On the shoulders of giants: Writing wrapper commands in Stata

Sebastian Kripfganz

University of Exeter Business School, Department of Economics, Exeter, UK
www.kripfganz.de @Kripfganz

29th London Stata Conference
September 7, 2023

On the shoulders of giants

- At the 2017 UK Stata Users Group Meeting, Nicholas Cox gave a talk entitled

  On the shoulders of giants, or not reinventing the wheel

  in which he presented a collection of (little known) commands that help reducing coding work. Many of those commands make a user's or programmer's life easier by

  1. providing a simplified syntax for a specific task, which is eventually performed under the hood by more powerful commands;
  2. extending the functionality of existing commands, and thus removing the need to resort on cumbersome workarounds.

Wrapper commands

- Defining characteristics of a wrapper subroutine (command or function):

Definition

A wrapper is a subroutine whose main purpose it is to call another subroutine. The wrapper extends this other subroutine or makes it easier to use, without reimplementing its functionality.

- Wrappers can wrap around other wrappers in a hierarchical way.
    - Often, the main purpose of the higher-layer wrappers is syntax parsing (plus some checks and data preparation tasks). Every layer translates the initially simple syntax into the more general syntax of the lower-layer command. The user only needs to be concerned with the basic syntax of the top-layer command.
- Wrappers facilitate the use of tailored (one-line) command lines, and thus avoid the repeated implementation of cumbersome (error-prone) tasks.

Wrapper commands

- Leading use cases for wrapper commands:
  1. Commands that perform a specific task as a special case of a more general procedure:
     - Many estimators for specific models are special cases of maximum-likelihood or generalized method-of-moments estimators. Wrapper commands translate relatively simple syntax into the more complex syntax of the flexible `ml` or `gmm` commands, or they directly wrap around Mata's `optimize()` or `moptimize()` functions.
     - Essentially any (community-contributed) command which implements specific types of graphs is a wrapper around the flexible `graph twoway`; see the dataviz packages of Asjad Naqvi, among others.
  2. Commands that extend the functionality of an existing command:
     - For example, the community-contributed `tsegen` command (Robert Picard and Nicholas Cox, 2015, SSC) extends `egen` to accommodate time series variable lists.

Example: mixed-effects estimation

- Many official commands can be regarded as wrapper commands, and often there is a hierarchical structure.
- For example, this applies to most `me` commands for mixed-effects estimation:
  1. `melogit` parses the user inputs for the specific regression model and translates them into the general `meglm` syntax.
  2. `meglm` processes the syntax and then wraps around the undocumented `_me_estimate`.
  3. `_me_estimate` performs further checks and translates the syntax into the format required by `gsem`.
  4. `gsem` prepares the maximum-likelihood estimation with `ml`.
  5. Since Stata 11, `ml` just passes through all input to the undocumented `mopt`.
  6. `mopt` sets up the numerical optimization with Mata's `moptimize()` function.
  7. The results are then passed back up the hierarchy, and eventually `melogit` wraps around the undocumented `_me_display`, which displays the estimation output.

Example: spatial data visualization

- The `grmap` command for drawing maps to visualize spatial data is another interesting example:
  1. The only purpose (besides a few additional checks) of `grmap` is to call the undocumented `g_spmap`, which is essentially identical to the community-contributed `spmap` command (Maurizio Pisati, 2007, SSC).
  2. `g_spmap` (or `spmap`) translate the input into the general `graph twoway` syntax, which displays the graph.
- The community-contributed `geoplot` (Ben Jann, 2023, SSC) provides even more flexibility for drawing maps. It, too, wraps around `graph twoway`.
- `bimap` (Asjad Naqvi, 2022, SSC) extends the functionality of `spmap`, and thus is an example of one community-contributed command wrapping around another community-contributed command.

Plugins

- Commands can wrap around subroutines written in different programming languages:
  - Compiled C (or C++) code can be linked to Stata code through a plugin; see `https://www.stata.com/plugins/`. The main purpose is to speed up execution time or to avoid reinventing the wheel by wrapping around already existing code. Much of Stata's low-level functionality is essentially implemented this way.
  - The `python` command allows to include third-party Python packages.
  - The community-contributed `rcall` package (E.F. Haghish, 2019, Stata Journal) and `rsource` package (Roger Newson, 2007, SSC) enable language interfacing with R.
  - With some creativity, subroutines written in other languages can be called as well.
    - For example, the community-contributed `boottest` command (David Roodman, Morten Nielsen, James MacKinnon, and Matthew Webb, 2019, Stata Journal) connects to a Julia back-end through Python.

Wrapper structure: r-class program

- Commands performing general tasks (other than estimation) are typically r-class commands. A *wrapper* might call another r-class command, the *wrappee*, and return the received results with or without amendment:

```
program define wrapper, rclass
    version 18
    ...                 // syntax processing and other preliminary tasks
    quietly wrappee ... // call to lower-layer command (e.g., summarize)
    return add          // add wrappee results to results returned by wrapper
    ...                 // potential amendments or additions to the results
    ...                 // display of results in desired format
    return scalar answer = 42 // add or change returned scalars, matrices, or locals
end
wrapper ...
return list       // display list of returned results
display r(answer)
```

Wrapper structure: e-class program

- Estimation commands are e-class commands. If a *wrapper* calls another e-class command, the estimation results can be passed through automatically:

```
program define wrapper, eclass
    version 18
    ...          // syntax processing and other preliminary tasks
    wrappee ... // call to the lower-layer command (e.g., regress)
    ...          // potential amendments or additions to the results
    ereturn local cmd "wrapper"           // add or change returned results
    ereturn local cmdline ‘"wrapper ‘0’"’
end
wrapper ...
ereturn list      // display list of returned results
matrix list e(b)
```

Wrapper structure: e-class program

- Alternatively, the estimation results can be processed further

```
program define wrapper, eclass
    version 18
    ...                 // syntax processing and other preliminary tasks
    quietly wrappee ... // call to the lower-layer command (e.g., regress)
    ...                 // potential amendments or additions to the results
    ereturn post ...    // return coefficient vector and variance-covariance matrix
    ereturn local cmd "wrapper"           // return further results
    ereturn local cmdline ‘"wrapper ‘0’"’
    ereturn display     // display coefficient table
end
wrapper ...
```

  - Note that `ereturn post` clears all estimation results returned by `wrappee`. Any relevant result needs to be processed and returned separately.
  - Alternatively, `ereturn repost` just amends the coefficient vector and variance-covariance matrix without clearing the other estimation results.

Version control

- Wrapping around other subroutines creates package dependencies. Changes in syntax and functionality of lower-layer *wrappees* may potentially break the code of higher-layer *wrappers* or lead to different results than under previous versions.
    - For official Stata commands, this is typically not a concern (from the user perspective) due to the strong emphasis on backward compatibility. Occasionally, there are more substantial changes to syntax, functionality, or the way results are reported, but version control with the `version` command ensures that old code continues to work as originally intended.
    - Version control does not ensure exact reproducibility of previous results if a command is updated to fix a bug.

Version control

- At the beginning of a program within an ado-file, a version statement should be included, e.g.

```
program define wrapper
    version 12.1
    ...
end
```

  - This ensures that (official) commands are interpreted as they were in the specified Stata version.
  - It does not preclude the use of new functionality introduced in later Stata versions, but extra precaution is needed.

Version control: system constants

- The version statement sets the minimum Stata version requirement for any user of the program; i.e., `c(stata_version)` $\geq$ `c(version)`, where
  - `c(stata_version)` is the version of Stata under which the code is running.
  - `c(version)` is the version set by the program's `version` statement.
- This still allows users to call the program from a do-file (or interactively) run under version control for an earlier version; i.e. `c(userversion)` $<$ `c(version)`, where
  - `c(userversion)` is the "user version" set in the user's do-file.
- Similarly, the program can be called from another program with a version statement referring to an earlier version; i.e. `_caller()` $<$ `c(version)`, where
  - `_caller()` is the version of the program or session invoking the current program.

Version control: system constants

- Consider the program `wrapper` called under version control (interactively or from a do-file), where `wrapper` calls another program `wrappee`, both written for different Stata versions:

```
program define wrapper
    version 17
    wrappee
end
program define wrappee
    version 11
    ...
end
version 13: wrapper
```

Version control: system constants

- Within `wrapper`, the system constants return the following values:
  - `c(stata_version)` = 18, the version of Stata that is running.
  - `c(version)` = 17, as set by the version statement within `wrapper`.
  - `c(userversion)` = 13, as set by the user when calling `wrapper`.
  - `_caller()` = 13, as set by the user when calling `wrapper`.
- Within `wrappee`, they return the following values:
  - `c(stata_version)` = 18, the version of Stata that is running.
  - `c(version)` = 11, as set by the version statement within `wrappee`.
  - `c(userversion)` = 13, as set by the user when calling `wrapper`.
  - `_caller()` = 17, as set by `wrapper` when calling `wrappee`.

Version control: system constants

- Official Stata commands normally rely on `_caller()` to determine the relevant version under which a command shall be run.
- For major Stata improvements that are intended to benefit all programs irrespective of the version statement, and which do not affect syntax or stored results, Stata checks `c(userversion)` instead of `_caller()`.
  - Unless a user explicitly runs a command under version control (in a do-file or interactively), the improvement will also benefit any program written for an earlier version before the improvement was implemented.
  - For example, the faster sort algorithm introduced in version 17 generally also benefits the `wrappee` program written for version 11. However, this is not the case here, if the user invoked `version 13`.

Quizzes & Grumbles

- Stata 15 introduced a major change regarding the naming convention for free parameters in fitted models, with an effect on row and column names in `e(b)` and `e(V)`. Consider the following example:

```
program define wrapper
    version 17
    melogit ‘0’ // melogit is the wrappee
end
webuse bangladesh
version 13: wrapper c_use urban age children || district: , nolog
```

- Does this code return `e(b)` and `e(V)` results equivalent to one of the following?
  1. `version 13: melogit c_use urban age children || district: , nolog`
  2. `version 17: melogit c_use urban age children || district: , nolog`

Quizzes & Grumbles

- Neither; Stata's `melogit` (or some command further down in the hierarchy) gets confused by the different versions in `c(userversion)` and `_caller()`:

```
(some output omitted)
invalid matrix stripe;
/:var(_cons[district])

Mixed-effects logistic regression               Number of obs     =      1,934
Group variable: district                        Number of groups  =         60

                                                Obs per group:
                                                              min =          2
                                                              avg =       32.2
                                                              max =        118

Integration method: mvaghermite                 Integration pts.  =          7

                                                Wald chi2(5)      =     125.36
Log likelihood = -1222.0022                     Prob > chi2       =     0.0000
------------------------------------------------------------------------------
       c_use | Coefficient  Std. err.      z    P>|z|     [95% conf. interval]
-------------+----------------------------------------------------------------
          c1 |   .7222158   .1183964     6.10   0.000     .4901631    .9542684
          c2 |  -.0307946   .0078383    -3.93   0.000    -.0461574   -.0154318
          c3 |   .4201486   .0576332     7.29   0.000     .3071897    .5331076
          c4 |  -1.432882      .1362   -10.52   0.000    -1.699829   -1.165935
          c5 |   .2189028   .0736679                      .1131865    .4233585
------------------------------------------------------------------------------
LR test vs. logistic model: chibar2(01) = 45.05       Prob >= chibar2 = 0.0000
```

Version control: system constants

- If it is important for `wrappee` (in the previous example: `melogit`) to know under which version `wrapper` was called, `wrapper` can be modified as follows:

```
program define wrapper
    local vv : display "version " string(_caller()) ":"
    version 17
    `vv' wrappee
end
```

    - This is how hierarchical official Stata commands are typically set up.

Quizzes & Grumbles

- Another major change came with Stata 14, when the 32-bit KISS random-number generator (RNG) was replaced by the 64-bit Mersenne Twister RNG. Consider:

```
program define wrapper
    version 17
    version 13: {
        set seed ‘0’
        display rnormal()
    }
end
version 18: wrapper 20230907
```

- Which random-number generator is invoked here?
  1. 32-bit KISS
  2. 64-bit Mersenne Twister

Quizzes & Grumbles

- The relevant variant of the random-number generator is determined based on `c(userversion)`, not `_caller()`. The version statements within `wrapper` are simply ignored. Here, a seed is set for the new 64-bit Mersenne Twister RNG.
- To force usage of the old 32-bit Kiss RNG, the `user` option needs to be specified in the version prefix:

```
program define wrapper
    version 17
    version 13, user: {
        set seed ‘0’
        display rnormal()
    }
end
version 18: wrapper 20230907
```

  - There is hardly any relevant case for a programmer to set the user version.

Quizzes & Grumbles

- Some care may be needed to ensure that `set seed` and `rnormal()` (or any other random-number function) are called under the same user version. The following example yields non-reproducible results:

```
program define wrapper
    version 17
    version 13, user: set seed ‘0’
    display rnormal()
end
version 18: wrapper 20230907
```

  - The seed is set under user version control for the old 32-bit Kiss RNG, while `rnormal()` obtains a random draw under the new 64-bit Mersenne Twister RNG, for which a seed has not been set here.
  - Technical note: `rnormal_kiss32()` always uses the old RNG, while `rnormal_mt64()` always uses the new RNG, but the seed still needs to be set under the appropriate user version.

Version control: system constants

- Unless the command intends to mimic the behavior of official commands under certain Stata versions, `c(userversion)` is not of much use for programmers of community-contributed commands.
- `c(stata_version)` can be used to utilize new features (e.g., frames introduced in Stata 16), while keeping the program accessible under older versions:

```
program define wrapper
    version 13
    if c(stata_version) >= 16 {
        ... // code using new Stata functionality (frames)
    }
    else {
        ... // workaround using old functionality (preserve, restore)
    }
end
```

Version control: system constants

- `_caller()` could still be useful in a community-contributed command to preserve old behavior if an update to the command introduces code-breaking changes:

```
program define wrapper
    version 13
    if _caller() >= 18 {
        wrapper_v18 ‘0’ // new code (fork to version 18 subroutine)
    }
    else {
        wrapper_v13 ‘0’ // old code (fork to version 13 subroutine)
    }
end
```

  - This way, any do-file or program with a version statement prior to Stata 18 (but at least version 13) will continue to run. However, the improvement will only be available to users with the latest Stata version. Moreover, do-files will still break if they were written since the release of Stata 18 but before the code-breaking update.

Version control for Mata functions

- For Mata functions, version control is achieved with a Stata version statement at the beginning of the do-file (mata-file), in which the Mata function is declared:

```
version 12.1
mata:
function myfcn() {
    ...
}
end
```

- Mata functions can be either compiled at run time or distributed with the package in compiled form (as a mo-file or together with other functions in an mlib-library). Crucially, a compiled Mata function/library cannot be used in older Stata versions. The Mata functions need to be compiled in the oldest Stata version that shall be supported by the package, or the source code must be supplied and compiled by the user at run time.

Version control for Mata functions

- It is challenging to write packages that are supported by older Stata versions while utilizing official Mata functions introduced in later versions.
    - Compilers in old Stata versions do not know the functions introduced in later versions.
    - If the source code is provided, any reference to the not yet supported function must be shielded in a function that is never called at run time by the old Stata version. `c(stata_version)` can be used to determine the relevant functions for compilation at run time.
    - If Mata libraries are provided, different libraries need to be distributed for different Stata versions, which requires careful version management. For a discussion of potential solutions, see `https://www.statalist.org/forums/forum/general-stata-discussion/mata/1319628-making-mata-libraries-for-multiple-stata-versions-gracefully`

Version control for community-contributed packages

- There is no common version control standard for community-contributed packages:
  - Some authors return a version statement in the returned results after running the command; e.g., `e(version)` in `xtabond2` (David Roodman, 2009, Stata Journal).
  - Some commands have a `version` option for displaying the package version; e.g., `ivreg2` (C. F. Baum, M. E. Schaffer, and S. Stillman, 2003, 2007, Stata Journal), `reghdfe` (Sergio Correia, 2014, SSC, prior to version 6), and `xtdcce2` (Jan Ditzen, 2018, 2021, Stata Journal).
  - The `moremata` package (Benn Jann, 2005, SSC) contains a Mata function `mm_version()` which returns the version number.
  - It is a convention (although not a requirement) to specify a package version number in the first line of an ado-file:

    ```
    *! version 1.0.0  07sep2023  Sebastian Kripfganz
    ```

    At the 2023 Stata Conference in Stanford, Sergio Correia (with Matthew Seay) presented the community-contributed `require` package, which extracts the version number from this "starbang" line and thus can be used to assert that a certain (minimum) package version is installed.

Version control for community-contributed packages

- When wrapping around community-contributed commands, version control can be implemented in a case-specific way if supported by the wrappee, especially if both the wrapper and the wrappee are developed by the same programmer.
- Unless community programmers are similarly careful as StataCorp's developers in ensuring backward compatiblity, guards are needed against potential code-breaking changes from updates to the wrappee.
  - Requiring a specific version of the wrappee is not a solution, because users typically only have access to the latest version.
  - Most updates to community-contributed packages are beneficial (bug fixes, new functionality, performance improvements, etc.) or at least harmless.
  - However, it cannot be taken for granted that wrappee updates will remain compatible with the wrapper code. Eventually, it is unavoidable to monitor whether community-contributed wrappees are still compatible with the wrapper code, and to make adjustments when necessary.

Package dependencies

- If the wrappee is another community-contributed package, the wrapper needs to assert that the package is installed. If necessary and possible, the correct version needs to be asserted as well.
- There are two possible approaches:
  1. Ascertain the existence of the (required version of the) wrappee package before it is used (ideally before running any other wrapper code):

     ```
     capture which wrappee        // check existence of a file in ado-path
     capture findfile wrappee.txt // occasionally useful in specific cases
     ```
  2. Capture any errors returned by the wrappee (or its nonexistence) when it is called:

     ```
     capture wrappee ...     // captured call of a Stata program
     capture mata: wrappee() // captured call of a Mata function
     ```
- The second approach is useful to guard against code-breaking wrappee updates. It often makes sense to combine both approaches.

Package dependencies

- capture can also be used for blocks of code:

```
capture {
    ...
}
```

- capture avoids that the code execution stops with an uninformative error message. If the wrappee is not found or returns an error message when called, capture returns a nonzero return code that needs to be processed subsequently:

```
if _rc != 0 {
    local rc = _rc
    ...              // potentially needed cleanup code
    exit ‘rc’        // any appropriate return code can be used here
}
```

  - The reaction can be tailored to the specific return code.

Package dependencies

- Example from the first few code lines of `geoplot` (Ben Jann, 2023, SSC):

```
capt which colorpalette
if _rc==1 exit _rc
local rc_colorpalette = _rc

capt findfile lcolrspace.mlib
if _rc==1 exit _rc
local rc_colrspace = _rc

capt mata: assert(mm_version()>=200)
if _rc==1 exit _rc
local rc_moremata = _rc

if `rc_colorpalette' | `rc_colrspace' | `rc_moremata' {
    if `rc_colorpalette' {
        di as err "{bf:colorpalette} is required; " _c
        di as err "type {stata ssc install palettes, replace}"
    }
    if `rc_colrspace' {
        di as err "{bf:colrspace} is required; " _c
        di as err "type {stata ssc install colrspace, replace}"
    }
    if `rc_moremata' {
        di as err "{bf:moremata} version 2.0.0 or newer is required; " _c
        di as err "type {stata ssc install moremata, replace}"
    }
    exit 499
}
```

Package update guidelines

- Avoid code-breaking changes as far as possible;
  - If new functionality changes the behavior of a command option, introduce a new option with different name, but keep the (now undocumented) old option for backward compatibility.
  - If you change the way you return results in `r()` or `e()`, keep returning previous results as hidden/historical results.
- If code-breaking changes are unavoidable, use Stata version control to call old code under old Stata versions. Ideally, try to time the release of a code-breaking update with the release of a new Stata version.
- In the case of substantial code-breaking changes, consider releasing the new version under a different package name.

Wishes & Grumbles

- Community-driven initiatives for version control are a way forward, but not an ultimate solution.
  - `require` is quite flexible, but it just creates another package dependency by itself.
- It would be desirable if StataCorp was recommending a version control standard for community-contributed packages and provide official tools (akin to `require`) for version checking.
- It would also be useful if there were more frequent Stata version increments: 18.0, 18.1, 18.2, . . .

Example 1: `spxtivdfreg`

- Vasilis Sarafidis and I recently released an update to our `xtivdfreg` package (2021, Stata Journal) for the instrumental-variables estimation of large-T panel data models with common factors.
- The new version contains the command `spxtivdfreg`, which adds functionality for the estimation of spatial panel data models:

$$y_{it} = \alpha y_{i,t-1} + \rho \sum_{j=1}^{N} w_{ij} y_{jt} + \phi \sum_{j=1}^{N} w_{ij} y_{j,t-1} + \boldsymbol{\beta}' \mathbf{x}_{it} + \boldsymbol{\delta}' \sum_{j=1}^{N} w_{ij} \mathbf{x}_{jt} + u_{it}$$

where w_{ij} are spatial weights collected in a spatial weights matrix $\mathbf{W}$, and the errors are assumed to have a common-factor structure:

$$u_{it} = \gamma'_{y,i} \mathbf{f}_{y,t} + \varepsilon_{it}$$

Example 1: spxtivdfreg

- spxtivdfreg is a wrapper for xtivdfreg:
    - spxtivdfreg has a few more options related to spatial model, but otherwise has the same syntax as xtivdfreg.
    - spxtivdfreg parses the spatial options, loads the spatial weights matrix (from an Excel or delimited text file, as a Stata or Mata matrix, or as an spmatrix object), carries out related checks, and creates temporary variables for spatial lags.
    - spxtivdfreg then calls xtivdfreg to perform the estimation.
    - Normally, the wrapper would need to process the output of the wrappee and display the results in the appropriate form. Here, I took a shortcut by adding a small subroutine to xtivdfreg, which amends the results table if called by spxtivdfreg.
    - The dedicated spxtivdfreg postestimation command estat impact can subsequently be used to compute short-run and long-run average direct, indirect, and total impacts (as well as corresponding Delta-method standard errors), which are typically the objects of interest when estimating spatial autoregressive models.

```
net install xtivdfreg, from(http://www.kripfganz.de/stata/)

help spxtivdfreg
```

Example 1: `spxtivdfreg`

- The minimum Stata version required for `spxtivdfreg` is Stata 13.
- Some options rely on new features introduced in later versions:
  - The option to import **W** from an external file requires at least Stata 14. For this purpose, if `c(stata_version)` $\geq$ 16, frames are used instead of `preserve`/`restore`.
  - To load **W** as an `spmatrix` object, using Stata's official `sp` tools, at least Stata 15 is required.
  - Option `std` for the factor extraction from standardized variables requires at least Stata 16.1 with `c(born_date)` $\geq$ `30jun2020`.
- In earlier Stata versions, the command can still be used but with reduced functionality.
- To absorb fixed effects, `xtivdfreg` requires `reghdfe` (Sergio Correia, 2014, SSC), which itself requires `ftools` (Sergio Correia, 2016, SSC).
  - Problems due to missing packages or incompatible versions are caught with `capture`.
- Version control between `spxtivdfreg` and `xtivdfreg` is not an issue because both commands are distributed as part of the same package.

Example 1: spxtivdfreg

```
. spxtivdfreg NPL INEFF CAR SIZE BUFFER PROFIT QUALITY LIQUIDITY,     ///
> absorb(ID) splag tlags(1) spmatrix("W.csv", import) std noheader     ///
> iv(INTEREST CAR SIZE BUFFER PROFIT QUALITY LIQUIDITY, splags lag(1))

Defactored instrumental variables estimation
------------------------------------------------------------------------------
             |               Robust
         NPL | Coefficient  std. err.      z    P>|z|     [95% conf. interval]
-------------+----------------------------------------------------------------
         NPL |
         L1. |   .2898517   .0543794     5.33   0.000     .1832699    .3964334
             |
       INEFF |   .4473766   .1045636     4.28   0.000     .2424357    .6523174
(some coefficients omitted)
-------------+----------------------------------------------------------------
W            |
         NPL |    .394323   .0848855     4.65   0.000     .2279505    .5606955
-------------+----------------------------------------------------------------
     sigma_f |  .64162383   (std. dev. of factor error component)
     sigma_e |  .90381826   (std. dev. of idiosyncratic error component)
         rho |  .33509007   (fraction of variance due to factors)
------------------------------------------------------------------------------
Hansen test of the overidentifying restrictions       chi2(19)    =   18.8252
H0: overidentifying restrictions are valid            Prob > chi2 =    0.4681
```

Example 1: spxtivdfreg

```
. estat impact, sr

Short-run impacts
(output omitted)

. estat impact, lr

Long-run impacts
------------------------------------------------------------------------------
             |              Delta-method
             |     Impact   std. err.      z    P>|z|     [95% conf. interval]
-------------+----------------------------------------------------------------
direct       |
       INEFF |   .6470571   .1593923     4.06   0.000     .3346539    .9594602
(some coefficients omitted)
-------------+----------------------------------------------------------------
indirect     |
       INEFF |   .7694746   .3352843     2.29   0.022     .1123295     1.42662
(some coefficients omitted)
-------------+----------------------------------------------------------------
total        |
       INEFF |   1.416532   .4274884     3.31   0.001     .5786698    2.254393
(some coefficients omitted)
------------------------------------------------------------------------------
```

Example 2: `xtdpdgmmfe`

- My `xtdpdgmm` command (2017, SSC) for the generalized method of moments estimation of linear dynamic panel data models has a fairly complicated syntax due to the flexibility it provides.
  - Correctly specifying all the options requires a good understanding of the underlying econometric theory.
- The latest version of the package contains the command `xtdpdgmmfe`, which acts as a wrapper for `xtdpdgmm`:
  - `xtdpdgmmfe` has a simplified (and hopefully more intuitive) syntax with slightly reduced functionality. Users specify options in accordance with a set of assumptions they make, which requires a less profound econometric background.
  - `xtdpdgmmfe` translates the inputs into the more complicated syntax of `xtdpdgmm` and executes the latter. It also displays the respective `xtdpdgmm` command line and stores it in `e(cmdline)`, which might help users to understand (and possibly amend) the latter's syntax.

```
net install xtdpdgmm, from(http://www.kripfganz.de/stata/)

help xtdpdgmmfe
```

Example 2: xtdpdgmmfe

```
. xtdpdgmmfe n w k, predetermined(w) exogenous(k) stationary collapse curtail(4) twostep vce(robust) nofooter noheader

  xtdpdgmm L(0/1).n w k , model(difference) gmmiv(k, lagrange(0 .)) gmmiv(k, lagrange(0 0) difference model(level))
> gmmiv(L.n w, lagrange(1 .)) gmmiv(L.n w, lagrange(0 0) difference model(level)) collapse curtail(4) twostep vce(robust)
> nofooter noheader

Generalized method of moments estimation

Fitting full model:
Step 1         f(b) =  .00375022
Step 2         f(b) =   .1724788

                                   (Std. err. adjusted for 140 clusters in id)
------------------------------------------------------------------------------
             |              WC-Robust
           n | Coefficient  std. err.      z    P>|z|     [95% conf. interval]
-------------+----------------------------------------------------------------
           n |
         L1. |   .4298095   .1159962     3.71   0.000     .2024612    .6571578
             |
           w |  -1.228313    .209562    -5.86   0.000    -1.639047   -.8175787
           k |   .2739143   .0514984     5.32   0.000     .1729793    .3748493
       _cons |   4.524447   .7540113     6.00   0.000     3.046612    6.002282
------------------------------------------------------------------------------
```

Conclusion: wrapper commands in Stata

- It is not difficult to write a wrapper command in Stata. In fact, most commands can be seen as wrappers, often with a hierarchical wrapping architecture.
- Managing package dependencies and version control is challenging for community-contributed packages, as there is no commonly accepted standard.
- As a programmer, keep in mind that others might want to write a wrapper for your command:
  - Return all relevant results in `r()` or `e()`.
  - As far as possible, retain backward compatibility.