Mata, the missing manual

William Gould

President and Head of Development
StataCorp LP

July 2011, Chicago

Mata, the missing manual

Before we begin, . . .

Apologies to Pogue Media and O'Reilly Media,
creators of the fine Missing Manual series,
"the book that should have been in the box".

(Unrelated to Mata, their web site is http://missingmanuals.com)

Introduction

- Mata is Stata's matrix programming language.

- StataCorp provides detailed documentation but has failed to provide any guidance as to when and how to use the language. This talk addresses StataCorp's omission. I will discuss
  - How to include Mata code in Stata ado-files.
  - When to include Mata code (and when not to).
  - Mata's broad concepts.

- This talk is the prelude to the *Mata Reference Manual*.

- This talk will be advanced.

Mata Matters (?)

- Cute title of *Stata Journal* column, title fashioned by Nicholas J. Cox; content by me.
- Why does Mata matter? Because it is an important development language for Stata.
    - StataCorp uses it.
    - `sem`, `mi`, `xtmixed`, *etc.* were developed using it.
- We at StataCorp can better and more quickly write code using it.
- You can, too.

Problem with Mata Reference Manual

The problem with the *Mata Reference* manual is that . . .

- It tells you all the details
- It never tells you how to put it all together.
- It gets into the details before it even motivates you.
- It's written at a high level.

. . . and because of that, we developers at StataCorp love this manual. It gets right to the details that are easy to forget.

We use it constantly.

Outline

1. Mechanics of including Mata code
    - We start gently, at the end of NC-151.
    - We end up discussing big—really big—systems.
2. Appropriate and inappropriate use of Mata
3. Mata concepts
4. Example
5. Back to concepts, this time advanced
6. and Debugging!

Do-file

script1.do:

```
version 12
clear all

. . .
(Stata code)
. . .
```

- Called a script.
- Used for data management, analysis, housekeeping.
- Should do just one of those tasks.
- Should be re-runnable.
- Exception: housekeeping (deleting old files).
- Another do-file will run all the scripts in order.
- Stored in c:/myproject.

Do-file with Stata program

script2.do:

```
version 12
clear all

...
program myutility
    ...
end

myutility ...
...
```

- Programs are typically short, 1–15 lines.
- Programs typically include no parsing; they are specific to the problem at hand.
- Programs often used to perform the same operation on many variables.

Do-file with in-line Mata

script3.do:

```
version 12
clear all

...
mata:
...
end
...
```

- Same as using Mata interactively.
- Better: You can modify and debug (do-file is re-runnable).
- See “Mata Matters” in *SJ* for examples.
- New `putmata` and `getmata` commands make it easy.

script4.do:

```
version 12
clear all
...
mata:
function myutility(...)
{
        ...
}
myutility("var1")
myutility("var2")
end
...
```

- More advanced form of do-file with in-line Mata, or do-file with Stata program.
- Mata function might take arguments, but regardless, it may have hardcoded variable names, etc., for the problem at hand.

Do-files, summary

- Do-files are used for specific project.
- Create a project directory (folder).
- Keep your data and do-files in it.

(Perhaps you keep the original data somewhere else.)

Ado-file (simple)

mycmd.ado:

```
*!  version 1.0.0 wwg 9sep2010

program mycmd
    version 12
    ...
end
```

- mycmd does something useful across projects.
- Stored in PERSONAL, e.g., C:/ado/personal/
- We're really programming now.

It is the generalization across projects that distinguishes real programs from mere do-files.

Serious and sophisticated work is sometimes put into project-specific do-files.

Ado-file with private subroutine(s)

mycmd.ado:

```
*!  version 1.0.0 wwg 9sep2010

program mycmd
    version 12
    ...
    mysubroutine ...
    ...
end

program mysubroutine
    ...
end
```

- Always good style, even for simple problems.
- Most programmers use too few subroutines.

Aside: How to write ado-files, step 1 of 3

mycmd.do:

```
clear all
program mycmd
    version 12
    ...
end

program mysubroutine
    ...
end

sysuse auto             /* test subroutines */
mysubroutine ...
assert ...

mycmd ...
                        /* test mycmd */
assert ...
...
```

1. To write `mycmd.ado`, first write `mycmd.do`
2. Save in c:/where_I'm_working/
3. Make it work
4. Don't even try to test `mycmd` until subroutines known to work
5. Keep adding to tests at bottom.

Aside: How to write ado-files, step 2 of 3

Split `mycmd.do` into `mycmd.ado` and `testmycmd.do`:

mycmd.ado:

```
*!  version 1.0.0 wwg 9sep2010

program mycmd
    version 12
    ...
end

program mysubroutine
    ...
end
```

(File `testmycmd.do` on next slide.)

Aside: How to write ado-files, step 2 of 3

mycmd.do:

see previous screen

testmycmd.do:

```
clear all

sysuse auto   /* test mycmd */
mycmd ...
assert ...
...
```

- Split `mycmd.do` into two files
- Save in c:/where_I'm_working/
- Make it work

Aside: how to write ado-files, step 3 of 3

1. Move `mycmd.ado` to PERSONAL (e.g., c:/ado/personal/)
2. Move `testmycmd.do` to c:/mycerts/
3. Add "`do testmycmd`" to c:/mycerts/master.do
4. Make it work
5. Optionally remove c:/where_I'm_working/

Back to the main topic . . .
We left off with **Ado-file with private subroutine**, meaning an ado-file subroutine.

Next is **Ado-file with Mata subroutine**.

Ado-file with Mata subroutine

```
*!  version 1.0.0 wwg 9sep2010

program mycmd
    version 12
    ...
    mata:  myfunction(...)
    ...
end

version 12
mata:
... myfunction(...)
{
    ...
}
end
```

(. . . finally)

Ado-file with Mata subroutine

I will show you a real example:

- Macro *varlist* contains a list of variables. It might be very, very long.
- I want to display "using . . ."
- If 1 variable, I want "using a"
- If 2 variables, I want "using a and b"
- if 3 variables, I want "using a, b and c"
- . . .

Ado-file with Mata subroutine

mycmd.do:

```
*!  version 1.0.0 wwg 9sep2010
program mycmd
    version 12
    ...
    mata:  st_local("toprint", printable("`varlist'"))
    display as txt "{p 0 4 2}"
    display as txt "using `toprint'"
    ...
end

version 12
mata:
string scalar printable(string scalar s)
{
    ... next slide ...
}
end
```

mycmd.ado:

. . . top of file on previous slide . . .

```
mata:
string scalar printable(string scalar s)
{
    real scalar         i
    string rowvector    tokens
    string scalar       toret

    tokens = tokens(s)
    if (cols(tokens)<2) return(strtrim(s))

    toret = tokens[1]
    for (i=2; i<cols(tokens); i++) {
        toret = toret + ", " + tokens[i]
    }
    return(toret + " and " + tokens[cols(tokens)])
}
end
```

Remember the outline . . .

```
*!  version 1.0.0 wwg 9sep2010

program mycmd
    version 12
    ...
    mata:  myfunction(...)
    ...
end

version 12
mata:
... myfunction(...)
{
    ...
}
end
```

Sometimes you don't need the bottom half!

mycmd.ado:

```
*!  version 1.0.0 wwg 9sep2010

program mycmd
    version 12
    ...
    mata:  st_local("macroname", ...)
    ...
    ...`macroname'...
    ...
end
```

- We use Mata in the ado-file.
- We have no Mata subroutine . . .
- . . . because we use only Mata built-in functions!

No bottom half, example

A popular question on Statalist is

“I have a macro that’s too long for Stata’s string-manipulation functions. What do I do?”

Answer: Use Mata. Macros are not too long for Mata’s string-manipulation functions.

Example: ”I need to reverse the string”

(Yes, I know there is a `reverse()` function among the extended macro functions.)

Solution, reverse the string:

mycmd.ado:

```
*!  version 1.0.0 wwg 9sep2010

program mycmd
    version 12
    ...
    mata:  st_local("reversed", strreverse("`yourmacro'"))
    ...
    ...`reversed'...
    ...
end
```

- `strreverse()` is a built-in function of Mata.
- Stata macro `reversed` now contains the reversed contents of `yourmacro`.

The ultimate ado-file

- The ultimate ado-file contains
  - The main routine
  - Stata subroutines
  - Mata subroutines
  - Mata sub-subroutines
- Sub-subroutines are Mata routines called by other Mata routines.
- We have not discussed when to use Stata and when to use Mata; we will.

Your ado file should look like this . . .

Ultimate ado-file

mycmd.ado:

```
*!  version 1.0.0 wwg 9sep2010

program mycmd
    version 12     ...
end

Stata subroutines go here

version 12
mata:

Mata subroutines go here

Mata sub-subroutines go here

end
```

How to write the ultimate ado-file

Create mycmd.ado as shown above, and create

mycmd.do:

```
clear all
set matastrict on
do mycmd.ado

sysuse auto, clear
mata:                    /* test Mata [sub-]subroutines */
assert(...)
...
end

                         /* test Stata subroutines */
assert ... ...

                         /* test mycmd */
assert ... ...
```

How to write the ultimate ado-file

- `mycmd.ado` and `mycmd.do` are stored in c:/where_I'm_working/
- You don't have to `set matastrict on`, but if you do not, budget more time for writing and debugging.
- Look at the notes produced by Mata when it compiles your code, and eliminate them. The notes are not just style issues. They often indicate conceptual errors on your part.
- Sometimes Mata is mistaken; you do intend what Mata has flagged. Learn about `#pragma`; see help `[m2] pragma`. You can suppress individual notes.

Systems

- A system is a set of commands that work together to solve one problem.
- Stata's `mi` command is an example of a system.
- A system has one or more of the following characteristics
  - multiple entry points
  - states
  - common subroutines
- We will postpone discussion of systems to end of talk and hope we get to it.

End mechanics / begin substantive

- End of the mechanical comments
  . . . but see Systems at end

- We begin the substantive
  . . . and talk exclusively about Mata.

When to use which

- Stata is better for . . .
    - Parsing standard syntax
    - Data management
    - Scripting existing Stata commands
    - Outputting (usually)
    - Posting saved results

- Mata is better for . . .
    - Parsing non-standard syntax (including files)
    - Performing matrix operations
    - Non-scripting applications
    - Outputting (when complicated)

When to use which

- Stata is a better scripting language than Mata
- Mata is a better programming language than Stata

Scripting versus programming

1. A script is a sequence of steps to be followed one after the other.
2. In real life, scripts are read and executed by intelligent people.
3. Stata is not intelligent, but it is more intelligent than Mata. Stata can understand big, broad instructions, along with a few detailed instructions.
4. With the exception of matrices, Mata doesn't understand big, broad instructions. Mata understands details and makes you spell them out, sometimes in painful detail.

Scripting versus programming

1. A script is a sequence of steps to be followed one after the other.

2. In real life, scripts are read and executed by intelligent people.

3. Stata is not intelligent, but it is more intelligent than Mata. Stata can understand big, broad instructions, along with a few detailed instructions.

4. With the exception of matrices, Mata doesn't understand big, broad instructions. Mata understands details and makes you spell them out, sometimes in painful detail.

Mata makes you spell out details

Mata makes you spell out details **and in return . . .**

- Mata is fast. In part, that's because you put the details in the most efficient order.

- Mata can do things Stata can't do; all you have to do is spell them out.

Mata has features to make spelling out the details easier.

You need to learn them.

When to use which II

I said, “Most programmers use too few subroutines”.

- In Stata, there’s an execution-time cost to subroutines.
- In Mata, that cost is near zero.

If you are not using subroutines in Stata for reasons of speed, that’s a sign you should be using Mata.

Scripting vs. programming, example
Linear regression

In Stata it's easy.

> Tell Stata you want to regress one variable on others and over what observations.

In Mata,

> You must not only provide the formula, you must provide lots more; see next slide

Scripting vs. programming, linear regression

Think of the conversation with Mata as going

1. What do you mean by an observation?
2. What do you mean by a variable?
3. Where shall we find this thing you call "data" that is a collection of "observations" on "variables"?
4. How shall we keep track of these "variables"?
   I love integers. Names, you say? Are those like strings?
   I have really long strings. Really, however, integers are better.

Scripting vs. programming, linear regression

Think of the conversation with Mata as going

1. What do you mean by an observation?
2. What do you mean by a variable?
3. Where shall we find this thing you call "data" that is a collection of "observations" on "variables"?
4. How shall we keep track of these "variables"?
   I love integers. Names, you say? Are those like strings?
   I have really long strings. Really, however, integers are better.

Scripting vs. programming, linear regression

Think of the conversation with Mata as going

1. What do you mean by an observation?
2. What do you mean by a variable?
3. Where shall we find this thing you call "data" that is a collection of "observations" on "variables"?
4. How shall we keep track of these "variables"?
   I love integers. Names, you say? Are those like strings?
   I have really long strings. Really, however, integers are better.

Scripting vs. programming, linear regression

What makes the conversation tolerable

- Mata can call Stata . . .
- . . . so we can use Stata's concepts.

Still, we specify more details. For instance, . . .

In Stata, you seldom think of row and column numbers.
Row and column numbers are all that Mata understands.

Mata concepts

- Forget about Stata macros and locals.
  Just thinking about them will will mislead you.

- Everything is a variable in Mata.
  Mata variables have nothing to do with Stata variables.

- If it's not a variable, it's a function.
  There are no other alternatives.
  - Functions accept arguments (variables).
  - Functions return results (which you store in variables).

Mata concepts

- Forget about Stata macros and locals.
  Just thinking about them will will mislead you.

- Everything is a variable in Mata.
  Mata variables have nothing to do with Stata variables.

- If it's not a variable, it's a function.
  There are no other alternatives.
  - Functions accept arguments (variables).
  - Functions return results (which you store in variables).

Mata concepts

- Forget about Stata macros and locals.
  Just thinking about them will will mislead you.

- Everything is a variable in Mata.
  Mata variables have nothing to do with Stata variables.

- If it's not a variable, it's a function.
  There are no other alternatives.
  - Functions accept arguments (variables).
  - Functions return results (which you store in variables).

Mata concepts

- Forget about Stata macros and locals.
  Just thinking about them will will mislead you.

- Everything is a variable in Mata.
  Mata variables have nothing to do with Stata variables.

- If it's not a variable, it's a function.
  There are no other alternatives.
    - Functions accept arguments (variables).
    - Functions return results (which you store in variables).

Mata concepts, continued

It's a variable or it's a function, ergo . . .

- There are no subroutines, programs, *etc.*, . . . that role is played by functions.
    - Functions accept arguments (variables).
    - Functions **optionally** return results (which you store in variables).
    - A function that returns nothing is said to "return void" or be a "void function". What everybody else calls a subroutine.
- There are no commands in Mata; that role is played by functions.

Mata concepts, continued

It's a variable or it's a function, ergo . . .

- There are no subroutines, programs, *etc.*, . . .
  that role is played by functions.
  - Functions accept arguments (variables).
  - Functions **optionally** return results (which you store in variables).
  - A function that returns nothing is said to "return void" or be a "void function". What everybody else calls a subroutine.

- There are no commands in Mata;
  that role is played by functions.

Mata concepts, continued

It's a variable or it's a function, ergo . . .

- There are no subroutines, programs, *etc.*, . . .
  that role is played by functions.
  - Functions accept arguments (variables).
  - Functions **optionally** return results (which you store in variables).
  - A function that returns nothing is said to "return void" or be a "void function". What everybody else calls a subroutine.

- There are no commands in Mata;
  that role is played by functions.

Mata concepts, continued

It's a variable or it's a function, ergo . . .

- There is no understanding of Stata by Mata.
  That role is played by functions.
  - Mata has functions that can access Stata.
  - Use the functions by filling in variables that you pass to them.
  - Get back results in variables. . .
    . . . which you then detail how to use.

Mata concepts, continued

It's a variable or it's a function, ergo . . .

- There is no understanding of Stata by Mata.
  That role is played by functions.
  - Mata has functions that can access Stata.
  - Use the functions by filling in variables that you pass to them.
  - Get back results in variables. . .
    . . . which you then detail how to use.

Mata concepts, continued

- Mata variables can contain
  - numbers (called real and complex)
  - characters (called strings)
  - memory addresses (called pointers)
  - collections of variables (called structures)
  - collections of functions and variables (called classes)

- Regardless of that, Mata variables can be scalars, vectors, row vectors, column vectors, or matrices.

Mata concepts, continued

- Mata variables can contain
  - numbers (called real and complex)
  - characters (called strings)
  - memory addresses (called pointers)
  - collections of variables (called structures)
  - collections of functions and variables (called classes)

- Regardless of that, Mata variables can be scalars, vectors, row vectors, column vectors, or matrices.

Mata concepts, continued

- Mata variables can contain numbers, . . .
- Mata variables can be scalars, . . .

This means you could have a matrix
  each element of which is a collection of functions and variables
    each variable of which is a vector of collections of variables
      each variable of which is a number or string

You will never want to do that.

Use of Mata concepts

- Mata functions that you write to be called by Stata typically return void:
  - Sometimes they return string scalars and, in the ado-file, you code `mata: st_local("macname", ...)` to store result in `macname`.
  - Usually, however, I write functions that return void.
    I make the first argument of the function a string scalar containing the name of a Stata macro, scalar, or matrix in which the result is to be returned, and code the `st_local()` in my function.
  - Sometimes I hard code Stata macro, scalar or matrix names in the Mata function, so the function has no arguments. That's considered bad style because it leads to hard-to-find bugs.

Use of Mata concepts

- Mata functions that you write to be called by Stata typically return void:
  - Sometimes they return string scalars and, in the ado-file, you code `mata: st_local("macname", ...)` to store result in `macname`.
  - Usually, however, I write functions that return void.
    I make the first argument of the function a string scalar containing the name of a Stata macro, scalar, or matrix in which the result is to be returned, and code the `st_local()` in my function.
  - Sometimes I hard code Stata macro, scalar or matrix names in the Mata function, so the function has no arguments. That's considered bad style because it leads to hard-to-find bugs.

Use of Mata concepts

- Mata functions that you write to be called by Stata typically return void:
    - Sometimes they return string scalars and, in the ado-file, you code `mata: st_local("macname", ...)` to store result in `macname`.
    - Usually, however, I write functions that return void.
      I make the first argument of the function a string scalar containing the name of a Stata macro, scalar, or matrix in which the result is to be returned, and code the `st_local()` in my function.
    - Sometimes I hard code Stata macro, scalar or matrix names in the Mata function, so the function has no arguments. That's considered bad style because it leads to hard-to-find bugs.

Use of Mata concepts

- Mata functions that you write to be called by Stata typically return void:
  - Sometimes they return string scalars and, in the ado-file, you code `mata: st_local("macname", ...)` to store result in `macname`.
  - Usually, however, I write functions that return void.
    I make the first argument of the function a string scalar containing the name of a Stata macro, scalar, or matrix in which the result is to be returned, and code the `st_local()` in my function.
  - Sometimes I hard code Stata macro, scalar or matrix names in the Mata function, so the function has no arguments. That's considered bad style because it leads to hard-to-find bugs.

Use of Mata concepts

- Ignore pointer variables unless you are programming something taught in a computer science course.
- Ignore structures and classes in most "simple" programming applications. Think of variables as containing numbers or strings.
- If you are programming a system, however, you should at least be using structures and perhaps classes.
  - If you do not, you are making your life more difficult than it needs to be.
  - Use classes only if you already know something about them or want to learn about them. Otherwise, structures will suffice.

Use of Mata concepts

- Ignore pointer variables unless you are programming something taught in a computer science course.
- Ignore structures and classes in most "simple" programming applications. Think of variables as containing numbers or strings.
- If you are programming a system, however, you should at least be using structures and perhaps classes.
  - If you do not, you are making your life more difficult than it needs to be.
  - Use classes only if you already know something about them or want to learn about them. Otherwise, structures will suffice.

Use of Mata concepts

- Ignore pointer variables unless you are programming something taught in a computer science course.
- Ignore structures and classes in most "simple" programming applications. Think of variables as containing numbers or strings.
- If you are programming a system, however, you should at least be using structures and perhaps classes.
    - If you do not, you are making your life more difficult than it needs to be.
    - Use classes only if you already know something about them or want to learn about them. Otherwise, structures will suffice.

Structures will suffice

- Structures are a general programming concept that are not unique to Mata.
- A structure would be better called a box.
    - You have 10 things laid out on a table which you hand to assistants to help you perform a task.
    - For some tasks, assistants need only a few of the things.
    - For other tasks, they need them all.
    - You would save yourself time if you put all ten things in a box and handed the box to your assistants.
    - In addition, you couldn't possibly forget to hand over something an assistant needs.

Structures, example

Here's an example of a structure (box) that you might find useful if you were programming linear regression:

```
struct regression_problem
{
    string scalar    lhs_var_name
    string rowvector rhs_var_names
    real scalar      first_obs_no, last_obs_no
}
```

If variable `rp` were a `struct regression_problem`

`rp.lhs_var_name` would be name of the dependent variable

`rp.rhs_var_names` would be names of the independent variables

Structures, example

```
struct regression_problem
{
    string scalar    lhs_var_name
    string rowvector rhs_var_names
    real scalar      first_obs_no, last_obs_no
}
...
struct regression_problem scalar   rp
```

rp contains four variables.

We can treat rp as if it were a single variable because it is a single variable.

If we had a routine called get_regression_results(), we might call it with a single variable, get_regression_results(rp).

Structures, example

rp is a struct regression_problem scalar.

We have written get_regression_results(rp).

We discover that we left something out of our structure!

So we add it:

```
struct regression_problem
{
    string scalar    lhs_var_name
    string rowvector rhs_var_names
    real scalar      first_obs_no, last_obs_no
    real scalar      include_intercept     // <- new
}
```

Structures, example

`rp` is a `struct regression_problem scalar`.

We have written `get_regression_results(rp)`.

We discovered that we left something out of our structure!

We added a new variable to our structure.

Now,

- We need to modify `get_regression_results()` to use new variable `rp.include_intercept`, but that's all we need to do!
- We do not have to back and change the number of arguments `get_regression_results()` and other functions receive, nor find all the calls to all the functions and modify them, *etc.*

Structures, example

Let's define another structure to hold regression results:

```
struct regression_results
{
    real vector       b
    real matrix       V
    real scalar       r_squared
    string scalar     lhs_var_name
    string rowvector rhs_var_names
}
```

Then we could code

```
rr = get_regression_results(rp)
```

I started off by mentioning how conceptually weak Mata is.

```
struct regression_problem {
    string scalar    lhs_var_name
    string rowvector rhs_var_names
    real scalar      first_obs_no, last_obs_no
    real scalar      include_intercept
}

struct regression_results {
    real vector      b
    real matrix      V
    real scalar      r_squared
    string scalar    lhs_var_name
    string rowvector rhs_var_names
}

rr = get_regression_results(rp)
```

Not weak at all.

Debugging code

- End of Substantive comments
- We begin debugging

We will debug the following Mata routine (which has no bugs):

Debugging a Mata routine

```
string scalar printable(string scalar s)
{
    real scalar        i
    string rowvector   tokens
    string scalar      toret

    tokens = tokens(s)
    if (cols(tokens)<2) return(strtrim(s))

    toret = tokens[1]
    for (i=2; i<cols(tokens); i++) {
        toret = toret + ", " + tokens[i]
    }
    return(toret + " and " + tokens[cols(tokens)])
}
```

Debugging a Mata routine

```
string scalar printable(string scalar s)
{
    ...(declarations omitted)...

"printable() begins; p0, s is"
s
    tokens = tokens(s)
    if (cols(tokens)<2) return(strtrim(s))

"p2, tokens are"; tokens
    toret = tokens[1]
    for (i=2; i<cols(tokens); i++) {
        toret = toret + ", " + tokens[i]
    }
"p3"
    return(toret + " and " + tokens[cols(tokens)])
}
```

Debugging a Mata routine
What I did

- I introduced messages that we will see when we execute the subroutine.
- I exploit the fact that in Mata the result of any expression which is not stored is displayed.
- The messages are on the left margin.
  They can easily be spotted and so removed later.
- I would add more messages—even inside loops—if necessary.

Debugging a Stata routine

I can use the same approach in Stata.

- Use Stata's `display` command.
- Problem: `quietly` will prevent output.
  Solution: Use Stata's `display as error`.
- Problem: `capture` will prevent all output.
  Solution: `set output proc` inside the capture block.
  (Put it on left margin so you remember to remove it later.)

(`set output proc` is not documented in the manuals)

Locating bugs

Now you know how to debug a routine.

Let's find the routine that needs debugging:

Problem:

- I have a thousand lines of code.
- It runs, and somewhere, it produces an error message.
- Find the offending line.

You may not use Stata's `set trace on` or
Mata's `mata set matalnum on`.

Locating bugs, related problem

Here's a related problem

- I'm thinking of a number between 1 and 1,000.
- You guess and I'll tell you if you're right.
  If you're wrong, I'll say lower or higher.

Everyone in this room knows a strategy that is guaranteed to get produce the number in 10 or fewer guesses; fewer 25% of the time.

I'll show you how to get Stata and Mata to say "lower" or "higher".

Locating bugs

My code:

```
"p1"
        ...(500 lines of code)...
"p1.5"
        ...(500 lines of code)...
"p2"
```

I will see

- `p1` and error message
  Ergo, the bug lies between `p1` and `p1.5`.
- `p1`, `p1.5`, and error message
  Ergo, the bug lies between `p1.5` and `p2`.

Repeat.

We're done, not

Now I would summarize what I've said, except . . .

Remember when we discussed Systems?

Well, we hardly discussed the subject because I was worried about time.

We'll discuss it now.

Systems

- A system is a set of commands that work together to solve one problem.
- Stata's `mi` command is an example of a system.
- A system has one or more of the following characteristics
  - multiple entry points
  - states
  - common subroutines

(You've seen this slide before)

Multiple entry points means that there are multiple commands users type as they work their way through the problem.

- The commands might be **related**
  - Rather than one complicated command with lots of options, features are presented as different commands.
  - Users use only one of the commands, which depending on problem. Users decide when it is appropriate to use which. Such systems are called **internally related**.
  - Users work their way through a problem using multiple commands. Such systems are called **externally related**.
- The commands might instead be **ordered**
  - Users use all the commands.
  - E.g., first they set, then they impute, then they estimate.

Or the system might be **single entry point** but the problem is so big you want to organize the code as a system.

The **state** of a system refers to information that needs to be available between subcommands.

- In single-entry-point systems, the state is recorded as just like any other variable. It comes into existence when the command starts, and is destroyed when it ends.
- In multiple-entry-point/related systems, there often is no state; the user is responsible for deciding when to use which subcommand.
- If there is an `xyz set` command, then there is a state.
- In multiple-entry-point/ordered systems, there is a state (even if there is no `xyz set` command).

Where to store states

Where you store states depends . . .

- If they are a property of the dataset, e.g., `mi`, store in Stata's `_dta[]` characteristics.
- If they are a property of the session, e.g., `ml`, store in
  - Stata's global macros and global scalars
  - or in Mata's global structures
  - and typically not both.

If states are stored in Stata's `_dta[]` characteristics, . . .

- To set them, use
  - Stata's `char _dta[`*name*`]` . . . command
  - Mata's `st_global("_dta[`*name*`]", "...")` function.
- To use them, use
  - Stata's `` `_dta[``*name*`]'` macro expansion
  - Mata's `st_global("_dta[`*name*`]")` function.

If states are stored in Stata's global macros and scalars, . . .

- To set them, use
  - Stata's `global` *name* . . . and `scalar` *name* = . . . commands
  - Mata's `st_global("`*name*`", "...")` and `st_numscalar("`*name*`",` *value*`)` functions.
- To access them, use
  - Stata's \``*name*`' macro expansion and `scalar(`*name*`)` pseudofunction
  - Mata's `st_global("`*name*`")` and `st_numscalar("`*name*`")` functions.

If states are stored in Mata's global structures, use Mata built-in functions

- `crexternal("`*name*`")`
- `findexternal("`*name*`")`
- `rmexternal("`*name*`")`

to create, find, and remove the global.

I will explain.

To create a global struct xyz_state scalar under the name _xyz_state, use this routine which I give you

```
void create_xyz_state()
{
    pointer(struct xyz_state scalar) scalar p

    if ((p=crexternal("_xyz_state"))==NULL) {
        _error("_xyz_state already exists")
        /*NOTREACHED*/
    }
    *p = xyz_state()
}
```

Call the function—code `create_xyz_state()`.

The global structure now exists. It will have missing values stored in it, so next find the global and fill it in.

To find the existing structure, use

```
pointer(struct xyz_state scalar) scalar find_xyz_state()
{
    pointer(struct xyz_state scalar) scalar p

    if ((p=findexternal("_xyz_state"))==NULL) {
        _error("_xyz_state not found")
        /*NOTREACHED*/
    }
    return(p)
}
```

call by coding

```
    pointer(struct xyz_state scalar) scalar p
    ...
    p = find_xyz_state()
```

We just said that to find the existing structure, use . . . and call by coding

```
pointer(struct xyz_state scalar) scalar p
...
p = find_xyz_state()
```

Now I add

- Because we are dealing with a global in a general way, pointers were unavoidable. Sorry.
- The only subsequent difference is that you will code `p->element1`, `p->element2`, . . . , where you would have coded *name*`.element1`, *name*`.element2`, . . .

Thus, to create a global structure and initialize it, code,

```
    pointer(struct xyz_state scalar) scalar p
    ...
    create_xyz_state()
    p = find_xyz_state()
    p->element1 = ...
    p->element2 = ...
    ...
```

When you call a subroutine, code

```
     ... mysubroutine(p, ...)  ...
```

and write `mysubroutine()` to receive a pointer(struct xyz_state scalar) scalar.

Call `find_xyz_state()` once at every entry point. After that, pass `p` to the subroutines you write.

To permanently remove (delete) the global `_xyz_state`, code

```
rmexternal("_xyz_state")
```

and we can even do that from Stata by coding

```
mata: rmexternal("_xyz_state")
```

Remember, a single structure can contain a lot of variables.
It can even contain other structures!
You can store lots of information under one name.

Aside on using structures

I hate typing

```
    pointer(struct xyz_state scalar) scalar
```

over and over.

So I type

```
    local StataPtr pointer(struct xyz_state scalar) scalar
```

once in Stata and then type `StatePtr' after that.

Watch . . .

```
version 12
local StataPtr pointer(struct xyz_state scalar) scalar
mata:
void create_xyz_state()
{
    `StatePtr' p

    if ((p=crexternal("_xyz_state"))==NULL) {
        _error("_xyz_state already exists")
        /*NOTREACHED*/
    }
    *p = xyz_state()
}

`StatePtr' find_xyz_state()
{
    ...
}
end
```

Aside on typing

You can use macros to define types based on meaning rather than types based on storage type. This makes your code more readable.

Structures are one way you define concepts. Defining types based on meaning is another way.

Remember our regression-problem structure?

```
struct regression_problem {
    string scalar    lhs_var_name
    string rowvector rhs_var_names
    real scalar      first_obs_no, last_obs_no
    real scalar      include_intercept
}
```

Better; variable types based on meaning

```
local RegrProb struct regression
local Varname  string scalar
local Varnames string rowvector
local ObsNo    real scalar
local Boolean  real scalar

mata:
`RegrProb' {
    `Varname'   lhs_var_name
    `Varnames'  rhs_var_names
    `Obsno'     first_obs_no, last_obs_no
    `Boolean'   include_intercept
}
end
```

Back to the topic

We took a long, substantive aside on

- States
- All the different ways they could be stored
- Structures
- Pointers
- A shorthand involving Stata macros to save typing
- Using the shorthand to improve readability

We were discussing systems, and in particular, the system `xyz` . . .

Systems, the goal

The goal is to write a command with syntax

```
xyz subcmd1 ...
xyz subcmd2 ...
...
```

- Could be written as one ado-file, `xyz.ado`.
- If the system is big,
  - might take too long to load
  - would be more difficult to write
  - would be more difficult to maintain
- Nonetheless, do not disregard the single ado-file approach for "small" systems.

Big systems, desired organization

The final layout of the system will be

xyz.ado	the `xyz` command switcher
xyz_cmd_subcmd1.ado	`xyz subcmd1` processor
xyz_cmd_subcmd2.ado	`xyz subcmd2` processor
...	
xyz_whatever1.ado	ado-file common subroutine
xyz_whatever1.ado	
...	
lxyz.mlib	common Mata subroutines, precompiled

Big systems can also have private subroutines, both ado and Mata.

Private subroutines are placed in the individual ado-files.

Big systems, initial organization

To begin, the contents of *.ado and lxyz.mlib will appear in xyz.do.

code.do	A single-line do-file containing “`do xyz`”
xyz.do	`xyz` code, Stata and Mata
xyzcheck.do	do-file to check compilation of our code
xyztest1.do	do-file to test something about our system
xyztest2.do	do-file to test something else about our system
. . .	
xyztest.do	do file to run all the tests

Big systems, initial organization

xyz.do:

```
program xyz
        version 12
        ...
end

(Stata subroutines go here)

version 12
set matastrict on
mata:

(Mata subroutines go here)

end
set matastrict off
```

Big systems, initial organization, continued

xyzcheck.do:

```
clear all
capture log close
log using xyzcheck.log, replace
do code
local rc = _rc
log close
exit `rc'
```

- `do xyzcheck` allows you to compile and thus see compile-time errors.
- If no errors, look at xyz.log and search for “note:”. Resolve them all.

Big systems, initial organization, continued

xyztest1.do:

```
clear all
run code.do          // I use run so I don't see the output

sysuse auto
...
```

- `xyztest2.do`, `xyztest3,do`, ..., all have the same structure as `xyztest1.do`.
- They are not really named `xyztest1.do`, `xyztest2.do`, ...
- I can run (`do`) any of the tests in isolation.
- I can run all the tests ...

Big systems, initial organization, continued

xyztest.do:

```
do xyztest1
do xyztest2
...
```

- I can run (do) any of the tests in isolation.
- I can run all the tests by typing
  `do xyztest`

Why I start like this

The organization is admittedly idiosyncratic, but when I start

- I don't yet have fixed ideas on the exact naming of the global state variables, or even what they all are.
- I fix ideas as I write, and I change my mind.
  I can easily make global changes.
- I write subroutines which I think will be private, but turn out to be useful globally. I can move them and easily change their names.

Why I start like this

The organization is admittedly idiosyncratic, but when I start

- I don't yet have fixed ideas on the exact naming of the global state variables, or even what they all are.
- I fix ideas as I write, and I change my mind.
  I can easily make global changes.
- I write subroutines which I think will be private, but turn out to be useful globally. I can move them and easily change their names.

Why I start like this

The organization is admittedly idiosyncratic, but when I start

- I don't yet have fixed ideas on the exact naming of the global state variables, or even what they all are.
- I fix ideas as I write, and I change my mind.
  I can easily make global changes.
- I write subroutines which I think will be private, but turn out to be useful globally. I can move them and easily change their names.

Why I start like this

The organization is admittedly idiosyncratic, but when I start

- I don't yet have fixed ideas on the exact naming of the global state variables, or even what they all are.
- I fix ideas as I write, and I change my mind.
  I can easily make global changes.
- I write subroutines which I think will be private, but turn out to be useful globally. I can move them and easily change their names.

As I work . . .

As I work, `xyz.do` evaporates

- I promote routines out of file xyz.do and into their own files.
- In the case of Stata code, they are promoted to ado-files.
- In the case of Mata code, they are promoted to *name*`.mata` files.

I add "`do` *name*`.ado`" or "`do` *name*`.mata`" to file `code.do`.

Eventually ...

Eventually, `xyz.do` is empty and

`code.do` looks like this

code.do:

```
do xyz.ado
do xyz_cmd_subcmd1.ado
...
do sub1.mata
do sub2.mata
...
```

Ado-files may have their own, private Mata subroutines, but the public Mata subroutines are in `*.mata`.

How to build a Mata library

Create file mklxyz.do from code.do:

mklxyz.do:

```
clear all
capture erase lxyz.mata

set matastrict on
do sub1.mata
do sub2.mata
set matastrict off
...

mata:
mata mlib create lxyz
mata mlib add    lxyz *(), complete
mata mlib index
end
```

We are nearly done

We have files

*.ado	move to PERSONAL
lxyz.mlib	move to PERSONAL
mklxyz.do	move to where you save Mata code
*.mata	move to where you save Mata code
code.do	make empty, move to where you store test scripts
xyztest.do	move to where you store test scripts
xyztest1.do	move to where you store test scripts
...	...

We are done

I ran through that pretty fast.

If you are the type of person who is writing big systems, however, I think you get the point. You need to get organized and develop guidelines for yourself.

Since there's not a chance I will get to this point when presenting this talk in person, I will not bother with a conclusion.

To those of you who stuck with me all the way to the end, I hope this was of help.