Writing Programs in Stata and Using them for the `gmm` Command

Roy Mill

November 3, 2010

Motivation

- So, you know how to run OLS in Stata, but it's not enough for the job market.
- Programs are used to encapsulate code
  - Instead of rewriting the code for every program that needs the same functionality.
  - To be used by programs that leave "black boxes".
- Example: OLS. What you actually do is:
  1. Take all independent variables and put them under X.
  2. Take the dependent variable and put it in Y.
  3. Remove observations that have a missing value in either variables.
  4. Calculate $b = (X'X)^{-1} X'Y$, the variance covariance matrix, t-tests, CI, and other statistics.
  5. Report everything.

  But instead of having you write all this by yourself, it was encapsulated in a program called `regress`.

Functions in Math and in Programming

- Just like programs, functions too encapsulate stuff.
  - Instead of writing $\frac{c^{1-\sigma}}{1-\sigma}$ we write $u(c)$.
  - If we think σ might vary, we write $u(c,\sigma)$ (or $u_\sigma(c)$ which is the same)
- Functions take arguments (c,σ) and return a value. In between they perform various operations.
- In programming, you do the same. Define the function, then you can call it and assign the returned value in the same line of code:

```
util_tomorrow = beta*u(c,sigma)
```

- Programs in Stata are not functions in the sense that you can't assign them to a value:

```
. reg y x1 x2
. local R2 = r(R2)
```

What's in a Program?

- Suppose we define a CRRA function in pseudocode (not aimed at a programming language syntax):

```
function utility(consumption, sigma) {
   if (sigma == 1) {
      utils = log(consumption)
   } else {
      utils = consumption ^ (1-sigma) / (1-sigma)
   }
   return utils
}
```

- As always, whenever we write a function or a program we do three things:
  1. Define our list of arguments (our (c, σ))
  2. Do something that depends on the arguments (calculate utility, regress variables, make coffee)
  3. Return something (or report to the user).

What's in a Program?

- Some programs don't return anything:

```
compress
```

- Others don't take any arguments:

```
count
```

- Actually, count can take arguments, but they are optional:

```
count if south == 1
```

- We will now write a program that calculates CRRA utility and prints it to the screen.

Our First Program (OMG!)

- Create a file named `utility.ado`.
  - Place it in your working directory (the one you "cd" into).
- Write the following code in it:

```
program utility
   args consumption

   local sigma = 2    // It MUST be 2!!
   local utils = `consumption' ^ (1-`sigma') / (1-`sigma')
   di "Utility for c=`consumption' is `utils'"
end
```

- Go to the command window in Stata and type: `utility 4`.

```
Utility for c=4 is -.25
```

Small ado-files notes

- When you type in the command `utility`. Stata:
    1. Looks for a program named `utility` that is already loaded into Stata's memory.
    2. If there is no program called `utility` in memory, Stata looks for a `utility.ado` file under a list of folders. One of them is the folder you "cd"-ed to.
- To see the list of folders Stata looks for the ado file in, type `adopath`
    - To add a folder to the list of folders, type `adopath + <added folder>`
- Note that after Stata read your ado file in step 2, next time it will find the program in step 1.
    - If you changed the ado file, type `program drop utility` before the next time you run it, to throw the program from Stata's memory.

Getting Arguments for the Program

- There are two statements that tell the program how to expect the arguments
  - The `args` command – for an ordered list of arguments.
  - The `syntax` command – puts more structure on the arguments and makes them more Stata-like.
- The syntax of the `args` command:

```
args local1 [local2 [local3 [...]]]
```

- For example:

```
args consumption sigma
```

- This means that after we're done with the program, when we run in Stata the command:

```
utility 41 3
```

- Stata will run the program `utility` and assign 41 to `` `consumption' `` and 3 to `` `sigma' `` inside `utility`.

Writing Generic Code

- We can write our program from above to be more generic
  - Not specific to $\sigma = 2$, but for any σ.

```
program define utility
    args consumption sigma   // List of arguments
    if (`sigma' == 1) {
        local utils = log(`consumption')
    }
    else {
        local utils = `consumption' ^ (1-`sigma') / (1-`sigma')
    }
    di "Utility for c=`consumption' and sigma=`sigma' is `utils'
end
```

- Letting σ be determined freely by the user makes your program suit more purposes.

Pros and Cons of Being Generic

- Pros:
  - If you want other users to use, program solves a wider range of problems.
  - If you want to change parameters to check sensitivity, no need to change the program.
    - Can loop easily over different values of σ
  - Overall, makes your code more reusable and do-files using this program easily adjustable.
- Cons
  - Input validation. What if someone runs `utility 41 risky`?
    - Should the program validate that the second argument is numeric?
    - When using the command `syntax` instead of `args`, Stata takes care of some validation
  - Code can get cumbersome, or slower
    - We added the `if` condition to take care of $\sigma = 1$.

Optional Arguments

- Actually, specifying k locals with **`args`** doesn't require the user to run the program with k arguments.
  - If given $m > k$ arguments, Stata will ignore arguments $k+1$ through m.
  - If given $m < k$ arguments, Stata will have locals $m+1$ through k undefined (empty).
- So we can let the user only **optionally** set some arguments:

```
program define utility
    args consumption sigma   // List of arguments

    if ("`sigma'" == "") { // If sigma wasn't specified...
        local sigma = 2    // Set it to be 2 by default.
    }

    // The rest of the program unchanged (I'm saving space)
end
```

Optional Arguments

```
. utility 48
Utility for c=48 and sigma=2 is -.0208333333333333
```

- However, can't omit any subset of arguments. If you specified m arguments, then
  - the first m locals will have these arguments assigned to them respectively
  - the last $k-m$ will be empty.
- In other words, can't omit consumption and specify sigma. First argument is always consumption.

Specifying Arguments Using `syntax`

- Allows your program to receive input like most Stata commands

```
command [varlist | namelist | anything]
    [if] [in] [using filename]
    [= exp] [weight]
    [, options]
```

- Takes care of some input validation
    - If you say `varlist`, Stata will check that all arguments there are variables list
    - Can limit the number of arguments.
- Parses the user's input smartly and puts it in locals inside the program
    - If user adds a condition using `if`, or `in`
    - If the user specifies weights variable
    - If the user specifies a `using` file.

An Example

- Suppose we write a program that calculates utility for each observation in our dataset. Name this program utilize.
  - It requires a variable containing consumption c_i.
  - It creates a new variable named `utility` containing $u_i = \frac{c_i^{1-\sigma_i}}{1-\sigma_i}$
  - It assumes $\sigma_i = 2$ for all observations.
- This is how the program will look like

```
program utilize
    syntax varlist(max=1)

    local sigma = 2
    local newvarname = "utility"

    gen `newvarname' = `varlist' ^ (1-`sigma') / (1-`sigma')
end
```

An Example

- Good. Now allow the program to do more
  - Specify a different name for the generated variable.
  - Allow σ_i to be another (still fixed) σ, but not 2 necessarily.
- This is how the program will look like now

```
program utilize
    syntax varlist(max=1) [, newvarname(name) sigma(real 2)]
    if ("‘newvarname’" == "") {
        local newvarname = "utility"
    }

    gen ‘newvarname’ = ‘varlist’ ^ (1-‘sigma’) / (1-‘sigma’)
end
```

- Had we wanted to make newvarname required, we could have rewriten the syntax line as follows:

```
syntax varlist(max=1) , newvarname(name) [sigma(real 2)]
```

Tweaking it a bit more

- Now, allow the program to optionally receive a variable name containing an i-specific σ_i.
  - Only if the second variable name is missing, assume $\sigma_i = 2$ for all observations.
  - If the σ_i variable is specified, assume $\sigma_i = 2$ only for observations with missing σ_i.
  - Allow the user to specify a different default σ.
- The program now becomes a bit longer...

Tweaking it a bit more

```
program utilize
    syntax varlist(max=2) [, newvarname(name) sigma(real 2)]

    if ("`newvarname'" == "") {
        local newvarname = "utility"
    }

    local consumpvar : word 1 of `varlist'
    local sigmavar : word 2 of `varlist'

    if ("`sigmavar'" != "") {
        gen `newvarname' = `consumpvar' ^ (1-`sigmavar') / (1-`sigmavar') ///
                                                    if `sigmavar' != .
        replace `newvarname' = `consumpvar' ^ (1-`sigma') / (1-`sigma') ///
                                                    if `sigmavar' == .
    }
    else {
        gen `newvarname' = `consumpvar' ^ (1-`sigma') / (1-`sigma')
    }
end
```

Passing Conditions to Your Program

- You can allow (or require) to specify a condition using `if`:

```
syntax varlist(max=2) [if/] [, Newvarname(name) SIgma(real 2)]
// rest of program the same, except the following part:
if ("`sigmavar'" != "") {
   gen `newvarname' = `consumpvar' ^ (1-`sigmavar') / (1-`sigmavar') ///
                                                  if `sigmavar' != . & `if'
   replace `newvarname' = `consumpvar' ^ (1-`sigma') / (1-`sigma') ///
                                                  if `sigmavar' == . & `if'
}
```

- The / after the if tells Stata to put inside the local `if` only the condition that is specified when the program is called.
- Without the / Stata will add the "if" word.
- Example: suppose I run `utilize consumption if female==0`
  - If I have `if/` in the syntax line, then `` `if' `` contains "`female==0`"
  - If I have `if` in the syntax line, then `` `if' `` contains "`if female==0`"

Small Comments about Programs in General

- `version #` statement
  - Usually invoked before the `syntax` or `args` command
  - Tells Stata to be compatible with the version in which you wrote the program.

```
program utilize
    version 11
    syntax varlist(max=2) [, newvarname(name) sigma(real 2)]

    // rest of program
end
```

- Scope (locals vs globals)
  - Locals set outside the program can't be used inside the program.
  - Locals set inside the program can't be used outside the program.
  - Globals set anywhere can be reached from anywhere.

Small Comments about the `syntax` Command

- You can specify on-off options that don't take additional arguments.
  - They will contain their own names if they were specified or nothing if not

```
program savesomething
   syntax varlist using [, replace]

    // `replace' contains the string "replace" if specified. "" otherwise.
    if ("`replace'" != "") {
        di "replace option was specified."
    }
end
```

- You can specify shorthands for your options capitalizing the letters of the shorthand.

```
syntax varlist(max=2) [, Newvarname(name) SIgma(real 2)]
    // rest of program

// outside the program, we can call the program this way:
utilize consumption, n(calcuated_util) si(3)
```

Returning Values from a Program

- We said that unlike functions, programs in Stata can't be assigned as an expression:

```
gen util = utility(consumption) // This is wrong!
```

- You can generate variables inside programs (like we did in `utilize`)

- You can also return values like other Stata programs do:
    - `return list` after you run an r-class program will show you the returned values
    - `ereturn list` will do the same for e-class programs.
- Perfect for returning different statistics or estimation results.
- `eclass` programs perform estimation. Other programs should be `rclass`.

Returning Values from a Program

- Suppose now we want `utility` to return the result too, and not just print it out.

```
program utility, rclass
    // everything else as before (calculating the local `utils')

    return scalar utility = `utils'

    matrix input = (`consumption', `sigma')
    return matrix arguments = input
end

// Outside the program
utility 41 3
di "The program returned " r(utility)
matrix list r(arguments)
```

- If the program is an e-class program, just specify `eclass` instead of `rclass` at the top, and use `ereturn` instead of `return`.
- See `help return` for further details.

With a little `help` from my Stata

- There is no time to go through everything that `syntax` lets you do.
- Moreover, you will probably get the `invalid syntax` `r(197);` error hundreds of times.
- Stata is your friend. Well, maybe not. But let it help you.
  - Always work with a window of `help syntax` open.
  - Same for `help return`

GMM Estimation in Stata 11

- As of version 11, Stata has a `gmm` command that allows to directly estimate GMM models.
- You can specify your equations using either **substitutable expressions** or **moment-evaluator programs**.
- Both methods are used in other commands in Stata such as `ml` (maximum likelihood) or `nl` (nonlinear LS).

- The main focus of these slides is not the econometrics, but the ability to implementing equations to code.
- We will use what we learned about programs above to write a moment-evaluator program.
- As always, more help in `help gmm`.
  - Many examples and more details in the manual pages (That's the PDF file you get when you click on Manual: [R] gmm at the bottom of the Stata help window)

GMM Quick Reminder

- A model that yields moment equations of the form:

  $$\mathbb{E}\left[\mathbf{z}_i \cdot u_i\left(\beta\right)\right] = 0$$

  if u_i is an additive error in the model, or more generally:

  $$\mathbb{E}\left[h_i\left(\mathbf{z}_i, \beta\right)\right] = 0$$

  where:
  - β is a vector of parameters we're interested in estimating using these moment equations.
  - $\mathbf{z}_i$ is a vector of instruments: variables that satisfy the moment condition.

- Numerically look for a $\hat{\beta}$ that minimizes the (weighted) distance of the sample-analogues from 0.
- Use derivatives of $\frac{\partial u_i}{\partial \beta}$ or $\frac{\partial h_i}{\partial \beta}$ to calculate $\widehat{Var}\left(\hat{\beta}\right)$ (and to find $\hat{\beta}$).

Estimating GMM Using Substitutable Expressions

- Suppose I try to estimate the following nonlinear (and stupid) model for cars' MPG

$$MPG_i = \alpha\,(WEIGHT_i)^{\beta}\,(LENGTH_i)^{\gamma} + u_i$$

- We have 3 parameters: (α, β, γ) and 3 instruments: weight, length and a constant term.

- Let's rewrite the equation for u_i and specify z_i:

```
gmm (mpg - {alpha=1}*weight^{beta=1}*length^{gamma=1}) ///
     , instruments(weight length)
```

- Parameters are in curly brackets. Initial values can be specified with $=$.
  - Stata will substitute the curly brackets with the current iteration's value of the parameter.

- Instruments were specified using the `instruments` option. Constant term added by default.

One more time

- Suppose our crazy theory says that:

$$HEADROOM_i = \exp\left(\beta_0 + \beta_1 TRUNK_i + \beta_2 MPG_i\right) + v_i$$

  - Also, we think MPG_i is correlated with the error so we use $LENGTH_i$ and $WEIGHT_i$ as instruments.
  - This means that we believe our moment conditions are:

$$\mathbb{E}\begin{bmatrix} v_i \\ TRUNK_i v_i \\ LENGTH_i v_i \\ WEIGHT_i v_i \end{bmatrix} = 0$$

- All we need to do is to express v_i in an equation:

$$v_i = HEADROOM_i - \exp\left(\beta_0 + \beta_1 TRUNK_i + \beta_2 MPG_i\right)$$

- And then run:

```
gmm (headroom - exp({b0} + {b1}*trunk + {b2}*mpg)) ///
     , instruments(trunk weight length)
```

Another way to specify initial values for the parameters

Initial values can be specified using the `from` option.

- Will override any initial values set with `{param=value}`
- Can pass a name of a vector of values:

```
matrix initials = (.01, .2, .1)
gmm ... , instruments(...) from(initials)
```

- Alternatively, can list parameters and their values:

```
gmm ... , instruments(...) from(alpha .01 beta .2 gamma .1)
```

Another way to specify linear combinations

Linear combinations can be specified using the `:` operator

- Instead of writing:

  ```
  gmm (headroom - exp({b0} + {b1}*trunk + {b2}*mpg)), ...
  ```

  can write:

  ```
  gmm (headroom - exp({b0} + {xb: trunk mpg})), ...
  ```

- Constant terms are not included, so we added `{b0}`.
- The `xb` is just a name, can put anything there: `{mylincom:  trunk mpg}`
- We can refer to the same linear combination again:
  - Suppose we had a model that looked like:

    $$\mathbb{E}\left\{\mathbf{z}\frac{y-\exp\left(\mathbf{x}'\beta\right)}{\exp\left(\mathbf{x}'\beta\right)}\right\}=0$$

    Then we should write (assume no constants, for simplicity):

    ```
    gmm ((y - exp({lin: x1 x2 x3}))/exp({lin:})), ...
    ```

Weighing Moment Conditions

- As you remember, the actual GMM estimator looks roughly like this

$$\hat{\beta}_{GMM} = \arg\min_{\beta} \left(U\left(\beta\right)' Z\right) W \left(Z' U\left(\beta\right)\right)$$

- If you're just-identified, the weighting matrix doesn't matter.
- If you're overidentified, then you need to choose a weighting matrix.

- Stata by default:
  - Runs once with $W = \left(Z'Z\right)^{-1}$ (like 2SLS).
  - Then, using the predicted $\hat{U}$, Stata allows for heteroskedastic u's and re-estimates using $W = \left(\hat{U}'ZZ'\hat{U}\right)^{-1}$
  - Stata by default reports the result of the second estimation.

Changing the Default Weighting Scheme

- To change the initial weighting matrix from $Z'Z$, use the `winitial` option.
- To change the way the weighting matrix is recalculated, use the `wmatrix` option.
  - Can take heteroskedasticity and autocorrelation (HAC), or clusters, into account. Can assume homoskedasticity instead.
- To change the number of times GMM is estimated (from the default twice), use:
  - `onestep` for only one estimation (to replicate 2SLS, say)
  - `twostep` is default
  - `igmm` allows for more iterations. Other options are required though. See `help`.

Specifying Analytical Derivatives

- Derivatives are used for minimizing the distance function and for calculating variance matrix.
- By default, Stata numerically calculates the Jacobian.
  - Basically, looking at the change in the function value following a small change of each parameter.
- Numerical derivatives are usually fine. But you can specify analytical derivatives.
  - More accurate, and makes optimization faster.
  - Can specify $\frac{\partial u_i}{\partial \beta}$ using the `derivative` option.
  - If you do, need to specify all derivatives in model.
- More on this in the help file and the manual (p. 593)

Optimization Options

- You can change the settings of the optimization **gmm** does each time it estimates.
- `technique(nr)` will change the optimization technique from Gauss Newton (default) to modified Newton-Raphson (nr).
    - `dfp`, `bfgs` also allowed for the `gmm` command.
- `<2->conv_maxiter(1000)` will set the maximum number of iterations to 1000.
    - This is not the GMM iterations! These are the iterations in the search for the minimizing $\hat{\beta}$.
    - GMM iterations are the ones that update the weighting matrix after errors are estimated.
- `<3->conv_ptol(), conv_vtol(), conv_nrtol(), tracelevel()` also available for you to set.

Estimating Multiple Equations Simultaneously

- Estimating one equation is for kids. You will never get a good placement if you estimate just one equation.
- If we assume there's covariance between the errors of the equations, it's better to estimate them jointly:

$$\mathbb{E}\begin{bmatrix} z_u u \\ z_v v \end{bmatrix} = \mathbb{E}\begin{bmatrix} z_u \left(MPG_i - \alpha \left(WEIGHT \right)^{\beta} \left(LENGTH \right)^{\gamma} \right) \\ z_v \left(HEADROOM - \exp\left(\beta_0 + \beta_1 TRUNK + \beta_2 MPG \right) \right) \end{bmatrix} = 0$$

- We can use the same instruments for all equations, or we can differ z_u from z_v.
- For now, assume the same instruments for both equations. Just add the equation.

```
gmm (mpg - {alpha=1}*weight^{beta=1}*length^{gamma=1}) ///
     (headroom - exp({b0} + {xb: trunk mpg})) ///
     , instruments(weight length trunk) ///
       winitial(unadjusted, independent)
```

Initial Weighting Matrix with Simultaneous Equations

- All we did was to add the 2nd equation in a separate parentheses.

- Recall that Stata by default sets the initial weighting matrix to be $W = \left(\frac{1}{n}\sum_{i=1}^{n} z_i^{'} z_i\right)^{-1} = \Lambda^{-1}$.
- When you have m equations, z_i contains all equations' $z_i = (z_{i1}, ..., z_{im})$
  - They can be identical, in which case they are copied.
- Λ now is composed of blocks according to the equation. For equations r, s we get:

$$\Lambda_{rs} = n^{-1} \sum_{i=1}^{n} z_{ir}^{'} z_{is}$$

- If you have a common instrument in all equations, Λ will not be positive-definite. $W = \Lambda^{-1}$ must be positive-definite.

Initial Weighting Matrix with Simultaneous Equations

Possible solutions:

1. Impose $\Lambda_{rs} = 0$ in the initial matrix by adding the `independent` suboption: `winitial(unadjusted, independent)`

1. Impose a different (and positive-definite) initial weighting matrix.

    1.1 One straightforward positive-definite matrix is the identity matrix `winitial(identity)`

        1.1.1 Usually performs badly – longer convergence.

    1.2 Alternatively, construct any matrix and impose it:

```
matrix W = ...
gmm ... , winitial(W)
```

- In any case, the second iteration of GMM doesn't have to impose these restrictions. Weighting matrix is recalculated next iteration and usually is fine.

Specifying Different Instruments for Differnet Equations

- If equations are not named $\Rightarrow$ identified by their order.

```
gmm (mpg - {alpha=1}*weight^{beta=1}*length^{gamma=1}) ///
     (headroom - exp({b0} + {xb: trunk mpg})) ///
     , instruments(1: weight length)  ///
       instruments(2: length trunk) ///  ...
```

- Alternatively, we can name the equations.

```
gmm (mpg_eq: mpg - {alpha=1}*weight^{beta=1}*length^{gamma=1}) ///
     (hdr_eq: headroom - exp({b0} + {xb: trunk mpg})) ///
     (price_eq: price - {prem: foreign mpg}) ///
     , instruments(weight length)  ///
       instruments(hdr_eq price_eq: trunk) ///
       instruments(price_eq: foreign gear_ratio) /// ...
```

- Note how colon inside () follows equation name and colon inside {} follows linear combination name

- When specifying `instruments`
  - No equation specified$\Rightarrow$ apply instruments to all equations
  - Can apply additional instruments to a subset of equations

Using GMM in the Moment-Evaluating Program Mode

- We can't always express the error as a function of variables and parameters.
  - Sometimes our moments are more complicated.
- As in the manual, following Blundell, Griffith and Windmeijer (2002), we have:

$$y_{it} = \exp\left(\mathbf{x}_{it}^{\prime}\beta\right)\nu_i + \epsilon_{it} = \mu_{it}\nu_i + \epsilon_{it}$$

  where ν_i is a multiplicative unobservable fixed-effect and we're interested in β
- We can estimate our betas by replacing ν_i with $\frac{\bar{y}_i}{\bar{\mu}_i}$, creating the following moment conditions (sample analogues):

$$\sum_i \sum_t \mathbf{x}_{it}\left(y_{it} - \mu_{it}\frac{\bar{y}_i}{\bar{\mu}_i}\right) = 0$$

- You can't write a substitutable expression that will do that.
- Luckily, we already know how to write programs!

A Moment-Evaluator Program's Possible `syntax`

```
syntax varlist if [weight], at(name) [derivatives(varlist)] [options]
```

- `varlist` will contain the variable names into which the program is supposed to save u_i
- `if` will hold the sample of observations that enter the analysis
  - Stata will take care of removing any observations with a missing value in one of the variables.
- `at` will get a name of a vector of the current iteration's $\hat{\beta}$.
- `weight` allows `gmm` to pass pweights, fweigths or aweights specified by the user.
- `derivatives` allows the program to save analytical derivatives in additional variables.
- `options` are any other options your program potentially needs from `gmm`.

A Simple Moment-Evaluating Program

- Write a program to evaluate the moments of the first example:

$$MPG_i = \alpha\,(WEIGHT_i)^{\beta}\,(LENGTH_i)^{\gamma} + u_i$$

```
program gmm_cobbdoug
    syntax varlist if, at(name)

    quietly replace `varlist' = ( mpg - `at'[1,1] * weight^(`at'[1,2]) * ///
                                              length^(`at'[1,3]) ) `if'
end

// outside the program
matrix initial = (1, 1, 1)
gmm gmm_cobbdoug, nequations(1) parameters(alpha beta gamma) ///
    instruments(weight length) from(initial)
```

- Note the use of varlist. The program is expecting one variable to hold the u it calculates.
- Also, ‘at’ contains a name of a vector that holds the values of the current $\hat{\alpha}, \hat{\beta}, \hat{\gamma}$

Adding Derivatives

```
program gmm_cobbdoug
    syntax varlist if, at(name) [derivatives(varlist)]

    quietly replace `varlist' = ( mpg - `at'[1,1] * weight^(`at'[1,2]) * ///
                               length^(`at'[1,3]) ) `if'
    if ("`derivatives'" == "") { // if derivatives unnecessary
         exit
    }

    local d_var_1 : word 1 of `derivatives'
    local d_var_2 : word 2 of `derivatives'
    local d_var_3 : word 3 of `derivatives'

    qui replace `d_var_1' = -1 * weight^(`at'[1,2]) * length^(`at'[1,3]) `if'
    qui replace `d_var_2' = -1 * `at'[1,1] * log(weight) * ///
                               weight^(`at'[1,2]) * length^(`at'[1,3]) `if'
    qui replace `d_var_3' = -1 * `at'[1,1] * log(length) * ///
                               weight^(`at'[1,2]) * length^(`at'[1,3]) `if'
end

// outside the program
matrix initial = (1, 1, 1)
gmm gmm_cobbdoug, nequations(1) parameters(alpha beta gamma) ///
     instruments(weight length) from(initial) hasderivatives
```

Generalizing our Program

- We can make the program estimate any equation of the form:

$$y = \beta_0 x_1^{\beta_1} x_2^{\beta_2} \cdot \cdots \cdot x_k^{\beta_k} + u$$

```
program gmm_cobbdoug
    syntax varlist if, at(name) lhs(varname) rhs(varlist) [derivatives(varlist)]

    local j = 2  // `at'[1,1] is the multiplicative term, start with `at'[1,2]
    local factor_str ""
    foreach var of varlist `rhs' {
        if (`j' > 2) {
             local factor_str "`factor_str' * "
        }
        local factor_str "`factor_str'`var'^(`at'[1,`j'])"
        local `++j'   // equivalent to local j = `j' + 1
    }

    quietly replace `varlist' = `lhs' - `at'[1,1] * `factor_str' `if'

    // Derivatives part on next slide...
```

Generalizing our Program

```
    // continued from last slide ...

    if ("‘derivatives’" == "") { // if derivatives unnecessary
         exit
    }
    local j = 1
    foreach var of varlist ‘derivatives’ {‘
        if (‘j’ == 1) {
            qui replace ‘var’ = -1 * ‘factor_str’ ‘if’
        }
        else {
            local rhs_counter = ‘j’ - 1
            local curvar : word ‘rhs_counter’ of ‘rhs’
            qui replace ‘var’ = -1 * ‘at’[1,1] * log(‘curvar’) * ‘factor_str’ ‘
        }
        local ‘++j’   // equivalent to local j = ‘j’ + 1
    }
end
```

Summary

- Programs allow you to encapsulate code and use it from multiple do-files or the command line.
- The new `gmm` command lets you estimate GMM models flexibly.
  - With substitutable expression you can estimate a system of equations with one (long) line of code.
  - With moment-evaluating programs you can increase functionality.
- Two help files will be your best guides, except for these magnificent slides
  - `help syntax` for how to get inputs to your program.
  - `help gmm` (but mainly the manual page) for how to translate your GMM model to code.