Title

stata.com

***mi impute usermethod* —** User-defined imputation methods

Description

This entry describes how to add your own imputation methods to the `mi impute` command.

Syntax

`mi impute` *usermethod userspec* [`,` *options*]

usermethod is the name of the method you would like to add to the `mi impute` command. When naming an `mi impute` method, you should follow the same convention as for naming the programs you add to Stata—do not pick "nice" names that may later be used by Stata's official methods.

userspec is a specification of an imputation model as supported by the user-defined method *usermethod*. It must include imputation variables *ivars*. It may also include independent variables *indepvars*, weights, and an *if* qualifier if those things are also supported by *usermethod*. The actual syntax of *userspec* will be specific to *usermethod*. We encourage users who are adding their own methods to `mi impute` to follow `mi impute`'s syntax or Stata's general syntax when designing their methods.

options	Description
impute_options	any option of `mi impute` except `noupdate` and `by()`
`orderasis`	impute variables in the specified order
user_options	additional options supported by *usermethod*

You must `mi set` your data before using `mi impute` *usermethod*; see [MI] **mi set**.

You must `mi register` imputation variables as `imputed` before using `mi impute` *usermethod*; see [MI] **mi set**.

Options

impute_options include `add()`, `replace`, `rseed()`, `double`, `dots`, `noisily`, `nolegend`, `force`; see [MI] **mi impute** for details.

`orderasis` requests that the variables be imputed in the specified order. By default, variables are imputed in order from the most observed to the least observed.

user_options specify any additional options supported by *usermethod*.

Remarks and examples

stata.com

Adding your own methods to `mi impute` is rather straightforward. Suppose that you want to add a method called `mymethod` to `mi impute`.

1. Write an ado-file that contains a program called `mi_impute_cmd_mymethod_parse` to parse your imputation model.

2. Write an ado-file that contains a program called `mi_impute_cmd_mymethod`, which will perform a single imputation on all of your imputation variables.
3. Place the ado-files where Stata can find them.

You are done. You can now use `mymethod` within `mi impute` like any other official `mi impute` method. `mi impute` will take care of performing your imputation step multiple times and will do it properly for any `mi` style.

Remarks are presented under the following headings:

Toy example: Naive regression imputation

As a quick example, let's write a method called `naivereg` to perform a naive regression imputation, also known as stochastic regression imputation, of a single variable *ivar* based on independent variables *xvars*.

First, let's describe our imputation procedure.

1. Regress *ivar* on *xvars* using the observed data.
2. Obtain the linear predictor, *xb*.
3. Replace missing values in *ivar* with *xb* plus a random error generated from $N(0, sigma\_mle)$, where *sigma_mle* is the estimated error standard deviation.

Let's now write our imputation program. We create an ado-file called `mi_impute_cmd_naivereg.ado` that contains the following Stata program:

```
// imputer
program mi_impute_cmd_naivereg
        version 14.1
        /* step 1: run regression on observed data */
        quietly regress $MI_IMPUTE_user_ivar $MI_IMPUTE_user_xvars
        /* step 2: compute linear prediction */
        tempvar xb
        quietly predict double `xb', xb
        /* step 3: replace missing values */
        quietly replace $MI_IMPUTE_user_ivar = `xb' + rnormal(0,e(rmse)) ///
              if $MI_IMPUTE_user_miss==1
end
```

Global macros `MI_IMPUTE_user_ivar` and `MI_IMPUTE_user_xvars` contain the names of the imputation and independent variables, respectively, and `MI_IMPUTE_user_miss` contains the indicator for missing values in the imputation variable. `ereturn` scalar `e(rmse)` contains the estimated error standard deviation from the `regress` command used in step 1. The `rnormal()` function is used to generate values from a normal distribution.

In addition to the imputer, we also need to write a parser program that passes the imputation model specification to `mi impute`. We create an ado-file called `mi_impute_cmd_naivereg_parse.ado` that contains the following simple program:

```
// parser
program mi_impute_cmd_naivereg_parse
        version 14.1
        syntax anything [, * ]
        gettoken ivar xvars : anything
        u_mi_impute_user_setup, ivars(`ivar') xvars(`xvars') `options'
end
```

The parser retrieves the information about imputation and independent variables to be supplied by the user and passes it to `mi impute` via the utility program `u_mi_impute_user_setup`, which will be discussed later.

We can now use `naivereg` with `mi impute`. For demonstration purposes only, let's use our new method to impute missing values of variable `rep78` from the `auto` dataset. We will use complete variables `mpg` and `weight` as predictors.

We load the data, declare the `mi` style, and register `rep78` as an imputation variable.

```
. sysuse auto, clear
(1978 Automobile Data)

. mi set wide

. mi register imputed rep78
```

We now use our new method `naivereg` within `mi impute`.

```
. mi impute naivereg rep78 mpg weight, add(2)

Multiple imputation                           Imputations =          2
User method naivereg                                added =          2
Imputed: m=1 through m=2                          updated =          0
```

	Observations per *m*			
Variable	Complete	Incomplete	Imputed	Total
rep78	69	5	5	74

```
(complete + incomplete = total; imputed is the minimum across m
 of the number of filled-in observations.)
```

We created two imputations using `mi impute`'s option `add()` and obtained the standard output from `mi impute`. We imputed all five missing values of variable `rep78` using the new `naivereg` method.

This is just a simple example. Your imputation model can be as complicated as you would like. See *Examples* for more complicated imputation models.

Steps for adding a new method to mi impute

Suppose you want to add your own method, *usermethod*, to the `mi impute` command. Here is an outline of the steps to follow:

1. Create a *parser*, a program called `mi_impute_cmd_`*usermethod*`_parse` and defined by the ado-file `mi_impute_cmd_`*usermethod*`_parse.ado` that parses the imputation model and checks the syntax of user-specific options, *user_options*. See *Writing an imputation parser*.

2. Optionally, create an *initializer*, a program called `mi_impute_cmd_`*usermethod*`_init` and defined by the ado-file `mi_impute_cmd_`*usermethod*`_init.ado` that performs certain tasks to be executed once on the observed data. For example, during monotone imputation, the estimation of model parameters can be done just once using the observed data. See *Writing an initializer*.
3. Create an *imputer*, a program called `mi_impute_cmd_`*usermethod* and defined by the ado-file `mi_impute_cmd_`*usermethod*`.ado` that performs one round of imputation for all imputation variables. See *Writing an imputer*.
4. Optionally, create a program for storing additional `r()` results called `mi_impute_cmd_`*usermethod*`_return` and defined by the ado-file `mi_impute_cmd_`*usermethod*`_return.ado`. See *Storing additional results*.
5. Optionally, create a *cleanup* program (or garbage collector), a program called `mi_impute_cmd_`*usermethod*`_cleanup` and defined by the ado-file `mi_impute_cmd_`*usermethod*`_cleanup.ado` that removes all the intermediate variables, global macros, etc., you created during parsing, initialization, or imputation. See *Writing a cleanup program*.
6. Place all of your programs where Stata can find them.

You can now use your *usermethod* with `mi impute`,

```
. mi impute usermethod ...
```

and access any of `mi impute`'s options (except `by()` and `noupdate`).

Writing an imputation parser

A parser is a program that parses the imputation model specification *userspec*, passes the necessary information to `mi impute`, and checks user-specified options. It must be defined within an ado file with the name `mi_impute_cmd_`*usermethod*`_parse.ado`. You can use any of Stata's parsing utilities such as the `syntax` command to write your parser. It may be more convenient for users if you follow the syntax of `mi impute` when designing your imputation methods.

At a minimum, your parser must supply information about the imputation variables to `mi impute`. This is done via the `ivars()` option of the utility command `u_mi_impute_user_setup`:

```
u_mi_impute_user_setup, ivars(varlist) ...
```

You may supply other information such as independent variables (complete predictors) in option `xvars()`, weights, an *if* qualifier, and so on.

A simple univariate parser may look as follows.

```
program mi_impute_cmd_usermethod_parse
        version ...
        syntax anything [if] [fw iw] [, * ]
        gettoken ivar xvars : anything
        u_mi_impute_user_setup `if' [`weight'`exp'],   ///
                ivars(`ivar') xvars(`xvars') `options'
end
```

The above parser corresponds to the following *userspec*,

ivar [*indepvars*] [*if*] [*weight*]

where only `fweights` and `iweights` are allowed.

A simple multivariate parser may look as follows.

```
program mi_impute_cmd_usermethod_parse
        version ...
        syntax anything(equalok) [if] [fw iw] [, * ]
        gettoken ivars xvars : anything, parse("=")
        gettoken eq xvars : xvars, parse("=")
        u_mi_impute_user_setup `if' [`weight'`exp'],   ///
                ivars(`ivars') xvars(`xvars') `options'
end
```

This parser corresponds to the following *userspec*,

ivars [= *indepvars*] [*if*] [*weight*]

where only `fweight`s and `iweight`s are allowed.

You may also supply complete predictors, *if* qualifiers, and weights specific to each imputation variable or control the order in which variables are imputed. Here is the full syntax of the utility program.

`u_mi_impute_user_setup` [*if*] [*weight*] [, *setup_options*]

setup_options	Description
Main	
* `ivars(`*varlist*`)`	specify imputation variables
`xvars(`*varlist*`)`	specify complete predictors for all imputation variables
`xvars#(`*varlist*`)`	specify complete predictors for the #th imputation variable; overrides `xvars()`
`if#(`*if*`)`	specify an *if* qualifier for the #th imputation variable (in addition to the global *if*)
`weight#(`*weight*`)`	specify weights for the #th imputation variable; overrides global weights
`orderasis`	impute variables in the specified order
[`no`]`fillmissing`	do not replace current imputed data with missing values
`title1(string)`	specify the main title
`title2(string)`	specify the secondary title

* `ivars(`*varlist*`)` is required.

`ivars(`*varlist*`)` specifies the names of the imputation variables. This option is required.

`xvars(`*varlist*`)` specifies the names of the independent variables (complete predictors) for all imputation variables. You may use `xvars#()` to override the complete predictors for the #th imputation variable.

`xvars#(`*varlist*`)` specifies the names of the independent variables for the #th imputation variable. This option overrides the `xvars()` option for that variable. If `xvars#()` is not specified, then `xvars()` (if specified) is assumed for that variable.

`if#(`*if*`)` specifies an *if* qualifier for the #th imputation variable. This option is used in conjunction with the global *if* qualifier specified with the program to define an imputation sample for that variable.

`weight#(`*weight*`)` specifies weights for the #th imputation variable. This option overrides the global weight specified with the program. If `weight#()` is not specified, then the global weight (if specified) is used for that variable.

`orderasis` requests that the variables be imputed in the specified order. By default, variables are imputed in order from the most observed to the least observed.

`fillmissing` or `nofillmissing` requests that the imputed data be filled in or not filled in with missing values prior to the imputation. The default is `fillmissing`. This option is rarely used.

`title1(`*string*`)` specifies the main title. The default is "Multiple imputation".

`title2(`*string*`)` specifies the secondary title. The default is "User method: *usermethod*".

`u_mi_impute_user_setup` sets certain global macros used by `mi impute`; see *Global macros* for details.

Writing an initializer

An initializer (in the context of `mi impute`) is a program that is executed once on the observed data, $m = 0$, before imputation. This program is optional. If you choose to write an initializer, it must be defined within an ado-file with the name `mi_impute_cmd_`*usermethod*`_init.ado`. This program is useful if you have an estimation task that needs to be performed only once on the observed data.

For example, a univariate regression imputation requires that the regression be performed on the observed data prior to imputation. A simple initializer for such imputation may look as follows.

```
program mi_impute_cmd_usermethod_init
        version ...
        quietly regress $MI_IMPUTE_user_ivar $MI_IMPUTE_user_xvars ///
                if $MI_IMPUTE_user_touse
end
```

Writing an imputer

An imputer is a program that imputes missing values of all specified imputation variables once. This program is required and must be defined within an ado-file with the name `mi_impute_cmd_`*usermethod*`.ado`. `mi impute` will execute this program multiple times to produce multiply imputed datasets.

A simple univariate imputer may look as follows.

```
program mi_impute_cmd_usermethod
        version ...
        quietly replace $MI_IMPUTE_user_ivar = ...  ///
                if $MI_IMPUTE_user_miss
end
```

Storing additional results

To store results in addition to those provided by `mi impute` (see *Stored results*), you need to create a r-class program called `mi_impute_cmd_`*usermethod*`_return`. Here is an example.

```
program mi_impute_cmd_ usermethod_return, rclass
        version ...
        syntax [, myopt(real 0) * ]
        return scalar myopt = 'myopt'
end
```

Writing a cleanup program

A "cleanup" program or garbage collector is a program that is called at the end of the imputation process to remove any intermediate results you created in your parser, initializer, or imputer that will not be removed automatically upon program completion. For example, such results may include new variables (except temporary variables), global macros, global names for estimation results, and so on. This program is optional but highly recommended when you have intermediate results that need to be cleared manually.

Examples

Naive regression imputation

Recall our introductory example from *Toy example: Naive regression imputation* of a naive (or stochastic) regression imputation.

Initializer. We can make our imputer more computationally efficient by separating the estimation and imputation tasks. Currently, regression is performed in each imputation. We can move this step into the initializer.

```
// initializer (naivereg)
program mi_impute_cmd_naivereg_init
        version 14.1
        /* step 1: run regression on observed data */
        quietly regress $MI_IMPUTE_user_ivar $MI_IMPUTE_user_xvars
end
```

Here is the updated imputer.

```
// imputer (naivereg)
program mi_impute_cmd_naivereg
        version 14.1
        /* step 2: compute linear prediction */
        tempvar xb
        quietly predict double `xb', xb
        /* step 3: replace missing values */
        quietly replace $MI_IMPUTE_user_ivar = `xb'+rnormal(0,e(rmse)) ///
            if $MI_IMPUTE_user_miss==1
end
```

If we now run `mi impute naivereg`, the `regress` command will be run only once, on the observed data $m = 0$.

If qualifier and weights. We can also extend our method to allow the specification of an *if* qualifier and, say, frequency weights.

```
// parser (naivereg, if and weights)
program mi_impute_cmd_naivereg_parse
        version 14.1
        syntax anything [if] [fw] [, * ]
        gettoken ivar xvars : anything
        u_mi_impute_user_setup `if' [`weight'`exp'] ,  ///
                        ivars(`ivar') xvars(`xvars') `options'
end
```

We updated the `syntax` statement to allow *if* and frequency weights and passed that information to the utility program `u_mi_impute_user_setup`.

```
// initializer (naivereg, if and weights)
program mi_impute_cmd_naivereg_init
        version 14.1
        step 1: run regression on observed data */
        quietly regress $MI_IMPUTE_user_ivar $MI_IMPUTE_user_xvars  ///
                    $MI_IMPUTE_user_weight if $MI_IMPUTE_user_touse
end
```

We included the global macros containing the information about weights and the imputation sample in our `regress` command.

```
// imputer (naivereg, if and weights)
program mi_impute_cmd_naivereg
        version 14.1
        /* step 2: compute linear prediction */
        tempvar xb
        quietly predict double `xb' if $MI_IMPUTE_user_touse, xb
        /* step 3: replace missing values */
        quietly replace $MI_IMPUTE_user_ivar = `xb'+rnormal(0,e(rmse)) ///
                 if $MI_IMPUTE_user_miss==1
end
```

We restricted the computation of the linear predictor for the sample determined by the specified *if* qualifier. A more efficient approach would be to also restrict the computation of the linear predictor for missing values only. This can be done by replacing `if $MI_IMPUTE_user_touse` in the `predict` line above with `if $MI_IMPUTE_user_miss`.

For example, we can now impute `rep78` separately for foreign and domestic cars and incorporate frequency weights. For the purpose of demonstration, we will use `turn` as a frequency weight.

```
. sysuse auto, clear
(1978 Automobile Data)

. mi set wide

. mi register imputed rep78

. mi impute naivereg rep78 mpg weight [fweight=turn] if foreign==1, add(2)
Multiple imputation                             Imputations =        2
User method naivereg                                  added =        2
Imputed: m=1 through m=2                            updated =        0

------------------------------------------------------------------------------
                             |                Observations per m
                             |----------------------------------------------
                    Variable |   Complete   Incomplete    Imputed      Total
-----------------------------+------------------------------------------------
                       rep78 |        741           38          38        779
------------------------------------------------------------------------------
(complete + incomplete = total; imputed is the minimum across m
 of the number of filled-in observations.)
```

```
. mi impute naivereg rep78 mpg weight [fweight=turn] if foreign==0, replace
Multiple imputation                          Imputations =        2
User method naivereg                               added =        0
Imputed: m=1 through m=2                         updated =        2
```

	Observations per *m*			
Variable	Complete	Incomplete	Imputed	Total
rep78	2005	150	150	2155

```
(complete + incomplete = total; imputed is the minimum across m
 of the number of filled-in observations.)
```

Univariate regression imputation

In *Naive regression imputation*, we added a new method, `naivereg`. The reason we called this imputation method naive is that it did not incorporate the uncertainty about the estimates of coefficients and error standard deviation when computing the linear predictor and simulating the imputed values.

Let's add a new method, `myregress`, that improves the `naivereg` method. The parser and the initializer stay the same (except they need to be renamed to `mi_impute_cmd_myregress_parse` and `mi_impute_cmd_myregress_init`, respectively). The imputer, however, changes substantially. Before we move on to the programming task, let's revisit the imputation procedure described in *Toy example: Naive regression imputation*.

The linear predictor from step 2 is computed using the maximum likelihood estimates of regression coefficients, *beta_mle*, from step 1. Also, the random normal variates are generated using the maximum likelihood estimate of the error standard deviation, *sigma_mle*. The proper regression imputation simulates a new set of parameters, *beta* and *sigma*, from their respective posterior distributions and uses them to compute results in steps 2 and 3. Let's update our imputation procedure.

1. Regress *ivar* on *xvars* using the observed data.
2. Simulate new regression coefficients *beta* and error standard deviation *sigma* from their respective posterior distributions, which are based on their maximum likelihood estimates, *beta_mle* and *sigma_mle*.
3. Obtain the linear predictor, *xb*, using the new regression coefficients *beta*.
4. Replace missing values in *ivar* with *xb* plus a random error generated from $N(0, sigma)$.

Let's now update our imputer.

```
// imputer (myregress)
program mi_impute_cmd_myregress, eclass
        version 14.1
        /* step 2: simulate new beta and sigma */
        tempname sigma beta sigma_mle beta_mle vce_chol rnorm
        matrix `beta_mle'  = e(b)
        scalar `sigma_mle' = e(rmse)
        matrix `vce_chol'  = cholesky(e(V))/`sigma_mle'
        local ncols        = colsof(`beta_mle')
        /* draw beta and sigma from the posterior distribution */
        scalar `sigma' = `sigma_mle'*sqrt(e(df_r)/rchi2(e(df_r)))
        mata: st_matrix("`rnorm'", rnormal(`ncols',1,0,1))
        matrix `beta' = `beta_mle'+(`sigma'*(`vce_chol'*`rnorm'))'
        /* step 3: compute linear prediction */
        ereturn repost b = `beta' // repost new beta
        tempvar xb
```

```
                quietly predict double `xb' if $MI_IMPUTE_user_miss, xb
                ereturn repost b = `beta_mle' // repost back beta_mle
                /* step 4: replace missing values */
                quietly replace $MI_IMPUTE_user_ivar = `xb'+`sigma'*rnormal() ///
                          if $MI_IMPUTE_user_miss==1
        end
```

Our new imputer is much more involved. In step 2, we generate a new (temporary) matrix of coefficients, `` `beta' ``, and a temporary scalar containing new error standard deviation. The new parameters are simulated from their posterior distribution. In step 3, we repost new coefficients to `e()` results to obtain the proper linear predictor, and we repost the old coefficients back to be used in the next imputation. In step 4, we use a new `` `sigma' `` to generate random errors.

We can check that we obtain the same imputed values as Stata's official `mi impute regress` command, provided that we use the same random-number seed. For example,

```
. sysuse auto, clear
(1978 Automobile Data)

. mi set wide

. mi register imputed rep78

. mi impute myregress rep78 mpg weight, add(1) rseed(234)

Multiple imputation                          Imputations =          1
User method myregress                              added =          1
Imputed: m=1                                     updated =          0

                     Observations per m
 ───────────────────┬──────────────────────────────────────────────
          Variable  │  Complete   Incomplete   Imputed │     Total
 ───────────────────┼──────────────────────────────────────────────
             rep78  │        69            5         5 │        74
 ───────────────────┴──────────────────────────────────────────────

(complete + incomplete = total; imputed is the minimum across m
 of the number of filled-in observations.)

. mi impute regress rep78 mpg weight, add(1) rseed(234)

Univariate imputation                        Imputations =          2
Linear regression                                  added =          1
Imputed: m=2                                     updated =          0

                     Observations per m
 ───────────────────┬──────────────────────────────────────────────
          Variable  │  Complete   Incomplete   Imputed │     Total
 ───────────────────┼──────────────────────────────────────────────
             rep78  │        69            5         5 │        74
 ───────────────────┴──────────────────────────────────────────────

(complete + incomplete = total; imputed is the minimum across m
 of the number of filled-in observations.)

. mi xeq 1 2: summarize rep78

m=1 data:
-> summarize rep78

    Variable │       Obs        Mean    Std. Dev.       Min        Max
 ────────────┼──────────────────────────────────────────────────────────
       rep78 │        74     3.37852    .9965215          1          5

m=2 data:
-> summarize rep78

    Variable │       Obs        Mean    Std. Dev.       Min        Max
 ────────────┼──────────────────────────────────────────────────────────
       rep78 │        74     3.37852    .9965215          1          5
```

Multivariate monotone imputation

Our previous examples demonstrated univariate imputation—imputation of a single variable. Here we demonstrate an example of multivariate imputation for variables with a monotone missing-value pattern. For simplicity, we will consider imputation of two variables using a new method, `mymonreg`.

We start with a parser.

```
// imputer (mymonreg)
program mi_impute_cmd_mymonreg_parse
        version 14.1
        syntax anything(equalok) [if] [, * ]
        gettoken ivars xvars : anything, parse("=")
        gettoken eq xvars : xvars, parse("=")
        u_mi_impute_user_setup ‘if’, ivars(‘ivars’) xvars(‘xvars’) ‘options’
end
```

We separate multiple-imputation variables from the complete predictors with the equality (=) sign. The same set of complete predictors will be used to impute all imputation variables.

```
// initializer (mymonreg)
program mi_impute_cmd_mymonreg_init
        version 14.1
        /* run regression on observed data for each imputation variable and
            store estimation results */
        quietly regress $MI_IMPUTE_user_ivar1               ///
              $MI_IMPUTE_user_xvars1 if $MI_IMPUTE_user_touse1
        quietly estimates store myreg1
        quietly regress $MI_IMPUTE_user_ivar2               ///
              $MI_IMPUTE_user_ivar1 $MI_IMPUTE_user_xvars2 ///
              if $MI_IMPUTE_user_touse2
        quietly estimates store myreg2
end
```

With multiple imputation variables, `mi impute` automatically orders them from the least missing to the most missing. In our example, `MI_IMPUTE_user_ivar1` will contain the name of the imputation variable with the least number of missing values, and `MI_IMPUTE_user_ivar2` with the most number. You can use the `orderasis` option to prevent `mi impute` from ordering the variables. Notice that during monotone imputation, the previously imputed variables are used as predictors of the subsequent imputation variables in addition to the complete predictors. So we used `MI_IMPUTE_user_ivar1` as an additional predictor of `MI_IMPUTE_user_ivar2`.

To avoid refitting models on each imputed dataset, we store estimation results as `myreg1` and `myreg2`. It is our responsibility to drop these estimation results from memory at the end of the imputation.

During imputation, we will need to apply the steps of the regression imputation described in *Univariate regression imputation* to each imputation variable. To simplify this task, we can create a subprogram within our imputer that performs these steps, `ImputeIvar`. Then, our imputer may look like this.

```
// imputer (mymonreg)
program mi_impute_cmd_mymonreg
        version 14.1
        ImputeIvar 1 myreg1
        ImputeIvar 2 myreg2
end
// subprogram defined within mi_impute_cmd_mymonreg.ado
program ImputeIvar, eclass
        args index estres
        /* load the appropriate estimation results */
        quietly estimates restore `estres'
        /* step 2: simulate new beta and sigma */
        tempname sigma beta sigma_mle beta_mle vce_chol rnorm
        matrix `beta_mle' = e(b)
        scalar `sigma_mle' = e(rmse)
        matrix `vce_chol' = cholesky(e(V))/`sigma_mle'
        local ncols = colsof(`beta_mle')
        /* draw beta and sigma from the posterior distribution */
        scalar `sigma' = `sigma_mle'*sqrt(e(df_r)/rchi2(e(df_r)))
        mata: st_matrix("`rnorm'", rnormal(`ncols',1,0,1))
        matrix `beta' = `beta_mle'+(`sigma'*(`vce_chol'*`rnorm'))'
        /* step 3: compute linear prediction */
        ereturn repost b = `beta' // repost new beta
        tempvar xb
        quietly predict double `xb' if ${MI_IMPUTE_user_miss`index'}, xb
        ereturn repost b = `beta_mle' // repost back beta_mle
        /* step 4: replace missing values */
        quietly replace ${MI_IMPUTE_user_ivar`index'} = `xb' +  ///
                    rnormal(0,`sigma') if ${c -(}MI_IMPUTE_user_miss`index'{c )-}==1
end
```

The `ImputeIvar` subprogram is almost the same as the imputer from the univariate regression imputation, except we replaced global macros with their analogs specific to each imputation variable. For example, we replaced `MI_IMPUTE_user_ivar` with `MI_IMPUTE_user_ivar`**`'index'`**, where local macro **`'index'`** will contain a value of 1 or 2. We also passed to the subprogram the corresponding names of the estimation results.

Finally, we write a cleanup program to drop the estimation results we created during initialization from memory.

```
// cleanup program (mymonreg)
program mi_impute_cmd_mymonreg_cleanup
        version 14.1
        capture estimates drop myreg1 myreg2
end
```

Returning to our `auto` example, we can replace missing values in `rep78` and `mpg`.

```
. sysuse auto, clear
(1978 Automobile Data)
. quietly replace mpg = . in 3
. mi set wide
. mi register imputed rep78 mpg
. mi impute mymonreg rep78 mpg = weight, add(1)
Multiple imputation                          Imputations =          1
User method mymonreg                               added =          1
Imputed: m=1                                     updated =          0
```

	Observations per m			
Variable	Complete	Incomplete	Imputed	Total
rep78	69	5	5	74
mpg	73	1	1	74

```
(complete + incomplete = total; imputed is the minimum across m
 of the number of filled-in observations.)
```

Global macros

`mi impute` *usermethod* stores global macros that can be consumed by the programmers of imputation methods. The global macros are `MI_IMPUTE_user_`*name*, where *name* is defined below. Global macro `MI_IMPUTE_user` is set to 1 for all user-defined imputation methods and to 0 for all official imputation methods.

name	Description
`method`	name of the imputation method
`user_options`	method-specific options
`k_ivars`	total number of specified imputation variables (complete and incomplete)
`allivars`	names of all specified imputation variables (complete and incomplete)
`k_ivarsinc`	number of incomplete imputation variables
`ivarsinc`	names of incomplete imputation variables in the original order
`ivars`	synonym for `ivarsinc`
`ivarscomplete`	names of complete imputation variables in the original order
`ivarsincord`	names of incomplete imputation variables ordered from the least missing to the most missing
`ordind`	indices of ordered imputation variables
`incordind`	indices for ordered incomplete imputation variables
`pattern`	`monotone` or `nonmonotone` pattern among all specified imputation variables with respect to the global imputation sample
`ivar#`	name of the #th incomplete imputation variable
`ivar`	synonym for `ivar1`; stored only with one imputation variable
`xvars`	names of complete predictors for all incomplete imputation variables
`xvars#`	names of the complete predictors for the #th incomplete imputation variable
`weight`	global weight expression
`weight#`	weight expression for the #th imputation variable
`touse`	indicator for the global imputation sample
`touse#`	indicator for the imputation sample for the #th imputation variable
`tousevars`	names of all imputation-sample indicators
`miss#`	missing-value indicator for the #th imputation variable
`miss`	synonym for `miss1`; stored only with one imputation variable
`missvars`	names of all missing-value indicators
`m`	current imputation number
`quietly`	contains `quietly` unless `mi impute`'s option `noisily` was specified
`opt_add`	content of option `add()`
`opt_replace`	content of option `replace`
`opt_rseed`	content of option `rseed()`
`opt_double`	content of option `double`
`opt_dots`	content of option `dots`
`opt_noisily`	content of option `noisily`
`opt_nolegend`	content of option `nolegend`
`opt_force`	content of option `force`
`opt_orderasis`	content of option `orderasis`

You may need to define your own global macros. In that case, you need to use the prefix `MI_IMPUTE_userdef_` for all of your global macros to avoid collision with `mi impute`'s internal global macros.

Stored results

`mi impute` *usermethod* stores the following in `r()`:

Scalars

`r(M)`	total number of imputations
`r(M_add)`	number of added imputations
`r(M_update)`	number of updated imputations
`r(k_ivars)`	number of imputed variables
`r(N_g)`	number of imputed groups

Macros

`r(method)`	name of imputation method (*usermethod*)
`r(ivars)`	names of imputation variables
`r(rngstate)`	random-number state used

Matrices

`r(N)`	number of observations in imputation sample
`r(N_complete)`	number of complete observations in imputation sample
`r(N_incomplete)`	number of incomplete observations in imputation sample
`r(N_imputed)`	number of imputed observations in imputation sample

You may also store your own results; see *Storing additional results* for details.

Acknowledgment

The development of this functionality was partially supported by the World Bank.

Also see

[MI] **mi impute** — Impute missing values

[MI] **mi estimate** — Estimation using multiple imputations

[MI] **intro** — Introduction to mi

[MI] **intro substantive** — Introduction to multiple-imputation analysis