* qui arrête le déroulement d'un programme ***Ects***, par exemple à l'intérieur d'une fonction, ou d'un bloc de commandes associé à une commande `if` ou `while`. Cette commande est `qquit`. Si, mais seulement si, la commande est exécutée, le programme est terminé. En revanche, `quit` est interprété comme une sorte de ponctuation servant à marquer la fin d'un bloc de commandes.

Un nouveau mot clé introduit dans ***Ects*** 4 est `include`. Si on a une ligne du type

`include` <*fichier de commandes*>

dans un fichier de commandes, la lecture de celui-ci est interrompu, et les commandes contenues dans le <*fichier de commandes*> sont insérées dans le programme. Quand on arrive à la fin du fichier dont le contenu est ainsi inséré, ou on arrive à un `quit` au milieu du fichier, on reprend la lecture des commandes du premier fichier. On verra ultérieurement comment et pourquoi le fonctionnement de `include` diffère de celui de `run`.

À la demande urgente de Pierre-Henri Bono, j'ai introduit dans ***Ects*** 4 la commande `readvariables`. Il avait un fichier contenant 43.583 observations pour une petite centaine de variables. La première ligne du fichier est une suite de noms de variables, soit une petite centaine de noms. Le reste du fichier est comme tout autre fichier de données que l'on peut soumettre à ***Ects***, avec 43.583 lignes après la première, contenant chacune une petite centaine de chiffres.

Si nous supposons que le fichier s'appelle `reg.dat` et qu'il se trouve dans le répertoire courant, nous pouvons exécuter la commande

```
readvariables reg.dat
```

et nous obtiendrons le résultat souhaité par Pierre-Henri. De la première ligne, ou, plus précisément, la première ligne qui n'est pas vide et qui ne commence pas par un '`#`' ou un '`%`', on obtient les noms des variables. Ensuite, après avoir compté le nombre restant de lignes non vides ne commençant pas par `#` ou `%`, on saisit normalement les valeurs à affecter à l'ensemble des variables. La taille d'échantillon en cours n'y est pour rien. Seul le contenu du fichier détermine le nombre et la taille des variables saisies.

On aura sans doute besoin pour la suite du programme du nombre de variables, de leurs noms, et du nombre d'observations. Le nombre de variables se trouve dans la variable `variablenumber`, et le nombre d'observations dans `obsnumber`. Les noms des variables sont mis ensemble dans une chaîne de caractères sous le nom de `variablenames` ; nous verrons plus tard ce que c'est pour ***Ects*** qu'une chaîne de caractères.

Le complément logique de la fonction `readvariables` est la fonction `writevariables`. La syntaxe est la suivante.

`writevariables` <*nom du fichier*> <*liste de noms*>

La seule différence entre cette fonction est la fonction `write` est que le format du fichier créé est celui dont on a besoin pour `readvariables`. Mais, à la différence de cette commande, on tient compte de la taille de l'échantillon en cours.

La lecture des données contenues dans `reg.dat` prend un certain temps même si on a un ordinateur rapide et performant (2 seconds environ sur ma machine actuelle, beaucoup plus longtemps sur le portable de Pierre-Henri). Pour

éviter cet inconvénient, on peut stocker les données sous forme binaire. La commande `writebinvariables` a exactement la même syntaxe que `writevariables`, mais son opération diffère par le fait que le fichier créé contient les données sous forme binaire plutôt que sous forme texte. Un inconvénient éventuel est que les formats binaires peuvent être différents sous des systèmes d'exploitation différents. Si tel est le cas, un fichier binaire créé sous le système X peut être incompatible avec le système Y.

Il faut bien sûr la commande `readbinvariables` pour aller dans l'autre sens. Sa syntaxe est toujours la même que celles des commandes `writevariables` et `writebinvariables` : on précise le nom du fichier binaire qui contient les données, suivi de la liste des noms des variables à saisir. Si le fichier binaire ne contient pas une variable qui correspond à un nom donné, un message d'erreur s'affiche, mais les variables trouvées sont effectivement saisies. Comme pour `readvariables`, la taille d'échantillon en cours n'a aucun effet sur cette commande, les dimension des matrices étant précisées dans le fichier binaire lui-même. La lecture des données du fichier `reg.dat` après leur conversion en forme binaire se fait en 0,2 secondes. On obtient donc un important avantage qui a pourtant son coût : le fichier binaire est entre trois et quatre fois plus gros que le fichier texte.

La dernière commande à traiter dans cette section est la commande `showinternal`. Elle n'est pas très utile normalement, mais elle peut être précieuse pour certaines opérations de débogage. La syntaxe:

```
showinternal <expression>
```

où <*expression*> est une expression que l'on peut mettre dans le membre de droite d'une commande `set`, `gen`, ou `mat`. Une représentation texte de la représentation interne de l'<*expression*> employée par ***Ecta*** est affichée à l'écran.

Un exemple. Si on fait

```
def residu = y-a*iota-b*x
showinternal diff(residu'*residu,b)
```

le résultat est

```
Expression diff(residu'*residu,b) is parsed as
+(*(trans(unary-(x1)),-(-(y,*(a,iota)),*(b,x1))),
*(trans(-(-(y,*(a,iota)),*(b,x1))),unary-(x1)))
```

Ce qui est facilement lisible par l'ordinateur ne l'est pas forcément par l'être humain ! On comprendra plus aisément quand on sait que les opérations binaires comme l'addition, la soustraction, *etc*, sont représentés exactement comme les autres fonctions, de sorte que, dans `-(-(y,*(a,iota)),*(b,x1))` par exemple, `*(a,iota)` représente ce que l'on taperait comme `a*iota`, `*(b,x1)` représente `b*x1`, `-(y,*(a,iota))` représente `y-a*iota`, et l'expression entière représente `y-a*iota-b*x1`.

EXERCICES:

La fonction `trans` signifie la tranposition d'une matrice : l'opération notée ' dans un programme ***Ects***. La fonction `unary-` est simplement l'opération qui remplace une variable par son opposé. Vérifiez que la dérivation de la somme des carrés des résidus a été effectuée correctement.

Démontrez que la dérivée de la fonction `digamma`, inconnue à ***Ects*** 3, est disponible sous ***Ects*** 4.

Plus haut j'ai évoqué la possibilité du chargement de modules qui augmentent les fonctionnalités d'***Ects***. Les commandes `load` et `loadlibrary` effectuent le chargement d'un module. On en parlera dans la section sur les modules.

Les commandes `appendnewline`, `appendnum`, `appendspace`, `appendtext`, `deftext`, `exec`, `putnewline`, `puttext`, et `showtext` sont les commandes associées à la manipulation des chaînes de caractères. On en parlera plus tard dans le détail.

3. Commandes Modifiées

Alors que j'ai tout fait, dans la mesure du possible, pour que les programmes qui tournent sous ***Ects*** 3 tournent aussi sous ***Ects*** 4, il y a une commande qui existe dans les deux versions, 3 et 4, où ce qui marche sous ***Ects*** 3 ne marche pas sous ***Ects*** 4 et *vice versa*. Il s'agit de la commande `halign` ; je reporte à plus tard la discussion de cette commande.

Dans le temps, à la demande de Christophe Flachot, la commande `beep` était capable, du moins sous certains systèmes d'exploitation, de faire sortir du petit haut parleur d'un ordinateur des sons « musicaux ». Des exemples sont contenus dans les fichiers `mrs.ect` et `pitch.ect`. Depuis quelque temps, ces fichiers de commandes ne produisent aucun effet sur ma machine. Les anciens mécanismes permettant un accès direct au haut parleur ne sont plus accessibles avec les systèmes d'exploitation modernes. D'ailleurs, une bien meilleure musique peut être obtenue si on a une carte son. Avec un je ne sais quoi de tristesse, alors, je me suis rappelé que le but d'***Ects*** n'est pas de faire la musique, et j'ai supprimé cette emploi de la commande `beep`. Il n'est plus capable que de faire ce grognement désagréable qui est le « bip » d'un ordinateur.

Les Estimations

Il y a plusieurs commandes qui, tout en restant compatible avec la syntaxe de la version 3, admettent aussi une syntaxe différente. Les variables soumises aux commandes `ols` et `iv` peuvent être des expressions à évaluer aussi bien que des variables qui existent dans les tables internes d'***Ects***. Par exemple, si on veut faire tourner la rgression

$$\boldsymbol{y} - \boldsymbol{x}_1 = \alpha + \beta(\boldsymbol{x}_2 - \boldsymbol{x}_1) + \boldsymbol{u},$$

on n'a qu'à faire

```
ols y-x1 c x2-x1
```

Avant la version 4, il aurait fallu procéder comme suit

```
gen yy = y-x1
gen z = x2-x1
ols yy c z
```

pour avoir le même effet. Attention ! Il faut parfois mettre des virgules afin d'éviter des ambiguïtés. Si par exemple on veut faire tourner

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

on ne peut pas faire

```
ols y c x1 -x2
```

parce que cette commande serait interprétée comme la rgression de $\boldsymbol{y}$ sur la constante et $\boldsymbol{x}_1 - \boldsymbol{x}_2$. Mais en faisant

```
y c x1, -x2
```

on a le résultat souhaité. Les virgules sont toujours admissibles pour séparer les variables.

Les variables sont évaluées comme si on avait fait une commande `gen`. Par conséquent, la constante peut être représentée simplement par `1`. Une tendance temporelle peut être représentée par `time(0)`, mais aussi par le mot clé `trend`. Ainsi la commande

```
ols y c trend
```

régresse `y` sur une constante et une tendance linéaire. Pour une tendance quadratique, on peut bien entendu écrire `time(0)^2`, mais il n'y a pas de mot clé.

* * * *

Si la constante est représentée autrement que par `c`, une rgression `ols` est interprétée comme étant sans constante, et le R^2 centré n'est ni affiché ni calculé.

* * * *

La commande `plot` bénéficie des mêmes avantages que `ols` et `iv`. On peut faire par exemple

```
sample 1 181
gen x = PI/90*time(-1)
set linestyle = 1
plot (x sin(x) cos(x))
```

et on obtient deux jolis tracés sinusoïdales superposés.

Depuis le début, les commandes `ols` et `iv` font des calculs dont les résultats sont disponibles dans des variables spéciales, comme `res` et `fit` pour les

résidus et les valeurs ajustées. Dans la version 4, ces calculs ne sont faits que s'ils sont demandés. Si on travaille sous l'influence d'une commande `silent`, les résultats d'une commande `ols` ou autre commande faisant tourner une procédure d'estimation ne figurent pas dans le fichier de sortie. La conséquence est que seuls les calculs les plus essentiels sont effectués. Cette politique permet d'accélérer le temps de calcul des simulations, où par exemple on a une boucle avec une rgression dont on ne se sert réellement que de certains résultats, comme les paramtres estimés. En fait, ne sont calculés d'office que ces paramtres estimés et la matrice de covariance estimée.

Si on n'est pas sous l'influence d'un `silent`, d'autres calculs sont faits, afin de pouvoir afficher les informations que l'on a habituellement dans le tableau de résultats. Pour une seule rgression, le temps de calcul supplémentaire est minime. Pour un million de rgressions, chose tout à fait concevable dans le contexte d'une simulation, ce temps devient un fardeau inutilement lourd.

Si la plupart des calculs ne se font que sur demande, il est possible d'augmenter leur nombre. Ainsi, suite à une commande `ols`, outre les variables disponibles dans les versions antérieures, ***Ects*** 4 met à disposition les variables `regressand` et `regressor`. Ces variables contiennent les résultats des calculs qui ont donné les matrices de variables dépendantes (`regressand`) et de variables explicatives (`regressor`). Avec `iv`, on a aussi `instrument`, qui contient la matrice de variables instrumentales.

Jusqu'ici, la matrice `vcov` est la matrice de covariance estimée pour la première colonne de la matrice `regressand` seulement. S'il n'y a qu'une seule colonne, ce n'est pas grave ! Mais dans le cas où il y a plusieurs colonnes, on aimerait pouvoir accéder aux matrices de covariance pour chaque colonne. On le fait maintenant de la manière suivante.

```
ols colcat(y1,y2) c x1 x2
show vcov
mats V = vcov(0)
mats V = vcov(1)
```

On voit que la rgression a deux variables dépendantes, `y1` et `y2`. En affichant `vcov` on voit la matrice de covariance pour `y1`. L'effet de la commande suivante, `mats V = vcov(0)`, fait afficher la même matrice. Mais, ensuite, `mats V = vcov(1)` fait afficher la matrice de covariance pour la régressande `y2`.

On ne peut pas faire simplement `show vcov(1)`. Ou plutôt si, mais on verra la matrice associée à `y1`. En faisant `show vcov(m)`, quel que soit `m`, c'est toujours la même matrice. Mais en se servant d'une commande `mats` l'argument est finalement pris en compte, parce que `vcov` est maintenant interprété comme une fonction. L'argument est l'indice de la colonne de la matrice de régressandes, *en commençant par* 0. Si on va trop loin, avec par exemple `mats V = vcov(2)`, un message d'erreur s'affiche.

Les trois versions de la matrice dite HCCME, pour Heteroskedasticity Consistent Covariance Matrix Estimator, ou, en français, estimateur de la matrice de covariance robuste à l'hétéroscédasticité, s'obtiennent de la même manière. Les variables `HC0`, `HC1`, `HC2`, et `HC3` sont associées à la première régressande. En utilisant les fonctions `HC0`, `HC1`, `HC2`, et `HC3`, on peut accéder aux matrices associées aux autres régressandes.

Les différentes matrices ont toutes la même forme :

$$(\boldsymbol{X}^\top\boldsymbol{X})^{-1}\boldsymbol{X}^\top\hat{\boldsymbol{\Omega}}\boldsymbol{X}(\boldsymbol{X}^\top\boldsymbol{X})^{-1}, \tag{1}$$

où $\boldsymbol{X}$ est la matrice de régresseurs. Les différences n'affectent que la matrice diagonale $\hat{\boldsymbol{\Omega}}$. Vous trouverez un exposé détaillé sur ces matrices dans la section 5.5 de l'ouvrage *Econometric Theory and Methods*, de moi-même et James MacKinnon, publié en 2004 par la Oxford University Press.

Une matrice qui, elle aussi, a la forme (1) est la matrice dite robuste à l'hétéroscédasticité et à l'autocorrélation, ou HAC, pour Heteroskedasticity and Autocorrelation Consistent. Cette matrice dépend d'un paramtre, dit le paramtre de troncature des retards. Voir la section 9.9 de l'ouvrage de Davidson et MacKinnon pour un exposé assez complet. La syntaxe de la fonction `HAC` est illustrée par la commande

```
mats V = HAC(3,0)
```

qui demande la matrice associée à la première régressande (dont l'indice est 0), avec un paramtre de troncature égal à 3.

Les variables `HC`i, et les fonctions `HC`i, $i = 0, 1, 2, 3$, et la fonction `HAC` sont disponibles non seulement suite aux commandes `ols` et `iv`, mais aussi après l'exécution des commandes `nls` et `nliv`, qui effectuent des estimations par moindres carrés non linéaires et variables instrumentales non linéaires respectivement.

La commande `gmmweight` a été introduite pour la première fois dans la version 3.3 d'***Ecta***. Des différentes commandes permettant de minimiser une fonction objectif, c'est elle qui est le plus fortement reliée à la méthodologie économétrique. Au lieu des différentes formes du HCCME, qui n'ont pas forcément un sens dans le contexte de la méthode généralisée des moments, on a simplement la variable `HCCME`. On rajoute à cela la fonction `HAC`, ici avec un seul argument, le paramtre de troncature des retards, parce qu'avec `gmmweight` on ne fait qu'une seule estimation à la fois. Un changement dans la nomenclature des variables : la variable appelée `res` est désormais remplacée par `zerofn`, parce qu'on a affaire à des **fonctions zéro élémentaires** (*elementary zero functions* en anglais) plutôt qu'à des résidus au sens habituel ; voir la section 9.5 de l'ouvrage de Davidson and MacKinnon. Une variable supplémentaire créée par `gmmweight` est justement `weightmatrix`. Elle contient la matrice de pondération telle qu'elle est évaluée à la fin de l'estimation.

Autres Commandes

Il faudra reporter aux chapitres suivants certains aspects des fonctionnalités étendues des commandes de la version 4. Ici, on se limite à quelques modifications mineures. La commande `differentiate`, peu utilisée sauf par moi, a changé de syntaxe.

* * * *

> Je pense que c'est par pure inadvertence. Mais la syntaxe de la version 4 me paraît légèrement plus logique

* * * *

On a maintenant

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

avec la <*variable*> avant l'<*expression*>.

La commande `expand` a toujours la même syntaxe, où en demandant

`expand` <*macro*>

on fait afficher la représentation interne de l'expression associée à la <*macro*> contenue dans la table de macros d'***Ects***. Mais, étant donné que la représentation interne des expressions est tout à fait différente avec ***Ects*** 4, la représentation texte de cette représentation interne est aussi différente. En fait, c'est la même que celle employée par `showinternal`, et que nous avons vu plus haut.

Le fait d'avoir plusieurs générateurs de nombres aléatoires nécessite un changement dans la syntaxe de la commande `setseed`. Pour tous les GNA, on peut faire maintenant

`setseed` <*expression*>

où <*expression*> est une expression qui, une fois évaluée, donne une matrice de la bonne taille pour le générateur actif au moment de la commande. Pour l'ancien générateur, `jgm`, `default`, ou `old`, on peut encore se servir de l'ancienne syntaxe :

`setseed` <*expression1*> <*expression2*>

où les deux <*expression*>s donne les deux points de départ (*seeds* en anglais) de ce GNA, pour lequel il en faut exactement deux. Mais la première syntaxe est aussi acceptable, si l'évaluation de l'<*expression*> donne lieu à une matrice de deux éléments.

Le GNA `250` a besoin de pas moins de 251 chiffres pour définir son point de départ ; 250 plus encore un, dont on a besoin pour une raison que je ne peux pas expliciter ici. Pour `mt`, comme pour l'ancien générateur, il faut 2 chiffres, ce qui implique que l'ancienne syntaxe marche aussi avec ce GNA. Le générateur `kiss` a besoin de 4 chiffres, tandis que `congruential` se limite à un seul.

Chapitre 2
Fonctionnalités Nouvelles

1. Les Chaînes de Caractères

Avec ***Ects***, on arrive sans trop de peine à manipuler les matrices. Mais jusqu'ici il a été tout à fait impossible de manipuler directement les chaînes de caractères. Il est possible, comme le démontre le code ***Ects*** 3 contenu dans le fichier de commandes `newlogit.ect`, ou même le code ***Ects*** 2 du fichier `logit.ect`, de construire dans le fichier de sortie des tableaux similaires à ceux produits par ***Ects*** même pour les procédures d'estimation comme `ols`. Mais le code est très fragile, et difficile à programmer.

Ects 4 possède une table de chaînes, tout comme les tables de matrices, de macros, d'équations, de procédures, et de fonctions définies par l'utilisateur. Pour ajouter une chaîne à cette table, il y a deux commandes qui servent. La première est `deftext`. Si on fait

```
deftext texte
```

une chaîne vide est associée au nom `texte`. Si une chaîne portant ce nom se trouve déjà dans la table, elle est écrasée et remplacée par la chaîne vide.

L'autre commande qui met une nouvelle chaîne dans la table est `settext`. Là, on peut faire

```
settext texte Un petit bout de texte
```

et la chaîne affectée au nom `texte` est en effet `Un petit bout de texte`. Une autre syntaxe est souvent plus utile. On ne met rien sur la première ligne de la commande après le nom `texte`, et, à la ligne suivante, on commence un texte qui peut s'étaler sur plusieurs lignes, terminées comme d'habitude par le mot clé `end`.

Afin d'illustrer le fonctionnement des différentes commandes de manipulation des chaînes, nous allons reprendre l'estimation du modèle logit par rgression artificielle, vue antérieurement dans le fichier `newlogit.ect`. Le fichier `logit.ect` contient le nouveau code. On refait le formatage des résultats de la manière suivante.

```
mat parms = rowcat(a,b1,b2,b3)
sample 1 4
gen T = parms/stderr
```

```
mat alignnums = colcat(parms,stderr,T)

settext tableau
  Estimation du Modèle Logit par la Régression Artificielle BRM

  Nombre d'itérations =
end
appendspace tableau
setprecision 0
appendnum tableau i
appendnewline tableau
setprecision 6
halign partable
\hskip2#\hfil&\hskip2\hfil#&\hskip2\hfil#&\hskip2\hfil#\cr
\cr
Paramètre&Paramètre Estimé&Écart-type&Student\hskip1\cr
\cr
a&\#/\hskip3&\#/\hskip1&\#/\cr
b1&\#/\hskip3&\#/\hskip1&\#/\cr
b2&\#/\hskip3&\#/\hskip1&\#/\cr
b3&\#/\hskip3&\#/\hskip1&\#/\cr
end

appendtext tableau partable
settext next
  Valeur maximisée de la fonction de logvraisemblance =
end
appendspace next
appendnum next newcrit
appendtext tableau next
settext next

  Matrice de covariance estimée:


end
mat alignnums = XtXinv
halign covmat
\hskip4\hfil#&\hskip2\hfil#&\hskip2\hfil#&\hskip2\hfil#\cr
\#/&\#/&\#/&\#/\cr
\#/&\#/&\#/&\#/\cr
\#/&\#/&\#/&\#/\cr
\#/&\#/&\#/&\#/\cr
end
appendtext next covmat
appendtext tableau next

puttext tableau
```

On peut étudier ici la syntaxe et les effets de plusieurs commandes. On commence par la création d'une chaîne `tableau` qui, à la fin, contiendra tout ce que nous voulons imprimer dans le fichier de sortie. Au début, la chaîne n'a que ses trois premières lignes, dont une vide. La troisième ligne contient le texte

```
Nombre d'itérations =
```

après quoi nous voulons mettre un espace blanc avant de mettre le nombre sous forme chiffrée. On le fait par

```
appendspace tableau
```

La commande `appendspace` est suivie du nom de la chaîne à la fin de laquelle on souhaite ajouter un espace blanc. La commande `appendnewline`, à la syntaxe identique, ajouterait un saut de ligne. Après avoir supprimé les décimales après la virgule par `setprecision 0`, on met le nombre d'itérations au moyen de la commande `appendnum` comme suit:

```
appendnum tableau i
```

Le premier argument est le nom de la chaîne, ensuite on a une expression susceptible d'être évaluée comme par une commande `set`. Ceci signifie qu'on aurait pu mettre `i+1` à la place de `i`, et la chaîne ajoutée à la fin de `tableau` aurait été la version texte (sans décimales) de la valeur de l'expression `i+1`.

Passons pour le moment sur la table créée par la commande `halign`. Après, cette table est rajoutée à la fin de la chaîne `tableau`. Ensuite, on crée une nouvelle chaîne nommée `next` (pour « la suite »). On veut mettre dans le tableau des résultats la valeur maximisée de la logvraisemblance. Après le formatage de cette ligne, elle est rajouté au `tableau` par la commande `appendtext` :

```
appendtext tableau next
```

où à la fin du `tableau` on annexe le contenu de `next`. On voit que la commande a deux arguments, le premier la chaîne qui reçoit le contenu du second ajouté à la fin de son contenu existant.

Ensuite on rajoute la matrice de covariance encore une fois formatée par un `halign`. Après tout ça, on arrive au moment où nous voulons insérer les résultats de notre travail dans la fichier de sortie. Ceci se fait au moyen de la commande `puttext`, dont la syntaxe n'a rien de mystérieux : le nom de la commande est suivi du nom d'une chaîne de caractères.

On aurait pu travailler de la manière dont il faut procéder avec les versions antérieures, c'est--dire, en mettant directement dans le fichier de sortie les bouts de textes, les chiffres, les tableaux *etc.*, au fûr et à mesure de leur création. Avec une telle politique, on aurait des commandes comme les suivantes.

```
settext tableau
  Estimation du Modèle Logit par la Régression Artificielle BRM

  Nombre d'itérations =
```

```
end
puttext tableau
putspace
setprecision 0
putnum i
putnewline
```

où on se sert de la commande `puttext`, et aussi des commandes `putspace`, `putnum`, et `putnewline`, qui mettent dans le fichier de sortie un espace blanc, un chiffre, et un saut de ligne respectivement.

Il existe encore une commande reliée à l'affichage des chaînes : `showtext`. On fait `showtext` suivi du nom d'une chaîne, et la chaîne est affichée à l'écran. C'est donc l'analogue pour l'écran de la commande `puttext` pour le fichier de sortie.

Une chaîne pour une matrice

La nouvelle commande `printmatrix` permet de créér une chaîne de caractères, et de la mettre dans la table, qui contient la représentation d'une matrice. La syntaxe est comme suit:

`printmatrix` <*chaîne*> <*matrice*> [<*indent*>]

Le premier mot après `printmatrix`, <*chaîne*>, est le nom qui sera affecté à la chaîne. Le deuxième mot, <*matrice*>, est le nom d'une matrice qui doit obligatoirement se trouver dans la table de matrices, sous peine d'erreur. Le troisième argument, <*indent*>, est facultatif. S'il est donné, il est interprété comme une expression algébrique, dont la valeur est le nombre d'espaces blancs au début de chaque ligne de la matrice. Si cet argument n'est pas donné, on utilise la valeur par défaut de 2.

EXERCICES:

Refaites la programmation de `logit.ect` pour utiliser la commande `printmatrix` afin d'afficher la matrice de covariance estimée.

La Syntaxe de `halign`

J'ai remarqué au chapitre dernier que la syntaxe de la commande `halign` a changé. Je ne reprends pas ici la discussion détaillée de comment se construit un tableau au moyen de cette commande. Cette discussion se trouve dans le deuxième volume de la documentation d'***Ects***, et elle est encore bonne. C'est simplement la ligne qui contient la commande elle-même qui a changé. Avant, on mettait simplement `halign`, avec, sur les lignes suivantes, la définition du tableau aligné. Par la suite, on pouvait faire appel au tableau, du moins dans certaines circonstances, en l'appelant simplement `halign`. Maintenant, la chaîne de caractères créée par `halign`, comme tout autre chaîne, est insérée dans la table des chaînes. Il lui faut donc un nom, et, comme on l'a vu dans le code plus haut, ce nom doit suivre le mot `halign` sur la première ligne.

Cette nouvelle approche est non seulement plus compatible avec les autres commandes ayant trait aux chaînes, mais elle permet de faire appel à plus d'un `halign` en même temps, chose impossible avant, parce qu'une seconde définition faisait écraser la définition précédente.

EXERCICES:

Écrivez un programme où, en utilisant les données du fichier `ols.dat`, vous faites tourner la rgression comme dans le fichier `ols.ect`, mais vous supprimez le listing habituel pour mettre à sa place un listing où la matrice de covariance estimée, ainsi que les écarts-type et les Students, sont donnée par la première version du HCCME, celle donnée par la fonction `HC0`.

Les Commandes Construites en Cours de Route

La dernière commande que nous traitons dans cette section est la commande `exec`. La syntaxe est simple:

```
exec <nom d'une chaîne>
```

Le contenu de la chaîne doit être une commande ***Ects*** en bonne et due forme, et l'effet de la commande est de faire exécuter cette commande. Le code suivant illustre bien comment on peut se servir de la commande, en combinaison avec les nouvelles commandes `readvariables` et `writevariables`, présentées au chapitre dernier.

```
readvariables reg.dat

sample 1 obsnumber

settext cmd writevariables new.dat
appendspace cmd
appendtext cmd variablenames
exec cmd
```

On saisit l'immense contenu du fichier `reg.dat`, et, sur la base de cette opération, on définit la taille de l'échantillon, en se servant de la variable `obsnumber`. Ensuite on crée une chaîne qui contient le début d'une commande `writevariables`, où on demande que certaines variables soient mises dans le fichier `new.dat` dans le format agréé par la commande `readvariables`. Ensuite la ligne de commande est complétée par un espace blanc suivi de la longue chaîne contenant tous les noms des variables saisies par `readvariables`, cette chaîne étant disponible sous le nom de `variablenames`. Ce qu'on a créé ressemble donc à

```
writevariables new.dat vble1 vble2 vble3 ...
```

À la dernière ligne du code, on fait exécuter la commande.

EXERCICES:

Essayez de faire la même chose sur une échelle plus modeste avec les données du fichier `ols.dat`. Vous créerez d'abord un nouveau fichier, où vous mettez les noms

des quatre variables de `ols.dat` dans la première ligne. Si vous ne connaissiez pas les noms contenus dans le fichier de données, comment pourriez-vous, au moyen d'une commande `exec`, faire tourner la rgression de la première variable sur la constante et les autres variables, comme dans `ols.ect` ?

Commentaires dans le Fichier de Sortie

La nouvelle programmation d'***Ecta*** a donné lieu à une conséquence qui pourrait gêner les utilisateurs qui aiment que les commentaires dans le fichier de commandes soient reproduits dans le fichier de sortie. Si on fait par exemple

```
rem Ceci est un commentaire
```

alors aucune trace de ce commentaire ne se trouvera dans le fichier de sortie. On se rappelle que la commande `rem` est une commande qui ne fait rien. Une autre manière d'insérer des commentaires dans le fichier de commandes est de les précéder d'un `#` ou d'un `%`. Là non plus, le commentaire ne figure pas dans le fichier de sortie.

Comment faire alors ? On peut se servir des commandes `settext` et `puttext` de la manière suivante :

```
@settext rem
## Ceci est un commentaire ##

end
@puttext rem
```

On se rappelle que l'utilisation du symbole `@` sert à supprimer l'affichage de la commande dans le fichier de sortie. La seule chose qui y apparaît avec ce bout de code est tout simplement

```
## Ceci est un commentaire ##
```

avec les `#`, qui, étant à l'intérieur d'une chaîne, n'ont pas leur sens habituel.

2. Les Blocs de Commandes

La manière dont ***Ecta*** lit un fichier de commandes et les exécute a complètement changé depuis la version 3. Jusque là, les commandes étaient lues et interprétées une à une, et exécutées aussitôt. Maintenant, ***Ecta*** lit d'abord tout le contenu du fichier de commandes, en faisant une interprétation partielle des commandes, ensuite gardées dans un format interne. Si des erreurs de syntaxe sont constatées, l'exécution n'a pas lieu, et on ne perd pas son temps à essayer d'exécuter des commandes probablement sans aucun sens.

Si la syntaxe est correcte, on passe à l'exécution des commandes. Bien entendu, d'autres erreurs peuvent être détectées pendant l'exécution, si, par exemple, on fait appel à des variables ou à des fonctions inexistantes. Mais,

dans ce cas, l'exécution continue, donnant lieu éventuellement à d'autres erreurs encore.

On peut voir maintenant pourquoi il nous faut une commande `run` et une directive donnée par le mot clé `include`, l'une bien différente de l'autre. Si on se sert de la directive `include`, le contenu du fichier auquel on fait référence est pris en compte lors de la première lecture des commandes, et les erreurs de syntaxe éventuelles sont détectées. Si on se sert de la commande `run`, la syntaxe de la commande `run` est vérifiée à la première lecture, mais non celle du contenu du fichier référencé. Si par exemple, on fait dans ce deuxième fichier des définitions, de macros ou autre chose, dont le premier fichier aurait besoin afin de vérifier la syntaxe, il faut faire un `include`. Sinon, les définitions ne seraient pas disponibles avant l'exécution, éventuellement trop tard. Nous trouverons plus loin une autre différence entre `run` et `include` qui peut donner lieu à des surprises.

Les macros sont principalement concernées par ces enjeux. Si on fait

```
def residu = y-a*iota-b*x1
```

alors la représentation interne du membre de droite `y-a*iota-b*x1` est insérée directement dans la table de macros, lors de la première lecture des commandes. Imaginons que par la suite on fait une deuxième définition qui fait appel à la première :

```
def scr = residu'*residu
```

Ce qui est inséré dans la table de macros, associé au nom `scr`, tient compte de la définition antérieure de `residu`, de sorte qu'on aboutit à la représentation interne de l'expression

```
(y-a*iota-b*x1)'*(y-a*iota-b*x1)
```

Si la définition de `residu` n'etait pas encore disponible lors de la définition de `scr`, on aurait plutôt la représentation interne de

```
residu'*residu
```

qui pourrait ne pas correspondre à ce que l'on souhaite.

Le fichier `proclogit.ect` est l'un des rares fichiers fournis avec le deuxième volume de documentation à titre d'illustration qui tourne sous ***Ects*** 3 mais donne lieu à des erreurs sous ***Ects*** 4. C'est précisément parce qu'une macro est redéfinie à un endroit qui ne gêne pas si l'exécution des commandes vient tout de suite après la lecture, mais qui ne convient pas du tout au nouveau système de lecture du fichier entier avant l'exécution. Par hasard, c'est dans ce même fichier qu'il se trouve deux autres incompatibilités entre les deux versions, entièrement indépendantes de celle due à la redéfinition d'une macro. J'en parlerai dans l'ordre.

Le premier changement qu'il faut pour que `proclogit.ect` tourne sous ***Ects*** 4 est tout simplement de changer l'endroit dans le fichier où la définition de la macro `X` se fait pour la troisième fois. Normalement, c'est un indice d'un mauvais style de programmation de redéfinir plusieurs fois la même macro.

Si je l'ai fait ici, c'est parce que je voulais illustrer quelques subtilités du fonctionnement d'***Ecta***. Cette macro est définie pour la première fois non loin du début du fichier, comme suit :

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

Ensuite, à l'intérieur de la définition de la procédure appelée `logit`, la macro est redéfinie :

```
def X = arg1*iota + x1*arg2 + x2*arg3 + x3*arg4
```

de manière à faire appel aux variables locales `arg`i, $i = 1, 2, 3, 4$, de la procédure. Cette technique, quoique possible, est fortement à déconseiller ! La raison en est que les variables locales disparaissent des tables d'***Ecta*** après l'évaluation d'une procédure ou autre bloc de commandes. Si la macro en garde la trace, on risque d'avoir des ennuis ultérieurs pour cause de variables non définies. La troisième définition de la macro `X`, identique à la première, se trouve juste avant la commande `mlhess` qui se sert de la procédure `logithess`. Mais cette procédure est définie *avant* la redéfinition de la macro, de sorte que celle-ci utilise la deuxième définition plutôt que la troisième. On tombe justement dans le piège de variables non définies lors de l'estimation par `mlhess`. Il suffit de déplacer la troisième définition *avant* la définition de la procédure `logithess` pour que la commande `mlhess` remarche.
Le deuxième changement nécessaire est dans l'estimation par `mlar` à la fin du fichier. Juste avant cette commande, on a la définition d'une équation :

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

Ensuite, dans la seconde ligne de la commande `mlar`, on a simplement

```
leftside
```

Ceci est tout à fait légitime selon les règles d'***Ecta*** 3. Mais ***Ecta*** 4 considère `lhs` comme un mot clé, qui doit paraître explicitement, et non implicitement, caché dans une `equation`. Là, il suffit de mettre explicitement

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

à la place de `leftside`, et l'estimation se déroulera correctement.

Le troisième changement est imposé par un changement de politique. Dans la documentation de la version 3, je dis clairement la chose suivante :

> Cette évaluation se fait comme si on était dans une commande `set`, assez logiquement, parce que les arguments doivent être des *scalaires*.

Et un peu plus loin :

> 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.

Ceci n'est plus vrai avec ***Ecta*** 4. Les arguments sont évalués désormais à la manière d'une commande `mat`. Il s'ensuit que les deux éléments de la réponse `bb2` sont conservés, avec des résultats peu souhaitables. Pour remédier à cette circonstance, il suffit de remplacer la commande `gen` dans la définition de la procédure `argt` par une commande `set`. Le fichier `newproclogit.ect` est une

modification de l'ancien fichier `proclogit.ect`, avec les trois changements qui permettent que le fichier tourne correctement sous ***Ecta*** 4.

Variables locales

Nous savons qu'il existe, dans le contexte des procédures, des variables locales. Celles qui existent avec ***Ecta*** 3 sont créées automatiquement. Avec ***Ecta*** 4, on peut en créer explicitement. À l'intérieur de tout bloc de commandes, on peut se servir du mot clé `local`, suivi d'une liste de noms de variables, pour que ces noms soient affectés à des variables locales, c'est--dire, à des variables qui n'existent qu'à l'intérieur du bloc. L'existence ou la non existence de variables portant les mêmes noms à l'extérieur du bloc n'est pas une contrainte. Elles existeront toujours après la fin de l'exécution des commandes du bloc. Seulement, pendant cette exécution, les variables extérieures ne sont pas disponibles : elles sont « masquées » par les variables locales.

Qu'est-ce qu'on entend par un bloc de commandes ? Le genre de bloc le plus commun est simplement l'ensemble des commandes contenues dans un fichier. Ce bloc est le bloc le plus global quand on soumet le fichier à ***Ecta***. Il est possible de déclarer que certaines variables sont locales, mais cette déclaration est inutile, étant donné qu'il n'y a pas d'extérieur où d'autres variables, moins locales, peuvent exister.

Si, dans le fichier « extérieur », on fait appel à un autre au moyen d'une commande `run`, alors les commandes du deuxième fichier constituent un bloc à l'intérieur du bloc global du fichier extérieur. Ceci a une conséquence qui peut conduire à un comportement différent sous ***Ecta*** 4 et les versions antérieures. Les variables qui déterminent la taille courante de l'échantillon sont toujours considérées comme étant locales pour tout bloc. S'il n'y a pas de commande `sample` dans le bloc intérieur, rien ne change, et le bloc hérite de l'échantillon courant du bloc extérieur. Mais si l'échantillon est changé dans le bloc intérieur, le bloc extérieur n'hérite pas de ce changement. Par conséquent, si on fait

```
run ols.ect
show y
```

on sera peut-être surpris de voir qu'un seul chiffre est affiché, correspondant au premier élément de la variable `y`, malgré le fait que le fichier `ols.ect` définit un échantillon de taille 100. Il faudrait la commande `sample 1 100` avant le lancement du fichier `ols.ect`, ou bien tout de suite après, pour que l'échantillon ait la même taille à la fois à l'intérieur et à l'extérieur.

Il est à noter que le même raisonnement ne s'applique pas du tout si on utilise `include`. C'est parce que les commandes dans le fichier qui fait l'objet d'un `include` sont mises dans le *même* bloc que celles du fichier extérieur. Une autre conséquence de ce fait est que, si on a

```
include ols.ect
show y
```

alors le `show y` ne sera pas exécuté. Pourquoi ? Parce que, à la fin du fichier `ols.ect`, il y a `quit`. Ce mot clé signifie la fin d'un bloc de commandes, et, étant donné qu'il n'y en a qu'un, le mot est pris au pied de la lettre, et ***Ects*** termine ses opérations. Si on supprime le `quit` dans `ols.ect`, l'effet du `show y` est d'afficher le vecteur entier, avec ses 100 composantes, parce que le `sample 1 100` se trouve dans le même bloc.

Des blocs imbriqués dans le bloc principal sont créés par les commandes `if` (deux blocs s'il y a un `else`), `while`, `procedure`, et `function`. Si on veut construire un bloc, afin d'y mettre des variables locales ou pour une autre raison, le moyen le plus simple est de faire comme suit :

```
if (1)
   local < liste de variables locales >
   < commandes du bloc >
end
```

La condition de la commande `if` étant toujours vrai, les commandes sont toujours exécutées, à l'intérieur d'un bloc.

D'autres blocs, d'une existence plus transitoire, sont créés par les commandes comme `nls` et `ml`, où la commande s'étale sur plusieurs lignes. La commande `recurrence` aussi fait appel à son propre bloc privé. Mais ces blocs, à la différence de ceux des autres commandes citées, n'admettent qu'un nombre de commandes limité, telle que `deriv`, selon la nature précise de la commande.

3. Les Modules

Beaucoup de logiciels scientifiques font appel à la possibilité de charger, à la demande de l'utilisateur, des fonctionnalités spéciales qui ne font pas partie du noyau du logiciel. La version 4 introduit ce type d'opération pour ***Ects***. Les instructions pour la création des modules sont données ailleurs, dans le code source d'***Ects*** ; ici je me limite à décrire comment on peut utiliser un module qui existe.

* * * *

> Le code source sera un jour disponible dans une version « littéraire », dans le sens donné à ce terme par le célèbre informaticien D. E. Knuth. En attendant, on n'a qu'à me demander la section qui porte sur les modules. Cette section sera sans aucun doute modifiée ultérieurement, mais une version préliminaire existe déjà.

* * * *

La réalisation d'un module est une librairie, compilée selon certaines règles permettant que les fonctions de la librairie puissent être employées par un autre programme, en l'occurrence, ***Ects***. Deux commandes ***Ects***, `load` et

`loadlibrary`, peuvent être utilisées pour demander le chargement d'un module. La deuxième commande, `loadlibrary` reçoit toutes ses instructions sur la ligne de commande ***Ects***, tandis que la première, `load`, lit ses instructions dans un fichier dont le nom est donné sur la ligne de commande. On peut spécifier le nom de ce fichier complètement, mais, comme pour un fichier de commandes, où l'extension `.ect` est fournie automatiquement au besoin, ici l'extension `.mod` sera rajoutée.

Un module complètement inutile, sauf pour des fins illustratives, est le module `demo`. Considérons à présent le petit fichier de commandes `demo.ect`, qui s'en sert. Voici son contenu :

```
load demo.mod
repeat What wonder is this?
sample 1 10
sets a = double(PI/3)
gens b = colcat(time(-1)*PI/6,double(time(-1)*PI/6),\
 2*(time(-1)*PI/6))
quit
```

Le fichier `demo.mod` contient les informations suivantes :

```
library ./libdemo.so
repeat
SetDouble
GenDouble
```

La première ligne doit obligatoirement commencer par le mot clé `library`. Après on met le nom de la librairie. Attention ! Le répertoire courant ne fait pas partie des répertoires où on cherche automatiquement la librairie, du moins sous Linux. La librairie n'est trouvée que si la librairie est installée dans un répertoire du chemin d'accès utilisé par le système pour la recherche de librairies dynamiques. C'est pour ça que le nom de la librairie `libdemo.so` est précédé de `./`, qui signifie que c'est justement un fichier dans le répertoire courant. On peut bien entendu mettre un autre chemin d'accès explicite, selon l'endroit où on met les librairies des modules.

Ensuite, on a les noms des fonctions contenues dans le module et qui deviennent disponibles après le chargement du module. Ici, la fonction `repeat` contient une commande, du même nom, et les fonctions `SetDouble` et `GenDouble` contiennent des fonctions, dont l'une marche sous `set` et l'autre sous `gen`, et qui font la même chose que notre fonction `deuxfois` du chapitre dernier.

Si on fait tourner `demo.ect`, on voit que la commande `repeat` répète justement, en l'affichant à l'écran, ce que l'on met sur sa ligne de commande. Le tableau généré par la commande `gens` a trois colonnes, dont les deux dernières sont identiques, les éléments égaux à deux fois les éléments correspondants de la première colonne.

Dans le cas de `demo.ect`, où il n'y a pas beaucoup d'informations dans le fichier `demo.mod`, on aurait pu préférer la commande `loadlibrary`. On lui

donne les mêmes informations que celles que l'on donne à load, mais sur la ligne de commande. La première ligne de `demo.ect` deviendrait

```
loadlibrary ./libdemo.so repeat SetDouble GenDouble
```

et tout se déroulerait par la suite de la même manière.

Génération de Variables Aléatoires

Un module beaucoup plus utile que `demo`, surtout pour les simulations, est le module `randdist`, contenu dans la librairie `librandist.so`. Le fichier `randdist.mod` répertorie 20 fonctions, dont 10 qui marchent sous `set` et 10 sous `gen`. Toutes ces fonctions génèrent des variables aléatoires qui sont des réalisations de différentes lois de probabilité. En chargeant ce module, on acquiert dix fonctions, utilisables soit sous `set` soit sous `gen`. Ces fonctions sont :

```
randbeta randCauchy randchi2 randexponential randFisher
randgamma randgeometric randlogistic randPoisson randStudent
```

La densité de probabilité de la distribution Cauchy s'écrit

$$f(x) = \frac{1}{\pi(1+x^2)}.$$

Elle est définie sur toute la droite réelle, de sorte que la fonction de répartition se calcule comme suit.

$$F(x) = \frac{1}{\pi}\int_{-\infty}^{x} \frac{dy}{1+y^2} = \frac{1}{\pi}\big[\tan^{-1} y\big]_{-\infty}^{x} = \frac{1}{2} + \frac{1}{\pi}\tan^{-1} x.$$

Des tirages de cette distribution sont donnés par la fonction `randCauchy`.

La fonction `randlogistic` est associée à la loi dite logistique, dont la fonction de répartition s'exprime comme

$$\Lambda(x) = \frac{1}{1+e^{-x}}.$$

La loi exponentielle dépend d'un paramtre, noté λ, égal à l'espérance de la variable. La fonction de répartition s'écrit $1-e^{-x/\lambda}$, et la densité $(1/\lambda)e^{-x/\lambda}$. La fonction `randexponential`, qui nécessite un argument correspondant à λ, génère des réalisations de cette loi.

La loi gamma est définie sur la droite réelle positive ; sa fonction de répartition est, pour $x > 0$ et pour un paramtre $a > 0$

$$f(x) = \frac{x^{a-1}e^{-x}}{\Gamma(a)}.$$

La fonction `randgamma`, qui prend un argument (a), tire de cette loi.

En conomtrie, on connaît bien la loi du khi-deux à n degrés de liberté. La loi est reliée à la loi gamma, mais il est plus simple conceptuellement de définir une variable χ^2_n comme la somme des carrés de n variables normales centrées et réduites indépendantes. La fonction `randchi2` donne des réalisations de la loi en fonction de l'argument n. En réalite, et malgré notre définition de la loi, n peut être un réel positif quelconque, pas forcément un entier.

La distribution F dite de Fisher mais réellement de Snedecor est définie en fonction du khi-deux. La distribution $F(n, m)$ est celle du rapport de deux variables khi-deux indépendantes, au numérateur à n degrés de libertés, au dénominateur m, chacune divisée par le nombre de degrés de liberté. On a

$$F(n.m) = \frac{\chi^2_n/n}{\chi^2_m/m}.$$

Les réalisations s'obtiennent en faisant appel à la fonction `randFisher`, qui prend deux arguments, m et n.

La loi de Student est presque un cas particulier de celle de Snedecor. Un Student à n degrés de liberté est le rapport d'une normale $\mathrm{N}(0, 1)$ à la racine carrée d'un khi-deux indépendant à n degrés de liberté, divisé par n, soit

$$t = \frac{\mathrm{N}(0, 1)}{\sqrt{\chi^2_n/n}}.$$

La fonction `randStudent`, avec un argument, n, génère des réalisations de cette loi.

Une distribution qui dépend de deux paramtres réels positifs est la distribution bêta. Elle est définie en fonction de deux variables gamma, par la relation

$$B(a, b) = \frac{\gamma(a)}{\gamma(a) + \gamma(b)}, \quad a, b > 0,$$

où les deux gamma sont indépendants. La fonction `randbeta` prend deux arguments, a et b.

Nous passons maintenant à des distributions discrètes. La loi géométrique se définit sur les entiers positifs. Si on a un événement qui se produit avec une probabilité p à chaque essai, alors le nombre d'essais indépendants qu'il faut avant que l'événement se produise est une variable qui suit la loi géométrique. Formellement, on a que, pour une variable géométrique N,

$$\Pr(N = n) = p(1 - p)^{n-1}, \quad n = 1, 2, \ldots.$$

C'est la fonction `randgeometric` qui permet de générer des réalisations de cette loi, avec l'argument p.

La distribution de Poisson est aussi une distribution discrète, théoriquement reliée à la distribution gamma. Comme celle-ci, la distribution dépend d'un

paramtre, interprété comme son espérance. Si le paramtre se note a, les probabilités sont données par la formule suivante, pour $n = 0, 1, 2, \ldots$.

$$\Pr(Y = n) = \frac{e^{-a} a^n}{n!}.$$

Les réalisations sont générées par la fonction `randPoisson`, avec le paramtre a en argument.

Ce n'est pas dans ce manuel que je vais décrire tous les modules en cours de réalisation. Le module `randdist` a fait l'objet d'un discours assez long pour deux raisons : il est déjà prêt à l'utilisation, et le descriptif permet d'apprécier les avantages des modules.

4. Fonctions Nouvelles

Le nombre de fonctions disponibles pour la première fois dans ***Ects*** 4 n'est pas énorme. Désormais, c'est la tâche des modules de compléter la liste des fonctions par des fonctions dont l'intérêt est limité à des applications très spécifiques. Mais il y a eu quelques rajouts, que je décris dans cette section.

Pour comparer les grandeurs numériques, ***Ects*** 3 n'a que trois opérations binaires. Elles ne portent pas des noms comme `sin` ou `cos`, car elles sont représentées par des symboles, comme les « fonctions » `+` et `*` pour l'addition et la multiplication. Les symboles sont `<`, `=`, et `>`. On n'admet donc que les inégalités strictes et l'égalité. ***Ects*** 4 ajoute à ces trois relations les relations d'inégalité non strictes, avec des symboles tout à fait intuitifs, `<=` et `>=`. On peut aussi avoir l'inégalité, sans en spécifier le signe. La relation `<>` sert à cette fin ; `a <> b` est vrai si et seulement si `a` est différent de `b`. Une autre souplesse, dont l'inspiration vient du C et du C++, est que la relation inverse, l'égalité, peut être représentée indifféremment par `=` ou par `==`.

Les relations d'égalité et d'inégalité existent aussi sous forme de fonctions normales. Par exemple, la valeur booléenne de `eq(a,b)` est identique à celle de la relation `a = b` ou `a == b`. L'inégalité, qui s'exprime comme `a <> b` s'exprime aussi comme `ne(a,b)`. Les inégalités strictes `a < b` et `a > b` peuvent également se noter `lt(a,b)` et `gt(a,b)` respectivement. De même, les inégalités non strictes `a <= b` et `a >= b` sont identiques à `le(a,b)` et `ge(a,b)`. Les noms de ces fonctions s'inspirent des fonctions analogues du Fortran.

Dans les calculs numériques, il est possible de demander des choses impossibles, du moins dans le champ des réels. Les exemples les plus fréquents sont la racine carrée ou le logarithme d'un nombre négatif. Le calcul d'une telle chose aboutit à quelque chose qui a une représentation dans l'arithmétique flottante utilisée par l'ordinateur. On l'appelle `NaN`, pour l'expression anglaise *Not a Number*. De même, quand on calcule quelque chose dont la valeur déborde les limites de l'arithmétique flottante, on obtient un résultat « infini », dont la

représentation symbolique est `Inf` ou `-Inf`. Bien entendu, aucune opération arithmétique ultérieure n'est possible sur ces objets-là.

On peut détecter la présence de ces « valeurs » en utilisant la fonction `badvalue`. Cette fonction donne une valeur booléenne, c'est--dire, 0 ou 1, correspondant à `faux` ou `vrai` respectivement. Si l'argument de la fonction est un `NaN` ou `Inf`, la valeur de la fonction est 1 (vrai), mais si l'argument est un réel ordinaire, dans le sens de l'arithmétique flottante, la valeur est 0 (faux).

EXERCICES:

Créez un vecteur dont les éléments sont des logarithmes de nombres dont certains sont négatifs. Purgez les éléments `NaN` ou `Inf` en utilisant la fonction `badvalue` en combinaison avec l'une des fonctions `select`, `rowselect`, ou `colselect`.
Le factoriel est une fonction dont la valeur tend très vite vers l'infini. Pour quel argument est-ce que votre ordinateur estime que la valeur de son factoriel est infinie?

Les anciennes versions d'***Ects*** connaissent la fonction `round`, qui sert à arrondir un réel (flottant) à l'entier le plus proche. À cette fonction s'ajoutent deux fonctions similaires, empruntées à la librairie standard du C : `ceil` et `floor`. La première arrondit toujours vers le haut, la seconde vers le bas. Dans tous les cas, si l'argument est déjà un entier, la valeur rendue est cet entier.

* * * *

Et si l'argument est infini, ou un `NaN`, la valeur est celle de l'argument inchangée.

* * * *

Malgré ce que j'ai dit plus haut concernant la fonction Gamma, il y a dans ***Ects*** 4 la fonction `gamma`, qui est tout simplement la fonction Gamma, l'exponentielle de ce qui est donné par la fonction `gln`. On aurait pu définir une fonction « factoriel » plus aisément en se servant de cette fonction nouvelle. Mais comme vous le savez déjà si vous avez fait l'exercice, la valeur tend très vite vers l'infini.

Ects 3 a vu l'introduction des fonctions de Bessel sous les noms de `j0`, `j1`, `y0`, et `y1`. Ce choix des noms des fonctions correspond bien aux notations habituelles, mais il a donné lieu à beaucoup de confusion, due simplement au fait qu'on a souvent envie de noter `y0` ou `y1` une variable. J'ai donc pris la décision de renommer ces fonctions, qui sont désormais `jbessel0`, `jbessel1`, `ybessel0`, et `ybessel1`. Pour de plus amples renseignements sur ces fonctions, voir l'ouvrage d'Abramowitz et Stegun, *Handbook of Mathematical Functions*, publié par Dover. L'édition originale porte la date de 1965, mais, depuis lors, il y a eu de très nombreuses réimpressions. Voir aussi le site

`http://www.math.sfu.ca/~cbm/aands/`

Une fonction qui manque dans ***Ects*** 3 est la fonction `polygamma`. En réalité, on a une suite de fonctions, qui se définissent de la manière suivante. On part de la fonction digamma, disponible depuis la version 3 du logiciel. Cette fonction est la dérivée du logarithme de la fonction Gamma :

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

La fonction est disponible sous deux noms différents, `digamma` et `psi`. Dans la documentation de la version 3, je remarque que la dérivée de la fonction `digamma` n'est pas disponible ; que la valeur de l'expression

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

est toujours nulle. Ce n'est plus le cas. La dérivée de la fonction digamma porte le nom de fonction trigamma, dont la dérivée est la fonction tetragamma, et ainsi de suite (du moins si on se souvient de la numérotation en grec !). La notation d'***Ects*** est décalée par rapport à cette numérotation grecque. Ainsi, la fonction trigamma se note `polygamma(x,1)`, la fonction tetragamma `polygamma(x,2)`. En principe, on peut dériver infiniment. Si la fonction `polygamma` est notée ψ_n, on a

$$\frac{d}{dx}\psi_n(x) = \psi_{n+1}(x).$$

La fonction ψ_0 est la fonction digamma, ψ_1 la trigamma, et ainsi de suite.

L'indice n de la fonction est forcément un entier non négatif. Cet indice est le deuxième argument de la fonction `polygamma`, le premier étant le vrai argument, que nous avons noté x. Un deuxième argument négatif est remplacé par 0, et un deuxième argument positif mais non entier est arrondi vers la bas.

Exercices:

Servez-vous de la commande `showinternal` pour voir directement comment ***Ects*** calcule la dérivée de la fonction `digamma`. Ensuite trouvez la représentation interne de la dérivée de la fonction `polygamma(x,1)`, et plus généralement de la fonction `polygamma(x,n)`.

Les fonctions `greatest` et `smallest` sont employées si on veut savoir le plus grand ou le plus petit élément d'un vecteur ou d'une matrice. Il est parfois utile de savoir non seulement la valeur de l'élément mais aussi où il se trouve dans le vecteur ou la matrice. Cette information est fournie par les fonctions `argmax` et `argmin`, disponibles, comme `greatest` et `smallest`, sous `set` et `gen` uniquement.

Un exemple permettra d'illustrer le fonctionnement de ces deux fonctions, et aussi celui d'une autre fonction nouvelle, `shuffle`, qui n'est disponible que sous `gen`. Considérez le code suivant.

```
sample 1 20
```

```
gen z = time(0))
gen x = shuffle(z)
sets mx = greatest(x)
sets mn = smallest(x)
sets ex = argmax(x)
sets en = argmin(x)
sets ttx = mx-x(ex)
sets ttn = mn-x(en)
show z x
```

On crée le vecteur `z` dont les éléments sont les entiers de 1 à 20 dans l'ordre. La fonction `shuffle` effectue une permutation aléatoire des éléments de `z`. Par conséquent, les éléments du vecteur `x` sont encore une fois les entiers de 1 à 20, mais l'ordre est aléatoire. Le calcul des variables `mx` et `mn`, dont les valeurs sont respectivement 20 et 1, permet de vérifier que le plus grand et le plus petit élément de `x` ont les valeurs attendues. Les variables `ex` et `en` nous indiquent les indices des éléments le plus grand et le plus petit respectivement. On vérifie que le calcul est exact en calculant les différences entre les valeurs `mx` et `mn` et les valeurs des éléments correspondant aux indices `ex` et `en`. Les variables `ttx` et `ttn` doivent avoir des valeurs nulles. Finalement, en affichant côte à côte les vecteurs `z` et `x`, on voit directement que les éléments sont là où `argmax` et `argmin` les ont trouvés.

EXERCICES:

Si vous faites tourner plus d'une fois un fichier contenant ce bout de code, vous constaterez que les variables `ex` et `en` sont toujours les mêmes. En revanche, si vous exécutez le bout de code plus d'une fois dans un même fichier, les résultats sont différents. Montrez aussi que les résultats ne sont plus les mêmes si vous changez de générateur de nombres aléatoires. Ceci démontre que l'opération de la fonction `shuffle` dépend du générateur de nombres aléatoires et de son état au moment de l'appel à la fonction `shuffle`.

La fonction `reorder` n'est disponible que sous `mat`. Le mot anglais *reorder* a le sens d'une permutation, ou changement d'ordre, mais le fonctionnement de cette fonction est entièrement différent de celui de `shuffle`. Si on a une matrice de la forme $n \times m$, on peut reformatter la matrice de façon qu'elle devient de la forme $p \times q$, à condition que $mn = pq$. Soit `A` la matrice. On fait alors

```
mat B = reorder(A,p,q)
```

et la matrice `B` a exactement les mêmes éléments que `A`, dans le même ordre, mais affectés différemment aux lignes et aux colonnes. Pour bien comprendre l'effet de la fonction, il faut savoir qu'***Ects*** range les éléments d'une matrice par lignes. Si `A` est une matrice $n \times m$ donc, les m premiers éléments sont ceux de la première ligne, suivis des m éléments de la deuxième ligne, et ainsi de suite.

Si la condition $mn = pq$ n'est pas vérifiée, la matrice `B` est toujours une matrice de la forme $p \times q$, mais tous les éléments sont nuls.

Exercices:

Montrez comment on peut transposer un vecteur colonne en un vecteur ligne en utilisant la fonction `reorder`. Est-ce qu'une manipulation de ce type permettrait de tranposer une matrice quelconque? Pourquoi ou pourquoi pas?

Convertissez le vecteur `x` de l'exercice précédent en une matrice 5×4. Vérifiez que les fonctions `argmax` et `argmin` trouvent toujours les éléments le plus petit et le plus grand aux endroits qui correspondent au rangement d'***Ects*** des éléments d'une matrice.

Construction de Matrices

La fonction `emptymatrix` existe depuis la version 3 d'***Ects***. Elle permet de construire une matrice vide, c'est--dire, une matrice qui a 0 lignes et 0 colonnes. Cette fonction a été étendue dans ***Ects*** 4 ; elle accepte maintenant un ou deux arguments. Dans le premier cas, si l'argument est l'entier non négatif k, la matrice créée est un vecteur $k \times 1$. Dans le deuxième cas, si les arguments sont les entiers non négatifs k et l, la matrice a la forme $k \times l$. Tous les éléments des matrices ainsi créées sont nuls. L'utilisation de `emptymatrix` est limitée à la commande `mat`.

Il est à noter que l'on peut créer avec la fonction `emptymatrix` des matrices vides, dont le nombre de lignes *ou* le nombre de colonnes est différent de zéro, si l'autre dimension est égale à zéro. En effet, le nombre d'éléments est égal au produit kl, qui s'annule si $k = 0$ ou $l = 0$. Je pense que ceci est anodin, mais on peut avoir des surprises auxquelles je ne m'attends pas moi-même si on joue avec des matrices vides dont les dimensions ne sont pas rigoureusement 0 et 0.

Si on donne des dimensions négatives à la fonction `emptymatrix`, elles sont remplacées par 0.

On peut avoir envie de définir une matrice par son élément type. Jusqu'ici, la construction d'une matrice sur la base d'une formule algébrique n'a pas été possible. Maintenant on peut le faire, en se servant d'une nouvelle syntaxe associée à la commande `mat`. Il faut d'abord créer une matrice de la bonne taille, par `emptymatrix` ou autrement. La suite de la construction est illustrée par le bout de code suivant.

```
mat Omega = emptymatrix(5,5)
set rho = 0.9
mats Omega(t,s) = rho^abs(t-s)
```

On se rappelle que la matrice de covariance aléatoire d'un processus AR(1) est une matrice dont l'élément (t, s) est donné par l'expression $\rho^{|t-s|}$, où ρ est le paramtre d'autocorrélation. Dans l'exemple, la taille de l'échantillon est faible, afin d'éviter la création de matrices trop volumineuses et difficiles à afficher.

La commande qui nous intéresse à présent est celle-ci :

```
mats Omega(t,s) = rho^abs(t-s)
```

La matrice `Omega` est créée avant le lancement de cette commande. Les variables `t` et `s` sont des variables locales, limitées à cette commande. Il n'est pas du tout nécessaire qu'elles se nomment `t` et `s` ; on aurait pu utiliser `i` et `j` ou toute autre combinaison. La seule contrainte est que les deux indices doivent être différents. Si on avait écrit `Omega(t,t)` à la place de `Omega(t,s)` on n'aurait pas eu le résultat souhaité.

Suite à la commande `mats`, la matrice `Omega` est affichée. Si on fait tourner le code, on verra que la matrice a la forme 5×5, et que l'élément (t, s) a bien la valeur $\rho^{|t-s|}$. L'expression algébrique qui constitue le membre de droite de la commande est interprétée comme si on avait lancé une commande `gen` avec le début de l'échantillon à l'observation 1 et la fin à l'observation dont l'indice est le nombre d'éléments de la matrice dans le membre de gauche de la commande.

Intégration Numérique

L'intégration numérique a été introduite dans la version 3 d'***Ecta***. La fonction pertinente se nomme `int`, et jusqu'ici elle n'est disponible que sous `set`.

* * * *

Une erreur dans la programmation de l'intégration numérique a été repérée. Cette erreur ne donne pas de résultats faux (heureusement !), mais il conduit à des temps de calcul trop longs. Dans la version 4 l'erreur est corrigée.

* * * *

Le fait de n'être disponible que sous `set` a pour conséquence que la commande ne savait calculer qu'une intégrale à la fois. Or, dans certaines estimations par le maximum de vraisemblance, chaque contribution à la fonction de log-vraisemblance est une intégrale que l'on ne peut calculer que numériquement. Mon collègue Stéphane Luchini a attiré mon attention sur des modèles de ce type.

À la demande de Stéphane Luchini donc, la fonction `int` est désormais disponible sous `gen`. Ce fait permet une grande souplesse, parce que non seulement la fonction à intégrer peut être différente selon l'observation, mais aussi les limites de l'intégration. Un exemple :

```
sample 1 10
gen t = 0.1*time(0)
gen I = int(x^t,t,t+1,x)
gen J = ((t+1)^(t+1)-t^(t+1))/(t+1)
show I J
```

Les valeurs de la variable `t` sont $0.1, 0.2, 0.3, \ldots, 0.9, 1$. Pour chacune de ces valeurs, on demande l'évaluation de l'intégrale

$$\int_t^{t+1} x^t \, dx.$$

Un calcul simple montre que la valeur de l'intégrale, en fonction de t, est donnée par l'expression

$$\frac{1}{t+1}\left((t+1)^{t+1} - t^{t+1}\right).$$

En faisant tourner le code, on peut voir que l'intégrale est calculée correctement par la commande `int`.

On se rappelle la syntaxe de la commande :

`int(<expn>, <inf>, <sup>, <variable>)`

où <*expn*> est l'expression algébrique de la fonction à intégrer, <*inf*> et <*sup*> sont respectivement les expressions algébriques des bornes inférieure et supérieure, et <*variable*> est la variable par rapport à laquelle on intègre.

La fonction à intégrer, <*expn*>, est évaluée selon les règles d'une commande `gen`, le début de l'échantillon à l'observation 1 et la fin là où il le faut dans l'évaluation itérative de l'intégrale. Les bornes sont évaluées comme sous `set` si la fonction `int` est utilisée dans une commande `set`, et comme sous `gen` si on est dans une commande `gen`. La <*variable*> doit être un symbole, et non pas une expression algébrique, sous peine d'erreur de syntaxe.

Exercices:

La fonction de répartition de la loi normale centrée réduite s'exprime comme

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp(-\tfrac{1}{2}z^2)\, dz.$$

La fonction Φ est fournie par ***Ecta*** sous le nom de `Phi`. Comparez les résultats donnés par `Phi` et `int` pour des valeurs de x entre 0 et 2, en remplaçant la borne de $-\infty$ par un réel négatif suffisamment grand en valeur absolue.

5. Fonctions Modifiées

Le nombre de fonctions dont l'opération a été modifiée pour la version 4 est assez faible. Le changement le plus important concerne la fonction `lag`. Cette fonction est d'une très grande utilité en conomtrie, parce qu'elle sert à retarder les variables, une opération indispensable pour les modèles dynamiques. La fonction n'est disponible que sous `gen`.

D'après les indications dans le premier volume, la syntaxe de la commande est la suivante:

`gen ylag = lag(<scalaire>, <expression>)`

et l'opération de la fonction est de générer une variable `ylag` dont les éléments sont ceux de la matrice qui résulte de l'évaluation de l'<*expression*> sous les règles de `gen`, les lignes décalées vers le bas par le nombre d'éléments spécifié

par le <*scalaire*>. Si le <*scalaire*> est négatif, le décalage des lignes se fait vers le haut.

La première modification est que le scalaire qui détermine le décalage n'est plus forcément un scalaire. Le premier argument de `lag` est lui aussi évalué selon les règles de `gen`, ce qui signifie que le degré de décalage est défini élément par élément plutôt que de façon globale, une fois pour l'ensemble des éléments du deuxième argument. J'ai fait cette modification dans le but d'une meilleure cohérence des syntaxes des différentes fonction, plutôt que parce que j'envisageais une grande utilité de la fonctionnalité nouvelle.

La seconde modification est plus intéressante. Elle m'a été imposée par l'interaction entre les règles de `gen` est la déclaration de la taille de l'échantillon courant. Pour toutes les fonctions dont l'opération est effectuée élément par élément, il n'est pas difficile de se limiter au bloc d'éléments compris entre le début de l'échantillon, tel qu'il est précisé par la variable `smplstart` et la fin, précisée par `smplend`. Mais l'opération de `lag` fait appel fatalement à des éléments en dehors de l'échantillon défini. Si ces éléments n'existent pas, il n'y a pas de problème. On s'appuie sur la règle d'***Ects*** selon laquelle tout élément inexistant est remplacé par zéro.

Mais il se peut que, pour une matrice donnée, il existe des éléments non nuls du deuxième argument avant ou après l'échantillon déclaré. Dans ce cas, on veut normalement les prendre en compte. La raison pour laquelle on définit un échantillon restreint est de limiter les effets d'une commande `gen` aux éléments de cet échantillon, en laissant inchangés tous les éléments de la variable dans le *membre de gauche* de la commande qui ne sont pas compris dans l'échantillon. Mais ceci n'est pas du tout incompatible avec un accès aux éléments en dehors de l'échantillon des variables qui se trouvent dans le *membre de droite* de la commande.

L'accès aux éléments avant le début de l'échantillon en cours ne pose aucun problème. Pendant l'évaluation de la fonction `lag`, ***Ects*** étend temporairement l'échantillon vers le bas, jusqu'aux premières lignes de toutes les matrices rencontrées ou évaluées dans l'évaluation du deuxième argument. Le début déclaré reprend sa force tout de suite après l'évaluation de cet argument.

Les choses ne sont pas aussi simple pour les éléments des lignes dont l'indice est supérieur à `smplend`. Il n'est pas possible d'étendre la fin de l'échantillon jusqu'à l'infini ! La solution de second rang adoptée par ***Ects*** est de traiter différemment les deuxièmes arguments qui consiste en une seule variable de tous les autres, pour lesquels il faudrait l'évaluation d'une ou de plusieurs fonctions. S'il faut une évaluation, on ne fait rien, et on perd ainsi l'accès aux éléments plus loins. Mais s'il n'y a qu'une seule variable, que l'on trouve dans les tables sans évaluation d'une fonction, on donne à la fonction `lag` l'accès à tous ses éléments.

Une illustration assez complète de la fonction `lag` nouvelle se trouve dans le fichier `lagtest.ect`. Le début de ce fichier est comme suit:

```
sample 1 20
gen x = time(0)
sample 5 15
gen x3 = lag(3,x)
gen x32 = lag(3,2*x)
gen xm3 = lag(-3,x)
gen xm32 = lag(-3,2*x)
sample 1 20
print x3 x32 xm3 xm32
```

Les éléments de la variable `x` sont simplement l'indice de l'élément, ce qui permet de voir aisément la nature des décalages effectués par la suite. Après avoir limité l'échantillon des éléments de 5 à 15, on génère des variables par `lag` avec un deuxième argument de `x` ou de `2*x`, et un premier argument de 3 ou de -3. Les résultats tels qu'il apparaissent :

```
x3          x32         xm3         xm32
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
2.000000    4.000000    8.000000   16.000000
3.000000    6.000000    9.000000   18.000000
4.000000    8.000000   10.000000   20.000000
5.000000   10.000000   11.000000   22.000000
6.000000   12.000000   12.000000   24.000000
7.000000   14.000000   13.000000   26.000000
8.000000   16.000000   14.000000   28.000000
9.000000   18.000000   15.000000   30.000000
10.000000  20.000000   16.000000    0.000000
11.000000  22.000000   17.000000    0.000000
12.000000  24.000000   18.000000    0.000000
```

On voit que les éléments avancés sont là si on évalue `lag(-3,x)`, mais non si on évalue `lag(-3,2*x)`.

Le code dans la suite du fichier illustre un moyen simple d'obtenir le résultat souhaité. La multiplication par 2 est transférée à l'extérieur le la fonction `lag`.

```
sample 5 15
gen xm23 = 2*lag(-3,x)
sample 1 20
print xm32 xm23
```

Si on fait tourner le programme, on verra que la variable `xm23` contient les éléments qui manquent à `xm32`.

Finalement, une toute autre approche est suggérée :

```
sample 1 4
gen fill = 0
sample 1 20
```

```
gen xmm3 = rowcat(fill,x(8,18,1,1))
print xm3 xmm3
```

Pour comprendre ce bout de code, il faut savoir que les arguments de la fonction `rowcat`, même sous `gen`, sont évalués selon les règles de `mat`. Les variables `xm3` et `xmm3` sont identiques.

Quoique plusieurs fonctions disponibles sous `gen` aient besoin d'un accès à plusieurs lignes de leurs arguments pour déterminer une ligne donnée de la valeur rendue, toutes ces fonctions, à l'exception de `lag` et de deux autres, n'ont pas besoin d'un accès à des éléments en dehors de l'échantillon déclaré. Les deux autres exceptions sont la fonction `conv` et la nouvelle fonction `invconv`, dont les propriétés sont exposées dans la section suivante. À la différence de `lag`, `conv` et `invconv` ne cherchent pas à accéder à des éléments « avancés » dont les indices sont supérieurs à `smplend`. On utilise donc la même astuce que pour `lag` : pendant l'évaluation des deux arguments des ces deux fonctions, l'échantillon est temporairement étendu de façon qu'il commence par les premiers éléments de chaque variable.

Les effets de la fonction `sum` dans toutes les utilisations permises par les versions antérieures d'***Ecta*** n'ont pas changé. La modification est que la fonction a un sens nouveau dans une commande `set`. Jusqu'ici, il a été un tant soit peu difficile de calculer la somme des éléments d'une série ou d'une matrice. Il y a plusieurs méthodes indirectes, par exemple,

```
sample 1 n
gen u = ...
gen iota = 1
mat somme = iota'*u
```

ou encore

```
sample 1 n
gen u = ...
gen somme = sum(u)
set somme = sum(n)
```

Maintenant on peut faire tout simplement

```
sample 1 n
gen u = ...
set somme = sum(u)
```

Utilisée sous `gen`, la fonction crée une matrice dont les lignes successives sont les additions de toutes les lignes de l'argument entre le début de l'échantillon et la ligne en question. Comme l'indique le deuxième exemple, seule la dernière ligne contient l'addition de toutes les lignes.

Mais, dans une commande `set`, l'argument est évalué selon les règles de `mat`, et la valeur scalaire de la fonction `sum` est simplement l'addition de l'ensemble des éléments de l'argument.

6. Polynômes en l'Opérateur Retard

Considérons deux polynômes A et B, de degrés p et q respectivement:

$$A(x) = \sum_{i=0}^{p} a_i x^i, \quad B(x) = \sum_{i=0}^{q} b_i x^i.$$

Le produit C de A et B est défini par les relations algébriques habituelles comme un polynôme de degré $p+q$. On peut vérifier sans peine que, si on écrit $C(x) = \sum_{i=0}^{p+q} c_i x^i$, alors les coefficients c_i vérifient la relation

$$c_i = \sum_{j=0}^{i} a_j b_{i-j}. \tag{2}$$

Cette relation nous dit que les coefficients du polynôme C sont donnés par la **convolution** de ceux des polynômes A et B.

Ects a la fonction `conv` pour le calcul des convolutions. Cette fonction existe depuis les premières versions d'***Ects*** ; elle est décrite dans la section 6.4 du premier volume de documentation. Par définition, l'élément t de l'expression `conv(a,b)` s'exprime comme

$$\texttt{conv}_t(\boldsymbol{a}, \boldsymbol{b}) = \sum_{i=1}^{t} a_i b_{t-i+1} = \sum_{i=1}^{t} b_i a_{t-i+1}, \tag{3}$$

d'où on voit que la fonction est symétrique par rapport à ses deux arguments. En fait, les indices des équations (2) et (3) seraient les mêmes si dans (3) le premier indice était 0 à la place de 1. Par conséquent, la fonction `conv` peut servir pour la multiplication des polynômes.

Processus AR et MA

Dans la section 3.2 de la documentation de la version 3, on a remarqué que la fonction `polylag` n'existe pas dans la version 4. En même temps, on a promis de proposer une meilleure manière de générer des processus ARMA. C'est le but de cette section. Ici, nous nous limitons aux processus univariés. Les processus multivariés peuvent bénéficier d'une approche similaire, mais la fonctionnalité nécessaire fait l'objet d'un module, et n'est pas comprise dans le logiciel de base.

On se rappelle que le processus générateur d'un ARMA(p, q) s'écrit

$$A(\mathrm{L})y_t = B(\mathrm{L})u_t, \tag{4}$$

où y_t est la série ARMA(p, q), u_t est un bruit blanc, et A et B sont deux polynômes en l'opérateur retard L, de degrés p et q respectivement.

Considérons d'abord un processus MA pur, à l'ordre q, pour lequel (4) devient

$$y_t = B(\mathrm{L})u_t = u_t + \sum_{j=1}^{q} b_j u_{t-j}, \tag{5}$$

où les b_j sont les coefficients du polynôme B :

$$B(x) = 1 + b_1 x + b_2 x^2 + \ldots + b_q x^q = 1 + \sum_{j=1}^{q} b_j x^j.$$

Un moyen simple de générer une série qui vérifie l'équation (5) est d'utiliser la fonction `conv`. La procédure est comme suit. On crée une variable `b` pour contenir les coefficients du polynôme B. Pour une série qui contiendra n éléments, on génère ensuite un vecteur `u` de $n + q$ éléments, parce que, comme l'équation (5) le montre clairement, le premier élément y_1 de la série à générer dépend non seulement de u_1, mais aussi de u_0, $u_{-1}, \ldots u_{-q+1}$. Ensuite on calcule la convolution de `u` et `b`, et on garde les n derniers éléments de la série `y` ainsi générée. On pourrait faire, par exemple,

```
mat b = rowcat(1,b1,...bq)
sample 1 n+q
gen u = random()
gen y = conv(u,b)
mat y = y(q+1,n+q,1,1)
```

L'équation (3) démontre que la commande `gen y = conv(u,b)` crée une variable `y` telle que, pour $t > q$,

$$y_t = \sum_{j=1}^{t} b_j u_{t-j+1} = b_1 u_t + b_2 u_{t-1} + \ldots b_{q+1} u_{t-q}.$$

Il est à noter que la somme n'a que $q + 1$ termes, parce que le vecteur `b` n'a que $q + 1$ termes. Les indices sont trompeurs parce qu'ils commencent par $j = 1$ plutôt que par $j = 0$; en fait $b_1 = 1$, $b_2 =$ `b1`, …, $b_{q+1} =$ `bq`. De ce fait, les éléments de `y`, à partir de l'élément $q + 1$, vérifie bien la relation (5).

Si on a envie de générer plusieurs séries à la fois, on peut noter que la fonction `conv` obéit à la règle imposée sur les fonctions évaluées dans une commande `gen`, à savoir que toutes les colonnes du premier argument sont prises en compte, mais uniquement la première colonne du deuxième argument. Ceci signifie que, si la variable `u` est une matrice avec plusieurs colonnes, alors `y` sera également une matrice avec le même nombre de colonnes, chaque colonne étant la convolution de la colonne correspondante de `u` et la première colonne de `b`.

Si on ne disposait que de la fonction `conv`, il serait très simple de générer une série MA, mais non une série AR. Supposons que la série AR(p) que nous souhaitons générer est décrite par la formule $A(\mathrm{L})y_t = u_t$, ou, explicitement,

$$y_t = a_1 y_{t-1} + \ldots + a_p y_{t-p} + u_t, \tag{6}$$

avec la définition suivante du polynôme A :

$$A(x) = 1 - a_1 x - a_2 x^2 - \ldots - a_p x^p = 1 - \sum_{i=1}^{p} a_i x^i.$$

Les signes négatifs dans le polynôme permettent l'écriture (6) de la série sous la forme d'une auto-rgression avec des signes positifs. Si nous notons les coefficients du polynôme A, toujours de cette manière éventuellement trompeuse, $\texttt{a1} = 1$, $\texttt{a2} = -a_1, \ldots, \texttt{ap} = -a_{p-1}, \texttt{app} = -a_p$, alors il n'est pas difficile de se convaincre que $\texttt{conv}(\texttt{y}, \texttt{a}) = \texttt{u}$. En effet, (5) et (6) ont la même forme algébrique avec les séries y_t et u_t interverties.

Si la relation `conv(y,a) = u` est considérée comme une équation qui définit la série `y`, on voit que ce qu'il faut faire pour générer `y` en fonction de `u` est de résoudre cette équation. La nouvelle fonction `invconv`, disponible uniquement sous `gen`, fait ce travail, en résolvant l'équation afin d'exprimer `y` en fonction de `a` et `u`. Formellement,

$$\texttt{conv}(\texttt{y}, \texttt{a}) = \texttt{u} \quad \text{si et seulement si} \quad \texttt{invconv}(\texttt{u}, \texttt{a}) = \texttt{y}. \tag{7}$$

Il s'ensuit que le gros du travail de génération d'une série AR(p) est effectué par la simple commande

```
gen y = invconv(u,a)
```

À la différence de `conv`, la fonction `invconv` n'est pas symétrique par rapport à ses deux arguments. Si on avait mis `invconv(a,u)` on n'aurait pas obtenu le résultat souhaité. Mais, à l'instar de `conv`, elle permet de traiter plusieurs séries à la fois si on les met dans les colonnes du premier argument. Du deuxième argument, qui représente le polynôme A, seule la première colonne est prise en compte.

On a vu que l'initialisation d'un processus MA(q) avec n éléments exige la génération de $n+q$ éléments du bruit `u`. L'initialisation d'un processus AR(p) est un peu plus subtile. Étant donné l'équivalence entre un processus AR dont l'ordre est fini et un processus MA d'ordre infini, il paraît qu'il faut un nombre infini d'éléments du bruit `u` pour l'initialisation du processus. Dans la pratique, on peut remplacer l'infini par un nombre « assez élevé ». Par exemple,

```
sample 1 n+200
gen y = invconv(random(),a)
mat y = y(201,200+n,1,1)
```

Quelle est la différence entre cette procédure et le code suivant, qui paraît plua naturel ?

```
sample 1 n
gen y = invconv(random(),a)
```

Dans les deux cas, la série `y` vérifie l'auto-rgression définissante (6). De ce fait même, on voit que cette auto-rgression ne définit pas la série `y` de manière unique. Le deuxième choix, plus naturel, fait implicitement l'hypothèse que $\mathtt{y}_i = 0$ pour tout $i \leq 0$. Il se peut que l'on souhaite imposer cette hypothèse, mais il y a certainement des circonstances où on aurait préféré une autre initialisation. En particulier, la série générée sous cette hypothèse n'est pas une réalisation d'un processus AR(p) *stationnaire*, alors que la première procédure, où on génère 200 ou un autre grand nombre d'éléments inutiles, en est une, du moins approximativement, l'approximation étant d'autant meilleure que le nombre d'éléments inutiles est grand.

Pour avoir une série unique, il faut non seulement les éléments de la série `u`, mais aussi les valeurs des p premiers éléments de `y`. Pour l'élément y_{p+1} on a de l'équation (6)que

$$y_{p+1} = u_{p+1} + \sum_{i=1}^{p} a_i y_{p-i+1},$$

où le membre de droite ne dépend que des p premiers éléments de `y` et l'élément $p+1$ du bruit `u`. Pareillement, pour $t > p+1$, les éléments successifs de `y` sont définis en fonction des éléments antérieurs et le bruit, de sorte que toute la série `y` a une définition unique. On note que seuls les éléments dont l'indice est supérieur à p sont nécessaires, Par conséquent, l'on a une définition unique des n éléments de `y` en fonction de n nombres, $y_1, \ldots, y_p$ et $u_{p+1}, \ldots, u_n$.

Deux types d'initialisation sont couramment utilisés dans les simulations. Le premier consiste à affecter aux éléments $y_1, \ldots, y_p$ des valeurs spécifiques. Par exemple, si on a observé une série dont on analyse les propriétés par des simulations, et qui est éventuellement une réalisation d'un processus AR(p), il est naturel d'initialiser les p premiers éléments de chaque série simulée avec les p premiers éléments de la série observée.

Le deuxième type d'initialisation utilisé couramment se sert justement de la distribution stationnaire du processus. On sait par exemple que, pour le processus le plus simple, défini par l'auto-rgression $y_t = \rho y_{t-1} + u_t$, la distribution stationnaire a une espérance nulle et une variance égale à $\sigma^2/(1-\rho^2)$, où σ^2 est la variance des éléments du bruit blanc u_t. On a une réalisation d'un AR(1) stationnaire si le premier élément, y_1, est un tirage de la loi stationnaire, une loi qui est très souvent une loi normale, mais pas forcément, et dont l'espérance est nulle et la variance égale à $\sigma^2/(1-\rho^2)$. Pour les processus AR(p) avec $p > 1$, il faut la distribution stationnaire *jointe* de p éléments consécutifs du processus afin de pouvoir générer les p premiers éléments. On

peut obtenir toutes les variances et covariances de cette distribution jointe en résolvant les équations de Yule-Walker ; voir le chapitre 13 du manuel de Davidson-MacKinnon (2004) pour les détails.

Supposons maintenant que nous avons choisi, d'une manière ou d'une autre, les valeurs des éléments $y_1, \ldots, y_p$. Comment générer le reste de la série ? On utilise la première des équations (7) pour l'obtention des p premiers éléments de `u` en fonction de ceux de `y` et le polynôme A. Après, on génère les $n - p$ éléments restants de `u` par un bruit. On peut maintenant générer `y` dans son intégralité sur la base de la deuxième équation de (7). Un exemple pour le cas $n = 12$ et $p = 3$ peut faciliter la compréhension de la démarche. Aux 3 premiers éléments de `y` sont affectées les valeurs 0.6, 0.7, et 0.8. Le polynôme A, de degré 3, est donné par $A(x) = 1 - 0.3x - 0.4x^2 + 0.2x^3$. On a le code suivant.

```
sample 1 12
mat y = rowcat(0.6,0.7,0.8)
mat a = rowcat(1,-0.3,-0.4,0.2)
gen u = conv(y,a)
sample 4 12
gen u = random()
sample 1 12
gen y = invconv(u,a)
```

Exercices:

Vérifiez directement que la série `y` générée par le code ci-dessus vérifie la relation auto-régressive

$$\mathtt{y}_t = a_1\mathtt{y}_{t-1} + a_2\mathtt{y}_{t-2} + a_3\mathtt{y}_{t-3} + \mathtt{u}_t,$$

où $a_1 = 0.3$, $a_2 = 0.4$, et $a_3 = -0.2$.

Processus ARMA

Nous allons maintenant mettre ensemble ce que nous venons d'apprendre sur les processus MA et AR, en considérant les processus ARMA. L'équation définissante (4) d'un processus ARMA(p, q) est formellement équivalente à l'équation

$$y_t = A^{-1}(\mathrm{L})B(\mathrm{L})u_t. \tag{8}$$

On peut donner un sens précis à cette relation formelle en interprétant $A^{-1}(\mathrm{L})$ comme un polynôme de degré infini qui vérifie les équations

$$A(\mathrm{L})A^{-1}(\mathrm{L}) = 1 \quad \text{et} \quad A^{-1}(\mathrm{L})A(\mathrm{L}) = 1,$$

où la multiplication est définie par (2), et 1 signifie le polynôme dont tous les coefficients sont nuls sauf le premier, qui est égal à 1. En termes de la fonction `conv`, donc, on a

$$\mathtt{conv}(A, A^{-1}) = 1 \quad \text{et} \quad \mathtt{conv}(A^{-1}, A) = 1.$$

Avec cette interprétation, l'équation (8) devient $y_t = C(\mathrm{L})u_t$, pour un polynôme de degré infini C dont les coefficients vérifient une relation qui peut s'exprimer comme $\texttt{conv}(A, C) = B$, ou, ce qui est équivalent, $\texttt{conv}(C, A) = B$. Mais par (7), cette relation est équivalente à $\texttt{invconv}(B, A) = C$. On peut donc calculer les coefficients de C par la fonction `invconv`.

Il reste l'inconvénient que le polynôme C a un nombre infini de coefficients. Mais à moins de vouloir générer une série d'un nombre infini d'éléments, ceci n'est pas un inconvénient réel. Pour une série de n éléments, on n'a besoin que des n premiers coefficients de C. En fait, si nous notons C^n le polynôme de degré n qui résulte de la troncature du polynôme infini C, il n'est pas difficile de voir que $C^n(x) - C(x)$ est un polynôme dont tout les termes dépendent de puissances de x supérieures à la puissance n. Si on arrive à initialiser la série `y` convenablement, les retards de u_t d'un ordre supérieur à n ne peuvent pas avoir une influence quelconque sur les éléments $y_1, \ldots, y_n$. On peut donc générer une série `y` qui vérifie à la fois (4) et (8) par la commande

```
gen y = conv(random(),invconv(b,a))
```

Mais cette commande effectue implicitement une initialisation qui n'est peut-être pas celle que nous souhaitons. On voit donc que, encore une fois, le vrai problème est l'initialisation de la série ARMA. L'équation définissante (8) n'a pas de solution unique en fonction seulement des éléments du bruit `u`. Mais on peut voir que la solution devient unique si, pour un ARMA(p, q), on spécifie les valeurs de $y_1, \ldots, y_p$ (comme pour un AR(p)), et aussi celles de $u_p, u_{p-1}, \ldots, u_{p-q+1}$, soit au total $p + q$ valeurs. En fait, l'équation définissante (4) nous donne

$$y_{p+1} = \sum_{i=1}^{p} a_i y_{p-i+1} + u_{p+1} + u_{p+1} + \sum_{j=1}^{q} b_j u_{p-j+1},$$

et on vérifie que le membre de droite ne dépend que des valeurs initialisées et de u_{p+1}. Les éléments de `y` après l'élément p sont aussi définis par (4) en fonction des élements antérieurs et les élements du bruit. Pour la génération des n éléments de `y` donc, il faut les $p + q$ valeurs initialisées plus les $n - p$ éléments $u_{p+1}, \ldots, u_n$. Il est à noter que, dans le cas où $p > q$, les valeurs (non affectées) des éléments $u_1, \ldots, u_{p-q}$ n'ont aucune influence sur la série `y`.

Dans le cas d'un AR(p) pur, l'initialisation des p premiers éléments est un cas particulier de l'initialisation du paragraphe précédent, avec $q = 0$. De même, pour un MA(q) pur, on retrouve l'initialisation de tout à l'heure par les q éléments du bruit `u` qui précède le processus généré.

Si on veut générer une réalisation d'un processus ARMA(p, q) *stationnaire*, on a encore une fois le choix entre deux approches, la première étant la génération d'une longue série initialisée n'importe comment, suivie de l'amputation des premiers éléments. L'autre approche est de tirer les $p + q$ valeurs de l'initialisation de la loi stationnaire jointe des variables $y_1, \ldots, y_p$ et

$u_p, \ldots, u_{p-q+1}$. Les variances et les covariances de cette loi jointe peuvent être trouvées en résolvant des équations Yule-Walker complétées de quelques autres relations ; voir encore une fois le chapitre 13 du manuel de Davidson-MacKinnon (2004).

Pour la mise en œuvre, on fait d'abord l'initialisation des éléments pertinents de `y` et `u`. Écrivons $B(\mathrm{L})u_t = v_t$, de sorte que $A(\mathrm{L})y_t = v_t$. Comme pour un processus AR(p), l'initialisation des p premiers éléments de `y` fixe en même temps les p premiers éléments de `v`. On les obtient par la commande `gen v = conv(y,a)`, où `a` représente le polynôme auto-régressif A. Ensuite, on peut générer les éléments du bruit `u` après l'élément p. La prochaine étape est de générer les élements de `v` après l'élément p par la relation `v = conv(u,b)`, où `b` représente le polynôme moyenne mobile B, ce qui permet finalement de générer `y` par `y = invconv(v,a)`. La procédure entière pour un ARMA$(2, 1)$ est contenu dans le code suivant.

```
mat a = rowcat(1,-0.3,-0.4)
mat b = rowcat(1,0.6)
mat y = rowcat(0.2,0.4)
gen v = conv(y,a)
mat u = rowcat(0,0.9)
sample 3 12
gen u = random()
sample 1 12
gen vv = conv(u,b)
sample 3 12
gen v = vv
sample 1 12
gen y = invconv(v,a)
```

On vérifie que l'on a obtenu ce que l'on voulait par les commandes

```
sample 3 12
gen yy = -a(2)*lag(1,y)-a(3)*lag(2,y)+u+b(2)*lag(1,u)
print y yy
```

Les variables `y` et `yy` doivent être identiques.

EXERCICES:

Démontrez que la série `y` générée par le code ci-dessus reste inchangée si on change la valeur de la première composante de `u`.

La multiplication des polynômes est une opération commutative. Démontrez ce fait par l'évaluation des trois expressions

```
conv(u,invconv(b,a))
invconv(conv(u,b),a)
conv(invconv(u,a),b)
```

Processus à Mémoire Longue

Une série temporelle peut être définie par la relation formelle

$$(1 - \mathrm{L})^d y_t = u_t. \tag{9}$$

L'utilisation de l'opérateur de **différence première**, $1-\mathrm{L}$, élevé à une puissance non entière d, donne lieu à ce qu'on appelle l'**intégration fractionnaire**. Les processus notés $\mathrm{I}(d)$ sont caractérisés par un phénomène de **mémoire longue**, ce qui signifie qu'un choc subi par le processus à un instant donné a des conséquences longtemps après. Un processus $\mathrm{MA}(q)$, par exemple, n'a pas de mémoire longue, parce que toute influence de l'innovation u_t est perdue à partir de l'élément $t+q$. Une marche aléatoire, que l'on peut assimiler à un processus $\mathrm{I}(1)$, a une mémoire parfaite, parce que l'élement t est le cumul de tous les chocs passés. Un processus $\mathrm{AR}(p)$ a une mémoire « assez longue », dans le sens que l'influence de l'innovation t ne disparaît jamais complètement, mais elle s'estompe assez vite – on se rappelle qu'un $\mathrm{AR}(p)$ est équivalent à un $\mathrm{MA}(\infty)$ caractérisé par un polynôme infini dont les coefficients tendent assez rapidement vers 0.

Le théorème du binomial nous dit que, pour un réel ou un complexe x tel que $|x| < 1$,

$$(1-x)^d = 1 + \sum_{m=1}^{\infty} (-1)^m \frac{d(d-1)\dots(d-m+1)}{m!} x^m. \tag{10}$$

Bien que l'opérateur retard L ne soit ni un réel ni un complexe, on utilise la formule (10) avec $x = \mathrm{L}$ pour l'explicitation de la relation formelle (9), qui s'écrit

$$D(\mathrm{L}) y_t = u_t$$

pour un polynôme infini D dont les coefficients sont les coefficients des x^m dans (10). Pour un échantillon de taille n, on peut bien sûr tronquer le polynôme après le terme en L^n.

Comment faire générer ces coefficients à ***Ects*** ? L'astuce la plus efficace est de créer d'abord une série dont les éléments sont les rapports entre les coefficients successifs de (10). Le rapport entre les coefficients de l'élément m et l'élément $m+1$ est égal à $-(d-m)/(m+1)$. Une série dont les éléments sont les coefficients de (10) est ensuite générée par une application de la fonction `product`. Le code suivant fait le nécessaire pour la génération d'une série $\mathrm{I}(d)$.

```
sample 1 n
gen m = time(-1)
gen r = -lag(1,(d-m)/(m+1))
set r(1) = 1
gen f = product(r)
gen y = invconv(random(),f)
```

La théorie des séries temporelles montrent que les processus I(d) peuvent être stationnaires pour $d < 0.5$, mais que, si $d > 0.5$, le processus est forcément non stationnaire. Dans le deuxième cas, il n'y a pas d'initialisation privilégiée. Dans le premier, on voudrait pouvoir initialiser en utilisant la distribution stationnaire du processus. Mais, même si on disposait de l'expression analytique de cette distribution, on aurait à initialiser un processus AR(∞), ce qui demande un nombre infini de valeurs pour l'initialisation. On est donc conseillé de se servir de l'autre approche si on veut générer une réalisation du processus stationnaire, en générant un grand nombre d'éléments dont on ne garde que les derniers.

EXERCICES:

On peut supposer que la relation (9) est équivalente à

$$y_t = (1 - \mathrm{L})^{-d} u_t. \tag{11}$$

L'expression $(1 - \mathrm{L})^{-d}$ est réalisée par la formule (10) avec $-d$ à la place de d. Pour le même bruit `u`, générez deux séries, l'une par la méthode donnée par le code ci-dessus, l'autre sur la base de (11) en utilisant `conv` plutôt que `invconv`.

Générez plusieurs réalisations du processus I(d) pour des valeurs de d entre 0 et 1. Pour $d < 0.5$, on verra que les réalisations restent dans un voisinage de l'origine, mais, pour $d > 0.5$, elles ont tendance à s'éloigner de leur point de départ, mais moins rapidement qu'une marche aléatoire.

Sachant que le développement en série de la fonction exponentielle s'écrit

$$\exp x = 1 + \sum_{m=1}^{\infty} \frac{x^m}{m!},$$

dressez une table des valeurs de cette fonction pour les arguments $0, 0.1, \ldots, 1.9, 2$ évaluées par un nombre suffisant de termes du développement. Comparez les résultats avec les valeurs données par la fonction `exp`.

7. Paramètres Vectoriels

Il s'est avéré fastidieux d'avoir toujours à traiter tous les paramtres d'une estimation non linéaire comme des scalaires. Les commandes linéaires `ols` et `iv` permettent de regrouper les variables, et par conséquent aussi les paramtres, dans des matrices. Il serait agréable si une telle opération était possible dans les commandes non linéaires comme `nls` et `ml`.

Ects 4 permet effectivement de regrouper les paramtres d'une estimation dans un vecteur de paramtres, moyennant quelques précautions. Le fichier `nlsvec.ect` illustre comment faire. Voici son contenu :

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

```
gen X = colcat(1,x1,x2,x3)
mat beta = emptymatrix(4,1)
while cols(X)
   nls y = mat(X*beta)
      deriv beta = X
   end
   ols y X
   if (cols(X) = 1)
      mat X = emptymatrix()
   else
      mat X = X(1,100,1,cols(X)-1)
   end
end
quit
```

Ce que l'on fait ici est assez trivial, car l'estimation n'est pas réellement non linéaire. On verra tout à l'heure que les mêmes principes s'étendent aisément à des estimations vraiment non linéaires.

Dans la première ligne de la commande `nls`, on voit

```
nls y = mat(X*beta)
```

où `X` est la matrice des explicatives, et `beta` est un vecteur colonne, créé par `emptymatrix`, de dimensions 4×1. L'aspect nouveau ici est la *fonction* `mat` utilisée dans l'expression de la fonction de rgression du modèle. Normalement `mat` est une *commande*, et non une *fonction*. La syntaxe est simple. L'argument de la fonction est une expression algébrique, que l'on veut évaluer comme si on travaillait sous `mat`. On se rappelle que toutes les expressions trouvées dans une commande `nls` sont normalement évaluées selon les règles de `gen`. Ceci reste vrai, mais, sous `gen` uniquement, la fonction `mat` suspend temporairement les règles habituelles, en faveur de celles de `mat`.

Ce fait signifie que le produit `X*beta` sera interprété comme un vrai produit matriciel, selon la règle de `mat`, plutôt qu'un produit calculé élément par élément comme sous `gen`. Autrement dit, la valeur de l'expression `mat(X*beta)`, sous `gen`, est la même que la valeur sous `gen` de l'expression plus compliquée

```
iota*beta1+x1*beta2+x2*beta3+x3*beta4
```

où on note `beta`i, $i = 1, 2, 3, 4$, les composantes du vecteur `beta`.

La suite de la commande `nls` ne comporte qu'une seule ligne avant le mot clé `end` :

```
deriv beta = X
```

ce qui correspond à la réalité, parce que la dérivée de la fonction de rgression par rapport au *vecteur* `beta` est la *matrice* `X`.

Pour que cette syntaxe marche, il faut non seulement la fonction `mat`, mais aussi un changement des règles concernant les paramtres, afin de reconnaître des vecteurs de paramtres, là où ***Ects*** 3 exige des paramtres scalaires. Nous

avons vu plus haut qu'un tel changement à été fait pour les arguments des `procedures`. La question que doit se poser ***Ects*** 4 est la suivante : Comment savoir la dimension du vecteur `beta` ? Ici, on le voit clairement. Le vecteur est défini à l'avance, par `emptymatrix`, et sa dimension est donc visible. Dans le cas présent, on aurait pu se passer de la commande

```
mat beta = emptymatrix(4,1)
```

parce que la dérivée par rapport à `beta`, ici `X`, est une matrice de la forme 100×4, d'où on peut conclure que le nombre de paramtres à associer à cette dérivée est 4. Depuis la version 3, il n'est plus nécessaire que les paramtres à estimer soient définis à l'avance. S'ils ne sont pas définis, ***Ects*** leur affecte des valeurs nulles. Bien entendu, si on veut que la procédure itérative d'estimation ait un point de départ différent de 0, il faut définir au préalable une variable dont la valeur est ce point de départ. Mais ici, dans le contexte d'une estimation à l'apparence non lineáire mais réellement linéaire, le point de départ n'a pas d'importance.

Si la fonction de rgression n'est pas linéaire, ses dérivées dépendent forcément des paramtres à estimer. Dans ce cas, il devient nécessaire de définir ceux de ces paramtres qui sont des vecteurs. Sinon, ***Ects*** ne sait pas calculer les dérivées afin de compter le nombre de colonnes. La règle est la suivante : si un paramtre n'est pas trouvé dans la table de symboles, on affecte au symbole une valeur nulle *scalaire*. Si ceci ne convient pas, il faut alors définir le symbole à l'avance avec le bon nombre de colonnes.

Dans la suite du fichier `nlsvec.ect`, on entame une boucle, qui nous permet d'observer que la dimension du vecteur `beta` est en effet déterminée par celle de la matrice `X`. Une à une, les colonnes de cette matrice disparaissent, jusqu'à ce qu'il n'en reste plus. À ce stade, on sort de la boucle. Pour chaque itération, les calcules effectués par la commande `nls` sont vérifiés par une commande `ols`. Dans le fichier de sortie, on voit que la commande `nls` s'exécute chaque fois avec une seule itération. Ceci est dû à la linéarité de la fonction de rgression.

* * * *

Pour la cohérence logique, il faut une fonction `gen`, disponible sous `mat`, et qui suspend les règles de `mat` pendant l'évaluation de son argument en faveur de celles de `gen`. Cette fonction existe, parce que non seulement son existence est logique, mais aussi sa programmation est très simple. Ce que je ne sais pas encore est si la fonction aura une utilité quelconque dans la pratique.

* * * *

Le Modèle Logit

Un exemple plus sérieux de comment utiliser un vecteur de paramtres se trouve dans le fichier `logit.ect`. Le modèle logit est un modèle réellement non linéaire, de sorte qu'on ne peut pas estimer les paramtres du modèle

par une procédure linéaire. Ce fichier contient des estimations multiples du même modèle, afin d'illustrer les performances des différentes commandes qui effectuent une estimation par le maximum de vraisemblance. Vers la fin du fichier, avant le code que nous avons étudié dans la section sur les chaînes de caractères, on trouve le code suivant.

```
gen XX = colcat(1,x1,x2,x3)
mat beta = emptymatrix(4)
def e = exp(mat(XX*beta))
ml lhd
deriv beta = XX*(y-F)
end
```

Plus haut, on avait défini des macros nécessaires à l'exécution de ce code :

```
def F = e/(1+e)
def lhd = y*log(F)+(1-y)*log(1-F)
```

On voit que la macro `lhd` exprime une contribution à la log-vraisemblance d'un modèle logit. Comme on peut le voir, la macro `e` correspond à l'expression $\exp(\boldsymbol{X\beta})$. L'expression algébrique de la macro `F` est alors

$$\boldsymbol{F} = \frac{\exp(\boldsymbol{X\beta})}{1+\exp(\boldsymbol{X\beta})}.$$

Pour l'observation t, la composante t du vecteur $\boldsymbol{F}$ est la probabilité que la variable dépendante y_t soit égale à 1. La contribution ℓ_t faite par l'observation t à la log-vraisemblance est

$$y_t \log F_t + (1-y_t)\log(1-F_t),$$

ce qui signifie que, si $y_t = 1$, la contribution est le log de la probabilité que $y_t = 1$, et si $y_t = 0$, elle est le log de la probabilité que $y_t = 0$.

Le vecteur de dérivées partielles de la contribution ℓ_t, si $y_t = 1$, est le gradient par rapport au vecteur $\boldsymbol{\beta}$ de $\log F_t$, soit

$$\frac{\partial}{\partial\boldsymbol{\beta}} \log \frac{\exp(\boldsymbol{X\beta})}{1+\exp(\boldsymbol{X\beta})} = \frac{\boldsymbol{X}}{1+\exp(\boldsymbol{X\beta})},$$

et, si $y_t = 0$, on a le gradient de $\log(1-F_t$, soit

$$\frac{\partial}{\partial\boldsymbol{\beta}} \log \frac{1}{1+\exp(\boldsymbol{X\beta})} = -\frac{\boldsymbol{X}\exp(\boldsymbol{X\beta})}{1+\exp(\boldsymbol{X\beta})},$$

En fonction de y_t donc, le gradient de ℓ_t s'écrit

$$\frac{\boldsymbol{X}}{1+\exp(\boldsymbol{X\beta})}\big(y_t - (1-y_t)\exp(\boldsymbol{X\beta})\big)$$
$$= \frac{\boldsymbol{X}}{1+\exp(\boldsymbol{X\beta})}\big(y_t(1+\exp(\boldsymbol{X\beta})) - \exp(\boldsymbol{X\beta})\big) = \boldsymbol{X}(y_t - F_t),$$

et c'est exactement ce qui correspond à la ligne

```
deriv beta = XX*(y-F)
```

dans le code.

Cette fois-ci, on ne peut pas se passer de la commande

```
mat beta = emptymatrix(4)
```

parce que, sans la définition de `beta`, l'expression `XX*(y-F)` ne peut pas être calculée, étant donné que la macro `F` dépend de `beta`. On aurait pu donner à `beta` des valeurs initiales différentes de 0, mais ceci n'a pas été nécessaire. L'important est que ***Ects*** a pu évaluer `XX*(y-F)` et a trouvé une matrice à 4 colonnes. La dimension de `beta` est ainsi établie pour la procédure d'estimation. En faisant tourner le code du fichier `logit.ect`, on verra que les résultats sont les mêmes que ceux obtenus par les méthodes plus classiques où tous les paramtres sont scalaires.

Il faut signaler un défaut de cette première version d'***Ects*** 4. Même si les vecteurs de paramtres sont reconnus pour les dérivées premières, ils ne le sont pas encore pour les dérivées secondes. La conséquence en est que, pour les commandes comme `mlhess` et `gmmhess` qui se servent de dérivées secondes analytiques, il faut que tous les paramtres soient scalaires. Ce défaut sera rectifié prochainement, auquel moment ce paragraphe sera enlevé au manuel !

Chapitre 3
Dernières Remarques

Il est important de rappeler que la documentation contenue dans ce volume est incomplète. Elle ne constitue qu'une annexe à la documentation des deux premiers volumes, rédigés pour les versions 2 et 3 du logiciel. Elle ne contient probablement pas toutes les informations nécessaires à la bonne utilisation d'***Ecte*** 4, mais les lacunes doivent être comblées prochainement. À plus long terme, on aura une documentation complète en un seul volume (volumineux probablement !), mais il me semble qu'il est préférable d'attendre à ce qu'***Ecte*** 4 ait fait ses preuves avant de me lancer dans le travail non négligeable de rédaction de ce volume.

Un autre travail de longue haleine, déjà bien entamé, est la documentation du code source. Le code lui-même est ou sera bientôt disponible, sous la licence GPL, mais il ne contient pratiquement aucun commentaire, de sorte qu'il n'est vraiment pas très lisible. La documentation du code sera sous la forme « littéraire » préconisée par D. E. Knuth, où le code est imbriqué dans une discussion de son sens, de ses buts, des choix faits pour son efficacité, *etc*. Cette programmation littéraire permet de créer un document relativement joli contenant le code et sa documentation, et aussi le code même, celui que l'on soumet au compilateur. À mon avis, la valeur d'un grand projet comme ***Ecte*** est largement augmentée par des efforts littéraires de ce type, parce que le code devient vraiment ouvert. En effet, toute personne souhaitant le comprendre n'a qu'à lire la documentation, ce qui permettrait de modifier le code, de l'améliorer, et de l'utiliser dans d'autres contextes.

J'ai fait mon possible pour que le code soit aussi modulaire que possible. Il est souvent plus agréable de développer des expériences de simulation, par exemple, d'abord avec ***Ecte***, ensuite en C++. Le passage d'***Ecte*** au C++ peut être simplifié si on peut se servir directement dans son code C++ des fonctions développées pour ***Ecte***. Tout ce qui n'est pas relié directement aux choses spécifiques à ***Ecte***, c'est--dire tout ce qui peut s'appliquer plus généralement, comme les opérations matricielles, les manipulations d'expressions algébriques, les estimations linéaires et les optimisations de fonctions non linéaires, est contenu dans des librairies essentiellement indépendantes d'***Ecte***. Ces librairies peuvent être employées directement dans du code C++.

Je compte bientôt rédiger, de manière littéraire, un exemple d'un module qui, chargé par ***Ecte***, permettra de lancer des commandes comme

```
logit y c x1 x2 x3
```

qui auront l'effet de faire l'estimation d'un modèle logit avec une variable dépendante binaire `y` et les explicatives `c` (la constante), `x1`, `x2`, et `x3`. La syntaxe est identique à celle utilisée par une commande `ols`. Mon but est de montrer clairement les étapes de la création d'un module qui étendrait les fonctionnalités d'***Ects***. J'espère que beaucoup de ces modules seront créés, au fur et à mesure du développement de techniques nouvelles d'conomtrie. Je ne peux pas faire tout ça moi-même, mais, si d'autres programmeurs savent comment créer les modules, ils ou elles peuvent m'aider à faire d'***Ects*** un outil digne de prendre sa place parmi les meilleurs logiciels d'conomtrie.

Ce manuel sera sans doute modifié assez régulièrement. Si vous pensez que votre exemplaire n'est plus actuel, connectez-vous à mon site

```
http://russell.vcharite.univ-mrs.fr/ects4
```

pour avoir la version la plus récente de la documentation, et du logiciel. Il y a aussi un site nouveau, créé par Pierre-Henri Bono, qui se trouve à

```
http://www.vcharite.univ-mrs.fr/ects
```

où on peut trouver non seulement le code mais aussi un forum où les utilisateurs d'***Ects*** peuvent s'exprimer.

GNU Free Documentation Licence

Version 1.2, November 2002

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

0. Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.
This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.
We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **"Document"**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as **"you"**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.
A **"Modified Version"** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.
A **"Secondary Section"** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of

mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.
The **"Invariant Sections"** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.
The **"Cover Texts"** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.
A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **"Opaque"**.
Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.
The **"Title Page"** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.
A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **"Acknowledgements"**, **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.
The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.
If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.
If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.
It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

[A.] Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

[B.] List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

[C.] State on the Title page the name of the publisher of the Modified Version, as the publisher.

[D.] Preserve all the copyright notices of the Document.

[E.] Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

[F.] Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

[G.] Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
[H.] Include an unaltered copy of this License.
[I.] Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
[J.] Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
[K.] For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
[L.] Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
[M.] Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
[N.] Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
[O.] Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant

Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.
In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.
You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.
If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.
If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.
Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

11. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

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

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.
If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

12. Loki Library Copyright and Permission

13. Boost Libraries Copyright and Permission

14. Cephes Mathematical Library Copyright

Index Gnral

Index *Ects*