Review of Stata II
AERC Training Workshop
Nairobi, 20-24 May 2002

This note provides more information on the basics of Stata that should help you with the exercises in the remaining sessions of the workshop. We will cover the following topics:

A. Submitting batch files (i.e. -do- files)
B. More on data manipulation
C. Dropping outliers
D. Basic regression analysis and post estimation results
E. Basics of matrices

A. Submitting Batch Files (i.e. –do- files)

While we can submit commands to Stata one-by-one interactively, we can also submit a bunch of commands in the form of batch files, or in Stata-speak, -do- files. This is very useful when (if you are at all like me) you make a coding error early in a series of commands. If you submit the commands as a -do- file, then you need only go back to the file, change the coding mistake, and resubmit the commands.

The way it works is that you create a text file using your favorite text editor (e.g. Notepad) and save it with the extension -do- (e.g. c:\pathname\myprog.do). You enter the commands in this file, save it, open up Stata, and from the Stata Command window type `do c:\pathname\myprog.do`. You can also change the directory to the one in which your file is located (e.g. type `cd c:\pathname`) and then type `do myprog.do`.

It is generally a good idea to create a -log- file that records all of the output that appears on the screen. AND, it's generally a good idea to give it the same name as your -do- file, but give it a '.log' extension. Here is an example of what a -do- file looks like (you'll see more in successive sessions):

```
version 7.0
clear
set memory 5m
#delimit ;

capture log close;
log using c:\pathname\myprog.log, replace;

* * * * * * * * * * * * * * * * * * * *
  c:\pathname\myprog

  Describe what the program does
* * * * * * * * * * * * * * * * * * * *;

use c:\datalocation\mydata.dta;

 -- commands . . ;

log close;
```

There are six things to note from the example:

1. *Version 7.0:* Since some Stata commands change as new versions of the software come out, it's a good idea to let Stata (and other users of your -do- file) know for which version of the program you are writing your commands.
2. *Memory:* By default, Stata allocates 1 megabyte of memory for storing data. If your dataset is larger than this, you need to change this allocation by setting the memory. In this example, we're setting the memory for data to 5 megabytes.
3. *#delimit ; :* By default, a carriage return (i.e. when you hit *Enter*) denotes the end of a command line. In the case of a -do- file, the end of the line is the end of a command – that is, unless you tell Stata otherwise. In this example, we assign a semi-colon (`;`) to denote the end of the line by including the "`#delimit ;`" command. Note that in the first two lines of the -do- file, the end of the line denotes the end of the command.
4. *capture log close:* Stata will not open a -log- file if another one is open. This line just tells Stata to close a -log- file if one is open.
5. **:* The '*' is used to cancel out a command line. This is useful because it lets us insert comments.
6. *log close:* This command tells Stata that this is the end of the -log- file, and to close it. This should not be confused with `log off`, which tells Stata to suppress output in the -log- file until the `log on` command is given.

An important thing to do when you write a -do- file is to annotate it so that if other's use your program or you come back to it after a month or so, it will be easy to understand what you did. This can be done by "starring" (*) out a command line and inserting your comments…

```
* comments here ;
```

Note that if you are using "`;`" to denote the end of the command line, then comments can take on multiple lines without "re-starring"…

```
* comments here
  and more comments here ;
```

Let's complete the remainder of this exercise by using a -do- file. Instead of writing each command in the **Stata Command Window**, we'll add them to our -do- file and then re-run it. To get this started, follow these steps:

1. Start Stata
2. Change the working directory to `c:\aerc\stata_review\programs\`
3. Open the **Do File-file Editor** (click on **Window**, then click on **Do File Editor**) or some other text editor that you prefer.
4. Save the file as `c:\aerc\stata_review\programs\exercise.do`
5. In this file write the following…

```
version 7.0
clear
set memory 5m

capture log close
log using c:\aerc\stata_review\programs\exercise.log, replace

#delimit ;
*********************************************

  c:\aerc\stata_review\programs\exercise.do

  Exercises for the second AERC Stata training
  session on Monday, May 20, 2002, Nairobi

*********************************************;
#delimit cr;

use c:\aerc\stata_review\data\anthmodel.dta
sum
describe


log close
```

6. Re-save the file
7. In the **Stata Command Window**, type `do exercise` (or in the **Stata Do-file Editor**, click on **Tools**, and then **Do**) and see what happens.

B. More on Data Manipulation:

We will eventually use the data that is now in memory to estimate very simple reduced form nutrition production functions to illustrate how to use Stata's regress command and to see what Stata produces in the realm of post-estimation results. For now though, we want to create some additional variables to include as explanatory variables. This will also give us an opportunity to run across some very useful commands. Let's start by creating region dummies…

1. ***tabulate, generate():***

One way to create region dummies is to use the `tabulate` command and the `generate()` *option*. Suppose that we want to name our region dummies: `reg1`…`reg7`. Then in our -do-file, insert the following line after `describe;` …

```
* Create region dummies with tabulate
tab region, gen(reg)
```

re-save the file, and in the **Stata Command Window**, type `do exercise` and see what happens. The following should appear in the **Stata Results Window**…

```
. * Create region dummies with tabulate
. tab region, gen(reg)

     Region |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |        463       18.32       18.32
          2 |        524       20.74       39.06
          3 |        369       14.60       53.66
          4 |        288       11.40       65.06
          5 |        120        4.75       69.81
          6 |        270       10.68       80.49
          7 |        493       19.51      100.00
------------+-----------------------------------
      Total |       2527      100.00
```

…which, of course, gives no indication that any dummy variables were made. BUT, look in the **Variables Window** and you will find 7 new variables. These can be `describe`d as…

```
. describe reg1-reg7

  16. reg1      byte   %8.0g                 region==     1.0000
  17. reg2      byte   %8.0g                 region==     2.0000
  18. reg3      byte   %8.0g                 region==     3.0000
  19. reg4      byte   %8.0g                 region==     4.0000
  20. reg5      byte   %8.0g                 region==     5.0000
  21. reg6      byte   %8.0g                 region==     6.0000
  22. reg7      byte   %8.0g                 region==     7.0000
```

… which is what we wanted.

2. ***for:***

Although we have already shown a very efficient way of producing region dummies, let's see how we can use the `for` command to do the same thing. Let's name these dummies `regi1`…`regi7`. Add the following lines to your -do- file and we'll see what `for` does before we explain it's syntax:

```
* Create region dummies with for
for num 1/7: gen int regiX = (region==X)
describe regi*
```

The output from this gives and idea of what `for` does…

```
. * Create region dummies with for
. for num 1/7: gen int regiX = (region==X)

->  gen int regi1 = (region==1)

->  gen int regi2 = (region==2)

->  gen int regi3 = (region==3)

->  gen int regi4 = (region==4)

->  gen int regi5 = (region==5)

->  gen int regi6 = (region==6)

->  gen int regi7 = (region==7)

. describe regi*;

  11. region    byte   %10.0g                 Region
  23. regi1     int    %8.0g
  24. regi2     int    %8.0g
  25. regi3     int    %8.0g
  26. regi4     int    %8.0g
  27. regi5     int    %8.0g
  28. regi6     int    %8.0g
  29. regi7     int    %8.0g
```

What the command line

```
for num 1/7: gen int regiX = (region==X)
```

tells Stata to do is to repeat the command on the right of the colon (:) for each number from 1 through 7. AND in each repetition, Stata is to replace the `X` with the appropriate number. So in the first repetition, where the value of the number is 1, then Stata reads

```
gen int regiX = (region==X)
```

as

```
gen int regi1 = (region==1)
```

while in the second repetition, where the number takes on a value of 2, Stata reads it as

```
gen int regi2 = (region==2)
```

and so on through 7. Notice, however, that the variables don't have labels. We can use `for` to define labels for them as well…

```
for num 1/7: label variable regiX "Dummy for region X"
describ regi1-regi7
```

which gives the following output…

```
. for num 1/7: label variable regiX "Dummy for region X"

->  label variable regi1 `"Dummy for region 1"'

->  label variable regi2 `"Dummy for region 2"'

->  label variable regi3 `"Dummy for region 3"'

->  label variable regi4 `"Dummy for region 4"'

->  label variable regi5 `"Dummy for region 5"'

->  label variable regi6 `"Dummy for region 6"'

->  label variable regi7 `"Dummy for region 7"'

. describ regi1-regi7

  23. regi1     int    %8.0g                 Dummy for region 1
  24. regi2     int    %8.0g                 Dummy for region 2
  25. regi3     int    %8.0g                 Dummy for region 3
  26. regi4     int    %8.0g                 Dummy for region 4
  27. regi5     int    %8.0g                 Dummy for region 5
  28. regi6     int    %8.0g                 Dummy for region 6
  29. regi7     int    %8.0g                 Dummy for region 7
```

Note that we could have written these two sets of commands (`gen` and `label variable`) using one `for` statement…

```
#delimit ;
for num 1/7: gen int regiX = (region==X)
           \ label variable regiX "Dummy for region X";
#delimit cr;
```

Before we describe the basic syntax for `for`, let's see how we can use it to compare the dummy variables that we created using `tabulate` and `for`. Of course we could simply type

```
for num 1/7: gen int regXerr = (regiX~=regX)
```

but to illustrate another feature of for, type the following command into your -do- file and run it…

```
* Check that the two sets of dummies are equivalent
#delimit ;
for var regi1-regi7 \ var reg1-reg7:
        gen int Yerr = (X~=Y)
     \ label variable Yerr “1 if X is not equal to Y”;
#delimit cr;
```

Again, inspecting the output is probably the easiest way to understand what `for` has just done:

```
. * Check that the two sets of dummies are equivalent;
. #delimit ;
delimiter now ;
. for var regi1-regi7 \ var reg1-reg7:
>      gen int Yerr = (X~=Y)
>    \ label variable Yerr "1 if X is not equal to Y";

->  gen int reg1err = (regi1~=reg1)

->  label variable reg1err `"1 if regi1 is not equal to reg1"'

->  gen int reg2err = (regi2~=reg2)

->  label variable reg2err `"1 if regi2 is not equal to reg2"'

->  gen int reg3err = (regi3~=reg3)

->  label variable reg3err `"1 if regi3 is not equal to reg3"'

->  gen int reg4err = (regi4~=reg4)

->  label variable reg4err `"1 if regi4 is not equal to reg4"'

->  gen int reg5err = (regi5~=reg5)

->  label variable reg5err `"1 if regi5 is not equal to reg5"'

->  gen int reg6err = (regi6~=reg6)

->  label variable reg6err `"1 if regi6 is not equal to reg6"'

->  gen int reg7err = (regi7~=reg7)

->  label variable reg7err `"1 if regi7 is not equal to reg7"'

. #delimit cr
delimiter now cr
```

```
. sum reg1err-reg7err

Variable |     Obs        Mean   Std. Dev.       Min        Max
---------+--------------------------------------------------------
 reg1err |    2527           0          0          0          0
 reg2err |    2527           0          0          0          0
 reg3err |    2527           0          0          0          0
 reg4err |    2527           0          0          0          0
 reg5err |    2527           0          0          0          0
 reg6err |    2527           0          0          0          0
 reg7err |    2527           0          0          0          0
```

As with the previous commands, we asked Stata to go through 7 repetitions. This time, however, the repetitions were defined over the 7 variables `regi1`…`regi7`, instead of numbers (hence the `var` instead of the `num` immediately after the `for`). In addition, we added an additional list, `reg1`…`reg7`, represented by the identifier `Y`, that is run in parallel with the `X` identifier. Stata allows up to 9 parallel lists with `X, Y, Z, A,… F` as identifiers for the 1st, 2nd, 3rd, 4th, … 9th lists, respectively.

Of course, instead of creating the error variables, we could just as easily have used the compare `command`:

```
for var regi1-regi7 \ var reg1-reg7: compare X Y
```

Now that we have an idea of what's going on, let's lay out the general syntax:

```
for type1 list1 \ type2 list2 \ … :   stata_cmd_containing_X
                                    \ stata_cmd_containing_Y
                                    \ …
```

So when there is only one list, then this becomes:

```
for type1 list1: stata_cmd_containing_X
```

Now if	`type` is a	then `list` is a
	`var`	list of existing variables
	`new`	list of new variables
	`num`	list of numbers
	`any`	list of words

OK, so `tabulate` is obviously a more efficient way to create dummy variables. Nonetheless, you will find that the `for` command is very useful and can save you much time and effort in what may be cumbersome coding.

3. ***`while`****:*

We can do the same thing by looping through the data using Stata's `while` statement. The barebone structure of a `while` statement is…

```
local i = min
```

```
while `i' <= max {
  command statements using `i' in them
  local i = `i' + 1
}
```

So the idea is that we use `i` as the index over which we'll loop, so we start by setting `i` to the minimum. We are defining `i` as a local scalar which is more important when writing canned programs. Nonetheless, it's good practice to do it this way rather than as a standard scalar. Then we define the condition that stops the looping through the data. Here we tell it to stop when `i` is less than some maximum value. Note that when we evaluate a local we must incase it in single quotation marks (`` `i' ``). Keep in mind that the left quotation mark is `` ` ``, and the right one is `'`. Now within the `{` and the `}`, are the command statements that incorporate the index `` `i' ``. The last line in the brackets augments the index by one. If you forget this line, `` `i' `` will never change and you'll get stuck in an endless loop.

Let's see how we might use this to create our region dummies…

```
local i = 1
while `i' <= 7 {
  gen byte regio`i' = (region == `i')
  label variable regio`i' "Dummy for region `i'"
  local i = `i' + 1
}
```

You can see that it is very similar to the concept of the `for` command. Again, `tab` is much easier for purposes of creating dummy variables. But this gave us an opportunity to see how the `while` statement works.

Exercise 1: Create a series of interactions between the urban dummy in the dataset and the following variables: `age`, `sex`, `bord`, `hhmemb`, *and your region dummies.*

Exercise 2: Use the dataset in memory and `sect01a.dta` *in the same directory to create a variable that indicates if the father or the mother of the child is the household head. (OK, so this doesn't have anything to do with using* `for` *or* `tabulate`*. What it does is to give you a chance to practice merging!)*

C. Dropping outliers:

A couple of things we want to do before we estimate any model is to check for outliers. There are a couple of helpful commands that can give us an idea of what the extreme values look like. Some of these include:

```
sum haz, detail

codebook haz

spikeplt haz

hist integer_variable

inspect haz
```

Deciding if observations are outliers is generally a judgment call which we won't go into here. But suppose that we decide to drop those observations below the 1st percentile and above the 99th percentile, then we can take advantage of `sum, detail`, and it's post estimation results. In your -do- file, include the following commands and run them…

```
* Drop outliers (below 1st and above 99th percentiles)
sum haz, detail
return list
```

So we see that Stata saves the 1st and 99th percentiles as scalars (in memory) after `sum, detail`...

```
. * Drop outliers (below 1st and above 99th percentiles)
. sum haz, detail

                          HAZ-score
-------------------------------------------------------------
      Percentiles      Smallest
 1%        -5.14          -5.92
 5%        -4.19          -5.84
10%        -3.64          -5.81       Obs                2527
25%        -2.82           -5.8       Sum of Wgt.        2527

50%           -2                      Mean           -1.898773
                        Largest       Std. Dev.       1.474881
75%        -1.09           4.83
90%          .01           4.99       Variance        2.175274
95%          .72            5.1       Skewness        .5852376
99%         2.21           5.47       Kurtosis        4.398814

. return list

scalars:
       r(N)        =  2527
       r(sum_w)    =  2527
       r(mean)     =  -1.898773248632098
       r(Var)      =  2.175273791427845
```

```
        r(sd)       =  1.474880941441662
        r(skewness) =  .585237591143191
        r(kurtosis) =  4.398814101760292
        r(sum)      =  -4798.199999293312
        r(min)      =  -5.920000076293945
        r(max)      =  5.46999979019165
        r(p1)       =  -5.139999866485596
        r(p5)       =  -4.190000057220459
        r(p10)      =  -3.640000104904175
        r(p25)      =  -2.819999933242798
        r(p50)      =  -2
        r(p75)      =  -1.090000033378601
        r(p90)      =  .0099999997764826
        r(p95)      =  .7200000286102295
        r(p99)      =  2.210000038146973
```

… these are represented by `r(p1)` and `r(p99)`. If we add one more command, we can drop the "outliers" as we have defined them:

```
drop if haz < r(p1) | haz > r(p99)
```

Once we also check for missing values among our dependent and independent variables, we can move onto estimating a model using the `regress` command.

D. Basic Regression Analysis and Post Estimation Results:

The most basic syntax for `regress` is

```
regress depvar varlist [weight] [if] [in] [, options]
```

The first variable after the regress command is always the dependent variable (or left-hand-side variable), and the remaining variables listed are the relevant independent variables. Let's try this using *"haz"* as the dependent variable:

```
* Estimate simple reduced form nutrition production
* function using regress;
regress haz age sex bord hhmemb urban reg2-reg7
estimates list
```

We see from the results that Stata saves many results from the estimation in the form of scalars and matrices…

```
. * Estimate simple reduced form nutrition production
. * function using regress
. regress haz age sex bord hhmemb urban reg2-reg7

  Source |       SS       df       MS                  Number of obs =    2478
---------+------------------------------               F( 11,  2466) =   34.65
   Model |  599.483487    11  54.4984988               Prob > F      =  0.0000
Residual |  3878.05748  2466  1.57261049               R-squared     =  0.1339
---------+------------------------------               Adj R-squared =  0.1300
   Total |  4477.54096  2477  1.80764674               Root MSE      =   1.254

------------------------------------------------------------------------------
     haz |      Coef.   Std. Err.       t     P>|t|       [95% Conf. Interval]
---------+--------------------------------------------------------------------
     age |  -.0229879   .0014914    -15.413   0.000      -.0259125   -.0200633
     sex |   .0404056   .0505052      0.800   0.424      -.0586314    .1394426
    bord |  -.0316171   .0152908     -2.068   0.039      -.0616012   -.0016329
  hhmemb |  -.0102254   .0102893     -0.994   0.320      -.0304019    .0099512
   urban |   .4091839   .0749723      5.458   0.000       .2621688    .5561991
    reg2 |   .2160381   .0812244      2.660   0.008       .0567631    .3753132
    reg3 |  -.0091246   .0882509     -0.103   0.918       -.182178    .1639288
    reg4 |   .2909512   .0966383      3.011   0.003       .1014507    .4804518
    reg5 |   .0733914   .1315888      0.558   0.577      -.1846445    .3314273
    reg6 |   .7873126   .0995141      7.912   0.000       .5921727    .9824525
    reg7 |    .423852    .082641      5.129   0.000       .2617992    .5859049
   _cons |  -1.392443   .0999404    -13.933   0.000      -1.588419   -1.196467
------------------------------------------------------------------------------

. estimates list

scalars:
       e(N)        =  2478
       e(df_m)     =  11
       e(df_r)     =  2466
       e(F)        =  34.65479791258936
       e(r2)       =  .1338867677823041
       e(rmse)     =  1.254037676222003
       e(mss)      =  599.483487277947
       e(rss)      =  3878.05747668564
       e(r2_a)     =  .1300233267626794
       e(ll)       =  -4071.05622014205
       e(ll_0)     =  -4249.149616544096

macros:
       e(depvar)   : "haz"
       e(cmd)      : "regress"
```

```
       e(predict)  : "regres_p"
       e(model)    : "ols"

matrices:
       e(b)        : 1 x 12
       e(V)        : 12 x 12

functions:
       e(sample)
```

Later when we discuss matrices, we will see how to use the matrix of saved coefficients (`e(b)`) and the variance-covariance matrix (`e(V)`). One way that we can get at the coefficients and at the standard errors is to use these values saved as scalars. After an estimation, Stata save the coefficients as `_b[`*`varname`*`]`, and the standard errors as `_se[`*`varname`*`]`. So for example, if we ask Stata to `display` the particular coefficients that are of interest to us, we will get…

```
. for any age sex bord hhmemb urban _cons: display _b[X]

->  display _b[age]
-.02298786

->  display _b[sex]
.04040557

->  display _b[bord]
-.03161708

->  display _b[hhmemb]
-.01022535

->  display _b[urban]
.40918392

->  display _b[_cons]
-1.392443
```

We could also use the standard error and the parameter estimate to recalculate the t-statistic for *"age"*…

```
. display _se[age]
.00149144

. display _b[age]/_se[age]
-15.413219
```

Which is exactly what appears in the results that Stata produced after we used the `regress` command. Since these results are up the **Stata Results Window**, you may want to ask Stata to display the results again. You can do so by typing only

```
regress
```

and the results will appear. Stata saves these results in memory until you run `regress` again.

OK, fine. We have just replicated the coefficients and a t-statistic that Stata already showed us. What is the big deal? The big deal is that we can use these estimates to manipulate our

data. For example, we can predict nutrition outcomes for the entire sample *as if* all of the children lived in region 1…

```
. * Predict z-scores for all kids as if they lived in region 1;

. gen double hazhat1 = age*_b[age] + sex*_b[sex] + bord*_b[bord] +
>                  hhmemb*_b[hhmemb] + urban*_b[urban] + _b[_cons];

. sum haz*;

Variable |     Obs        Mean   Std. Dev.       Min        Max
---------+--------------------------------------------------------
     haz |    2478    -1.91406   1.344488      -5.14       2.21
 hazhat1 |    2478   -2.160121   .4178733  -3.122771  -1.058811
```

If we want to compare this predicted value with the predicted value for kids in their actual areas of residence, we can use the saved coefficients and generate the latter fitted values as well. This, however, can become cumbersome when there are many explanatory variables. Fortunately, we don't have to do this because Stata will do it automatically for us with `predict`…

```
. * Predicted z-scores for all kids in their regions of residence
. predict hazhat
(option xb assumed; fitted values)

. * See how the summary statistics differ for regions 1 and 2
. for num 1 2: sum haz* if region == X

->  sum haz* if region == 1

Variable |     Obs        Mean   Std. Dev.       Min        Max
---------+--------------------------------------------------------
     haz |     454   -2.154185    1.28131      -5.14       1.64
 hazhat1 |     454   -2.154185   .4162141  -2.946772  -1.058811
  hazhat |     454   -2.154185   .4162141  -2.946773  -1.058811

->  sum haz* if region == 2

Variable |     Obs        Mean   Std. Dev.       Min        Max
---------+--------------------------------------------------------
     haz |     517   -1.954855   1.236939      -5.11       2.21
 hazhat1 |     517   -2.170893   .4199959  -3.084903  -1.058811
  hazhat |     517   -1.954855   .4199959  -2.868865  -.8427724
```

This is just as we expected. In region 1, our two predicted (fitted) values are exactly the same, whereas in region 2, their means differ (by 0.216, which is the value of the parameter estimate on `reg2`).

Now that we know about predict, it should be clear to you that the following commands should create a variable identical to `hazhat`:

```
predict double hazhat2
for num 2/7: replace hazhat2 = hazhat2 - _b[regX] if regX == 1
```

Estimating separate models:

If we believe that the data generating process in urban and rural areas is truly different, we can estimate separate models using `if`

```
. * Estimate separate urban & rural models
. * Rural
. regress haz age sex bord hhmemb reg2-reg7 if urban==0

  Source |       SS       df       MS                  Number of obs =    2121
---------+------------------------------                F( 10,  2110) =   28.51
   Model |   459.32168    10   45.932168                Prob > F      =  0.0000
Residual |  3399.75742  2110  1.61125944                R-squared     =  0.1190
---------+------------------------------                Adj R-squared =  0.1148
   Total |   3859.0791  2120  1.82032033                Root MSE      =  1.2694

------------------------------------------------------------------------------
         haz |      Coef.   Std. Err.       t     P>|t|       [95% Conf. Interval]
-------------+----------------------------------------------------------------
         age |  -.0243027    .001631    -14.900   0.000      -.0275012   -.0211041
         sex |   .0255223   .0552374      0.462   0.644      -.0828032    .1338478
        bord |  -.0273322   .0166175     -1.645   0.100      -.0599206    .0052563
      hhmemb |   -.007908   .0116995     -0.676   0.499      -.0308517    .0150358
        reg2 |   .1962734   .0867003      2.264   0.024       .0262464    .3663005
        reg3 |   .0023259   .0930157      0.025   0.980      -.1800862    .1847379
        reg4 |   .2826988   .1086381      2.602   0.009       .0696498    .4957478
        reg5 |   .0587171   .1344982      0.437   0.662      -.2050458    .3224801
        reg6 |   .8043588   .1156887      6.953   0.000        .577483    1.031235
        reg7 |   .4093896   .0894388      4.577   0.000       .2339922     .584787
       _cons |  -1.364371   .1086921    -12.553   0.000      -1.577526   -1.151217
------------------------------------------------------------------------------

. * Urban
. regress haz age sex bord hhmemb reg2-reg7 if urban==1

  Source |       SS       df       MS                  Number of obs =     357
---------+------------------------------                F(  9,   347) =    5.30
   Model |  63.6196786     9  7.06885318                Prob > F      =  0.0000
Residual |   463.16862   347  1.33477988                R-squared     =  0.1208
---------+------------------------------                Adj R-squared =  0.0980
   Total |  526.788298   356  1.47974241                Root MSE      =  1.1553

------------------------------------------------------------------------------
         haz |      Coef.   Std. Err.       t     P>|t|       [95% Conf. Interval]
-------------+----------------------------------------------------------------
         age |  -.0149482   .0036827     -4.059   0.000      -.0221913    -.007705
         sex |   .1451551   .1234553      1.176   0.240      -.0976597    .3879699
        bord |  -.0767909   .0420499     -1.826   0.069      -.1594956    .0059139
      hhmemb |  -.0271518   .0227105     -1.196   0.233      -.0718194    .0175158
        reg2 |   .3886436   .2389425      1.627   0.105      -.0813142    .8586014
        reg3 |  -.3105804   .3001925     -1.035   0.302      -.9010061    .2798453
        reg4 |   .3679373   .2269424      1.621   0.106      -.0784185     .814293
        reg5 |  (dropped)
        reg6 |   .8414193   .2232163      3.770   0.000       .4023921    1.280446
        reg7 |    .538519   .2237401      2.407   0.017       .0984616    .9785765
       _cons |  -1.152788   .2507334     -4.598   0.000      -1.645936   -.6596395
------------------------------------------------------------------------------
```

If we typed `predict` at the end of these sets of commands, we would have predicted values of *"haz"* for all kids (urban and rural) *as if* they all lived in urban areas. If that is not what we want (i.e. we want fitted values for kids in rural areas to be predicted from the rural model), then the following sets of commands will generate the fitted values we want…

```
* Estimate separate urban & rural models
* Rural
 regress haz age sex bord hhmemb reg2-reg7 if urban==0
* Rural fitted values
 predict hazhatr if e(sample)
* Urban
 regress haz age sex bord hhmemb reg2-reg7 if urban==1
* urban fitted values
 predict hazhatu if e(sample)
egen hazhatur = rsum(hazhatr hazhatu)
```

Try this and check that they are indeed the same.

Of course, we could have used the interactions between the urban dummy and the other explanatory variables that you created in exercise 1, and included them in one pooled model to allow for the parameter estimates to differ across urban and rural areas. This as we shall see in a later session facilitates testing.

Finally, we created all of these dummy variables ourselves. Stata has an option called `xt` that will do this (as well as interactions) for you when you run a model. This is a nice command, but if you are going to run several models with the same dummy variables and if your dataset is large, then you will save time by creating the dummies yourself rather than have Stata do it for each run of the model. In addition, you will be able to give the dummies names that you prefer rather than those imposed by Stata. Now that we know how to use `for` to make the dummies, it's not a big deal to do so.

Hypothesis Testing

Now that we have the parameter estimates and the variance-covariance matrix of these estimators, we can manually construct our own statistics to perform hypothesis tests on the coefficients, *OR* we can use Stata's `test` command.

Suppose for some reason we want to test the null hypothesis that the effect of age and sex (being male) on nutritional outcomes is the same. Anytime after using `regress`, the `test` command can be invoked. Here we run the pooled model again and run the test:

```
. * Estimate pooled model and perform hypothesis tests
. regress haz age sex bord hhmemb reg2-reg7

-- results deleted (we already know them)

. test age = sex

 ( 1)  age - sex = 0.0

       F(  1,  2467) =    1.50
            Prob > F =    0.2212
```

So we cannot reject the null hypothesis that these effects are the same (not that it has any real meaning). We could just as easily have used the `lincom` command (which tests the significance of linear combinations of estimators) to test the same hypothesis…

```
. lincom age - sex

 ( 1)  age - sex = 0.0

------------------------------------------------------------------------------
     haz |      Coef.   Std. Err.       t     P>|t|       [95% Conf. Interval]
---------+--------------------------------------------------------------------
     (1) |  -.0622214   .0508495     -1.224   0.221      -.1619334    .0374906
------------------------------------------------------------------------------
```

Now suppose that we want to test joint hypothesis that there are no region effects…

```
. test reg2 reg3 reg4 reg5 reg6 reg7

 ( 1)  reg2 = 0.0
 ( 2)  reg3 = 0.0
 ( 3)  reg4 = 0.0
 ( 4)  reg5 = 0.0
 ( 5)  reg6 = 0.0
 ( 6)  reg7 = 0.0

       F(  6,  2467) =   19.51
            Prob > F =    0.0000
```

`test` with just one argument and no condition tests whether that argument is different from zero (the null hypothesis is that it is zero). When there are multiple arguments, as we have here, a joint test that they are all together different from zero is conducted with the appropriate F-statistic reported. In this case we soundly reject the null hypothesis that all of the regional effects are equal to zero.

If we want to test whether the effects of regions 3-7 are different from region 2, we can't just list the arguments as we did above, but we can use `test` with the `accumulate` option. We'll first show how this works with regions 3 and 4, and then show a compact way of writing the code for more conditions…

```
. test reg2 = reg3, notest

 ( 1)  reg2 - reg3 = 0.0

. test reg2 = reg4, accum

 ( 1)  reg2 - reg3 = 0.0
 ( 2)  reg2 - reg4 = 0.0

       F(  2,  2467) =    7.32
            Prob > F =    0.0007
```

The `notest` option is used to suppress the results of the first test (between regions 2 and 3), and the `accum` option tells Stata that the hypothesis is to be jointly tested with the previously tested hypotheses. So here we again reject the hypothesis that the effects of region 3 and 4 are no different from region 2.

To test whether regions 3-7 are different from region 2, we could write out a bunch of lines consistent with what we just did above, or we can use our favorite `for` command:

```
test reg2 = reg3, notest
for num 4/6: test reg2 = regX, notest accum
test reg2 = reg7, accum
```

Try this and confirm that the test statistic indeed represents a test of our hypothesis.

E. Basics of Matrices:

In the previous section we saw that after estimation commands such as regress, Stata saves matrices with the parameter estimates and with the variance-covariance of these estimates in them. Now we will briefly get a feel for how matrices are used by creating them and manipulating them.

Let's first define a matrix called "beta" to be the transpose of `e(b)`, and another called "covb" to be `e(V)`, and then `list` them:

```
. matrix beta = e(b)

. matrix covb = e(V)

. matrix list beta

beta[11,1]
               y1
   age  -.02275691
   sex   .03946451
  bord  -.04225822
hhmemb  -.00833378
  reg2   .21444858
  reg3  -.02241319
  reg4   .34935303
  reg5   .04375658
  reg6   .89001386
  reg7   .45196462
 _cons  -1.3432511

. matrix list covb, format(%4.3f)

symmetric covb[11,11]
           age     sex    bord  hhmemb    reg2    reg3    reg4    reg5    reg6
   age   0.000
   sex   0.000   0.003
  bord   0.000   0.000   0.000
hhmemb   0.000   0.000   0.000   0.000
  reg2   0.000   0.000   0.000   0.000   0.007
  reg3   0.000   0.000   0.000   0.000   0.004   0.008
  reg4   0.000   0.000   0.000   0.000   0.003   0.003   0.009
  reg5   0.000   0.000   0.000   0.000   0.003   0.003   0.004   0.017
  reg6   0.000   0.000   0.000   0.000   0.003   0.003   0.004   0.004   0.010
  reg7   0.000   0.000   0.000   0.000   0.003   0.003   0.004   0.004   0.004
 _cons   0.000  -0.001   0.000  -0.001  -0.004  -0.004  -0.003  -0.002  -0.003

          reg7   _cons
  reg7   0.007
 _cons  -0.003   0.010
```

Note that since the variance-covariance matrix is symmetric, Stata saves memory and only stores the lower triangle. To assure you that those are not true zeros in `covb`, let's look at its inverse…

```
. matrix list invcovb, format(%4.1f)

symmetric invcovb[11,11]
              age         sex        bord      hhmemb        reg2        reg3
   age  1911235.6
   sex    24631.6       795.1
  bord   123013.0      2044.0     15023.0
hhmemb   284070.4      4739.3     25612.8     65739.5
  reg2    10203.3       172.9       707.7      1578.3       325.0
  reg3     6922.2       110.6       581.4      1253.3         0.0       229.4
  reg4     5296.8        95.5       462.6      1267.2         0.0         0.0
  reg5     2123.9        36.5       269.6       515.4         0.0         0.0
  reg6     5166.7        93.7       449.4      1128.2         0.0         0.0
  reg7     9563.4       140.2       826.5      1878.7         0.0         0.0
 _cons    47768.6       795.1      4014.6      9252.9       325.0       229.4

              reg4        reg5        reg6        reg7       _cons
  reg4       177.9
  reg5         0.0        73.5
  reg6         0.0         0.0       165.9
  reg7         0.0         0.0         0.0       300.4
 _cons       177.9        73.5       165.9       300.4      1557.5
```

… of course, those aren't true zeros either. To assure you that this is indeed the case, let's just look at the region columns and not restrict the number of digits to the right of the decimal point…

```
. matrix reginv = invcovb[1...,"reg2".."reg7"]

. matrix list reginv

reginv[11,6]
              reg2         reg3         reg4         reg5         reg6         reg7
   age   10203.265    6922.2296    5296.7973     2123.873    5166.6873    9563.3999
   sex    172.85146    110.62493    95.539716    36.455944    93.654064    140.16682
  bord    707.74816    581.40946    462.61336    269.64828     449.4138    826.54426
hhmemb     1578.291    1253.3302    1267.1583    515.41163    1128.2486    1878.7382
  reg2    324.96075   -1.300e-12    1.002e-12   -5.294e-13   -4.832e-13   -7.248e-13
  reg3   -1.300e-12    229.42103   -4.050e-13   -2.043e-14    1.350e-13    2.224e-12
  reg4    1.002e-12   -4.050e-13    177.87987    3.064e-13    1.030e-13   -7.532e-13
  reg5   -5.294e-13   -2.043e-14    3.064e-13    73.540439   -2.531e-13   -1.172e-13
  reg6   -4.832e-13    1.350e-13    1.030e-13   -2.531e-13     165.9374   -4.050e-13
  reg7   -7.248e-13    2.224e-12   -7.532e-13   -1.172e-13   -4.050e-13    300.44727
 _cons    324.96075    229.42103    177.87987    73.540439     165.9374    300.44727
```

So there you go, there are no zeros. But what's more interesting for us is how we extracted part of the `invcovb` matrix with the following command

```
matrix reginv = invcovb[1...,"reg2".."reg7"]
```

This could just as easily been written as

```
matrix reginv = invcovb[1...,5..10]
```

or even as

```
matrix reginv = invcovb[1..11,5..10]
```

This last expression more explicitly illustrates what were extracting: the matrix made up of all 11 rows and the 5th through 10th columns. The general form for submatrices is

```
matrix submat = matname[r1..rN,c1..cN]
```

In the first line above, we substituted the column names for the column indices ($c_1..c_N$). And when Stata evaluates r_N (or c_N for that matter) as missing (.), it is taken as referring to the last row (or column) of *matname*. Because there are 11 rows in `invcovb`, Stata interpreted `invcovb[1...,5..10]` as `invcovb[1..11,5..10]`.

One final thing that we shall do with matrices is to calculate the t-statistic for "age" again. We already did this in part D, but with saved scalars. Now let's do it with matrices (and scalars)…

```
. scalar bage = beta[1,1]

. scalar vage = covb[1,1]

. scalar tage = bage / sqrt(vage)

. display %4.3f tage
-15.176
```

Type `regress`, and you will see that this is the same as in the Stata output. Something to note here is that to our eyes, `beta[1,1]` is a scalar. But as far a Stata is concerned it's a matrix that just happens to have 1x1 dimensions. Consequently, we had to create scalars to do what we wanted. If we had tried the following command

```
matrix tage = beta[1,1] / sqrt(covb[1,1])
```

we would have had a problem because:

a. The `sqrt()` function can only evaluate scalars, not matrices, and
b. Even if our denominator was a 1x1 matrix, the forward slash (`/`) tells Stata to stack the matrices, not divide their elements.

There are many more features to matrices and functions that can be applied to them. Your best bet is to read through chapter 17 ("Matrix Expressions") of the Stata User's Guide to get a better feel for what can be done with matrices in Stata.

This concludes the Review of Stata for the Technical Workshop. Because time has been short, and there has been much to learn or brush up on, many important issues were not covered. Foremost among them is the use of sampling weights and controlling for sampling design in estimates of standard errors following an estimation procedure. Since the data being used in this workshop are self-weighted, this will not hinder the remaining sessions in terms of weighted estimation.[1] *Nonetheless, I recommend that you read the section of the User's Guide on estimation and use of weights since you will inevitably need to use them in the future. Similarly, a read through the section on* ***svy estimators*** *in the reference manual will motivate the need for controlling for sampling design in the estimation of standard errors. I am happy to talk with one and all of you about these and any other issues!!*

[1] Note that even with self-weighted samples of households, weights might be necessary. This is the case if we want to estimate the average expenditure of individuals in the population. In our data, we have a variable that estimates per capita expenditures for each household. If we want an estimate of the mean of this variable over the population of individuals (not households), we need to use the household size as the weight.