

a_eff Package Help

Oleh Komashko

July 19, 2026

v. 0.93

Contents

1	Introduction	1
2	Short description	1
2.1	mf x ()	2
2.2	ey ex ()	2
3	Existing packages interaction	2
4	Public functions	2
4.1	mf x ()	2
4.2	ey ex ()	4
4.3	eff_confint()	4
4.4	smart_formula2()	5
4.5	smart_formula2()	5
4.6	double_formula()	6
4.7	mk_formula2()	6
4.8	norm_dvar()	7
4.9	check_datanames()	7
4.10	sq_times()	8
4.11	cb_inters()	8
4.12	sq_inters()	9
4.13	inters()	9

1 Introduction

The current version of the package provides 12 public functions:

mf x (), ey ex (), eff_confint(), smart_formula2(), mk_formula2(), double_formula(),
check_datanames(), norm_dvar(), cb_inters(), sq_inters(), inters(), and sq_times().

2 Short description

The package provides the following facilities:

- computation of marginal effects after `ols`, `hsk`, `wls`, `tsls`, `arma` (with exogenous regressors), `logit` (including ordered and multinomial models), `probit` (including ordered models), `logistic`, `poisson`, `negbin`, `tobit` (two types of effects), and `heckit` (two types of effects) via `mfx()`.
- computation of elasticities for the same class of models via `eyex()`.
- computation of confidence intervals for marginal effects via `eff_confint()`.
- parsing model formulas using `smart_formula2()`, `double_formula()`, and `mk_formula2()`.
By default, `mfx()` calls `smart_formula2()` internally. In most applications these functions need not be called directly.
- checking the dataset for potentially confusing variable names via `check_datanames()`.
- normalization of discrete variables to the standard sequence 1, 2, ... via `norm_dvar()`.
- creation of polynomial and interaction terms using `cb_inters()`, `sq_inters()`, `inters()`, and `sq_times()`.

2.1 mfx()

The `mfx()` function computes marginal effects.

For continuous variables, marginal effects are evaluated as the corresponding partial derivatives.

For dummy variables, they are computed as discrete changes from 0 to 1.

For discrete variables, they are computed as discrete changes relative to the base level.

By default, marginal effects are computed at the sample means.

They can also be computed at user-specified values.

2.2 eyex()

The `eyex()` function computes elasticities.

It supports the same class of models as `mfx()` and uses the same rules for specifying the points at which the elasticities are computed.

3 Existing packages interaction

Unlike the existing `*mfx` packages, `mfx()` correctly handles level dummies created from discrete variables.

The `*mfx` packages (`lp-mfx` by Allin Cottrell, `poisson_mfx` and `tobit_mfx` by Oleh Komashko) are preferable for models that contain neither transformed regressors nor discrete variables.

The `margeff_el` package by Oleh Komashko computes marginal effects and elasticities for user-specified models.

4 Public functions

4.1 mfx()

```
function bundle mfx (  
  bundle *b           "b is $model in most cases",  
  string formula[null] "the model formula",  
  string at_wrt_type[null] "at and wrt vars names, eff type",  
  matrix at_vals[null]   "values of 'at' variables",  
  string formulaZ[null]  "for zlist-containing models",  
  list da[null]          "dataset, for 2018c only")  
  
command arguments [options]  
# e.g.  
  
tobit y 0 x1 sq_x1  
  
bundle b = $model  
  
mfx(&b)  
# the most typical usage  
  
mfx(&b, formula)  
# saves parsing time when marginal effects are  
# computed repeatedly at many evaluation points,  
# or when using an idiosyncratic formula  
  
mfx(&b, formula ,  
  "wrt = x1; at = x2",  
  {min(x2)})  
  
# computes only the marginal effects of x1  
# at x2 = min(x2), while all other variables  
# are held at their sample means
```

Different variable names are separated by spaces.

```
mfx(&b, formula ,  
  "wrt = x1 x2; at = x2 x3",  
  {min(x2), max(x3)})
```

The number of variables specified after **at** must equal the number of elements in **at_vals**, which may be either a row or a column vector.

The **at_wrt_type** string may contain the components

at, wrt, type

(one, two, or all three in any order), for example

"wrt = x1; type = exp"

The option

"type = exp"

is available after `ols`, `hsk`, `wls`, and `tsls`.

It is useful when the dependent variable is specified in logarithms.

Suppose that the dependent variable is `log(y)`.

With

"type = exp"

marginal effects are computed for `y` rather than for `log(y)`.

See also the Translog production function example in the sample script. After the `tobit` command, the `type` option has a different meaning.

With the default null value, effects on

$$E(y)$$

are computed.

With

"type = inside"

effects on

$$E(y \mid ll < y < ul)$$

are computed, where `ll` and `ul` denote the lower and upper limits, respectively.

After the `heckit` command, the `type` option is interpreted differently.

With the default null value, conditional effects on

$$E(y \mid y \text{ is selected})$$

are computed.

With

"type = Ey"

effects on

$$E(y)$$

are computed.

For all other supported commands, the `type` option is currently ignored.

At present, `formulaZ` is required only after `heckit`, and only when the selection-equation formula cannot be parsed correctly by the default call

`smart_formula2(b.zlist)`

4.2 eyex()

```
function bundle eyex (  
  bundle *b "b is $model in most cases",  
  string formula[null] "the model formula",  
  string at_wrt_type[null] "at and wrt vars names, eff type",  
  matrix at_vals[null] "values of 'at' variables")
```

Computes elasticities.

The syntax is identical to that of `mfx()`.

4.3 eff_confint()

```
function matrix eff_confint (  
  bundle *b "output of mfx()",  
  int df[0] "z- or t-based intervals",  
  scalar cl[NA] "confidence level; NA = 0.95")
```

Computes confidence intervals for marginal effects.

4.4 smart_formula2()

```
function string smart_formula2 (  
  list x "$xlist in most cases",  
  list indlist[null] "forced independent variables")
```

```
smart_formula2($xlist)
```

Idiosyncratic usage The function parses the names and descriptions of the variables in `xlist`. It recognizes formulas such as

$$x1^2 * x7,$$

standard square terms, logarithms, products, and interaction terms.

In some cases we may have, for example,

$$\text{rgdp} = \text{gdp} / \text{deflator}$$

Naturally, we may wish to interpret `rgdp` as an independent variable rather than as a ratio. There are two ways to achieve this.

1. Recommended Assign a description to the variable, for example,

```
setinfo rgdp —description="real gdp"
```

2. Forced interpretation Specify the variable explicitly as independent:

```
smart_formula2($xlist , rgdp)
```

Remember that the output of `smart_formula2()` can also be edited manually.

Alternatively, the model formula may be entered manually and supplied directly to `mfx()`.

See also `mk_formula2()`.

Do not use chained formulas such as

```
x2 = x^2
z = x2*y
```

Instead, write

```
z = x^2*y
```

See also

`sq_times()`, `sq_inters()`, and `cb_inters()`

for convenient creation of common polynomial and interaction terms.

4.5 smart_formula2()

```
function string smart_formula2 (
  list x "$xlist in most cases",
  list indlist[null] "forced independent variables")
```

```
smart_formula2($xlist)
```

Idiosyncratic usage The function parses the names and descriptions of the variables in `xlist`. It recognizes formulas such as `x1^2*x7`, standard square terms, logarithms, products, and interaction terms.

In some cases we may have, for example,

```
rgdp = gdp/deflator
```

Naturally, we may wish to interpret `rgdp` as an independent variable rather than as a ratio. There are two ways to achieve this.

1. Recommended:

```
setinfo rgdp —description="real gdp"
```

2. Alternatively,

```
smart_formula2($xlist , rgdp)
```

Remember that the output of `smart_formula2()` can also be edited manually. Alternatively, the model formula may be entered manually and supplied directly to `mfx()`. See also `mk_formula2()`.

Do not use chained formulas such as

```
x2 = x^2
z = x2*y
```

Instead, use

```
z = x^2*y
```

See also `sq_times()`, `sq_inters()`, and `cb_inters()` for convenient creation of common polynomial and interaction terms.

4.6 double_formula()

```
function string double_formula (
  list xlist "$xlist in most cases",
  list indlist[null] "forced independent variables")
```

This function is similar to `smart_formula2()`, but differs in two respects:

- It is approximately three times slower.
- It can handle two-stage chained variable creation for continuous variables only. It is difficult to imagine practical applications beyond interaction terms for dummy and discrete variables. See the Translog production function example in the sample script.

4.7 mk_formula2()

```
function string mk_formula2 (
  list xlist "$xlist in most cases",
  list vlist[null] "independent continuous and dummy variables",
  list discrlist[null] "independent discrete variables")
```

```
xlist = x1 square(x1) dummify(z)
```

```
mk_formula2(xlist , x1 , z)
```

Compared with `smart_formula2()`, this function requires a more explicit syntax: all functionally independent regressors must be specified.

Since formula parsing is relatively slow, this approach may be more convenient in some situations.

4.8 norm_dvar()

```
function series norm_dvar (
  series *y "a discrete series")
```

The main `mfx()` function works properly with discrete variables only if they take the standard sequence of values

1, 2, 3, ...

This function transforms a discrete variable so that it takes the normalized sequence

1, 2, 3, ...

```
list dilist = list of discrete variables

loop foreach i dilist
  $i = norm_dvar(&($i))
  discrete $i
endloop
```

If some discrete variables take arbitrary values, it is recommended to run this script before estimating the model, before using `dummify()`, or before applying `list^list`.

4.9 `check_datanames()`

```
function void check_datanames (void)
```

Checks the dataset for potentially confusing variable names.
The names

`Dvarname_N`

and

`varname_N`

where `N` is a positive integer, are reserved for level dummies created from discrete variables. The function therefore searches the dataset for other uses of such names and reports any variables that should be renamed.

4.10 `sq_times()`

```
function list sq_times (
  list li1 "list of variables with second-order terms",
  list li2 "list of variables as is",
  bool cubes[0] "whether to include cubes")
```

```
nulldata 20
```

```
x1 = normal()
x2 = normal()
x3 = normal()
```

```
list li1 = x*
```

```
x4 = normal()
```

```
list li3 = sq_times(li1 , x4)
```

This script creates the following new variables (variable descriptions are shown after the colon):

```
sq_x1_x4 : x1^2*x4
x1_x2_x4 : x1*x2*x4
x1_x3_x4 : x1*x3*x4
sq_x2_x4 : x2^2*x4
x2_x3_x4 : x2*x3*x4
sq_x3_x4 : x3^2*x4
```

4.11 cb_inters()

```
function list cb_inters (
  list li1 "list of series",
  bool cubes[0] "whether to include cubes")
```

```
  nulldata 20
```

```
  x1 = normal()
  x2 = normal()
```

```
  list li = x1 x2
```

```
  list li3 = cb_inters(li , 1)
```

This script creates the following new variables (variable descriptions are shown after the colon):

```
cb_x1      : x1^3
sq_x1_x2   : x1^2*x2
sq_x2_x1   : x2^2*x1
cb_x2      : x2^3
```

The corresponding formula generated by

```
eval smart_formula2(li3)
```

is

```
x1^3 x1^2*x2 x2^2*x1 x2^3; x1 x2; _empty
```

4.12 sq_inters()

```
function list sq_inters (
  list li1 "list of series",
  bool squares[0] "whether to include squares")
```

Sometimes only interaction terms are required.

```

nulldata 20

x1 = normal()
x2 = normal()
x3 = normal()

list li = x*

list li2 = sq_inters(li)

```

This script creates the following new variables (variable descriptions are shown after the colon):

```

x1_x2 : x1*x2
x1_x3 : x1*x3
x2_x3 : x2*x3

```

4.13 inters()

```

function list inters (
  list li1 "list of series",
  list li2 "list of series",
  bool squares[0] "whether to include squares")

```

Creates cross-interaction terms between the variables in two lists. If a variable appears in both lists and **squares** = 1, its square is also included in the output list.

The native gretl expression

dummy[^]list

treats **dummy** as a binary (0/1) variable.

On the other hand, **mfx()** and **eyex()** require all discrete variables to be normalized to the sequence

1, 2, 3, ...

This function makes it easy to compute effects for models containing interactions between discrete and continuous variables.

For example,

```

nulldata 20
set seed 13

y = normal()
x1 = normal()
x2 = normal()

# a discrete variable taking values from {1,2,3}

```

```
di = randgen(i,1,3)
```

```
list X = x*
```

Running

```
inters(dummify(di,NA), X)
```

creates

```
Ddi_1      : dummy for di = 1
Ddi_2      : dummy for di = 2
Ddi_3      : dummy for di = 3
Ddi_1_x1   : Ddi_1*x1
Ddi_1_x2   : Ddi_1*x2
Ddi_2_x1   : Ddi_2*x1
Ddi_2_x2   : Ddi_2*x2
Ddi_3_x1   : Ddi_3*x1
Ddi_3_x2   : Ddi_3*x2
```

where

- Ddi_1, Ddi_2, and Ddi_3 are created by `dummify(di,NA)`;
- the remaining variables are generated by `inters()`.

Variable descriptions are shown after the colon.

```
ols y 0 inters(dummify(di,NA), X)
```

The resulting regressor list is

```
const Ddi_1_x1 Ddi_1_x2 Ddi_2_x1 Ddi_2_x2 Ddi_3_x1 Ddi_3_x2
```

The following commands estimate the model and compute the marginal effects:

```
bundle b2 = $model
```

```
b2fla = smart_formula2($xlist)
```

```
mfx(&b2, b2fla)
```

We can now easily compute three different slopes for `x1` and `x2`:

```
mfx(&b2, b2fla, "wrt = x1 x2; at = di", 1)
```

```
mfx(&b2, b2fla, "wrt = x1 x2; at = di", 2)
```

```
mfx(&b2, b2fla, "wrt = x1 x2; at = di", 3)
```

Changelog

NEWS in v0.3

- Significantly improved execution speed without changing the external interface.

NEWS in v0.4

- Improved the implementation of `smart_formula2()` without changing its syntax.
- Removed the `clist` and `dilist` arguments from `mfx()`. With the current implementation, these arguments are no longer needed.
- Added the `link` argument for use with linear models when the dependent variable is specified in logarithms. It allows different types of effects to be computed.
- Added the new function `double_formula()`. Although slower than `smart_formula2()`, it is more powerful.
- Rewrote the sample script: timing tests are no longer required, unnecessary repetitions have been removed, and a new example illustrating the `link` argument has been added.

NEWS in v0.6

- Added a new type of marginal effect for the Tobit model, $E(y \mid ll < y < ul)$.
- Added the new function `eyex()` to compute elasticities for the same class of models as `mfx()`.
- Added examples illustrating the use of `eyex()` in the sample script.

NEWS in v0.7

Bug fixes

- Corrected `mfx()` after Tobit and multinomial logit. In some cases, models estimated without a constant produced errors.
- Corrected `eyex()` after multinomial logit. Although the computed results were correct, the printed output contained incomplete sets of marginal effects and elasticities.
- Expanded the output bundle to support the m-functions provided by the new version of `waldTest.gfn`.
- Modified the syntax in preparation for future support of two-equation models (`heckit` and `mixed logit`).
- Deprecated the `link` argument. Instead, the third argument may now contain up to three components: `at` (as before), `wrt` (as before), and `type` (new).

Examples:

```
"wrt = x1; type = exp; at = x2"
```

for linear models, and

```
"type = inside"
```

for Tobit models.

NEWS in v0.8

- Improved the expert system used by `smart_formula2()` and `double_formula()`.
- Formulas containing non-differentiable functions are now treated as independent variables. For example,

```
x = randgen (...)
```

is interpreted as an independent variable rather than as a function.

- Variables with empty descriptions are now treated as independent variables by default.
- Added the `inters()` function for creating cross-interaction terms between variables from two lists.
- Corrected several bugs that were not visible in the sample script.
- Eliminated warnings caused by variable names coinciding with function names.
- Added two types of marginal effects and two types of elasticities for `heckit`.
- Added support for `type = exp` in `eyex()` with the same meaning as in `mfx()`.
- Avoided the "penetration" problem in gretl 2018c.

Note. This problem has been fixed in snapshot versions of gretl released after 19 October 2018.

Changes in v0.9 (by the gretl team)

- Required gretl 2018d, thereby removing warnings related to gretl 2018c.

Changes in v0.91 (by the gretl team)

- Removed internally deprecated syntax and performed additional code cleanup.

NEWS in v0.93

- Converted the package documentation from text help to PDF help.
- Revised and expanded the documentation.
- Improved the English, consistency, and readability throughout the help file.
- Updated several examples and function descriptions for greater clarity.