

STATA POUR LES NULS

Olivier Cadot
Juin 2012

Contents

Comment mal démarrer : leçon un.....	3
Do- et log-files	3
Garbage in, garbage out : Formatage de l'output	6
Manipuler des données : comment faire du faux avec du vrai	6
Importer des données	7
Trier les variables et les maltraiter.....	9
Générer des expressions vides de sens.....	11
Moyenne, écart-type, max et min	11
Compter les sous-catégories à l'intérieur de catégories.....	12
Variables aléatoires	12
Variables en différences et retardées (mentales)	16
Variables muettes, aveugles et idiotes	16
Variables en string	17
Mettre le string et l'enlever.....	17
Manipuler des variables en string	18
Fusionner des fichiers	20
Fusion « horizontale ».....	20
Fusion « verticale ».....	22
Variables avec indices (boucles).....	23
Principe général	23
Itérations	24
Boucles sur des observations	25
Matrices et vecteurs	25
Mettre une matrice en vecteur-colonne.....	25
Mettre un vecteur-colonne en matrice	27
Multiplier une matrice par un vecteur.....	28
Cylindrer un panel.....	29
Un peu de programmation pour les ado.....	30
Programme	30
Fichiers ado	30
If/else.....	31
While.....	33
Estimation : quelques bidouillages	33
Sauvegarder des résultats, même quand ils sont absurdes	33
Sauvegarder des statistiques descriptives	33

Sauvegarder valeurs prédites et résidus	33
Sauvegarder des coefficients absurdes.....	33
Estimation	36
OLS and WLS	36
Panels	36
Sure	37
2sls	37
GMM.....	38
Variables dépendantes limitées	39
Switching reg	42
Propensity score matching	42
Analyse de survie	45
Graphiques	49
Graphiques avec courbes ou barres.....	50
Nuages de points	51
Regressions non paramétriques (« smoother reg »)	55
Enquêtes sur les scènes de ménages	56
Statistiques descriptives et manipulation de données	56
Moyennes, totaux et corrélations	56
Calculer des indices d'inégalité	58
Densités	58
Effet de changements de prix par tranche de revenu	59
Estimation sur des enquêtes	62
Modèles de sélection.....	62
Quelques trucs en Mata.....	64

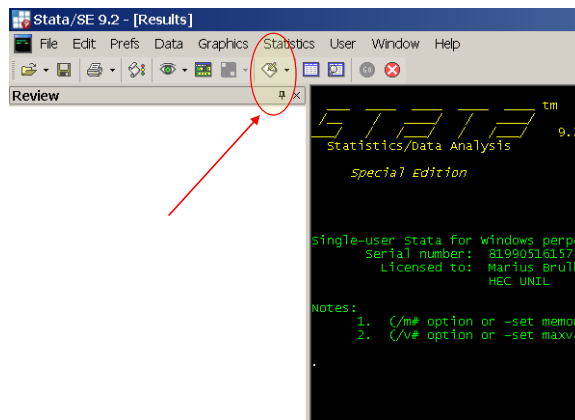
Comment mal démarrer : leçon un

« I think there is a world market for about five computers. »

Thomas J. Watson
Chairman of the Board, IBM.¹

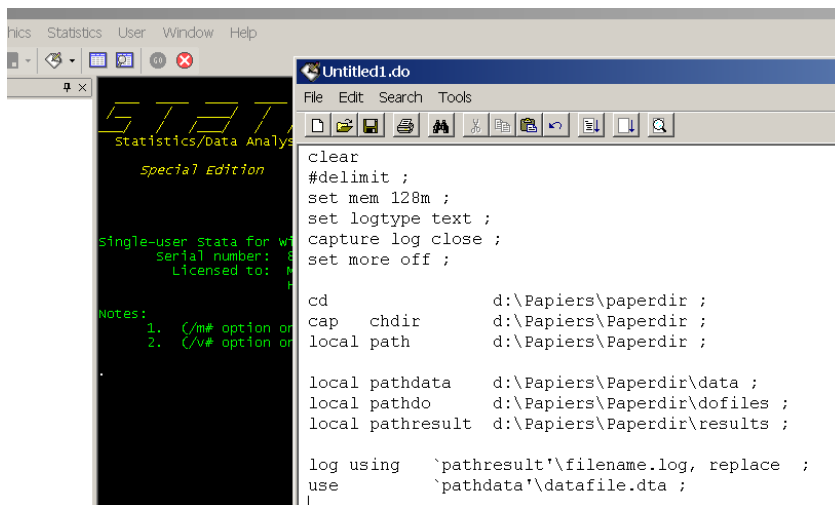
Do- et log-files

Dès les premières manipulations malheureuses, il faut regrouper toutes les erreurs, y compris celles qui mettent les données définitivement cul par-dessus tête, dans un (don't)-do-file avec le *do-file editor* de Stata qui est ici :



A la fin, le fichier en question (ou la séquence de fichiers) doit prendre les données de base et garantir la traçabilité totale de toutes les âneries que vous avez faites. C'est seulement ainsi que vous saurez pourquoi vous avez des résultats absurdes depuis six mois. Les premières commandes du do-file devraient être

¹ <http://www.gocreate.com/QuotAmaze/qlun.htm>. Cité dans le manuel de reference de Shazam v.8, McGraw-Hill, 1997. Il faut préciser que la citation est de 1943.



Explication des commandes de départ :

`clear` vide la mémoire

`#delimit ;` définit le symbole que Stata comprendra comme la fin d'une commande ; par défaut, c'est un retour à la ligne (qu'on retrouve avec `#delimit cr`); utile si on a des commandes très longues et qu'on souhaite revenir à la ligne pour pouvoir tout lire à l'écran ; par contre, il ne faudra plus l'oublier à la fin de *toutes* les commandes. Si on choisit cette syntaxe, les lignes de commentaires doivent être écrites comme

```
* commentaire ;
```

Alternativement, si on veut écrire sans point-virgule, les commentaires s'écrivent

```
// commentaire
```

et les commandes longues peuvent être « cassées » en plusieurs lignes avec `///` :

```
début de la commande ///
suite de la commande
commande suivante
```

Attention il faut impérativement un espace avant `///`.

Le bloc de commandes

```
cd          d:\Papiers\paperdir ;
cap  chdir  d:\Papiers\Paperdir ;
local path  d:\Papiers\Paperdir ;
```

indique à Stata de se mettre sur le répertoire `d:\Papiers\paperdir` et de s'en souvenir. La commande `set mem` est inutile à partir de Stata 12. Le bloc de commandes

```
local pathdata d:\Papiers\Paperdir\data ;
```

```
local pathdo      d:\Papiers\Paperdir\dofiles ;
local pathresult  d:\Papiers\Paperdir\results ;
```

met dans sa mémoire des sentiers qui mènent vers trois répertoires (qu'il aura bien sûr fallu créer avant dans le file manager) : data, où on met les données, dofiles où on met les programmes, et results où on met les résultats. Ce départ permet de garder la suite dans un semblant d'ordre. On pourra alors se référer à ces sentiers sous forme abrégée ; par exemple, on appelle le fichier données par la commande

```
use `pathdata'\datafile.dta ;
```

au lieu de mettre tout le sentier. Ne pas oublier de mettre les guillemets comme ils sont (noter le sens !).

`set logtype text` ; enregistre le fichier log sous format texte au lieu d'un format Stata (par défaut)

`capture log close` ; permet au programme de s'arrêter sans bug même si, par exemple, on l'interrompt par `/*` sans refermer l'interruption par `*/` avant « log close » (qui signale la fin). en fait, `capture` est également utile devant d'autres commandes sensibles pour éviter des bugs. Ceci dit ça bugge quand même ; par exemple, chaque fois qu'un do-file est interrompu par une erreur, le log ne se ferme pas et on a la fois suivante un message d'erreur « log file already open ».

`set more off` ; permet a tout le programme de se dérouler sur l'écran

`log using filename.log, replace` ; crée un fichier log nommé filename ; si ce nom de fichier existe déjà **replace** remplace l'ancien par ce nouveau fichier log

`log close` ; ferme le fichier log;

La commande `save datafile1.dta` est très importante : elle sauvegarde le fichier-données (.dta) modifié par le programme sous un *autre* nom que le fichier initial, ce qui permet de laisser ce dernier intouché. Sinon on altère le fichier initial de façon permanente, ce qui est en général un désastre.

De façon générale, les guillemets (comme dans `cd "c:\path\directory"`) sont obligatoires quand les noms spécifiés ne sont pas liés en un seul mot ; par exemple, Stata comprend `use "le nom que je veux.dta"` mais pas `use le nom que je veux.dta`.

Si on a fait une série de commandes sans do-file (ce qui une mauvaise idée, très souvent) et que tout d'un coup on voudrait pouvoir les reproduire, on peut créer un log file après coup avec

```
log using filename.log
# review x
log close
```

ce qui va imprimer, dans le nouveau log file créé, les x dernières commandes (autant que l'on veut).

Sur un Mac, la syntaxe pour les sentiers n'est pas tout à fait la même puisque le système d'opération est fondé sur Unix et non sur le DOS. Pour ouvrir un fichier, on tape :

```
use "/Users/admin/Documents/subdirectory/filename.dta"
```

Les commandes s'adaptent facilement :

```
cd /Users/admin/Documents/subdirectory  
local pathdata /Users/admin/Documents/subdirectory/data
```

etc. Les gros fichiers peuvent être compressés par la commande

```
compress x y z
```

Il faut mettre toute la liste de variables et on peut réduire d'un tiers au moins la taille du fichier sans perte d'information bien sûr.

Garbage in, garbage out : Formatage de l'output

Si l'on veut que les variables apparaissent dans l'output sous un autre nom (par exemple sous un nom plus simple) que celui donné par le dingue qui a fait les premières manipulations de données, et ceci tout en préservant le nom original des variables, on utilise la commande `label` comme ceci :

```
label variable lnparcellesexpl land ;
```

Manipuler des données : comment faire du faux avec du vrai

« When the President does it, that means it is not illegal. »
Richard Nixon²

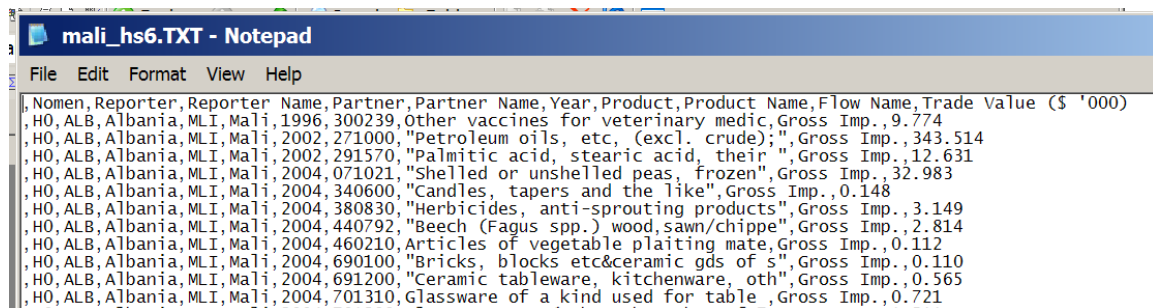
Un petit préambule. Pourquoi manipuler les données en Stata et pas en Excel ? La raison est simple : pas mal des commandes que l'on va voir ci-dessous existent aussi en Excel et sont certes quelquefois plus simples (si on arrive à les trouver), mais par contre on perd vite le fil de ce que l'on a fait subir aux données avant de passer à l'estimation, et c'est parfois là que se cachent soit les quelques erreurs à l'origine de résultats grotesques soit, au contraire, les mauvais traitements infligés aux chiffres pour obtenir le résultat désiré.

² Citation encore trouvée dans le manuel de Shazam, p. 51.

Avec Stata, on peut garder la trace de toutes les manipulations dans le do-file. Celui-ci doit contenir toutes les commandes permettant de passer du fichier-données brut à celui qui est prêt à l'estimation. Il est alors facile de retrouver l'erreur qui tue ou bien de vérifier ce que les chiffres ont subi entre les mains du bourreau avant d'avouer.

Importer des données

On peut copier-coller à partir d'excel, mais c'est trop facile. Supposons que l'on veuille importer un fichier-données avec la tête suivante :



```
,Nomen,Reporter,Partner,Partner Name,Year,Product,Product Name,Flow Name,Trade Value ($ '000)
,H0,ALB,Albania,MLI,Mali,1996,300239,other vaccines for veterinary medic,Gross Imp.,9.774
,H0,ALB,Albania,MLI,Mali,2002,271000,"Petroleum oils, etc, (excl. crude);",Gross Imp.,343.514
,H0,ALB,Albania,MLI,Mali,2002,291570,"Palmitic acid, stearic acid, their",Gross Imp.,12.631
,H0,ALB,Albania,MLI,Mali,2004,071021,"Shelled or unshelled peas, frozen",Gross Imp.,32.983
,H0,ALB,Albania,MLI,Mali,2004,340600,"Candles, tapers and the like",Gross Imp.,0.148
,H0,ALB,Albania,MLI,Mali,2004,380830,"Herbicides, anti-sprouting products",Gross Imp.,3.149
,H0,ALB,Albania,MLI,Mali,2004,440792,"Beech (Fagus spp.) wood,sawn/chippe",Gross Imp.,2.814
,H0,ALB,Albania,MLI,Mali,2004,460210,Articles of vegetable plaiting mate,Gross Imp.,0.112
,H0,ALB,Albania,MLI,Mali,2004,690100,"Bricks, blocks etc&ceramic gds of s",Gross Imp.,0.110
,H0,ALB,Albania,MLI,Mali,2004,691200,"Ceramic tableware, kitchenware, oth",Gross Imp.,0.565
,H0,ALB,Albania,MLI,Mali,2004,701310,Glassware of a kind used for table,Gross Imp.,0.721
```

Horrible à voir, n'est-il pas. Mais tout va se passer dans la bonne humeur. On commence par

```
insheet using mali_hs6.TXT
```

et le fichier vient tout gentiment. Puis on se débarrasse de toutes les cochonneries dont on n'a pas besoin et on renomme la variable tradevalue000 « tv » ce qui, vous en conviendrez, est plus commode :

```
keep reporter year product tradevalue000 product
rename tradevalue000 tv
```

ce qui donne ceci :

Nettement plus clair. Par contre, observez le monstre tapi dans l'ombre : à la ligne 11544, la variable « tv » a une virgule pour les milliers. Cette virgule va profondément perturber stata et il faut à tout prix s'en débarrasser. A la bonne heure ! Rien de plus facile. On tape

```
split tv, parse(,)  
egen export = concat(tv1 tv2)
```

et le tour est joué. Il n'y a plus de virgule (on a « cassé » la variable autour de la virgule puis on a réuni les deux parties, nommées automatiquement tv1 et tv2 par stata, sans la virgule). Notez d'ailleurs que l'interdiction des virgules dans les nombres est une preuve (de plus) du complot anglo-saxon pour tuer la culture française.

Pour éviter de passer par excel qui n'admet que 65'000 observations et met automatiquement les variables chiffrées (genre classification *hs*) en mode numérique, ce qui élimine inopportunistement les zéros du début, on peut télécharger les données en spécifiant qu'on les veut séparées par des espaces pour pouvoir utiliser *infile* au lieu de *insheet* (qui n'admet pas qu'on spécifie un format string pour certaines variables). Il faut commencer par sauvegarder le fichier données en ASCII (en lui donnant l'extension *.txt*). Si la première variable est « *hs8* » par exemple et doit être en string et la deuxième est « *value* », la commande est alors

```
Infile str10 hs8 value using path/filename.txt.
```

On note que les séries de quantités physiques doivent être téléchargées avec un délimiteur (tab, point-virgule) mais surtout pas des espaces car la définition des unités est quelquefois formée de plusieurs mots, ce qui produit un galimatias.

Pour contrôler la façon dont stata formate les variables dans le data editor, on tape quelque chose comme

```
format firm_name %15s
```

pour les string, où % contrôle le formatage, 15 donne le nombre d'espaces dans la colonne (plus il y en a plus elle est large), un moins (optionnel) aligne les entrées à gauche dans la colonne, et le s est pour string. Voir aussi la section « Mettre le string et l'enlever ». Si c'est des données numériques, on tape

```
format var %11.3f
```

où 11 donne le nombre de chiffres à gauche de la virgule, 3 donne le nombre de décimales, et f impose la notation décimale (par opposition à la notation scientifique).

Trier les variables et les maltraiter

Les maniaco-dépressifs à tendance paranoïaque désirent parfois voir les variables apparaître sur l'écran dans un certain ordre, de gauche à droite. C'est totalement futile mais si c'est votre cas, ne vous privez pas : il suffit de taper

```
order x y z
```

On trie les variables (verticalement, en fonction de leurs observations) par `sort` ou `gsort`. La première ne peut les trier que par ordre ascendant, alors que la seconde peut les trier par ordre ascendant ou descendant. Pour trier une variable x par ordre *descendant*, on ajoute un signe «-» devant la variable, par exemple

```
gsort -x
```

Pour générer une variable t indexant les observations (par exemple le temps), la commande est

```
gen t=_n
```

Pour modifier des valeurs ou des groupes de valeurs dans une variable, on utilise la commande `recode`. Par exemple,

```
recode x .=0  
recode y 1/12 = 1
```

changent toutes les valeurs manquantes dans x (`.`) en 0 et toutes les valeurs de y comprises entre 1 et 12 en 1, respectivement. Cela fonctionne aussi avec un `if`.

Attention à distinguer les commandes `recode` et `replace`. `recode x .=value` s'emploie si on veut remplacer x par une constante (ici *value*) quand x est égale à une certaine valeur (ici une valeur manquante). La commande

```
replace x=log(x) if x!=0
```

s'emploie si on veut remplacer la variable x par une variable (ici son log), éventuellement quand x satisfait une condition (ici, « n'est pas égale à 0 »). Noter la syntaxe légèrement différente. Pour renommer simplement une variable, c'est

```
rename year81 yr81
```

Si l'on veut additionner deux variables, x et y , qui ont des valeurs manquantes, la commande

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

va générer une nouvelle variable qui aura une valeur manquante pour chaque observation où soit x soit y a une valeur manquante. Pour éviter cela, utiliser la commande `rsum` avec `egen` (extensions plus élaborées de la commande `gen`) :

```
egen z = rsum(x y)
```

qui traitera les valeurs manquantes comme des zéros. (On peut utiliser des commandes similaires pour d'autres fonctions comme `rmean`, `rmax`, `rmin` ; toutes avec `egen` et toutes sans virgules dans la parenthèse). Si on veut faire la somme des observations d'une variable et non pas une somme de variables, la commande est:

```
egen sumx = sum(x)
```

Pour faire des manipulations sur les observations d'une variable x , les suffixes sont `[_n]`, `[_n-1]`, etc. Supposons que l'on veuille vérifier si un code (disons `hs8`) n'est pas répété (c'est-à-dire qu'on n'a pas deux entrées pour le même code). On compte ces doublons possibles avec la commande

```
count if hs8==hs8[_n-1]
```

Sinon, l'élimination des doublons complets (deux observations ayant les mêmes valeurs pour toutes les variables dans la base de données) la commande est

```
duplicate drop
```

Quand il n'y a pas assez d'erreurs dans les données, on peut en fabriquer en interpolant et en extrapolant linéairement avec la commande `ipolate`:

```
ipolate x year, gen(new_x)
```

L'option `gen` (obligatoire) génère une nouvelle variable au lieu d'écraser l'ancienne (non interpolée) ce qui permet de vérifier si le résultat a du sens ou pas. En ajoutant l'option `epolate`, on interpole et on extrapole :

```
ipolate x year, gen(new_x) epolate
```

En panel avec une variable d'individus, disons `individual`, on ajoute `by individual`

```
by individual : ipolate x year, gen(new_x) epolate
```

Les statistiques descriptives sont générées par `sum x` («sum» pour summarize, pas somme). Pour leur sauvegarde voir la section Estimation/Sauvegarder des résultats. On peut utiliser aussi `codebook` pour avoir une idée plus complète de la distribution d'une variable ; en particulier pour savoir combien de valeurs distinctes elle prend.

Générer des expressions vides de sens

Moyenne, écart-type, max et min

Pour créer des nouvelles variables (constantes) égales respectivement à la moyenne, la médiane, l'écart-type, le minimum ou le maximum d'une variable existante, les commandes sont

```
egen x_bar = mean(x)
egen x_med = pctlile(x)
egen x_sd = sd(x)
egen maxy = max(y)
egen miny = min(y)
```

Ne pas oublier qu'il s'agit d'un "egen" et non d'un "gen".

Si l'on a n variables, x_1 à x_n , (dont on va appeler l'énumération sans virgule *varlist*) et que l'on veut exporter la matrice des corrélations, voici la séquence de commandes :

```
corr varlist
mat c = r(C)
svmat c
outsheet varlist using filename
```

ce qui donne un fichier .out qui peut être importé en Excel.

Max de deux variables (i.e. la plus grande des deux)

Pour créer une expression du style $y = \max\{x, c\}$ où c est une constante, la commande est

```
gen y=x if x>=c
recode y . =c
```

La première ligne crée des valeurs manquantes dans y quand $x < c$. La deuxième remplace ces valeurs manquantes par c . Attention, cependant : si la variable x a déjà elle-même des valeurs manquantes, celles-ci seront codées par Stata comme 999 dans sa petite tête et vont alors être interprétées comme de grandes valeurs de y et rendues égales à la constante. Le problème est alors de préserver les valeurs manquantes comme telles, par exemple en adoptant une convention ad hoc:

```
recode x . = -989898
gen y=x if x>=c & x != -989898
recode y . =c
```

Compter les sous-catégories à l'intérieur de catégories

Supposons que l'on ait une catégorie large x , à l'intérieur de laquelle on a une sous-catégorie y , et que l'on veuille avoir une nouvelle variable indiquant, pour chaque x , combien il y a de valeurs différentes de y (par exemple, combien de produits par firme). Il y a plusieurs façons de le faire, mais une façon simple consiste à compter les changements de valeur de y (avec un "by x ")

```
bysort x: egen nbry = count(y == y[_n-1])
```

Plus compliqué: supposons que l'on ait deux sous-catégories, y et z , variant à l'intérieur de x (par exemple des produits et des destinations). On veut avoir maintenant deux variables caractérisant chaque x : combien de valeurs différentes de y et combien de valeurs différentes de z .

Pour y , on fait

```
* Number of y per x

sort x y
by x: gen newy = (y != y[_n-1])
by x: egen nbry = sum(newy)
```

Pour z , c'est symétrique:

```
* Number of z per x

sort x z
by x: gen newz = (z != z[_n-1])
by x: egen nbrz = sum(newz)
```

Variables aléatoires

Pour générer une variable aléatoire d'une distribution uniforme, la commande est

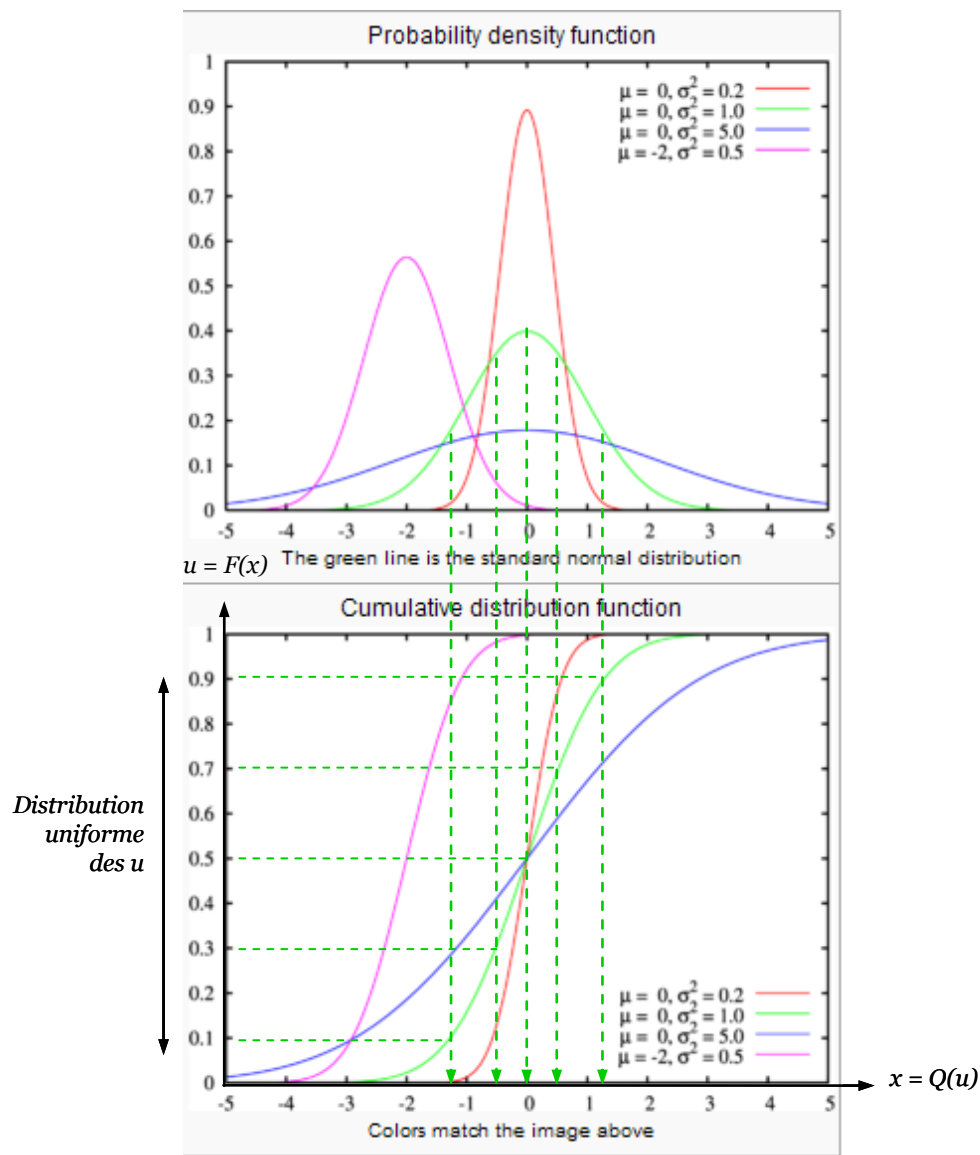
```
gen x=uniform()
```

pour une variable x avec une distribution uniforme sur un intervalle unitaire. `set seed` donne la valeur avec laquelle Stata va commencer l'algorithme de génération de nombres pseudo-aléatoires. Par défaut, Stata prend ce chiffre sur la montre de l'ordinateur ; `set seed` est donc nécessaire seulement si l'on veut pouvoir reproduire *exactement* les résultats.

Stata peut générer des variables aléatoires de distributions autre que la distribution uniforme. Pour cela, il utilise la fonction quantile qui est l'inverse de la distribution cumulative : si $F(x) = u$ alors $u = Q(x)$. Pour générer une variable tirée d'une distribution normale, Stata génère d'abord une variable uniforme u , puis génère x avec la fonction Q , comme le montre la Figure 1 dans laquelle les u sont 0.1, 0.3,..., 0.9 et les x générés

graphiquement par la courbe $F(\cdot)$. On voit dans le graphique du dessus que leur fréquence correspond (bien logiquement) à la densité normale.

Figure 1
Génération d'une variable normale par Stata



Les commandes pour générer cent observations d'une variable tirée d'une distribution standard normale centrée sur zéro sont donc

```
set obs 100
gen u=uniform()
gen x=invnorm(u)
```

où `invnorm` est la fonction quantile gaussienne ; ou bien, tout simplement,

```
gen x=invnorm(uniform())
```

Stata peut également générer des variables aléatoires tirées d'autres distributions (gamma etc.). Par exemple pour générer une variable aléatoire non-négative tirée d'une distribution Gamma avec paramètre 1, c'est

```
gen u=uniform()
gen x=invgammap(1,u)
```

Pour les distributions qui ne sont pas pré-programmées, on peut rentrer directement la forme algébrique de la fonction quantile (quand on la connaît) dans la troisième ligne. Par exemple, pour générer les dates de 100 événements tirés d'un processus dans lequel le temps d'attente entre deux événements suit une distribution de Weibull avec paramètre $p = 3$, on utilise la fonction quantile de Weibull $Q(u) = [-\ln(1-u)]^{1/p}$ pour écrire

```
set obs 100
gen u=uniform()
gen t=(-ln(1-u))^(1/3)
```

où t la date aléatoire de chacun des 100 événements observés.

Pour générer deux variables tirées d'une distribution gaussienne bivariée avec un coefficient de corrélation de son choix, la commande est

```
matrix C = (1.0,      ///
            0.5,  1.0)
drawnorm u v, corr(C) cstorage(lower) seed(1.0)
```

La matrice C donne les corrélations sous la forme d'une matrice triangulaire inférieure (ici le coefficient de corrélation est 0.5), ce qui est indiqué à Stata par l'option `cstorage(lower)` ; et la commande optionnelle `seed(1.0)` permet de générer le même vecteur de variables aléatoires à chaque tirage.

Les excités de la gravité (qui se reconnaîtront) peuvent générer une variable aléatoire de type « *zero-inflated Poisson* », par exemple pour faire des simulations, avec la série de commandes suivante³ :

```
clear
set obs 1000
gen x = uniform()
gen u = x
local lambda = 10
local p0 = .4
local cp = 0
local n = 0
while `cp'<=.99999 {
    local p = `p0'*(`n'==0) + ///
    (1-`p0')*exp(-1*`lambda')*(`lambda'^`n')/exp(lnfactorial(`n'))
    local cp= `cp'+`p'
    local pc = `cp'-`p'
```

³ Source: http://www.ats.ucla.edu/stat/stata/code/discrete_rv_v2.htm

```

        quietly recode x (`pcp'/'cp' = `n')
        local n = `n' + 1
    }

```

Variables en différences et retardées (mentales)

On génère les premières différences d'une variable x par la commande `gen dx = D.x` et des lags par la commande `gen lx = L.x`. Mais d'abord il faut déclarer l'échantillon soit comme une série temporelle par la commande

```
tsset year
```

(si `year` est la variable qui indice le temps) soit comme un panel par la commande

```
xtset ind year
```

où `ind` est la variable marquant les individus (toujours dans cet ordre). Attention quand les individus sont marqués par des strings Stata va couiner. Pour le calmer il va falloir taper

```

egen newind = group(ind)
xtset newind year
sort newind year
gen dx = D.x
gen lx = L.x

```

c'est-à-dire qu'il n'y a plus besoin de lui dire « `by ind` » pour le `D.x` (ou pour le `L.x`) il va comprendre tout seul dans sa petite tête.

Variables muettes, aveugles et idiots

Pour générer une variable muette égale à 1 quand *flowname* est *Export* (et 0 quand *flowname* n'est pas *Export*), la commande est

```
gen export=(flowname=="Export")
```

Attention, Stata distingue les majuscules. Si la vraie variable est « `export` » et non « `Export` » la commande renverra un message d'erreur.

Pour générer une variable muette qu'on veut appeler *dummy* à partir d'une variable numérique x (égale à 1 quand x est égale à *value1* ou⁴ *value2* et 0 sinon) la commande est

```
gen dummy=(x==value1 | x==value2)
```

⁴ Le signe « `|` » n'est pas toujours présent sur les claviers. On peut le copier-coller depuis l'aide de Stata : « `help operator` » ou « `search op_logical` »

Pour générer plusieurs variables muettes à partir d'une variable continue,

```
gen dummy1=(x<=value1)
gen dummy2=(x>value1 & x<=value2)
gen dummy3=(x>value2 & x != .)
```

Il faut noter la condition dans la troisième ligne. Stata code les valeurs manquantes comme 9999 ou quelque chose comme cela, et il faut donc éviter de les attraper par erreur. On vérifie que les variables muettes correspondent bien à ce que l'on veut et que les valeurs manquantes sont préservées avec

```
tab x dummy3, missing
```

Pour créer un vecteur de variables muettes à partir d'une variable numérique servant d'identifiant (disons *countrycode*) allant de 1 à *n*, la commande de base est

```
tab countrycode, gen(country)
```

et Stata crée *n* variables muettes qu'il appellera *country1-country_n*⁵. On note que cette commande fonctionne aussi lorsque l'on a une liste de codes-pays avec des trous : avec trois pays dont les codes sont 21, 47, 803, la commande `tab/gen` va donner trois dummies *country1-country3*. Pour effacer toutes ces nouvelles variables, il suffit de taper

```
drop country1-countryn
```

La commande `xi` permet de générer facilement des variables muettes à partir de catégories. Supposons que l'on veuille régresser *y* sur *x*, *z* et un vecteur de variables muettes codant les régions. La syntaxe est

```
xi: reg y x z i.location
```

Variables en string

Mettre le string et l'enlever

Pour changer une variable numérique en une variable alphanumérique⁶ (« string »), la commande est

```
tostring x, replace
```

Si on veut se compliquer la vie, on peut aussi taper

⁵ Par exemple, *country4* vaudra 1 pour toutes les observations qui ont la même (quatrième) valeur dans *countrycode*, 0 sinon.

⁶ Les strings sont rouges dans le Data Editor.

```
gen str8 newvar=string(oldvar,"8.0f")
```

où `str8` indique combien de chiffres (ici, huit) la nouvelle variable (ici `newvar`) doit avoir, au maximum. Si la variable initiale (`oldvar`) contenait des nombres à plus de huit chiffres, Stata ne gardera que les huit premiers d'entre eux (à partir de la gauche). Si on ne met pas l'option `%8.0f`, à partir de huit chiffres ce crétin de Stata met même les strings sous forme exponentielle et ça fait du galimatia par exemple quand on agrège. Par contre des fois il n'aime pas cette option, à voir.

Pour changer une variable alphanumérique en numérique la commande est

```
destring oldvar, replace
```

et pour générer sa sœur jumelle en version numérique (noter les parenthèses) :

```
destring oldvar, gen(newvar)
```

Souvent dans les données brutes il y a un mélange de données chiffrées et de cochonneries mal tapées style XXX. Pour transformer ces codes-lettres en valeurs manquantes sans s'en soucier, on utilise l'option « force » :

```
destring oldvar, replace force
```

mais attention, on ne voit alors pas ce qu'on fait... On peut faire des dégâts sans s'en rendre compte. Plus prudent, quand on a repéré un certain type de cochonneries à nettoyer, on peut spécifier l'option « ignore XXX ».

Manipuler des variables en string

Pour changer la valeur d'une variable alphanumérique pour certaines observations, la commande `recode` ne fonctionne pas : il faut employer `replace` et ne pas oublier les guillemets. Exemple :

```
replace reporter="Bahamas, The" if reporter=="Bahamas"
```

Supposons qu'une question d'enquête (disons la question codée comme variable `q85`) comprenne six sous-questions, chacune admettant une réponse de 1 (tout à fait d'accord) à 4 (pas du tout d'accord). Les réponses aux six sous-questions sont codées dans une seule variable en format string 6, style "412232". Le problème est d'extraire la réponse à la sous-question 4, qu'on appelle `newvar`. La commande est alors

```
gen str1 newvar=substr(q85,4,1)
```

où l'argument « 4 » de la commande `substring` est la place du chiffre à garder en partant de la gauche et l'argument « 1 » est le nombre de chiffres à garder. Si on voulait garder le quatrième et le cinquième chiffres et pas seulement le quatrième, la commande serait

```
gen str2 newvar=substr(q85,4,2)
```

Supposons maintenant que l'on s'intéresse à la fin d'un string et pas à son début. Il y a alors une ruse. Le principe est le même : le premier argument entre parenthèses indique où il faut commencer et le deuxième le nombre de caractères à garder. Mais si le premier argument est négatif, stata compte en partant de la gauche et garde le nombre de caractères donnés par le deuxième argument *en allant vers la droite*. Par exemple si *dutyrate* est égal à 1.23%, avec

```
gen str1 newvar=substr(dutyrate,-1,1)
```

stata démarre au dernier caractère de *dutyrate* et le garde, ce qui donne le % ; avec

```
gen str1 newvar=substr(dutyrate,-2,1)
```

stata démarre à l'avant dernier (le 3) et ne garde que celui-là ; avec

```
gen str2 newvar=substr(dutyrate,-2,2)
```

stata démarre encore à l'avant dernier et en garde deux, ce qui donne « 3% ».

Supposons qu'on ait trois variables alphanumériques : *reporter*, *partner* et *flowname*, cette dernière étant soit *Import* soit *Export*. Les deux premières sont des codes de pays à trois lettres. On peut générer une nouvelle variable égale à *reporter* quand *flowname* est *Export* et à *partner* quand *flowname* est *Import* par

```
gen str3 exporter1=reporter if flowname=="Export"  
gen str3 exporter2=partner if flowname=="Import"
```

Pour écrire des commandes conditionnelles sur la longueur du string, la fonction est `length()`. Par exemple si on veut ajouter un 0 au début d'un string quand il a deux caractères (par exemple parce que le 0 a disparu quand un code a été enregistré en numérique) la commande serait

```
gen str1 zero="0" ;  
gen str3 newcode = zero + oldcode if length(oldcode)==2 ;  
replace newcode oldcode if length(oldcode)!=2 ;  
drop oldcode ;
```

Avec exactement le même exemple mais en voulant ajouter deux 0, la deuxième ligne serait

```
gen str3 newcode = zero + zero + oldcode if length(oldcode)==2 ;
```

Supposons que l'on ait un ensemble de variables en string (des noms de produits par exemple) chacun revenant plusieurs fois dans l'échantillon (pour chaque ménage par exemple) et que l'on veuille créer un indice pour ces produits allant de 1 à n. Une façon

de le faire est la suivante : on garde seulement la variable nom de produit et on la sauvegarde dans un fichier à part. Dans ce fichier, on crée une variable indiquant les observations, et on collapse cette variable par le nom du produit. Puis on la supprime et on la remplace par une nouvelle variable indiquant les observations, qui va maintenant aussi indiquer les produits de 1 à n. On sauvegarde et on fusionne avec le fichier original :

```
keep product_name
gen n=_n
collapse n, by(product_name)
drop n
gen prod_id=_n
sort product_name
save products, replace

use original_file, clear
sort product_name
merge products using product_name
```

et voilà.

Fusionner des fichiers

On ne le dira jamais assez : les grands désastres commencent souvent par un merge mal fait. Regardez cinq fois le résultat du merge de la façon la plus soupçonneuse. Vous gagnerez du temps !

Fusion « horizontale »

Pour fusionner deux fichiers contenant des variables différentes mais aussi une variable commune (identificatrice), la variable identificatrice (*var*) doit avoir le même nom dans les fichiers fusion1.dta et fusion2.dta. La première chose à faire est un tri des deux fichiers par cette variable. Procédure :

```
use fusion1 ;
sort var ;
save fusion1, replace ;
clear ;
use fusion2 ;
sort var ;
save fusion2, replace ;
```

On fait la fusion à partir de fusion2.dta (le fichier ouvert) comme suit : La syntaxe a changé dans Stata 12. L'ancienne syntaxe était

```
merge var using fusion1
save fusion3
```

La nouvelle est

```
Merge m:1 var using fusion1
```

```
save fusion3
```

si on a plusieurs observations dans le fichier maître à associer à une observation dans le fichier esclave, 1:m si c'est le contraire, 1:1 si c'est un pour un, et m:m si c'est plusieurs pour plusieurs (l'ancien joinby, je crois).

Les valeurs manquantes dans un fichier apparaissent dans le fichier fusionné. Les lignes omises dans un fichier (ici, fusion2.dta) apparaissent dans le fichier fusionné avec une valeur manquante pour les variables de ce fichier mais en bas du fichier fusionné, qui doit alors être re-trié.

La fusion horizontale comporte un grave risque. Il se peut que la variable commune (identificatrice) ne soit pas suffisante pour identifier chaque observation : dans notre cas, il se peut que plusieurs observations aient la même valeur dans *class*. Dès lors, il convient de trier les données avec le maximum de variables communes possibles, par exemple

```
use fusion1 ;
sort class code country year id ;
save fusion1, replace ;
clear ;
use fusion2 ;
sort class code country year id ;
save fusion2, replace ;
```

Puis, à nouveau,

```
merge class code country year id using fusion1 ;
save fusion3 ;
```

Attention, beaucoup de désastres ont leur origine dans des fusions hasardeuses. S'il y a une chose à vérifier cent fois c'est celle-là. Exemple : imaginons que l'on veuille fusionner un fichier dans lequel les entreprises sont classées par leur code NACE avec un fichier de tarifs par code HS comprenant une concordance NACE-HS (NACE et HS sont des nomenclatures de produits). Le fichier entreprises est

	nace	firm	
1	100	1	
2	100	2	
3	200	3	
4	200	4	
5	200	5	

et le fichier concordance est

	nace	hs	tariff	
1	100	0001	10	
2	100	0010	15	
3	200	0011	20	
4	200	0100	25	

On les fusionne par NACE et on obtient...

	nace	firm	hs	tariff	_merge
1	100	1	0001	10	3
2	100	2	0010	15	3
3	200	3	0011	20	3
4	200	4	0100	25	3
5	200	5	0100	25	3

... du pipi de chat ! L'entreprise 1 est classée dans hs1 et a donc un tarif de 10%, l'entreprise dans hs2 avec un tarif de 15%, alors que rien dans le fichier-entreprises ne permet de dire qu'elles sont différenciées par code HS (puisque le fichier-entreprises n'a pas de code HS !). Le problème, c'est que chaque catégorie NACE a plusieurs images à la fois dans le fichier entreprises et dans le fichier concordance, et Stata les apparie selon sa fantaisie.

Il y a plusieurs possibilités dans ce cas. On peut d'abord calculer un tarif moyen par code NACE, puis faire un collapse (voir la section « agréger des données ») par code NACE, et ensuite seulement faire la fusion. Ou bien (plus élégant) utiliser la commande

```
joinby varlist using file2
```

ce qui va éclater la variable nace par hs et entreprise (c'est-à-dire faire un produit cartésien de toutes les combinaisons de hs et firmes), ce qui donne

	nace	firm	hs	tariff
1	100	1	0001	10
2	100	1	0010	15
3	100	2	0001	10
4	100	2	0010	15
5	200	3	0011	20
6	200	3	0100	25
7	200	4	0011	20
8	200	4	0100	25
9	200	5	0011	20
10	200	5	0100	25

et on peut, à partir de ce fichier (dont la taille a bien sûr augmenté), calculer des moyennes simples de tarifs par entreprise.

Fusion « verticale »

Pour fusionner deux fichiers contenant les *mêmes* variables mais des *observations* différentes (par exemple s'ils sont tellement gros qu'ils ont été préalablement saucissonnés), c'est encore plus simple :

```
use fusion1;
append using fusion2 ;
```

```
save fusion3 ;
```

Variables avec indices (boucles)

Principe général

Deux commandes : `foreach` et `forvalues`. Supposons que l'on veuille créer une série de variables muettes marquant cinq catégories codées dans une variable *type* allant de 1 à 5. On peut écrire

```
foreach i of numlist 1/5 {  
  gen dummy`i' = (type==`i') ;  
} ;
```

en notant que 1/5 signifie « de 1 à 5 » et `numlist` est (évidemment) une liste de nombres. On peut aussi énumérer toute la liste, sans virgule, ou combiner :

```
foreach i of numlist 1 2 3/5 {  
  gen dummy`i' = (type==`i') ;  
} ;
```

C'est justement la combinaison qui rend `foreach` flexible (au cas où une boucle plante on peut la sauter par exemple). Par contre `foreach` n'admet que des valeurs numériques. Si on veut énumérer jusqu'à la fin de l'échantillon sans préciser sa taille en utilisant `_N` (par exemple pour une procédure à répéter sur des échantillons de taille variable), il faut utiliser `forvalues`.

La syntaxe pour `forvalues` consiste à donner le « i » initial (le premier 1), l'incrément du i (le 1 entre parenthèses), et le i final. Pour l'exemple précédent, on écrit

```
local N = _N  
forvalues i = 1(1) N {  
  gen dummy`i' = (type==`i')  
}
```

Donc

```
forvalues i = 0(2) 4 {  
  gen x`i' = 1990 + `i'  
}
```

donne trois nouvelles variables, `x0=1990`, `x2=1992` et `x4=1994`, la valeur entre parenthèses (2) disant à Stata de sauter de deux en deux.

Comme exemple de l'application de `foreach`, supposons que l'on veuille calculer un « poverty gap » par année pour quatre années : 1993, 97, 99 et 2001. On a dans la base de données quatre variables binaires appelées *txchange1993* à *txchange2001* et une autre

appelée *year* qui prend comme valeur le chiffre de l'année (1993 à 2001). Soit *j* l'indice marquant les années (*j* = 1993, etc...) On définit les variables avec l'indice *j* de la façon suivante

```
foreach j of numlist 1993 1997 1999 2001{

gen dollardep`j'      = dep/txchange`j' if year==`j'
gen dollarperday`j' = dollardep`j'/360

gen gap`j'            = 1 - dollarperday`j'
mean gap`j' if gap`j' <=.
display gap`j'
}
```

Itérations

Jusqu'à présent on a considéré des boucles consistant à répéter une commande (ou un bloc de commandes) pour chaque valeur de l'indice *i*, disons de un à *n*. Il ne s'agit pas d'itérations à proprement parler puisque chaque boucle est indépendante de la suivante. On considère maintenant une série de commandes dans lesquelles la valeur initiale d'une itération est la valeur donnée par l'itération précédente. Supposons que l'on veuille générer un processus de Markov

$$x_t = \lambda x_{t-1} + u_t$$

où u_t est un choc avec une distribution normale, $\lambda = 0.9$ et une valeur initiale $x_0 = 1$. La liste de commandes pour cinq itérations est

```
gen x0=1 ;
gen lambda=0.9 ;
foreach i of numlist 1/5 {;
local j = `i'-1 ;
set seed 0 ;
gen u`j'=invnorm(uniform()) ;
gen x`i' = lambda*x`j' + u`j' ;
};
display "valeur finale egale a" x5 ;
end ;
```

On note que x_t et u_t sont en fait des vecteurs si on a plusieurs observations. Les conditions et les commandes qui suivent sont à interpréter en gardant cela à l'esprit.

Si on veut choisir le nombre d'itérations simplement en appelant le programme, on mettra le bloc de commandes sous forme d'un programme (ou d'un fichier ado) et on utilisera `forvalues`:

```
program drop newloop ;
program define newloop ;
```



```

gen x0=1 ;
gen lambda=0.9 ;
local n = `1' ;
forvalues i = 1(1) `n' {;
  local j = `i'-1 ;
  set seed 0 ;
  gen u`j'=invnorm(uniform()) ;
  gen x`i' = lambda*x`j' + u`j' ;
};
display "valeur finale egale a " x`n' ;
end ;

newloop 100 ;

```

Boucles sur des observations

Attention : `forvalues` ne fonctionne pas sur des observations. Par exemple, supposons que l'on veuille générer une variable de stock de capital K à partir d'une variable d'investissement I par l'inventaire perpétuel. La commande

```

local N = _N
gen K = 100
forvalues j = 2(1)N {
  replace K = 0.9*K[_n-1] + I[_n-1] if _n == `j'
}

```

ne fonctionne pas. Il n'y a pas besoin de boucle explicite. Il suffit d'écrire

```

gen K = 100
replace K = 0.9*K[_n-1] + I[_n-1] if _n != 1

```

et stata procède séquentiellement en commençant par la deuxième observation.

Matrices et vecteurs

Si on veut vraiment faire des opérations matricielles, Stata a un sous-programme, Mata, avec sa propre syntaxe (voir une petite introduction à Mata à la fin de cet aide-mémoire). On ne présente ici que quelques manipulations très simples.

Mettre une matrice en vecteur-colonne

Avec la commande `reshape`. On démarre avec une matrice 3x2 de la forme suivante

Editor

ve Restore Sort << >>

var9[14] =

	source	dest1	dest2
1	1	3	2
2	2	5	4
3	3	7	6

L'indice de l'industrie-source est $i = 1, 2, 3$ dans la première colonne et l'indice de l'industrie-destination est $j = 1, 2$ dans le suffixe des têtes de la deuxième et troisième colonnes. Les variables (coefficients input-output si c'est une matrice IO mais j'ai pris une matrice rectangulaire pour montrer que la procédure ne demande pas une matrice carrée) sont $x_{11} = 3$, $x_{12} = 2$, $x_{21} = 5$, $x_{22} = 4$, $x_{31} = 7$ et $x_{32} = 6$. Cet exemple est dans le fichier IOwide.dta dans DATA. On suppose aussi, c'est en général le cas, que la première colonne est en format string (le code du secteur).

La série de commandes est

```
destring source, replace ;
reshape long dest, i(source) j(destination) ;
gen coeff=dest ;
drop dest ;
```

L'écran de Stata donne alors les détails de l'opération :

```
. reshape long dest, i(source) j(destination)
(note: j = 1 2)

Data               wide  ->  long
-----
Number of obs.      3    ->    6
Number of variables  3    ->    3
j variable (2 values) -> destination
xij variables:
                   dest1 dest2 -> dest
```

dont le résultat est

Editor

e Restore Sort << >> H

var10[13] =

	source	destination	coeff
1	1	1	3
2	1	2	2
3	2	1	5
4	2	2	4
5	3	1	7
6	3	2	6

les coefficients input-output étant maintenant stockés dans la variable *coeff*. La variable *destination* correspondant à l'indice j a été créée.

Pour que la syntaxe marche, on note que le nom des variables correspondant aux secteurs de destination doit avoir un suffixe correspondant à la variable i (code-produit). Je suppose qu'il faut donc renommer toutes les colonnes à la main, un peu casse-pieds mais

bon. Il faut aussi que le code du secteur i soit numérisable, c'est-à-dire ne contienne pas de lettres ou d'espaces.

Mettre un vecteur-colonne en matrice

On repart du tableau ci-dessus, et on veut remettre destination en colonnes. La commande est

```
reshape wide coeff, i(source) j(destination)
```

et les deux variables correspondant à "source" et "destination" dans notre exemple doivent être toutes les deux numériques, sauf si on utilise l'option string (voir ci-dessous). Si on a une variable qui ne change pas pour un i donné on a pas besoin de la mettre dans la liste de variables de `reshape` (exemple, i est l'identifiant d'un individu et la variable est son sexe). Bien évidemment on ne peut mettre qu'une seule variable dans la matrice.

Plus vicieux : supposons que l'on ait des données du type

var6[4] =				
	year	hs6	value	
1	2002	10690	8827	
2	2003	10110	1038	
3	2003	30110	862	

c'est-à-dire 3 catégories de produits et deux années mais (i) des valeurs manquantes et (ii) des lignes carrément omises pour ces valeurs, au lieu de lignes vides (typique des données commerce). On veut construire un tableau avec `hs6` en ligne et les années en colonne (`hs6` va donc jouer le rôle de *source* et `year` de *destination* dans l'exemple précédent), et des cellules vides pour les valeurs manquantes. D'autre part on veut garder `hs6` en format string, sinon le zéro du début va disparaître.

Il faut commencer par trier les données d'abord par `hs6` puis par année, sinon ça ne fonctionnera pas. Supposons que `year` et `hs6` soient toutes les deux en string au départ. Pour le tri il faut les mettre en format numérique, mais en gardant une version string de `hs6` puisque l'on en aura besoin plus tard (la transformation n'est pas neutre à cause du zéro au début alors que pour `year` elle l'est). La liste de commandes est alors

```
destring hs6, gen(nhs6) ;  
destring year, replace ;  
sort nhs6 year ;
```

Après le tri, comme on veut `hs6` en string, il faudra utiliser l'option string dans `reshape` et il faudra alors que les deux variables (`hs6` et `year`) soient en string, ce qui veut dire les transformer et supprimer leurs versions numériques :

```
gen str4 syear=string(year) ;
drop nhs6 year ;
```

enfin on peut utiliser la commande *reshape wide*

```
reshape wide value, i(hs6) j(syear) string ;
```

et on obtient

var11[20] =			
hs6	value2002	value2003	
010110	.	1038	
010690	8827	.	
030110	.	862	

Multiplier une matrice par un vecteur

Voilà le programme concocté à partir des fichiers *IOLong.dta* et *vector.dta*, tous les deux dans le répertoire DATA. La matrice dans le fichier *IOLong.dta* doit déjà avoir été mise en format long (voir section précédente). Soit i la source (lignes) et j la destination (colonnes) si on a en tête une matrice input-output par exemple. On commence par les bidouillages habituels:

```
use IOLong.dta ;
destring source, replace ;
sort destination ;
save matrix.dta, replace ;
```

puis on fait la même chose avec le fichier contenant le vecteur-colonne:

```
clear ;
use vector.dta ;
sort destination ;
save newvector.dta, replace ;
```

Ensuite on fusionne les deux fichiers sur la base de l'indice j (appelé *destination* et contenu dans la deuxième colonne du fichier matrice en format long)

```
merge destination using matrix ;
drop _merge ;
save matrixvector.dta, replace ;
```

Puis on génère les produits par “cellule” $x_{ij} \cdot y_j$ que l'on appelle *product*:

```
gen product=coeff*y ;
sort source destination coeff product ;
```

Enfin on somme ces produits sur j , c'est-à-dire que l'on génère les expressions $\sum_j x_{ij} \cdot y_j$. Attention, on va utiliser l'option “*by source*” qui en Stata signifie que l'on

garde i constant, c'est-à-dire que l'on somme sur j . En d'autres termes, "by" indique l'indice (la variable) gardé constant et non pas l'indice (la variable) sur lequel on somme.

```
by source: egen final=sum(product) ;  
collapse final, by(source) ;  
save matrixvector.dta, replace ;
```

et surprise, ça marche, si, si. Mais franchement ça va plus vite avec Mata (voir ci-dessous).

Cylindrer un panel

Supposons que l'on veuille cylindrer un panel en rajoutant des zéros pour les périodes manquantes pour chaque individu. Pour fixer les idées, les variables sont `firm` (exportateur), `hs6` (produit), `iso3` (code destination) et `value`.

La commande est `fillin`, mais elle va mettre des valeurs manquantes pour toutes les variables (y compris `value`) dans les lignes qu'elle remplit. Pour `value`, qui est une variable numérique, on remplacera simplement les valeurs manquantes par des zéros. Par contre, pour les autres variables qui sont des string, c'est plus compliqué. Deux méthodes pour que `firm`, `hs6` et `iso3` soient renseignées dans chaque ligne, les deux un peu bourrin.

Méthode 1

```
egen cell = concat(firm hs6 iso3), punct(,)  
fillin cell year  
split cell, parse(,)  
drop firm hs6 iso3  
rename cell1 firm  
rename cell2 hs6  
rename cell3 iso3
```

Méthode 2

On utilise le fait que `value` va être codée en valeur manquante pour toutes les années rajoutées par `fillin`. En faisant un `sort` par `value`, Stata mettra les valeurs renseignées en premier. Après, on lui dit de remplacer chaque ligne par la ligne précédente à l'intérieur de chaque cellule (`firm hs6 iso3`) :

```
sort firm hs6 iso3 value  
by firm hs6 iso3: gen n = _n  
  
by firm hs6 iso3: replace firm = firm[_n-1] if n > 1  
by firm hs6 iso3: replace hs6 = hs6[_n-1] if n > 1  
by firm hs6 iso3: replace iso3 = iso3[_n-1] if n > 1
```

Finalement on remplace les valeurs manquantes de `value` par des zéros:

```
replace value = 0 if value == .
```

Un peu de programmation pour les ado

Programme

On peut inclure un programme à l'intérieur du do-file. Attention, si on fait marcher le do-file plusieurs fois de suite (par exemple pour debugger) on aura le message d'erreur « program dummy already defined ». A partir de la deuxième fois il faudra donc écrire `program drop dummy` avant `program define dummy` (pas très élégant mais bon). La syntaxe de base pour ouvrir et fermer le programme (par exemple de création des variables muettes) est

```
program define dummy ;  
    forvalues i = 1(1) 3 {;  
        gen type`i' = (type==`i') ;  
    } ;  
end ;  
dummy
```

et le programme est exécuté par le do-file par la dernière commande qui est simplement son nom (les lignes qui précèdent ne font que le définir).

Fichiers ado

Une alternative plus élégante à l'inclusion d'un programme dans le do-file (par exemple s'il est long) est de créer un fichier *.ado* (en utilisant notepad ou n'importe quel éditeur ASCII). Si on veut utiliser le programme en l'appelant par une commande il doit être sauvegardé dans *c:\Stata\ado\personal* ou plus simplement dans un répertoire ado directement sur *c:* avec des sous-répertoires chacun ayant comme nom une lettre de l'alphabet. On classe les fichiers ado dans le répertoire de son initiale. Un fichier ado est mis en route dans le fichier *.do* par une commande portant simplement son nom (juste le nom, pas le chemin pour y arriver) suivie du nombre de boucles à faire s'il y a des boucles (voir ci-dessous pour les macros).

Macros. Première chose utile à savoir : comment créer un macro (au vin blanc). Stata en a de deux sortes : local et global. Un macro local défini dans un fichier ado ne peut pas être utilisé dans un autre fichier ado. Par contre, un macro global est défini une fois pour toutes et peut être utilisé par tous les fichiers ado (donc à utiliser prudemment). Un exemple de macro local consiste à donner un nom à une liste de variables, disons *x y z* en l'appelant *truc* :

```
local truc "x y z"
```

On peut alors utiliser ``truc'` (avec un simple guillemet) dans n'importe quelle commande, style

```
sum `truc'  
reg depvar `truc'
```

Par contre les doubles guillemets dans la définition du macro sont optionnels. On peut bien sûr combiner le macro avec une boucle. Supposons que l'on veuille mettre au carré 10 variables nommées respectivement z_1 - z_{10} . On commence par faire un macro local avec la liste des variables, puis on fait la boucle, ce qui donne

```
local truc z1-z10 ;  
foreach x of local truc { ;  
  gen new`x'=`x'^2 ;  
} ;
```

On note que le nom de la variable générique de la boucle (x) est libre. Jusque-là, le macro peut être utilisé dans un do-file normal, il n'y a rien qui requière un programme.

Supposons maintenant qu'on ait 100 types possibles d'une variable, indexés dans une variable-code allant de 1 à 100. Pour créer des variables muettes disons pour les 50 derniers types, le contenu du fichier ado pour créer la liste de variables muettes serait:

```
program define dummy ;  
  local n = `1' ;  
  forvalues i = 50(1) `n' {;  
    gen type`i' = (type==`i') ;  
  } ;  
end ;
```

qui est appelé dans le do-file par

```
dummy 100
```

c'est-à-dire avec le nom du programme (`dummy`) et le numéro de la dernière boucle désirée (100). La signification de la ligne `local n = `1'` n'est pas trop claire mais bon. La définition du macro local par ``1'` avec des guillemets uniques ne peut être utilisée que dans un programme. L'intérêt de cette syntaxe est que la fin des boucles peut maintenant être choisie directement par la commande d'appel du programme `dummy`.

If/else

Supposons maintenant que l'on veuille faire de zéro une barrière absorbante, c'est-à-dire que dès que le processus passe à zéro ou en-dessous, il reste coincé à zéro. On utilise la commande `if/else` :

```
gen x0=1 ;  
gen lambda=0.9 ;
```

```

foreach i of numlist 1/100 {;
local j = `i'-1 ;
set seed 0 ;
gen u`j'=invnorm(uniform()) ;
if x`j' <= 0 {;
    gen x`i' = 0 ;
};
else {;
    gen x`i' = lambda*x`j' + u`j' ;
};
};
display "x100 =" x100 ;

```

Attention la syntaxe est très sensible. L'ouverture d'accolade doit être sur la même ligne que le `if`, la commande en dessous, et la fermeture d'accolade sur une ligne à part.

En utilisant `forvalues` on peut mettre aussi une barrière « endogène » en disant à Stata de s'arrêter dès que la différence entre x_t et x_{t-1} est plus petite que 0.1, ce qui illustre en même temps l'imbrication de commandes `if/else` :

```

gen x0=1 ;
egen X0=mean(x0) ;
gen lambda=0.9 ;

program drop newloop ;
program define newloop ;

local n = `1' ;
forvalues i = 1(1) `n' {;
local j = `i'-1 ;
set seed 0 ;
gen u`j'=invnorm(uniform()) ;
gen x`i' = lambda*x`j' + u`j' ;
egen X`i' = mean(x`i') ;
if X`j+1'-X`j' > -0.01 {;
    continue ;
};
else if x`j'>5 {;
    continue ;
    else {;
        break ;
    };
};
};
display "valeur finale " X`n' ;
end ;

newloop 20 ;

```

et là encore on peut choisir combien de boucles on fait (ce qui ne sert pas forcément à grand'chose si l'arrêt est endogène ! Mais bon). On note que la condition du `if` ($x_{j+1}' - x_j' > -0.01$) est exprimée en termes de l'indice j , pas i .

While

La syntaxe de while est en gros la même que celle de if/else :

```
while condition { ;  
command ;  
} ;
```

A utiliser prudemment, `while` peut facilement produire des boucles sans fin...

Estimation : quelques bidouillages

Sauvegarder des résultats, même quand ils sont absurdes

Sauvegarder des statistiques descriptives

```
sum x ;  
gen x_bar = r(mean) ;  
gen sigma_x = r(sd) ;
```

Sauvegarder valeurs prédites et résidus

Pour sauvegarder les valeurs prédites (en générant une nouvelle variable `yhat`) et résidus (en générant une nouvelle variable `residuals`), les commandes sont

```
predict yhat  
predict e, residuals
```

Par défaut c'est les valeurs prédites qui sortent.

Sauvegarder des coefficients absurdes

Deux façons de sauvegarder des estimés. On peut le faire simplement par la commande

```
gen beta_1 = _b[x1]  
gen constant = _b[_cons] ;
```

pour les coefficients, et

```
gen sigma_1 = _se[x1]
```

pour les écarts-types. Plus généralement, toutes les commandes de stata gardent leur résultat en mémoire dans trois types de macros, les « r », les « e » et les « s ». On peut accéder à la liste de ces macros en tapant `return list`, `ereturn list` ou `sreturn list`.

Alternativement, les estimés de régression et leurs variances sont sauvegardés sous forme de macros dans `e()`, et on peut les transformer en variables apparaissant dans la base de données (sur la première ligne uniquement, puisqu'il s'agit de scalaires) en tapant :

```
reg y x
mat b=e(b)
svmat b
```

La première commande (`mat b=e(b)`) génère un vecteur-ligne b dont les éléments sont ceux de $e(b)$, i.e. les coefficients. La deuxième (`svmat b`) génère k variables b_1, b_2, \dots, b_k , chacune avec une seule observation (sur la première ligne), correspondant aux k coefficients estimés par la régression précédente, mais attention le coefficient de la constante vient toujours en dernier, donc on peut ajouter `rename bk b0` pour être sûr de ne pas confondre. Pour éviter de charger la base de données, on peut définir un scalaire par coefficient :

```
scalar beta1 = b1
scalar beta0 = b2
```

et faire des opérations dessus du genre

```
gen yhat = beta0 + beta1*x1
```

On peut donner le nom qu'on veut à la place de b . Si on voulait garder les coefficients dans une seule variable $A1$ (en colonne), on sauverait la transposé de la matrice `e(b)`,

```
mat b=e(b)' ;
svmat b ;
```

Pour sauvegarder la matrice de variances-covariances des coefficients, la commande est

```
mat v=e(V) ;
svmat v ;
```

On note que l'argument dans `e(V)` est en majuscule. Ce qu'on obtient est une matrice $k \times k$ (ou k est le nombre de coefficients y compris celui de la constante, qui vient toujours en dernier dans stata) et les variances sont sur la diagonale (donc $\sigma^2(\hat{\beta}_1)$ est sur la première ligne et la première colonne, et ainsi de suite jusqu'à $\sigma^2(\hat{\beta}_k)$, la variance du coefficient sur la constante, qui est sur la k -ième ligne et la k -ième colonne.

Générer un tableau de régressions

Si l'on veut sauvegarder des résultats d'estimation sous une forme directement exportable dans excel, la commande qui vient immédiatement après la commande d'estimation est

```
outreg varlist using table1, br coeastr 3aster bdec(3) tdec(3)
bfmt(fc) replace;
```

qui génère un fichier (*table1.out*) contenant un tableau de résultats avec les statistiques *t* entre crochets (*br*) pour qu'Excel ne les prenne pas pour des nombres négatifs⁷, des astérisques par niveau de significativité (*coeastr 3aster*: trois pour 1%, 2 pour 5% et une pour 10%), trois chiffres après la virgule pour les coefficients (*bdec(3)*) et les statistiques *t* (*tdec(3)*) et des coefficients formatés en fixe (et non en notation scientifique, pas pratique) et avec une virgule pour les milliers (*bfmt(fc)*). Par défaut, *outreg* utilise les labels et non les noms de variables.

Pour que plusieurs colonnes de résultats soient regroupées dans un seul tableau, la commande *outreg* est comme ci-dessus après la première estimation mais pour les suivantes, on ajoute l'option *append* en gardant le même nom de fichier et les mêmes options de formatage ; style

```
reg y x1 x2 x3;
outreg x1 x2 x3 using table1, coeastr 3aster replace;
reg y x2 x3 x4;
outreg x2 x3 x4 using table1, append coeastr 3aster replace;
```

La commande *outreg* n'est pas dans le package standard de Stata8 (ni de Stata9 d'ailleurs) et doit être téléchargée depuis le web avec *findit outreg*. Une fois qu'un fichier excel est généré par *outreg*, il peut être ré-exporté vers Latex en incorporant dans Excel la macro *Excel2Latex* (fichier téléchargeable depuis ma page web, il suffit de cliquer dessus et il se met tout seul dans Excel)⁸.

⁷ Si on veut garder des parenthèses et pas des crochets, quand on ouvre le fichier *table1.out*, à la troisième (dernière) étape du wizard il faut sélectionner chaque colonne et la formater en texte. Excel ne changera alors pas les parenthèses en nombres négatifs.

⁸ A propos, pour passer d'un format de page portrait à landscape pour des tableaux larges, les commandes en latex sont:

Au début du document: `\usepackage{rotating}`

Dans le document, juste avant le tableau:

`\clearpage`

`\begin{sidewaystable}[!h]`

`\centering`

`\begin{tabular}`

Données du tableau

`\end{tabular}`

`\caption{Le nom de mon tableau}`

`\end{sidewaystable}`

`\clearpage`

Estimation

OLS and WLS

OLS c'est

```
reg y x
```

Avec des contraintes (par exemple $\beta_1 = 1$ et $\beta_2 = \beta_3$) c'est

```
constraint define 1 x1 = 1  
constraint define 2 x2 = x3  
cnsreg y x1 x2 x3, c(1,2)
```

Pour WLS, la commande est

```
reg y x [aweight = z], options
```

où `[aweight = z]` n'est pas une option mais vient *avant* la liste des options (par exemple `nocons`)

Le clustering des écarts-type se fait par l'option `vce(cluster clustervariable)`.

Panels

Tant de choses à dire ! Le mieux est d'aller voir le manuel de Stata. Quelques trucs à noter en priorité, cependant. La commande pour indiquer à Stata que l'échantillon est un panel avec `product` comme variable de coupe et `year` comme variable temporelle est

```
xtset product year
```

Après quoi on peut utiliser toutes les commandes qui commencent par `xt`. Si la variable qui indice les individus est en string, exemple un code HS, Stata va couiner ; il faut alors la déclarer comme « groupe » :

```
egen product = group(hs6)
```

Attention ça ne marche pas très bien en Scientific Workplace. A propos de tableaux en SW, quand un tableau se met n'importe où dans le fichier, en général c'est qu'il manque le « caption ». Il se peut aussi que les données rentrées par le macro latex aient des lignes sautées, auquel cas SW va couiner.

Si l'on a un panel multidimensionnel (par exemple pays importateur \times produit) on définit comme nouvel individu une paire pays importateur \times produit que l'on appelle un « groupe ». La commande est alors

```
egen mgroup=group(reporter sitc2)
xtset mgroup year
```

On peut utiliser `foreach` pour des estimations séparées par année. Supposons que la variable dépendante est y_t avec $t = 2001, 2002, 2003$ et les régresseurs x_t et z_t . La commande est tout simplement

```
foreach t of numlist 2001 2002 2003 { ;
reg y x z if year==`t' ;
} ;
```

et on pourra récupérer les valeurs prédites et les résidus année par année, par exemple, avec

```
predict yhat`j' ;
predict resid`j', resid ;
```

Supposons qu'on estime un effet de traitement en DID. La correction suggérée par Bertrand Duflo et Mulainathan (2004) pour la corrélation sérielle de la variable de traitement est de permettre une matrice de covariances non-contrainte pour chaque individu dans la dimension temporelle. Supposons que ce soient des firmes, l'option est alors

```
egen newfirm = group(firm)
xtset newfirm year
xtreg y x year1-year10, fe vce(cluster newfirm)
```

Sure

La commande est

```
sureg (eq1 : y1 x1 x2 x3, options) (eq2 : y2 x1 x2)
predict y1hat, equation(eq1) ;
```

Parmi les options, `aweight` et `fweight` fonctionnent mais pas `pweight`.

2sls

Supposons que l'on veuille estimer le système suivant

$$y = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 + u$$

$$x_1 = \beta_0 + \beta_1 y + \beta_2 z_2 + \beta_3 z_3 + v.$$

La commande est alors

```
ivreg y x2 x3 (x1 = z1 z2 z3)
```

et la plupart des options de `reg` (OLS) sont permises. Le test d'endogénéité de Hausman est entré de la façon suivante :

```
ivreg y x2 x3 (x1 = z1 z2 z3) ;
hausman, save ;
reg y x1 x2 x3 ;
hausman, constant sigmamore ;
```

et l'output dit si la différence des coefficients entre les deux regressions (un chi carré) est significative ou non. Si elle est significative ($\text{Prob} > \chi^2$ inférieure à 0.05) l'hypothèse nulle (pas d'endogénéité) est rejetée et les estimés OLS sont inconsistants.

Si pour une raison ou pour une autre on veut calculer des écarts-types par bootstrap, il y a plusieurs commandes. Supposons par exemple qu'on veuille calculer l'écart-type du coefficient sur `x1` par bootstrap. On peut utiliser la commande

```
bootstrap _se[x1] _se[x2] _se[_cons], reps(#) saving(filename, replace)
: command
```

où `#` est le nombre de répétitions et `command` est la commande de régression (ou autre). Si la procédure est plus compliquée qu'une commande en une ligne, il faut utiliser un ado file.

GMM

Si on veut utiliser l'estimateur dit « system-GMM » de Blundell et Bond, le programme doit être descendu du web par findit xtabond2. Le system GMM estime une équation simultanément en niveaux et en différences et instrumente les niveaux par les différences contemporaines et les différences par les niveaux retardés. Si on n'a pas d'instruments externes, la commande est

```
xtabond2 y L.y x_exo x_endo, gmmstyle(y, lag(2 2) collapse) \\\
gmmstyle(L.y x_endo, lag(2 2) collapse) \\\
ivstyle(x_exo) robust h(2) nomata
```

L'estimateur utilise par défaut toutes les valeurs retardées des variables endogènes disponibles dans l'échantillon. Il y a alors souvent trop d'instruments, un problème qui se manifeste dans une p-valeur du test de Hansen proche de 1. En général il vaut mieux qu'il n'y ait pas plus d'instruments que d'individus dans le panel. L'option `lag(2 2)` permet

de limiter le nombre de retards utilisés à deux périodes. L'option `robust` contrôle l'hétéroscédasticité. L'option `nomata` évite un problème que je n'ai jamais compris (il y a toute une explication dans le help).

Si on a un vecteur `z` d'instruments, la commande est

```
xtabond2 y L.y x_exo x_endo, gmmstyle(y, lag(2 2) collapse) \\\
gmmstyle(x_endo, lag(2 2) collapse) \\\
ivstyle(x_exo z) robust h(2) nomata
```

Pour utiliser le GMM efficient en deux étapes, on ajoute l'option `twostep`. Attention il faut faire un `tsset` avant la commande.

Variables dépendantes limitées

Tobit

Supposons que l'on veuille régresser une variable `y` censurée à zéro et à un sur `x`. La commande est

```
tobit y x z, ll(0) ul(1)
```

où `ll(.)` est `ul(.)` sont les bornes inférieures et supérieures respectivement. Pour avoir les effets marginaux, il faut descendre `dtobit2` en tapant « findit dtobit2 » (Dieu sait pourquoi la commande `dtobit` bugge). Sinon on peut utiliser la commande `postestimation`

```
mfx compute
```

mais elle est bestialement gourmande en calculs.

Probit

L'estimation et les effets marginaux s'obtiennent de la façon suivante. Soient `x`, `y`, et `z` trois variables respectivement catégorielle (`x`), binaire (`y`) et continue (`z`). La variable dépendante est `y`. Supposons qu'`x` aille de 1 à 8 et que l'on veuille son effet marginal à la valeur `x = 7`. Les commandes sont

```
probit y x z
matrix xvalue=(7)
dprobit y x z, at(xvalue)
```

La première ligne estime le probit de façon conventionnelle et donne les coefficients. La seconde définit la valeur à laquelle l'effet marginal est calculé (par défaut, c'est-à-dire sans la seconde ligne et l'option dans la troisième, c'est la moyenne). La troisième ligne demande les résultats sous forme d'effets marginaux plutôt que de coefficients. L'option `robust` peut être utilisée avec `probit`.

Supposons que l'on veuille compter le nombre d'observations pour lesquelles $y = 1$ et l'équation donne $\text{Prob}(y=1) > 0.5$ ou bien $y = 0$ et $\text{Prob}(y=1) < 0.5$ (une mesure ad hoc du fit de l'équation). La série de commandes est

```
predict yhat ;
gen predicted=(yhat>=0.5) ;
count if (predicted==1 & y ==1) | (predicted==0 & y ==0) ;
```

Enfin, à noter que `outreg` utilisé avec un probit ne donne pas par défaut le pseudo-R². Il faut préciser

```
outreg x1 x2, addstat(Pseudo R-squared, e(r2_p))
```

A savoir: on peut pas interpréter le coefficient sur un terme d'interaction comme l'effet de l'interaction des deux variables dans un modèle non-linéaire (logit, probit ou autre), ça se voit en cinq mn dans l'algèbre. S'il n'y a pas de terme exponentiel, on peut utiliser la commande `inteff`:

```
logit y age educ ageeduc male ins_pub ins_uni x, cluster(pid)
inteff y age educ ageeduc male ins_pub ins_uni x, ///
savedata(d:\data\logit_inteff,replace) savegraph1(d:\data\figure1,
replace) savegraph2(d:\data\figure2, replace)
```

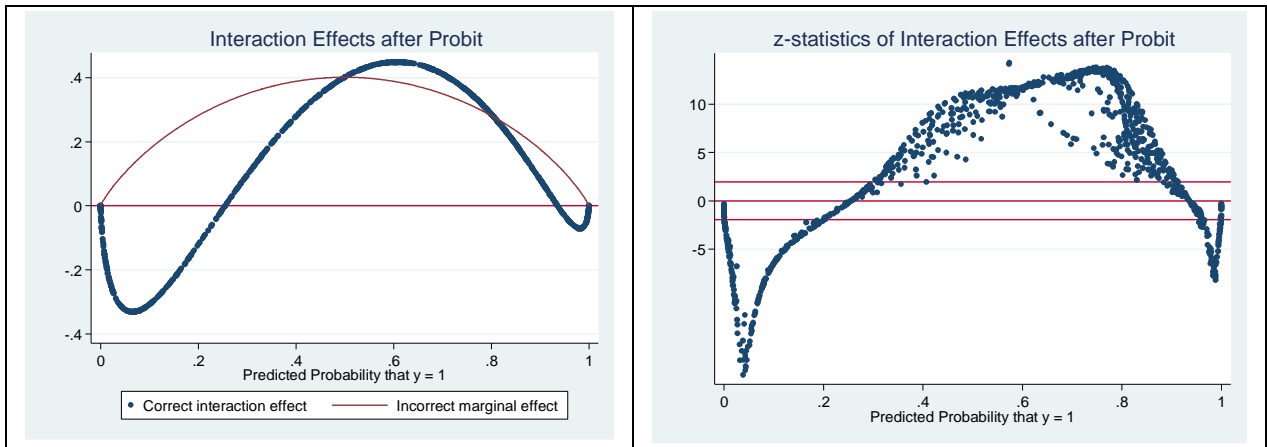
La commande donne l'effet correct de l'interaction, l'écart-type et la stat z pour chaque observation. On peut se donner une idée de ce que `inteff` produit avec le petit programme suivant

```
clear
set obs 1000
gen x = invnorm(uniform())
gen z = invnorm(uniform())
gen latent = x - z + x*z + invnorm(uniform())
gen y = (latent > 0)
gen interact = x*z

probit y x z interact
inteff y x z interact, ///
savedata(d:\databases\made_up\inteff1,replace) ///
savegraph1(d:\databases\made_up\inteff_figure1,replace) ///
savegraph2(d:\databases\made_up\inteff_figure2, replace)
```

ce qui produit les graphiques bizarroïdes suivants:

--	--



Variables instrumentales en probit :

La commande est `ivprobit` et la syntaxe est

```
ivprobit y x2 x3 (x1 = z1 z2 z3)
```

où $x1$ est le régresseur endogène et $z1$ - $z3$ sont les instruments.

Multinomial logit

Syntaxe du multilogit : soit $y \in \{y_1, y_2, \dots\}$ la variable de choix. La commande pour choisir la catégorie de base par rapport à laquelle tous les coefficients seront calculés (disons y_1) est

```
mlogit y x1 x2 x3, basecategory(y1)
```

En `mlogit` les coefficients ne sont pas interprétables directement (ratios de probabilités) ; les effets marginaux des variables $x1$ et $x2$ (par défaut toutes) calculés aux valeurs *valeur1* et *valeur2* (par défaut aux moyennes) sur la catégorie $y1$ (par défaut toutes) sont obtenus avec

```
prchange x1 x2, x(x1=valeur1) x(x2=valeur2), outcome(1)
```

Pour les régresseurs binaires le delta par défaut est de un. A noter, *prchange* ne fonctionne pas avec *svymlogit*, la commande de multilogit pour les enquêtes.

Une commande alternative qui fonctionne avec *svymlogit* est

```
mfx compute, predict (outcome(1)), eyex at(median x1)
```

qui donne également les écarts-type et peut donner des élasticités directement (*eyex*) ou en semi-log (voir [R] **mfx**), et peut être évaluée là où on veut (ici médiane du régresseur

x1). Il faut répéter la commande pour chaque catégorie de la variable dépendante. Attention le calcul est lourd et prend quatre siècles.

Negative binomial

```
nbreg y x, exposure(z)
```

Switching reg

Les programmes de Laure (des ado-file qui sont sur mon disque dur) sont les suivants: Pour un switching inconnu et exogène, c'est `switch_exo`, pour un switching inconnu et endogène, c'est `switch_endo`, il y a des help files pour la syntaxe exacte (c'est un cauchemard). On peut faire tourner une grid search (option `gs`) ou un algorithme EM (option `em`) mais il faut mettre un cierge à l'église pour que ça tourne.

Une fois qu'on a fait tourner l'un des deux et qu'on a trouvé le switchpoint, on peut faire tourner `movestay1` qui est pour le switchpoint connu mais endogène qui tourne beaucoup plus vite et qui peut générer quelques fonctions post-estimations (valeurs prédites etc., voir le fichier `help movestay`).

Si tout foire on peut utiliser le premier do file que Laure a fait qui s'appelle `switchshventeprod.do`. Si si c'est bien ça le nom. Il fait une boucle sur `movestay1` pour le grid search.

Propensity score matching

La commande `psmatch2` (à googler et à descendre) permet de faire des comparaisons d'outcomes entre un groupe de traitement et un groupe de contrôle construit par PSM.

Première étape : Propensity score estimation

Supposons que la variable muette marquant le groupe de traitement soit `TG`. Le propensity score s'obtient en faisant un probit sur les caractéristiques individuelles `x1` et `x2` en utilisant tout l'échantillon, en utilisant la commande `pscore` :

```
pscore TG x1 x2, pscore(pscore) blockid(blockf1) comsup level(0.001)  
outreg using `pathresults'\pscore_file, 3aster replace
```

On vérifie le support commun avec des densités (voir ci-dessous pour les kernel densities) et des histogrammes en miroir

```
gen treated      = pscore if TG == 1  
gen untreated    = pscore if TG == 0
```

```
* Densities
twoway (kdensity untreated) (kdensity treated)
graph save Graph `pathresults'\densities, replace

* Mirror histograms
psgraph, treated (TG) pscore(pscore)
graph save Graph `pathresults'\mirror_histograms, replace
```

Seconde étape : Matching

Pour la procédure elle-même (comparaison de la valeur moyenne de la variable outcome pour les deux groupes), la commande de base est `psmatch2`. On peut en fait sauter l'étape `pscore` car la commande `psmatch2` fait cette étape elle-même et génère le `pscore` (`_pscore`). Dans ce cas on ne met pas l'option `pscore()`.

Matching avec la méthode des *n nearest neighbors* (si c'est 1 on met 1!) :

```
psmatch2 TG, outcome(y) pscore(pscore) neighbor(n) com cal(0.01)
```

La variable `TG` est la muette qui définit le groupe de traitement (construite préalablement). L'option `pscore(pscore)` utilise la variable `pscore` construite et sauvegardée dans la première étape. L'option `com` définit le support commun en excluant les individus traités ayant un score plus grand que tous les non traités et vice-versa. L'option `cal(0.01)` met la distance maximum entre voisins (le « caliper ») à 0.01. Matching en kernels :

```
psmatch2 treatment, outcome(y) pscore(pscore) kernel com
```

et on peut choisir le type de poids dans les kernels (voir le help).

La commande `psmatch2` génère automatiquement plusieurs variables qui apparaissent dans le fichier-données. `_treated` est la variable de traitement (égale à `TG` ici). `_support` est une muette égale à un si l'observation est dans le support commun. `_pscore` est le score. `_weight` est générée par le nearest neighbor et indique la fréquence avec laquelle l'observation est utilisée comme match ; elle est donc plus grande ou égale à un. `_id` est un identifiant ad-hoc utilisé pour `_n1`, qui est une étiquette mise à toutes les observations *traitées* indiquant l'identité de son match dans le groupe de *contrôle* si on utilise le nearest neighbor (si on utilise les *n nearest neighbors*, c'est le premier). Par exemple, si l'observation 1 dans le groupe de traitement est matchée avec l'observation 3 dans le groupe de contrôle, `_n1 = 3` pour l'observation 1. C'est clair ? Toujours avec le nearest neighbor, `_nn` donne le nombre de matches pour chaque observation traitée. Enfin `_pdif` donne la distance au voisin.

Il y a plusieurs “balancing tests” (pour vérifier que les caractéristiques individuelles moyennes ne diffèrent pas trop entre le groupe de contrôle et le groupe de traitement. La source est Smith & Todd (1985, 1986). La commande de base est

```
pstest TG x1 x2, support(_support), summ
```

où la variable `_support` est générée par la procédure `psmatch2` (voir ci-dessus). La commande `ptest` fait deux choses. La première est une régression de chaque covarié x_1, \dots, x_n sur la variable TG et donne le t-test. S'il n'est pas significatif, il n'y a pas de différence significative entre le groupe de traitement et le groupe de contrôle pour la variable x_i . La régression est courue avant matching sur tout l'échantillon, et après matching sur le support commun en moindres carrés pondérés utilisant les poids définis dans `_weight`. La deuxième donne le « standardized bias » pour le covarié x défini dans Rosenbaum et Rubin (1985) :

$$SB(x) = \frac{(100/n) \sum_{i \in TG} x_i - w_{ij} x_j}{\sqrt{[\text{var}_{i \in TG}(x) + \text{var}(x)_{i \in CG}] / 2}}$$

Il n'y a pas de théorie sur la valeur maximale admissible de $SB(x)$ mais on considère généralement que 20 c'est déjà beaucoup. Si on a peur de s'ennuyer, on peut aussi courir une régression polynomiale d'ordre 4 pour chaque covarié x de la forme

$$x_j = \alpha + \sum_{k=1}^4 \beta_k (\hat{p}_j)^k + \sum_{k=1}^4 \gamma_k TG_j (\hat{p}_j)^k + u_j$$

où \hat{p}_j est le score pour l'observation j (pas tout à fait sûr pour le terme d'interaction avec TG—à vérifier). On peut faire des MCP en utilisant `_weight` aussi si on veut. Si les γ sont conjointement non significatifs (F-test), c'est bon ! En pratique, la commande est donc

```
foreach j of numlist 1/4 {
    gen pscore`j' = pscore^`j'
    gen interact`j' = TG*pscore`j'
}

set seed 123456789
qui reg x pscore1-pscore4 interact1-interact4 if com == 1,
[iweight=_weight]
testparm TG interact1-interact4
```

Pour faire du matching diff-in-diff, il faut définir la variable d'outcome en premières différences :

```
gen dy = D.y
psmatch2 treatment, outcome(dy) pscore(pscore) kernel com
```

Mais attention ! C'est là que les vraies difficultés commencent. Première difficulté : la période de traitement est bien définie pour les firmes traitées, mais pas pour les firmes non traitées. Il faudrait contraindre `psmatch2` à ne prendre comme contrôles pour une firme traitée en 2005 que des firmes non traitées en 2005 aussi, mais il ne le fait pas. Il y a une variante appelée `matchyear` qui est censée le faire, développée par Jens Arnold, mais chaque fois que j'ai essayé de la faire marcher j'ai dû amener l'ordinateur à la réparation.

Deuxième problème. Supposons qu'une firme soit observée entre 2000 et 2010 et qu'elle se fasse traiter en 2005. Première possibilité, on définit la période de traitement comme 2005-2010. La variable $TG \times TP$ (groupe de traitement et période de traitement) est égale à 1 pour toutes les années à partir de 2005. Alors `psmatch2` va faire un test joint de la différence entre traitement et contrôle pour $y_{2005}-y_{2004}$ (la première année de traitement), mais aussi pour $y_{2006}-y_{2005}$, et ainsi de suite. Pourtant il n'y a pas de raison que $y_{2009}-y_{2008}$ soit plus élevé pour une firme traitée : c'est 4 ans après le traitement ! Deuxième possibilité, par contre, si on définit la période de traitement comme seulement 2005 et qu'on laisse la variable $TG \times TP$ retourner à zéro à partir de 2006, alors `psmatch2` risque de prendre la firme traitée en 2008 comme contrôle pour elle-même en 2005. Pour éviter ce problème, la méthode ad-hoc consiste à remplacer la variable $TG \times TP$ par des valeurs manquantes, pour toutes les firmes traitées, à partir de l'année *après* le traitement (2006-2010 pour la firme traitée en 2005).

Dernière chose à noter, on peut courir une régression en WLS avec des poids égaux à 1 pour le groupe de traitement et $r = _pscore/(1-_pscore)$ pour le groupe de contrôle, et on obtient des résultats très voisins du matching DID (voir Morgan et Harding 2006), avec l'avantage que comme c'est une régression on peut ajouter des contrôles additionnels (par exemple l'année calendaire de traitement).

Analyse de survie

Supposons que l'on ait des données d'exportation par firme, année, produit et marché de destination sous la forme suivante :

firm	year	hs6	market_d	ln_value
2084169A	2006	100190	GMB	9.493021
2223965A	2004	100620	MLI	11.48171
0078396A	2005	100630	ESP	8.888981
2292415A	2005	100640	MLI	9.987593
0055956A	2006	100820	FRA	8.249949
0463294A	2004	10392	COG	12.05565
0176033A	2004	110412	LBR	5.008934

La première chose à faire est de générer des zéros pour les combinaisons (firm hs6 market_d year) pour lesquelles il n'y a pas d'exportation, qui n'apparaissent pas ici. En d'autres termes, il faut créer un panel cylindré. On fait cela avec la commande `fillin` (voir les détails dans la section xx) puis on remplace les valeurs manquantes par des zéros (ici il faut exponentier les exportations puisqu'elles sont en logs) :

```
egen indiv = group(firm hs6 market_d)
fillin indiv year

gen outcome = exp(ln_value)
replace outcome = 0 if outcome == .
```

Ensuite on marque les événements de début et de mort des spells par les muettes suivantes :

```
gen start = (outcome != 0 & outcome[_n-1] == 0) | (outcome != 0 & year == 1)
gen fail = (outcome==0 & outcome[_n-1]!=0) & year!=1
```

Puis on donne une étiquette (`spell_id`) à chaque spell. On marque chaque ligne par n , puis on sauve (sous un nouveau nom, ici `survival2`). Ensuite, on génère un fichier avec seulement la muette marquant le départ de chaque spell et n (`keep n start`), on élimine toutes les observations autres que les démarrages de spells (`keep if start == 1`), et on génère l'étiquette (`spell_id`) comme l'identifiant de ligne (`_n`) dans ce nouveau fichier.

```
sort countrypair year
keep countrypair year start fail outcome
order countrypair year start fail outcome
```

```
* Tag spells
```

```
gen n = _n
sort n
save survival2, replace
```

```
keep n start
keep if start == 1
gen spell_id = _n
sort n
```

ce qui donne

		start	n	spell_id
	1	1	4	1
Jr...	2	1	11	2
	3	1	15	3
OW.	4	1	23	4
	5	1	32	5
	6	1	41	6
	7	1	48	7
	8	1	52	8

Enfin on fusionne par n avec le fichier `survival2` pour attacher à chaque démarrage de spell l'identifiant `spell_id` :

```
merge n using survival2
sort n
drop _merge n
```

et on remplace toutes les valeurs manquantes de l'étiquette (`spell_id`):

```
replace spell_id = spell_id[_n-1] if (spell_id==. & spell_id[_n-1]!=. & outcome > 0)
order countrypair year spell_id start fail outcome
```

ce qui donne

	indiv	year	spell_id	start	fail	outcome	firm	hs6	market_d	ln_value
1	1	2001	.	0	1	0				.
2	1	2002	.	0	0	0				.
3	1	2003	.	0	0	0				.
4	1	2004	1	1	0	34072.54	0000589A	540620	GMB	10.43625
5	1	2005	.	0	1	0				.
6	1	2006	.	0	0	0				.
7	1	2007	.	0	0	0				.
8	2	2001	.	0	0	0				.

Dans l'étape suivante, on génère des caractéristiques de spell (par exemple durée, valeur maximum, valeur initiale)

```

gen spell_alive = (outcome > 0)
sort spell_id year
by spell_id: egen duration = sum(spell_alive)
by spell_id: egen startyear = min(year)
by spell_id: egen endyear = max(year)
gen failtime = endyear + 1

by spell_id: egen max_outcome = max(outcome)
gen start_outcome = outcome if year == startyear
replace start_outcome = start_outcome[_n-1] if (start_outcome ==. &
start_outcome[_n-1] !=. & outcome > 0)

```

La base de données est maintenant prête à être transformée de façon à définir la spell comme unité d'observation, ce qui va réduire considérablement sa dimensionalité :

```

collapse(mean) countrypair startyear endyear failtime duration
max_outcome, by(spell_id)
gen lcensored = (startyear == 1)
gen rcensored = (endyear == 10)
gen fail2 = 1 - rcensored

```

	spell_id	indiv	startyear	endyear	duration	failtime	lcensored	rcensored	year	start	fail	outcome
1	1	1	2004	2004	1	2005	0	0	2004	1	0	34072.54
2	2	2	2004	2004	1	2005	0	0	2004	1	0	37856.48
3	3	3	2001	2001	1	2002	0	0	2001	1	0	3997.386
4	4	4	2002	2002	1	2003	0	0	2002	1	0	3914.268
5	5	5	2004	2004	1	2005	0	0	2004	1	0	12190.39
6	6	6	2006	2006	1	2007	0	0	2006	1	0	44220.06
7	7	7	2006	2006	1	2007	0	0	2006	1	0	46158.73
8	8	8	2003	2003	1	2004	0	0	2003	1	0	20646.93
9	9	9	2005	2005	1	2006	0	0	2005	1	0	8000.481
10	10	10	2002	2002	1	2003	0	0	2002	1	0	3421.351

On peut maintenant la déclarer comme base de données de survie par la commande stset :

```
stset failtime, id(spell_id) failure(fail2==1) origin(time startyear)
```

ce qui génère un certain nombre de variables ad hoc :

	spell_id	countrypair	startyear	endyear	failyear	duration	max_outcome	lcensored	rcensored	fail2	_st	_d	_origin	_t
1	1	1	1	1	2	1	.834618	1	0	1	1	1	1	1
2	2	1	5	10	11	6	2.130242	0	1	0	1	0	5	6
3	3	2	2	4	5	3	1.896178	0	0	1	1	1	2	3
4	4	2	9	9	10	1	.523202	0	0	1	1	1	9	1
5	5	3	1	2	3	2	.8336083	1	0	1	1	1	1	2
6	6	3	5	5	6	1	2.001825	0	0	1	1	1	5	1
7	7	3	9	9	10	1	.4624494	0	0	1	1	1	9	1
8	8	4	2	2	3	1	3700284	0	0	1	1	1	2	1

La variable `_st` indique les observations qui seront gardées pour l'analyse de survie (ici, toutes). La variable `_d` est égale à un pour les observations non censurées à droite (elle est égale à $1 - \text{rcensored}$). La variable `_origin` indique l'année de départ de la spell (elle est égale à `startyear`). Enfin `stset` génère une variable `_t0` ici égale à zéro pour toutes les observations—pas clair ce qu'elle est censée faire.

Les caractéristiques de la base sont résumées sur l'écran stata :

```
. stset failyear, id(spell_id) failure(fail2==1) origin(time startyear)
      id:      spell_id
failure event: fail2 == 1
obs. time interval: (failyear[_n-1], failyear]
exit on or before: failure
t for analysis: (time-origin)
origin:      time startyear
```

```
57 total obs.
1 ignored because spell_id missing
```

```
56 obs. remaining, representing
56 subjects
49 failures in single failure-per-subject data
86 total analysis time at risk, at risk from t =      0
   earliest observed entry t =      0
   last observed exit t =      6
```

Ce qui permet de vérifier un peu la bouille de tout le truc. On peut maintenant faire tourner une régression de Cox sur les données non censurées à gauche (la censure à droite est contrôlée par la procédure d'estimation) avec la commande

```
stcox max_outcome if lcensored==0, nohr
```

dans laquelle l'option `nohr` donne des coefficients plutôt que des ratios de taux de succès (c'est-à-dire des coefficients non exponentiés). On teste le postulat de proportionnalité des taux de succès avec un test de Schönfeld. Pour cela, il faut d'abord utiliser les options suivantes dans la régression :

```
stcox max_outcome start_outcome if lcensored==0, nohr schoenfeld(sch*)
scaledsch(sca*)
```

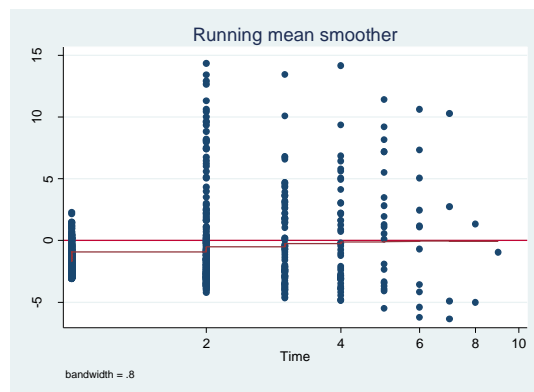
puis la commande

```
stphtest, log detail
```

qui donne l'output suivant

Test of proportional-hazards assumption				
Time: Log(t)				
	rho	chi2	df	Prob>chi2
max_outcome	0.35287	3.31	1	0.0688
start_outcome	-0.27380	2.55	1	0.1101
global test		3.40	2	0.1830

L'hypothèse nulle est celle de proportionnalité des taux de succès, sous laquelle la régression de Cox est valide. Ici, par exemple, elle est acceptée (p value = 0.1830) pour l'ensemble des régresseurs, mais pas pour `max_outcome`. On peut aussi faire un nuage des résidus de Schönfeld en fonction du temps (en logs) avec une régression 'smoother' (voir ci-dessous). L'hypothèse nulle est que la pente du smoother est zéro partout



Ce qui n'est pas forcément très facile à vérifier comme on peut le constater.

Dans le cas de temps discret et quand le postulat de proportionnalité des taux de succès est violé, l'alternative est de courir un probit en effets aléatoires (`xtprobit`) sur la base de données avant de l'avoir transformée en spells. Il faut simplement remplacer la variable d'outcome par « missing » l'année après la mort du spell. Pour fixer les idées, supposons qu'il s'agisse d'exportations. On garde toutes les années d'une spell à partir du moment où la valeur est positive (on ignore les années avant) plus la première années où elles retournent à zéro.

Graphiques

La commande de base est `twoway` suivie du type de graphique (line, bar, scatter, hist) et des variables, d'abord celle sur l'axe vertical puis celle sur l'axe horizontal. Pour sauvegarder un graphique, on tape

```
graph save graph1, replace
```

et pour le ré-ouvrir,

```
graph use graph1
```

Graphiques avec courbes ou barres

Pour un graphe avec des courbes c'est

```
graph twoway line y x, xscale(range(0 100)) yscale(range(0 100))  
xlabel(0(2)18)
```

où `xscale` et `yscale` donnent le minimum et le maximum sur les axes (utile par exemple si on veut une symétrie), `xlabel(0 10 to 100)` et `ylabel(0 10 to 100)` fixent les marques sur les axes (ici toutes les 10 unités), `c(1)` dit à Stata de relier les points par une ligne, et `s(i)` lui dit de rendre les marqueurs invisibles, ce qui donne une simple ligne. Les variables apparaissent automatiquement sous leur label et pas sous leur nom de variable si on a spécifié un label par la commande

```
label variable x "n'importe quoi"
```

Il y a plusieurs façons de mettre deux courbes sur le même graphique. En voici une :

```
twoway line y1 x, lpattern(1) xtitle() ytitle() || line y2 x,  
lpattern(_)
```

où `lpattern(1)` indique que la ligne représentant `y1` sera pleine et `lpattern(_)` que la ligne représentant `y2` aura des tirets longs.

Pour des barres avec ajustement de la couleur, c'est

```
twoway bar count revokekey, fcolor(gs15) xtitle(Revocation year) ytitle(#  
of cases) ;
```

où `gs15` donne un gris très clair (pour économiser l'encre de l'imprimante).

Pour un histogramme de la variable `x`, la commande la plus simple est

```
hist x, bin(10) xscale(range(0 100)) xtick(0(10)100) xlabel(0(10)100)
```

et le nom de la variable devra être mis dans l'option `xtitle("my title")`. Si on veut la densité à la place d'un histogramme, la commande est

```
kdensity x, n(100) width(0.01) normal ;  
twoway (kdensity GDPpc, gaussian width(0.01) n(100))
```

où l'option `n(100)` donne le nombre d'observations utilisées (plus il y en a moins la densité est lissée), `width(0.01)` détermine la largeur des intervalles (important pour éviter que le graphique ne bave sur des valeurs hors de l'intervalle permissible à droite et

à gauche) et l'option `normal` superimpose sur le tout une distribution normale (je suppose avec les mêmes moyennes et variances) pour amuser le lecteur.

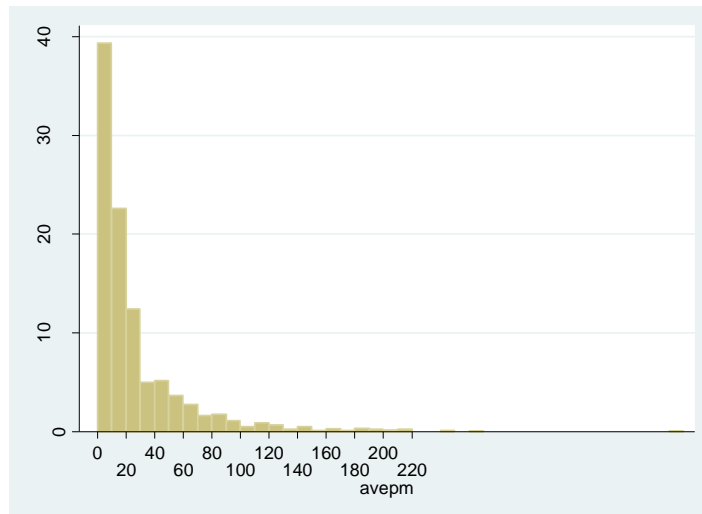
Si on veut raffiner la présentation, on peut donner des précisions sur l'apparence des lignes, ce qui peut être important quand on veut superimposer plusieurs courbes sur le même graphique ; exemple

```
twoway (kdensity x, lpattern(longdash) lwidth(medthick)) ///  
(kdensity y, lwidth(medthick) lcolor(red)) ///  
(kdensity z, lpattern(shortdash_dot) lwidth(medthick) lcolor(blue))
```

On peut aussi spécifier des sous-options. Ainsi dans

```
hist x, width(10) xlabel(0(20)220, alternate format(%9.0fc)) percent
```

la sous-option `alternate` à l'intérieur de `xlabel(0(20)220,...)` fait alterner la position des chiffres le long de l'axe pour qu'ils ne se chevauchent pas et la sous-option `format(%9.0fc)` élimine les décimales de ces chiffres, ce qui donne



On note que les étiquettes s'arrêtent à 220 alors que les données vont plus loin ; à éviter bien sûr.

Nuages de points

Pour un nuage de points la commande est

```
graph twoway scatter y x, xtitle() ytitle()
```

De la même façon, un scatterplot de y sur x avec la droite de régression sera obtenu par

```
reg y x  
twoway (scatter y x) (lfit y x)
```

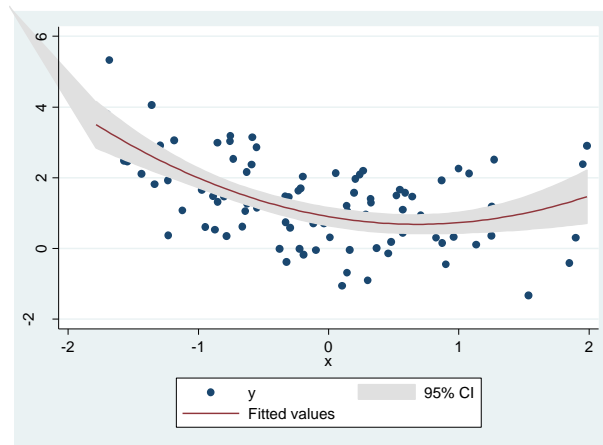
ou bien

```
reg y x
twoway scatter y x || lfit y x
```

Si on veut avoir une courbe pour une régression polynomiale, on peut taper

```
twoway (scatter y x) (qfit y x)
```

et si on veut l'intervalle de confiance autour des valeurs prédites on tape `lfitci` ou `qfitci` au lieu de `lfit` ou `qfit`, ce qui donne le joli graphe



Un nuage de points pour la corrélation partielle entre une variable explicative x et une variable dépendante y , contrôlant pour les autres variables explicatives (disons z) s'obtient soit en écrivant

```
reg y x z
gen bz = _b[z]
gen y_tilde = y - bz*z
twoway scatter y_tilde x || lfit y_tilde x
```

ou plus simplement

```
reg y x z
avplot x
```

mais la droite de régression ne sera pas en rouge et on ne peut pas raffiner avec `qfit` ou avec les intervalles de confiance par exemple.

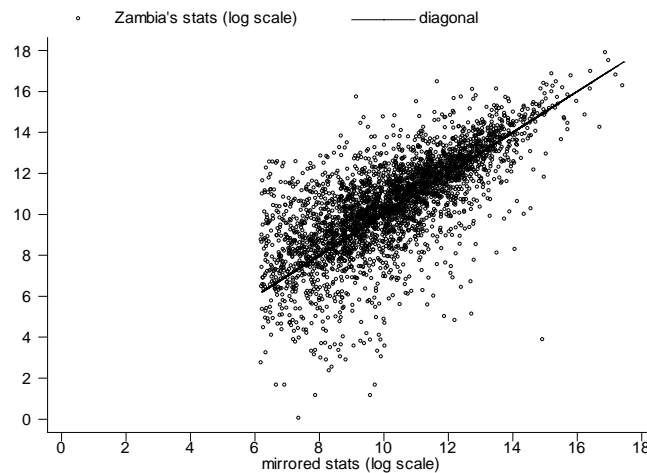
Lorsque l'on veut mettre l'un des axes (disons l'axe horizontal) en logs, on peut soit mettre la variable sur l'axe horizontal en logs avec `gen lx = ln(x)` et garder une échelle normale soit ajouter l'option `xscale(log)`. Si l'on choisit la première solution la droite de régression obtenue avec `lfit` a une drôle de tête, avec un coude à la fin. Si on choisit la seconde, la droite de régression a la bonne tête mais par contre la lecture est

moins facile (si on a des parts sur l'axe horizontal, par exemple, on aura des logs négatifs).

Supposons maintenant que l'on veuille un scatterplot de y par rapport à x avec une droite représentant la diagonale. On veut donc éviter de relier les observations de y en fonction de x (un nuage) mais par contre on veut relier les observations le long de la diagonale pour avoir une ligne à 45° . La série de commandes est alors

```
gen diagonal=x ;  
label variable y "Zambia's stats (log scale)" ;  
label variable x "mirrored stats (log scale)" ;  
twoway (scatter d x) (line diagonal x), xscale(range(0 18))  
yscale(range(0 18)) xlabel(0(2)18) ylabel(0(2)18) ;
```

Le tout donne

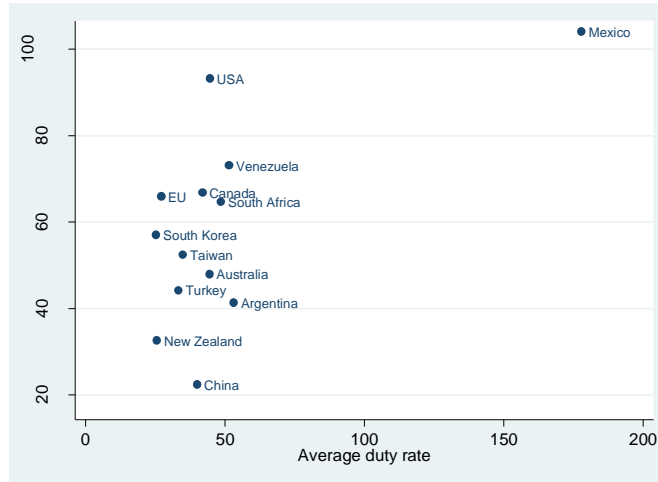


ce qui ne paye peut-être pas de mine mais en couleur est beaucoup mieux (pas la moindre idée comment récupérer les couleurs, par contre). En l'occurrence le nuage est tronqué horizontalement à 6 parce qu'il n'y a pas de valeur miroir au-dessous de $e^6=403'000$ dollars (pas de raison spéciale).

Pour un nuage de points avec pour chacun une étiquette donnée par la variable `cty_name`, la commande est

```
twoway scatter y x, mlabel(cty_name) legend(off)
```

Souvent le problème est que les étiquettes se chevauchent ou sont trop près, comme c'est le cas ici avec Canada et South Africa:

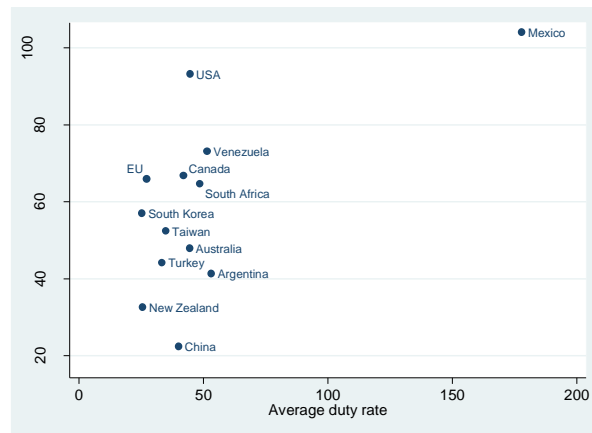


Le problème se résout en générant une variable, *pos*, dont la valeur est la position de l'étiquette par rapport au point exprimée en heure d'horloge (12 = en haut, 3 = à droite, 6 = en bas etc.), et en ajustant cette variable :

```
gen pos = 3
replace pos = 11 if cty_name=="EU"
replace pos = 4 if cty_name=="South Africa"
replace pos = 2 if cty_name=="Canada"

twoway scatter l_bar t_bar, mlabel(cty_name) mlabv(pos) legend(off)
```

ce qui donne



Si on ne veut des étiquettes que pour certains points, il faut définir une nouvelle variable comme ceci

```
gen show_tag =
  (cty_name=="ARG" | cty_name=="BRA" | cty_name=="BEL" | cty_name=="BRA"
  | cty_name=="CHL" | cty_name=="COL" | cty_name=="ECU" | cty_name=="PER"
  | cty_name=="PRY" | cty_name=="URY" | cty_name=="VEN") ;
```

```
gen str3 tag = cty_name if show_tag==1 ;
twoway scatter l_bar t_bar, mlabel(tag) mlab(pos) legend(off)
```

Si on veut des boulettes vides pour tous les pays et des boulettes rouges pour disons le botswana, la façon bourrin de le faire c'est

```
gen      t2 = t if countrycode == "BWA"
replace t = . if countrycode == "BWA"
label variable t  "Theil index"
label variable t2 "Theil index, Botswana"
```

```
twoway  scatter t  gdp_pc_PPP_2005, mfcolor(black) mlwidth(thin) ///
      || scatter t2 gdp_pc_PPP_2005, mfcolor(red)    mlwidth(thin)
```

L'option `mfcolor` c'est pour le « fill » des points, et `mlwidth` c'est pour leur entourage. Si à la place des points on veut avoir des bulles, disons bleues, avec une bordure mince, et dont la taille est proportionnelle à une troisième variable *z*, la commande devient

```
twoway scatter y x [weight = z], msymbol(circle) mfcolor(ltblue)
mlwidth(thin)
```

Si on veut des bulles et des étiquettes, il faut superimposer deux graphes, un nuage avec les étiquettes sans bulles et un avec bulles sans étiquettes, en mettant celui avec les étiquettes en dernier pour que les étiquettes ne soient pas cachées par les bulles :

```
twoway scatter y x [w=z], msymbol(circle) mfcolor(gs15) mlwidth(thin)
|| scatter y x, mlabel(cty_name) msymbol(none) mlabcolor(black)
```

Pour la couleur, `gs15` c'est gris clair, c'est le mieux.

Regressions non paramétriques (« smoother reg »)

Pour une régression non paramétrique d'une variable *x* sur les centiles d'une distribution (disons le coefficient de Gini calculé sur les exportations d'un pays, régressé sur le revenu par habitant pour voir comment la diversification évolue avec le revenu), la commande est

```
xtile centile = gdpcap, nquantiles(100) ;
collapse(mean) Gini gdpcap, by(centile) ;
lowess Gini centile, bwidth(0.8) ;
```

On revient sur cette méthode plus bas pour les effets de changements de prix par tranche de revenu.

Enquêtes sur les scènes de ménages

« The goverment are very keen on amassing statistics. They collect them, add them, raise them to the nth power, take the cube root and prepare wonderful diagrams. But you must never forget that every one of these figures comes in the first instance from the village watchman, who just puts down what he damn well pleases. »

Anonyme, cité dand Sir Josiah Stamp,
Some Economic Factors in Modern Life.⁹

Les commandes statistiques ordinaires (moyenne, regression etc.) appliquées à une enquête de ménages donnent des estimés biaisés (voir Deaton 1997). Stata a plusieurs commandes qui permettent de tenir compte de la structure de l'enquête.

Supposons que l'enquête soit divisée en différentes strates, que dans chaque strate, on ait tiré au sort un certain nombre de villages (qu'on appelle *Primary Sampling Unit*, ou *PSU*) et que dans chaque village un certain nombre de ménages soit tiré. Supposons aussi que la variable identifiant la strate s'appelle *strataid*, que celle identifiant le village s'appelle *id_comm* et que celle indiquant le poids du ménage s'appelle *poids*. La syntaxe pour indiquer cette structure à stata est

```
svyset [pweight=poids], strata(strataid) psu(id_comm)
```

Une fois que cette structure est déclarée on n'a pas besoin de la répéter à chaque commande d'estimation. La commande `svydes` donne une description de l'enquête sous forme de tableau (combien de strates, de PSU etc...)

Statistiques descriptives et manipulation de données

Moyennes, totaux et corrélations

La commande de moyenne est `svymean` et la commande de somme est `svytotal`. Celle-ci n'est pas exactement équivalente à `sum` car on ne peut pas enregistrer le résultat comme une nouvelle variable comme dans `egen somme = sum(output)`. A la place, la procédure est

```
svytotal output;  
mat A=e(b);  
svmat A;  
rename A1 somme;
```

La première ligne calcule la somme de *output* mais n'enregistre pas le résultat qui est stocké momentanément dans *e(b)*. La deuxième ligne génère un scalaire *A* égal à *e(b)*,

⁹ Toujours le manuel de Shazam...

dont la troisième fait une variable (*A1*) que la dernière renomme « somme ». Cette nouvelle variable est la somme sur toutes les observations de la variable *output* dans laquelle chaque observation est pondérée selon la structure de l'enquête. Pour trouver la moyenne, même chose avec `svymean` au lieu de `svytotal`.

Les moyennes conditionnelles ne peuvent pas non plus se faire avec la commande *by*, comme par exemple `by year: egen average=mean(output)`. On est obligé de faire ça catégorie par catégorie, par exemple

```
svymean output if year==1993;
mat B=e(b);
svmat B;
rename B1 average93;
```

et ainsi de suite pour chaque année.

Supposons que l'on veuille calculer les valeurs médianes d'un certain nombre de variables. Il y a une façon subtile d'utiliser la procédure introduite pour des variables indicées :

```
foreach x in "lnfamily" "lnchefmage" "lnchefmage2"
"chefmeduc" "lnmembresage" "inputs" "lndepenses"
"lnparcellesexpl" {;
gen `x'x=`x';
```

On note que la procédure `foreach` est ici utilisée avec des variables sans indice, le *x* étant la variable elle-même ; les huit variables générées sont `lnfamilyx`, `lnchefmagex` etc. Ensuite on refait une procédure similaire pour les indiquer par année

```
foreach j of numlist 1993 1997 1999 2001{;

/*egen `x'p`j'=median(`x') if year==`j';*/

svymean `x' if year==`j';
mat A=e(b);
svmat A;
egen `x'p`j'=mean(A1);
mat drop A;
drop A1;
};
};
```

Les corrélations sont difficiles à calculer en tenant compte des strates, clusters etc... Or si l'on n'en tient pas compte les estimés risquent d'être gravement biaisés. La commande pour les calculer est

```
corr_svy
```

qui peut être descendue du web par `findit corr_svy`.

Calculer des indices d'inégalité

Il y a quelques trucs à descendre du web en commençant par `search ineqerr`, d'où Stata dirige sur son site pour descendre les fichiers nécessaires. Il y a aussi une commande `inequal` mais je ne sais pas quelle est la différence. La commande `ineqerr` donne le coefficient de Gini et l'indice d'entropie de Theil. Je crois qu'elle admet les poids de redressement et tous les bidouillages utilisés dans les enquêtes de ménage mais pas très clair comment. La syntaxe pour calculer par exemple les indices d'inégalité pour le revenu des ménages ruraux (*income* avec une variable muette *rural* égale à un) est

```
ineqerr income if rural==1
```

L'output donne aussi les écarts-type des estimés de ces indices calculés par bootstrapping (par défaut, le nombre de tirages est 100).

La commande en Stata 7 pour les courbes de Lorenz est `glcurve7` et la syntaxe est la suivante. Supposons que les poids de redressement (*fw*) soient donnés par la variable *wgt* :

```
glcurve7 income fw=wgt if rural==1, Lorenz saving lorenz1.wmf ;
```

Supposons que l'on veuille comparer la courbe de Lorenz en 1993 et 1997. L'option `saving lorenz1.wmf` sert à exporter le graphique en format *.wmf*, qui s'ouvre en ACDSee ou autre chose et peut être copié-collé dans word ou importé dans latex.

Si la base de données contient une variable *year* égale à 1993 ou 1997 selon les observations (on suppose qu'on a mélangé deux enquêtes de ménages), on définit dans la commande une variable *gl(income)* que Stata va appeler, dans le graphe, *income_1993* ou *income_1997*, et on lui précise *by(year)* et *split* :

```
glcurve7 income fw=wgt if rural==1, gl(income) by(year) split Lorenz  
saving lorenz1.wmf ;
```

et on a les deux courbes sur le même graphique.

Densités

Pour tracer une distribution du revenu (variable *income*), on utilise la commande de « kernel density » de la façon suivante :

```
kdensity income [fweight=poids], nogr gen(y fy) ;
```

Quelquefois la densité est écrasée par des outliers. On lui met alors une borne supérieure avec `if income<=level`. Les intervalles sur lesquels la densité est calculée sont ajustés par l'option `width(#)` où *#* est exprimé en termes de *income* (la variable sur l'axe horizontal). Plus il est petit moins la densité est lissée. Il est possible que la densité

démarre à un niveau de revenu négatif « inventé », c'est juste que l'intervalle sur lequel la première observation est centrée déborde sur les chiffres négatifs. Il se peut aussi que la courbe soit plus lissée sans les poids, c'est à voir.

Si on veut plusieurs densités sur le même graphique, la série de commandes est un peu plus compliquée :

```
kdensity income [fweight=poids] if income<=level, nogr gen(y fy) ;
kdensity income [fweight=poids] if year==1993 & income<=level93, nogr
gen(fy93) at(y) ;
kdensity income [fweight=poids] if year==1999 & income<=level99, nogr
gen(fy99) at(y) ;
label var fy93 "1993" ;
label var fy99 "1999" ;
gr fy93 fy99 y, c(ll) xlab ylab ;
```

La première commande génère la densité pour les deux années et les variables *y* (*income*) et *fy* (fréquence). La deuxième et la troisième font la même chose pour les années 1993 et 99 séparément. Les deux commandes *label* font apparaître les labels plutôt que les noms de variables sur le graphique ; enfin la dernière génère le graphique combiné, avec des lignes (ll) dans la commande *c(ll)* (*c* est pour « connect ») et les labels sur les axes horizontal (*xlab*) et vertical (*ylab*). Les symboles sur les courbes (triangles etc.) sont choisis avec la commande *s()*.

Effet de changements de prix par tranche de revenu

Une procédure sympa consiste à tracer une courbe représentant l'impact d'un changement de prix par tranche de revenu (disons centiles). La procédure est la suivante. Soit 1993 l'année de base sur laquelle on analyse l'impact du changement de prix (*delta_p*) sur le revenu (*income*). La première commande génère les centiles de la distribution du revenu :

```
xtile centile = income, nquantiles(100) ;
```

puis pour chaque centile on calcule le *delta_p* moyen

```
foreach j of numlist 1/100{ ;
svymean delta_p if centile==`j' ;
```

et on sauvegarde le résultat par la procédure habituelle

```
mat A=e(b) ;
svmat A ;
rename A1 delta_p`j' ;
mat drop A ;
```

On a alors cent variables $\delta_{p,j}$ que l'on veut combiner en une seule. Problème: elles ont toutes le même format, à savoir un chiffre positif sur la première ligne (première observation) et des valeurs manquantes pour tout le reste. On les transforme alors en générant cent nouvelles variables $\delta_{2,p,j}$, des constantes dans lesquelles toutes les valeurs manquantes sont remplacées par la valeur positive de la première ligne:

```
egen delta2_p`j'=mean(delta_p`j') ;
```

puis on transforme une nouvelle fois ces constantes pour qu'elles soient égales à cette valeur positive seulement dans le bon centile j (et codées comme valeurs manquantes pour tous les autres centiles):

```
egen delta3_p`j'=mean(delta2_p`j') if centile==`j' ;
```

puis on laisse tomber les variables de calculs intermédiaires et on ferme l'accolade car on n'a plus besoin de l'indice j :

```
drop delta_p`j' delta2_p`j' ;  
} ;
```

et on combine ces cent variables $\delta_{3,p,j}$ en une seule en en faisant la somme horizontale (avec `egen`, ce qui permet d'additionner "à travers" les valeurs manquantes)

```
egen newdelta_p=rsum(delta3_p1-delta3_p100) ;
```

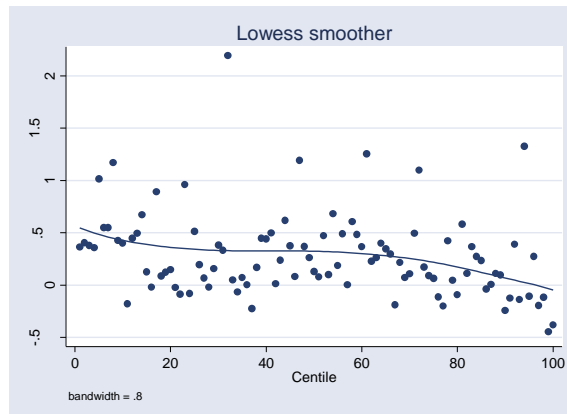
enfin on garde seulement ce dont on a besoin (on ne met ici que newdelta_p et centile mais on peut vouloir garder d'autres variables pour autre chose!)

```
collapse newdelta_p, by(centile) ;
```

et finalement on régresse newdelta_p sur centile en utilisant le lowess smoother qui recalcule la pente de la droite de régression pour chaque observation en prenant 40% de l'échantillon à gauche de l'observation et 40% à droite, soit 80% en tout (ce qui peut être ajusté avec `bwidth()`).

```
lowess newdelta_p centile, bwidth(0.8) ;
```

Le résultat est une courbe indiquant que le changement de prix est favorable aux pauvres si elle est à pente négative et favorable aux riches si elle est à pente positive. Une procédure similaire peut être utilisée pour calculer l'effet d'un tarif par tranche de revenu (le δ_p est alors l'effet du tarif sur le revenu réel moyen des ménages dans la tranche, soit via les prix à la consommation soit via les revenus soit les deux). Exemple :



Ici (Madagascar 1999-2001) le changement des prix à la production est une fonction négative des centiles (classés par ordre croissant de revenu, les plus riches à droite), donc il est « pro-pauvre ». On note la distribution uniforme des points le long de l'axe horizontal : c'est parce que l'on a bien régressé les Δp sur les centiles et non sur les niveaux de revenu. Pas très intuitif, certes, mais on s'aperçoit vite à l'usage que la régression sur les niveaux de revenu donne un graphique complètement différent. A noter, aussi, avant de s'exciter, que le sous-échantillon « glissant » recalculé par Stata pour chaque observation est plus petit aux extrêmes qu'au milieu (bien nécessairement, sinon il ne pourrait pas être différent pour chaque observation). Les non-linéarités plus marquées observées souvent près des extrêmes en sont le reflet, donc à interpréter prudemment.

Pour s'amuser voici un do-file qui fait la procédure (avec les commandes à jour pour Stata 12) sur un toy sample

```
/* THIS FILE GENERATES A SMOOTHER REG */

clear
set more off

* Create bogus data with 1000 people and lognormal income distribution

set obs 1000
gen lincome = invnorm(uniform())
gen income = exp(lincome)
gen delta_p = uniform() + 0.1*income

* Give it HH survey format, with psu as indiv. and random[0,1] sampling weights

gen psu = _n
gen weight = uniform()
svyset psu [pweight=weight]

* generate delta p by centile

xtile centile = income, nquantiles(100)
sort centile income
```

```

foreach j of numlist 1/100{
    svy: mean delta_p if centile==`j'
    mat A=e(b)
    svmat A
    rename A1 delta_p`j'
    mat drop A
    egen delta2_p`j'=mean(delta_p`j')
    replace delta2_p`j'= 0 if centile!=`j'
    drop delta_p`j'
}

egen newdelta_p = rsum(delta2_p1-delta2_p100)
collapse newdelta_p, by(centile)

* Run smoother reg

ksm newdelta_p centile, bwidth(0.25)

```

Et voila.

Estimation sur des enquêtes

Une fois que l'on a indiqué à Stata la structure de l'enquête avec `svyset/svydes`, on peut utiliser les commandes spéciales (`svyreg`, `svyprobit`, `svyivreg`, `svyheckman`,...) sans avoir à préciser à chaque fois les poids, les clusters et les strates de l'échantillon, par exemple

```
svyreg y x1 x2 x3 ;
```

Si par contre on n'a rien déclaré au début comme structure d'enquête, la commande est

```
svyreg [pweight=poids], strata(strataid) psu(id_comm);
```

Modèles de sélection

Supposons que l'on veuille estimer sur des données d'enquête le modèle

$$\begin{aligned}
 y &= X\beta + u_1, \\
 I^* &= Z\gamma + u_2, \\
 I &= \begin{cases} 1 & \text{si } I^* > 0 \\ 0 & \text{sinon,} \end{cases}
 \end{aligned}$$

avec u_1 and u_2 suivant une distribution normale avec moyenne zéro et variances σ^2 et un respectivement, et corrélation ρ . La version la plus simple de la commande est

```
svyheckman y x1 x2 x3, select(I=x1 x2 z1 z2)
```

et bien entendu de nombreux raffinements peuvent être ajoutés.

Quelques trucs en Mata

Pour construire une matrice mata avec des variables sauvegardées dans un fichier stata, la commande est

```
st_view(X=.,., ("x1", "x2"))
```

où le premier argument donne le nom, le deuxième lui dit de prendre toutes les observations, et le troisième définit les colonnes par le nom des variables. La fonction `st_view` fait la même chose que la fonction `st_data` mais prend moins de mémoire car elle ne fait que « regarder » les données au lieu de les stocker séparément.

Pour générer des variables mata contenant le nombre d'observations et de paramètres, c'est

```
N = rows(X)
k = cols(X)
```

Pour générer le vecteur beta en mata après une regression stata, c'est pareil:

```
reg y x1 x2
mat coeff = e(b)
svmat coeff

mata
    st_view(b=.,1, ("coeff1", "coeff2", "coeff3"))
end
```

où le deuxième argument lui dit de ne garder que la première observation, ce qui transforme les variables stata créées par `mat coeff = e(b)` en un vecteur-ligne 1 x k.

Si on veut calculer en mata l'estimateur des moindres carrés à partir d'un échantillon fabriqué pour s'amuser, voilà le programme :

```
set obs 100 ;
gen const = 1 ;
gen x1 = invnormal(uniform()) ;
gen x2 = invnormal(uniform()) ;
save regressors, replace ;

use regressors, clear ;
gen u = 3*invnormal(uniform()) ;
gen y = 2 + 3*x1 -4*x2 + u ;

mata ;

st_view(y=.,., "y") ;
```



```

st_view(X=.,., ("const", "x1", "x2")) ;
b = invsym(X'X)*X'y ;
b ;

e = y - X*b ;
n = rows(X) ;
k = cols(X) ;
s2 = (e'e)/(n-k) ;
s2 ;

end ;

```

Pour définir un vecteur-colonne $m \times 1$ contenant des « 0 », on peut taper

```

real matrix iota(real scalar m) {
    real matrix iota
    iota = J(m,1,0)
    return(iota)
}

```

Une variable contenant la trace d'un produit de matrices se définit par

```
t = trace(A,B)
```

pour $\text{tr}(AB)$ et

```
t = trace(A,B,1)
```

pour $\text{tr}(A'B)$.

Pour introduire une commande stata dans mata, c'est

```

{
stata(`"drop _all"')
}

```

Je n'ai pas trouvé comment faire de programme mata dans un programme : deux « end » l'un après l'autre le font capoter. Il faut donc mettre toutes les commandes mata une par une avec le préfixe mata : au début de chaque commande.