Title

moptimize( ) — Model optimization

Syntax

If you are reading this entry for the first time, skip down to Description and to Remarks, and more especially, to Mathematical statement of the moptimize( ) problem under Remarks.

Syntax is presented under the following headings:

Step 1: Initialization

M = `moptimize_init()`

Step 2: Definition of maximization or minimization problem

In each of the functions, the last argument is optional. If specified, the function sets the value and returns void. If not specified, no change is made, and instead what is currently set is returned.

(varies) `moptimize_init_which(`*M*`, {"max"|"min"})`

(varies) `moptimize_init_evaluator(`*M*`, &`*functionname*`())`

(varies) `moptimize_init_evaluator(`*M*`, "`*programname*`")`

(varies) `moptimize_init_evaluatortype(`*M*`, `*evaluatortype*`)`

(varies) `moptimize_init_negH(`*M*`, {"off"|"on"})`

(varies) `moptimize_init_touse(`*M*`, "`*tousevarname*`")`

(varies) `moptimize_init_ndepvars(`*M*`, `*D*`)`

(varies) `moptimize_init_depvar(`*M*`, `*j*`, `*y*`)`

(varies) `moptimize_init_eq_n(`*M*`, `*m*`)`

(varies) `moptimize_init_eq_indepvars(`*M*`, `*i*`, `*X*`)`

(varies) `moptimize_init_eq_cons(`*M*`, `*i*`, {"on"|"off"})`

(varies) `moptimize_init_eq_offset(`*M*`, `*i*`, `*o*`)`

(varies) `moptimize_init_eq_exposure(`*M*`, `*i*`, `*t*`)`

(varies) `moptimize_init_eq_name(`*M*`, `*i*`, `*name*`)`

(varies) `moptimize_init_eq_colnames(`*M*`, `*i*`, `*names*`)`

(varies) `moptimize_init_eq_coefs(`*M*`, `*i*`, `*b0*`)`

(varies) `moptimize_init_constraints(`*M*`, `*Cc*`)`

(varies) `moptimize_init_search(`*M*`, {"on"|"off"})`

(varies) `moptimize_init_search_random(`*M*`, {"off"|"on"})`

(varies) `moptimize_init_search_repeat(`*M*`, `*nr*`)`

(varies) `moptimize_init_search_bounds(`*M*`, `*i*`, `*minmax*`)`

(varies) `moptimize_init_search_rescale(`*M*`, {"on"|"off"})`

(varies) `moptimize_init_weight(`*M*`,` *w*`)`

(varies) `moptimize_init_weighttype(`*M*`,` *weighttype*`)`

(varies) `moptimize_init_cluster(`*M*`,` *c*`)`

(varies) `moptimize_init_svy(`*M*`,` { `"off"` | `"on"` }`)`

(varies) `moptimize_init_by(`*M*`,` *by*`)`

(varies) `moptimize_init_nuserinfo(`*M*`,` *n_user*`)`

(varies) `moptimize_init_userinfo(`*M*`,` *l*`,` *Z*`)`

(varies) `moptimize_init_technique(`*M*`,` *technique*`)`

(varies) `moptimize_init_vcetype(`*M*`,` *vcetype*`)`

(varies) `moptimize_init_nmsimplexdeltas(`*M*`,` *delta*`)`

(varies) `moptimize_init_gnweightmatrix(`*M*`,` *W*`)`

(varies) `moptimize_init_singularHmethod(`*M*`,` *singularHmethod*`)`

(varies) `moptimize_init_conv_maxiter(`*M*`,` *maxiter*`)`

(varies) `moptimize_init_conv_warning(`*M*`,` { `"on"` | `"off"` }`)`

(varies) `moptimize_init_conv_ptol(`*M*`,` *ptol*`)`

(varies) `moptimize_init_conv_vtol(`*M*`,` *vtol*`)`

(varies) `moptimize_init_conv_nrtol(`*M*`,` *nrtol*`)`

(varies) `moptimize_init_conv_ignorenrtol(`*M*`,` { `"off"` | `"on"` }`)`

(varies) `moptimize_init_iterid(`*M*`,` *id*`)`

(varies) `moptimize_init_valueid(`*M*`,` *id*`)`

(varies) `moptimize_init_tracelevel(`*M*`,` *tracelevel*`)`

(varies) `moptimize_init_trace_ado(`*M*`,` { `"off"` | `"on"` }`)`

(varies) `moptimize_init_trace_dots(`*M*`,` { `"off"` | `"on"` }`)`

(varies) `moptimize_init_trace_value(`*M*`,` { `"on"` | `"off"` }`)`

(varies) `moptimize_init_trace_tol(`*M*`,` { `"off"` | `"on"` }`)`

(varies) `moptimize_init_trace_step(`*M*`,` { `"off"` | `"on"` }`)`

(varies) `moptimize_init_trace_coefdiffs(`*M*`,` { `"off"` | `"on"` }`)`

(varies) `moptimize_init_trace_coefs(`*M*`,` { `"off"` | `"on"` }`)`

(varies) `moptimize_init_trace_gradient(`*M*`,` { `"off"` | `"on"` }`)`

(varies) `moptimize_init_trace_Hessian(`*M*`,` { `"off"` | `"on"` }`)`

(varies) `moptimize_init_evaluations(`*M*`,` { `"off"` | `"on"` }`)`

(varies) `moptimize_init_verbose(`*M*`,` { `"on"` | `"off"` }`)`

Step 3: Perform optimization or perform a single function evaluation

void `moptimize(`*M*`)`

real scalar `_moptimize(`*M*`)`

void `moptimize_evaluate(`*M*`)`

real scalar `_moptimize_evaluate(`*M*`)`

Step 4: Post, display, or obtain results

void `moptimize_result_post(`*M* [, *vcetype*]`)`

void `moptimize_result_display(`[*M* [, *vcetype*]]`)`

real scalar `moptimize_result_value(`*M*`)`

real scalar `moptimize_result_value0(`*M*`)`

real rowvector `moptimize_result_eq_coefs(`*M* [, *i*]`)`

real rowvector `moptimize_result_coefs(`*M*`)`

string matrix `moptimize_result_colstripe(`*M* [, *i*]`)`

real matrix `moptimize_result_scores(`*M*`)`

real rowvector `moptimize_result_gradient(`*M* [, *i*]`)`

real matrix `moptimize_result_Hessian(`*M* [, *i*]`)`

real matrix `moptimize_result_V(`*M* [, *i*]`)`

string scalar `moptimize_result_Vtype(`*M*`)`

real matrix `moptimize_result_V_oim(`*M* [, *i*]`)`

real matrix `moptimize_result_V_opg(`*M* [, *i*]`)`

real matrix `moptimize_result_V_robust(`*M* [, *i*]`)`

real scalar	`moptimize_result_iterations(`*M*`)`
real scalar	`moptimize_result_converged(`*M*`)`
real colvector	`moptimize_result_iterationlog(`*M*`)`
real rowvector	`moptimize_result_evaluations(`*M*`)`
real scalar	`moptimize_result_errorcode(`*M*`)`
string scalar	`moptimize_result_errortext(`*M*`)`
real scalar	`moptimize_result_returncode(`*M*`)`
void	`moptimize_ado_cleanup(`*M*`)`

Utility functions for use in all steps

void	`moptimize_query(`*M*`)`
real matrix	`moptimize_util_eq_indices(`*M*`,` *i* [`,` *i2*]`)`
(varies)	`moptimize_util_depvar(`*M*`,` *j*`)`
	returns y set by `moptimize_init_depvar(`*M*`,` *j*`,` *y*`)`*, which is usually a real colvector*
real colvector	`moptimize_util_xb(`*M*`,` *b*`,` *i*`)`
real scalar	`moptimize_util_sum(`*M*`,` *real colvector v*`)`
real rowvector	`moptimize_util_vecsum(`*M*`,` *i*`,` *real colvector s*`,` *real scalar value*`)`
real matrix	`moptimize_util_matsum(`*M*`,` *i*`,` *i2*`,` *real colvector s*`,` *real scalar value*`)`
real matrix	`moptimize_util_matbysum(`*M*`,` *i*`,` *real colvector a*`,` *real colvector b*`,` *real scalar value*`)`
real matrix	`moptimize_util_matbysum(`*M*`,` *i*`,` *i2*`,` *real colvector a*`,` *real colvector b*`,` *real colvector c*`,` *real scalar value*`)`
pointer scalar	`moptimize_util_by(`*M*`)`

Definition of M

M, if it is declared, should be declared transmorphic. *M* is obtained from `moptimize_init()` and then passed as an argument to the other `moptimize()` functions.

`moptimize_init()` returns *M*, called an `moptimize()` problem handle. The function takes no arguments. *M* holds the information about the optimization problem.

Setting the sample

Various `moptimize_init_*()` functions set values for dependent variables, independent variables, etc. When you set those values, you do that either by specifying Stata variable names or by specifying Mata matrices containing the data themselves. Function `moptimize_init_touse()` specifies the sample to be used when you specify Stata variable names.

`moptimize_init_touse(`*M*`, "`*tousevarname*`")` specifies the name of the variable in the Stata dataset that marks the observations to be included. Observations for which the Stata variable is nonzero are included. The default is `""`, meaning all observations are to be used.

You need to specify *tousevarname* only if you specify Stata variable names in the other `moptimize_init_*()` functions, and even then it is not required. Setting `tousevar` when you specify the data themselves via Mata matrices, whether views or not, has no effect.

Specifying dependent variables

D and *j* index dependent variables:

index	description
D	number of dependent variables, $D \geq 0$
j	dependent variable index, $1 \leq j \leq D$

D and *j* are real scalars.

You set the dependent variables one at a time. In a particular optimization problem, you may have no dependent variables or have more dependent variables than equations.

`moptimize_init_depvar(`*M*`, `*j*`, `*y*`)` sets the *j*th dependent variable to be *y*. *y* may be a string scalar containing a Stata variable name that in turn contains the values of the *j*th dependent variable, or *y* may be a real colvector directly containing the values.

`moptimize_init_ndepvars(`*M*`, `*D*`)` sets the total number of dependent variables. You can set *D* before defining dependent variables, and that speeds execution slightly, but it is not necessary because *D* is automatically set to the maximum *j*.

Specifying independent variables

Independent variables are defined within parameters or, equivalently, equations. The words parameter and equation mean the same thing. *m*, *i*, and *i2* index parameters:

index	description
m	number of parameters (equations), $m \geq 1$
i	equation index, $1 \leq i \leq m$
i2	equation index, $1 \leq i2 \leq m$

m, *i*, and *i2* are real scalars.

The function to be optimized is $f(p_1, p_2, \ldots, p_m)$. The ith parameter (equation) is defined as

$$pi = Xi \times bi' + oi + \ln(ti) \;\texttt{:+}\; b0i$$

where

pi: $Ni \times 1$	(*i*th parameter)
Xi: $Ni \times ki$	(*Ni* observations on *ki* independent variables)
bi: $1 \times ki$	(coefficients to be fit)
oi: $Ni \times 1$	(exposure/offset in offset form, optional)
ti: $Ni \times 1$	(exposure/offset in exposure form, optional)
$b0i$: 1×1	(constant or intercept, optional)

Any of the terms may be omitted. The most common forms for a parameter are $pi = Xi \times bi' + b0i$ (standard model), $pi = Xi \times bi'$ (no-constant model), and $pi = b0i$ (constant-only model).

In addition, define b: $1 \times K$ as the entire coefficient vector, which is to say,

$$b = (b1, [b01,] \quad b2, [b02,] \quad \ldots)$$

That is, because bi is $1 \times ki$ for $i = 1, 2, \ldots, m$, then b is $1 \times K$, where $K = \sum_i ki + ci$, where ci is 1 if equation i contains an intercept and is 0 otherwise. Note that bi does not contain the constant or intercept, if there is one, but b contains all the coefficients, including the intercepts. b is called *the full set of coefficients.*

Parameters are defined one at a time by using the following functions:

`moptimize_init_eq_n(`*M*`,` *m*`)` sets the number of parameters. Use of this function is optional; *m* will be automatically determined from the other `moptimize_init_eq_*()` functions you issue.

`moptimize_init_eq_indepvars(`*M*`,` *i*`,` *X*`)` sets *X* to be the data (independent variables) for the *i*th parameter. *X* may be a $1 \times ki$ string rowvector containing Stata variable names, or *X* may be a string scalar containing the same names in space-separated format, or *X* may be an $Ni \times ki$ real matrix containing the data for the independent variables. Specify *X* as `""` to omit term $Xi \times bi'$, for instance, as when fitting a constant-only model. The default is `""`.

`moptimize_init_eq_cons(`*M*`,` *i*`,` {`"on"`|`"off"`}`)` specifies whether the equation for the *i*th parameter includes *b0i*, a constant or intercept. Specify `"on"` to include *b0i*, `"off"` to exclude it. The default is `"on"`.

`moptimize_init_eq_offset(`*M*`,` *i*`,` *o*`)` specifies *oi* in the equation for the *i*th parameter. *o* may be a string scalar containing a Stata variable name, or *o* may be an $Ni \times 1$ real colvector containing the offsets. The default is `""`, meaning term *oi* is omitted. Parameters may not have both *oi* and ln(*ti*) terms.

`moptimize_init_eq_exposure(`*M*`,` *i*`,` *t*`)` specifies *ti* in term ln(*ti*) of the equation for the *i*th parameter. *t* may be a string scalar containing a Stata variable name, or *t* may be an $Ni \times 1$ real colvector containing the exposure values. The default is `""`, meaning term ln(*ti*) is omitted.

`moptimize_init_eq_name(`*M*`,` *i*`,` *name*`)` specifies a string scalar, *name*, to be used in the output to label the *i*th parameter. The default is to use an automatically generated name.

`moptimize_init_eq_colnames(`*M*`,` *i*`,` *names*`)` specifies a $1 \times ki$ string rowvector, *names*, to be used in the output to label the coefficients for the *i*th parameter. The default is to use automatically generated names.

Specifying constraints

Linear constraints may be placed on the coefficients, *b*, which may be either within equation or between equations.

`moptimize_init_constraints(`*M*`,` *Cc*`)` specifies an $R \times K+1$ real matrix, *Cc*, that places *R* linear restrictions on the $1 \times K$ full set of coefficients, *b*. Think of *Cc* as being (*C*,*c*), *C*: $R \times K$ and *c*: $R \times 1$. Optimization will be performed subject to the constraint $Cb' = c$. The default is no constraints.

Specifying weights or survey data

You may specify weights, and once you do, everything is automatic, assuming you implement your evaluator by using the provided utility functions.

`moptimize_init_weight(`*M*`,` *w*`)` specifies the weighting variable or data. *w* may be a string scalar containing a Stata variable name, or *w* may be a real colvector directly containing the weight values. The default is `""`, meaning no weights.

`moptimize_init_weighttype(`*M*`,` *weighttype*`)` specifies how *w* is to be treated. *weighttype* may be `"fweight"`, `"aweight"`, `"pweight"`, or `"iweight"`. You may set *w* first and then *weighttype*, or the reverse. If you set *w* without setting *weighttype*, then `"fweight"` is assumed. If you set *weighttype* without setting *w*, then *weighttype* is ignored. The default *weighttype* is `"fweight"`.

Alternatively, you may inherit the full set of survey settings from Stata by using `moptimize_init_svy()`. If you do this, do not use `moptimize_init_weight()`, `moptimize_init_weighttype()`, or `moptimize_init_cluster()`. When you use the survey settings, everything is nearly automatic, assuming you use the provided utility functions to implement your evaluator. The proviso is that your evaluator must be of evaluatortype `lf`, `lf*`, `gf`, or `q`.

`moptimize_init_svy(`*M*`,` { `"off"` | `"on"` }`)` specifies whether Stata's survey settings should be used. The default is `"off"`. Using the survey settings changes the default *vcetype* to `"svy"`, which is equivalent to `"robust"`.

Specifying clusters and panels

Clustering refers to possible nonindependence of the observations within groups called clusters. A cluster variable takes on the same value within a cluster and different values across clusters. After setting the cluster variable, there is nothing special you have to do, but be aware that clustering is allowed only if you use a type `lf`, `lf*`, `gf`, or `q` evaluator. `moptimize_init_cluster()` allows you to set a cluster variable.

Panels refer to likelihood functions or other objective functions that can only be calculated at the panel level, for which there is no observation-by-observation decomposition. Unlike clusters, these panel likelihood functions are difficult to calculate and require the use of type `d` or `gf` evaluators. A panel variable takes on the same value within a panel and different values across panels. `moptimize_init_by()` allows you to set a panel variable.

You may set both a cluster variable and a panel variable, but be careful because, for most likelihood functions, panels are mathematically required to be nested within cluster.

`moptimize_init_cluster(`*M*`,` *c*`)` specifies a cluster variable. *c* may be a string scalar containing a Stata variable name, or *c* may be a real colvector directly containing the cluster values. The default is `""`, meaning no clustering. If clustering is specified, the default *vcetype* becomes `"robust"`.

`moptimize_init_by(`*M*`,` *by*`)` specifies a panel variable and specifies that only panel-level calculations are meaningful. *by* may be a string scalar containing a Stata variable name, or *by* may be a real colvector directly containing the panel ID values. The default is `""`, meaning no panels. If panels are specified, the default *vcetype* remains unchanged, but if the opg variance estimator is used, the opg calculation is modified so that it is clustered at the panel level.

Specifying optimization technique

Technique refers to the numerical methods used to solve the optimization problem. The default is Newton–Raphson maximization.

`moptimize_init_which(`*M*`,` { `"max"` | `"min"` }`)` sets whether the maximum or minimum of the objective function is to be found. The default is `"max"`.

`moptimize_init_technique(`*M*`,` *technique*`)` specifies the technique to be used to find the coefficient vector *b* that maximizes or minimizes the objective function. Allowed values are

technique	description
`"nr"`	modified Newton–Raphson
`"dfp"`	Davidon–Fletcher–Powell
`"bfgs"`	Broyden–Fletcher–Goldfarb–Shanno
`"bhhh"`	Berndt–Hall–Hall–Hausman
`"nm"`	Nelder–Mead
`"gn"`	Gauss–Newton (quadratic optimization)

The default is `"nr"`.

You can switch between `"nr"`, `"dfp"`, `"bfgs"`, and `"bhhh"` by specifying two or more of them in a space-separated list. By default, `moptimize()` will use an algorithm for five iterations before switching to the next algorithm. To specify a different number of iterations, include the number after the technique. For example, specifying `moptimize_init_technique(`*M*`,` `"bhhh 10 nr 1000")` requests that `moptimize()` perform 10 iterations using the Berndt–Hall–Hall–Hausman algorithm, followed by 1,000 iterations using the modified Newton-Raphson algorithm, and then switch back to Berndt–Hall–Hall–Hausman for 10 iterations, and so on. The process continues until convergence or until *maxiter* is exceeded.

`moptimize_init_singularHmethod(`*M*`,` *singularHmethod*`)` specifies the action to be taken during optimization if the Hessian is found to be singular and the *technique* requires the Hessian be of full rank. Allowed values are

singularHmethod	description
`"m-marquardt"`	modified Marquardt algorithm
`"hybrid"`	mixture of steepest descent and Newton

The default is `"m-marquardt"`.
`"hybrid"` is equivalent to `ml`'s `difficult` option; see [R] **ml**.

`moptimize_init_nmsimplexdeltas(`*M*`,` *delta*`)` is for use with Nelder–Mead, also known as technique `nm`. This function sets the values of *delta* to be used, along with the initial parameters, to build the simplex required by Nelder–Mead. Use of this function is required only in the Nelder–Mead case. The values in *delta* must be at least 10 times larger than *ptol*. The initial simplex will be $\{p, p+(d_1,0,\dots 0), p+(0,d_2,0,\dots,0),\dots, p+(0,0,\dots,0,d_K)\}$.

Specifying initial values

Initial values are values you optionally specify that via a search procedure result in starting values that are then used for the first iteration of the optimization technique. That is,

$$\text{initial values} \xrightarrow{\textit{(searching)}} \text{starting values} \xrightarrow{\substack{\textit{(optimization}\\ \textit{technique)}}} \text{final results}$$

Initial values are specified parameter by parameter.

`moptimize_init_eq_coefs(`*M*`,` *i*`,` *b0*`)` sets the initial values of the coefficients for the *i*th parameter to be *b0*: $1 \times (ki + ci)$. The default is (0, 0, ..., 0).

The following functions control whether searching is used to improve on the initial values to produce better starting values. In addition to searching a predetermined set of hardcoded starting values, there are two other methods that can be used to improve on the initial values: random and rescaling. By default, random is off and rescaling is on. You can use one, the other, or both.

`moptimize_init_search(`*M*`,` { `"on"` | `"off"` }`)` determines whether any attempts are to be made to improve on the initial values via a search technique. The default is `"on"`. If you specify `"off"`, the initial values become the starting values.

`moptimize_init_search_random(`*M*`,` { `"off"` | `"on"` }`)` determines whether the random method of improving initial values is to be attempted. The default is `"off"`. Use of the random method is recommended when the initial values are or might be infeasible. Infeasible means that the function cannot be evaluated, which mechanically corresponds to the user-written evaluator returning a missing value. The random method is seldom able to improve on feasible initial values. It works well when the initial values are or might be infeasible.

`moptimize_init_search_repeat(`*M*`,` *nr*`)` controls how many times random values are tried if the random method is turned on. The default is 10.

`moptimize_init_search_bounds(`*M*`,` *i*`,` *minmax*`)` specifies the bounds for the random search. *minmax* is a 1×2 real rowvector containing the minimum and maximum values for the *i*th parameter (equation). The default is (`.`, `.`), meaning no lower and no upper bounds.

`moptimize_init_search_rescale(`*M*`,` { `"on"` | `"off"` }`)` determines whether rescaling is attempted. The default is `"on"`. Rescaling is a deterministic (not random) method. It also usually improves initial values, and usually reduces the number of subsequent iterations required by the optimization technique.

Performing one evaluation of the objective function

`moptimize_evaluate(`*M*`)` and `_moptimize_evaluate(`*M*`)` perform one evaluation of the function evaluated at the initial values. Results can be accessed by using `moptimize_result()`, including first- and second-derivative-based results.

`moptimize_evaluate()` and `_moptimize_evaluate()` do the same thing, differing only in that `moptimize_evaluate()` aborts with a nonzero return code if things go badly, whereas `_moptimize_evaluate()` returns the real scalar error code. An infeasible initial value is an error.

The evaluation is performed at the initial values, not the starting values, and this is true even if search is turned on. If you want to perform an evaluation at the starting values, then perform optimization with *maxiter* set to 0.

Performing optimization of the objective function

`moptimize(`*M*`)` and `_moptimize(`*M*`)` perform optimization. Both routines do the same thing; they differ only in their behavior when things go badly. `moptimize()` returns nothing and aborts with error. `_moptimize()` returns a real scalar error code. `moptimize()` is best for interactive use and often adequate for use in programs that do not want to consider the possibility that optimization might fail.

The optimization process is as follows:

1. The initial values are used to create starting values. The value of the function at the starting values is calculated. If that results in a missing value, the starting values are declared infeasible. `moptimize()` aborts with return code 430; `_moptimize()` returns a nonzero error code, which maps to 430 via `moptimize_result_returncode()`. This step is called iteration 0.

2. The starting values are passed to the technique to produce better values. Usually this involves the technique calculating first and second derivatives, numerically or analytically, and then stepping multiple times in the appropriate direction, but techniques can vary on this. In general, the technique performs what it calls one iteration, the result of which is to produce better values. Those new values then become the starting values and the process repeats.

   An iteration is said to fail if the new coefficient vector is infeasible (results in a missing value). Then attempts are made to recover and, if those attempts are successful, optimization continues. If they fail, `moptimize()` aborts with error and `_moptimize()` returns a nonzero error code.

Other problems may arise, such as singular Hessians or the inability to find better values. Various fix-ups are made and optimization continues. These are not failures.

This step is called iterations 1, 2, and so on.

3. Step 2 continues either until the process converges or until the maximum number of iterations (*maxiter*) is exceeded. Stopping due to *maxiter* is not considered an error. Upon completion, programmers should check `moptimize_result_converged()`.

If optimization succeeds, which is to say, if `moptimize()` does not abort or `_moptimize()` returns 0, you can use the `moptimize_result()` functions to access results.

Tracing optimization

`moptimize()` and `_moptimize()` will produce output like

```
Iteration 0:   f(p) = .........
Iteration 1:   f(p) = .........
```

You can change the $f(p)$ to be "log likelihood" or whatever else you want. You can also change "Iteration".

`moptimize_init_iterid(`*M*`,` *id*`)` sets the string to be used to label the iterations in the iteration log. *id* is a string scalar. The default is `"Iteration"`.

`moptimize_init_valueid(`*M*`,` *id*`)` sets the string to be used to label the objective function value in the iteration log. *id* is a string scalar. The default is `"f(p)"`.

Additional results can be displayed during optimization, which can be useful when you are debugging your evaluator. This is called tracing the execution.

`moptimize_init_tracelevel(`*M*`,` *tracelevel*`)` specifies the output to be displayed during the optimization process. Allowed values are

tracelevel	to be displayed each iteration
`"none"`	nothing
`"value"`	function value
`"tolerance"`	previous + convergence values
`"step"`	previous + stepping information
`"coefdiffs"`	previous + parameter relative differences
`"paramdiffs"`	same as `"coefdiffs"`
`"coefs"`	previous + parameter values
`"params"`	same as `"coefs"`
`"gradient"`	previous + gradient vector
`"hessian"`	previous + Hessian matrix

The default is `"value"`.

Setting *tracelevel* is a shortcut. The other trace functions allow you to turn on and off individual features. In what follows, the documented defaults are the defaults when *tracelevel* is `"value"`.

`moptimize_init_trace_ado(`*M*`, {"off"|"on"})` traces the execution of evaluators written as ado-files. This topic is not discussed in this manual entry. The default is `"off"`.

`moptimize_init_trace_dots(`*M*`,` { `"off"` | `"on"` }`)` displays a dot each time your evaluator is called. The default is `"off"`.

`moptimize_init_trace_value(`*M*`,` { `"on"` | `"off"` }`)` displays the function value at the start of each iteration. The default is `"on"`.

`moptimize_init_trace_tol(`*M*`,` { `"off"` | `"on"` }`)` displays the value of the calculated result that is compared with the effective convergence criterion at the end of each iteration. The default is `"off"`.

`moptimize_init_trace_step(`*M*`,` { `"off"` | `"on"` }`)` displays the steps within iteration. Listed are the value of objective function along with the word forward or backward. The default is `"off"`.

`moptimize_init_trace_coefdiffs(`*M*`,` { `"off"` | `"on"` }`)` displays the coefficient relative differences from the previous iteration that are greater than the coefficient tolerance *ptol*. The default is `"off"`.

`moptimize_init_trace_coefs(`*M*`,` { `"off"` | `"on"` }`)` displays the coefficients at the start of each iteration. The default is `"off"`.

`moptimize_init_trace_gradient(`*M*`,` { `"off"` | `"on"` }`)` displays the gradient vector at the start of each iteration. The default is `"off"`.

`moptimize_init_trace_Hessian(`*M*`,` { `"off"` | `"on"` }`)` displays the Hessian matrix at the start of each iteration. The default is `"off"`.

Specifying convergence criteria

Convergence is based on several rules controlled by four parameters: *maxiter*, *ptol*, *vtol*, and *nrtol*. The first rule is not a convergence rule, but a stopping rule, and it is controlled by *maxiter*.

`moptimize_init_conv_maxiter(`*M*`,` *maxiter*`)` specifies the maximum number of iterations. If this number is exceeded, optimization stops and results are posted where they are accessible by using the `moptimize_result_*()` functions, just as if convergence had been achieved. `moptimize_result_converged()`, however, is set to 0 rather than 1. The default *maxiter* is Stata's `c(maxiter)`, which is usually 16,000.

`moptimize_init_conv_warning(`*M*`,` { `"on"` | `"off"` }`)` specifies whether the warning message "convergence not achieved" is to be displayed when this stopping rule is invoked. The default is `"on"`.

Usually, convergence occurs before the stopping rule comes into effect. The convergence criterion is a function of three real scalar values: *ptol*, *vtol*, and *nrtol*. Let

$$\begin{aligned} b &= \text{full set of coefficients} \\ b\_prior &= \text{value of } b \text{ from prior iteration} \\ v &= \text{value of objective function} \\ v\_prior &= \text{value of } v \text{ from prior iteration} \\ g &= \text{gradient vector from this iteration} \\ H &= \text{Hessian matrix from this iteration} \end{aligned}$$

Define, for maximization,

$$\begin{aligned} C\_ptol&: \texttt{mreldif(}b\texttt{, } b\_prior\texttt{)} \le ptol \\ C\_vtol&: \texttt{reldif(}v\texttt{, } v\_prior\texttt{)} \le vtol \\ C\_nrtol&: g \times \texttt{invsym(} -H\texttt{)} \times g' < nrtol \\ C\_concave&: -H \text{ is positive semidefinite} \end{aligned}$$

For minimization, think in terms of maximization of $-f(p)$. Convergence is declared when

$$(C\_ptol \,|\, C\_vtol) \;\&\; C\_nrtol \;\&\; C\_concave$$

The above applies in cases of derivative-based optimization, which currently is all techniques except `"nm"` (Nelder–Mead). In the Nelder–Mead case, the criterion is

$$\begin{aligned} C\_ptol&: \texttt{mreldif(}\text{vertices of } R\texttt{)} \le ptol \\ C\_vtol&: \texttt{reldif(}R\texttt{)} \le vtol \end{aligned}$$

where *R* is the minimum and maximum values on the simplex. Convergence is declared when *C_ptol* | *C_vtol*.

The values of *ptol*, *vtol*, and *nrtol* are set by the following functions:

`moptimize_init_conv_ptol(`*M*`, ` *ptol*`)` sets *ptol*. The default is 1e–6.

`moptimize_init_conv_vtol(`*M*`, ` *vtol*`)` sets *vtol*. The default is 1e–7.

`moptimize_init_conv_nrtol(`*M*`, ` *nrtol*`)` sets *nrtol*. The default is 1e–5.

`moptimize_init_conv_ignorenrtol(`*M*`, {"off"|"on"})` sets whether *C_nrtol* should always be treated as true, which in effect removes the *nrtol* criterion from the convergence rule. The default is `"off"`.

Accessing results

Once you have successfully performed optimization, or you have successfully performed a single function evaluation, you may display results, post results to Stata, or access individual results.

To display results, use `moptimize_result_display()`.

`moptimize_result_display(`*M*`)` displays estimation results. Standard errors are shown using the default *vcetype*.

`moptimize_result_display(`*M*`, ` *vcetype*`)` displays estimation results. Standard errors are shown using the specified *vcetype*.

Also there is a third syntax for use after results have been posted to Stata, which we will discuss below.

`moptimize_result_display()` without arguments (not even *M*) displays the estimation results currently posted in Stata.

vcetype specifies how the variance–covariance matrix of the estimators (VCE) is to be calculated. Allowed values are

vcetype	description
`""`	use default for technique
`"oim"`	observed information matrix
`"opg"`	outer product of gradients
`"robust"`	Huber/White/sandwich estimator
`"svy"`	survey estimator; equivalent to `robust`

The default *vcetype* is `oim` except for technique `bhhh`, where it is `opg`. If survey, `pweights`, or clusters are used, the default becomes `robust` or `svy`.

As an aside, if you set `moptimize_init_vcetype()` during initialization, that changes the default.

`moptimize_init_vcetype(`*M*`,` *vcetype*`)`, *vcetype* being a string scalar, resets the default *vcetype*.

To post results to Stata, use `moptimize_result_post()`.

`moptimize_result_post(`*M*`)` posts estimation results to Stata where they can be displayed with Mata function `moptimize_result_post()` (without arguments) or with Stata command `ereturn display` (see [P] **ereturn**). The posted VCE will be of the default *vcetype*.

`moptimize_result_post(`*M*`,` *vcetype*`)` does the same thing, except the VCE will be of the specified *vcetype*.

The remaining `moptimize_result_*()` functions simply return the requested result. It does not matter whether results have been posted or previously displayed.

`moptimize_result_value(`*M*`)` returns the real scalar value of the objective function.

`moptimize_result_value0(`*M*`)` returns the real scalar value of the objective function at the starting values.

`moptimize_result_eq_coefs(`*M* $\left[\text{, } i\right]$`)` returns the $1 \times (ki + ci)$ coefficient rowvector for the *i*th equation. If $i \geq$ `.` or argument *i* is omitted, the $1 \times K$ full set of coefficients is returned.

`moptimize_result_coefs(`*M*`)` returns the $1 \times K$ full set of coefficients.

`moptimize_result_colstripe(`*M* $\left[\text{, } i\right]$`)` returns a $(ki + ci) \times 2$ string matrix containing, for the *i*th equation, the equation names in the first column and the coefficient names in the second. If $i \geq$ `.` or argument *i* is omitted, the result is $K \times 2$.

`moptimize_result_scores(`*M*`)` returns an $N \times m$ (evaluator types `lf` and `lf*`), or an $N \times K$ (evaluator type `gf`), or an $L \times K$ (evaluator type `q`) real matrix containing the observation-by-observation scores. For all other evaluator types, `J(0,0,.)` is returned. For evaluator types `lf` and `lf*`, scores are defined as the derivative of the objective function with respect to the parameters. For evaluator type `gf`, scores are defined as the derivative of the objective function with respect to the coefficients. For evaluator type `q`, scores are defined as the derivatives of the *L* independent elements with respect to the coefficients.

`moptimize_result_gradient(`*M* $\left[\text{, } i\right]$`)` returns the $1 \times (ki + ci)$ gradient rowvector for the *i*th equation. If $i \geq$ `.` or argument *i* is omitted, the $1 \times K$ gradient corresponding to the full set of coefficients is returned. Gradient is defined as the derivative of the objective function with respect to the coefficients.

`moptimize_result_Hessian(`*M* [`,` *i*]`)` returns the $(ki + ci) \times (ki + ci)$ Hessian matrix for the *i*th equation. If $i \geq$ `.` or argument *i* is omitted, the $K \times K$ Hessian corresponding to the full set of coefficients is returned. The Hessian is defined as the second derivative of the objective function with respect to the coefficients.

`moptimize_result_V(`*M* [`,` *i*]`)` returns the appropriate $(ki+ci) \times (ki+ci)$ submatrix of the full variance matrix calculated according to the default *vcetype*. If $i \geq$ `.` or argument *i* is omitted, the full $K \times K$ variance matrix corresponding to the full set of coefficients is returned.

`moptimize_result_Vtype(`*M*`)` returns a string scalar containing the default *vcetype*.

`moptimize_result_V_oim(`*M* [`,` *i*]`)` returns the appropriate $(ki + ci) \times (ki + ci)$ submatrix of the full variance matrix calculated as the inverse of the negative Hessian matrix (the observed information matrix). If $i \geq$ `.` or argument *i* is omitted, the full $K \times K$ variance matrix corresponding to the full set of coefficients is returned.

`moptimize_result_V_opg(`*M* [`,` *i*]`)` returns the appropriate $(ki + ci) \times (ki + ci)$ submatrix of the full variance matrix calculated as the inverse of the outer product of the gradients. If $i \geq$ `.` or argument *i* is omitted, the full $K \times K$ variance matrix corresponding to the full set of coefficients is returned. If `moptimize_result_V_opg()` is used with evaluator types other than `lf`, `lf*`, `gf`, or `q`, an appropriately dimensioned matrix of zeros is returned.

`moptimize_result_V_robust(`*M* [`,` *i*]`)` returns the appropriate $(ki+ci) \times (ki+ci)$ submatrix of the full variance matrix calculated via the sandwich estimator. If $i \geq$ `.` or argument *i* is omitted, the full $K \times K$ variance matrix corresponding to the full set of coefficients is returned. If `moptimize_result_V_robust()` is used with evaluator types other than `lf`, `lf*`, `gf`, or `q`, an appropriately dimensioned matrix of zeros is returned.

`moptimize_result_iterations(`*M*`)` returns a real scalar containing the number of iterations performed.

`moptimize_result_converged(`*M*`)` returns a real scalar containing 1 if convergence was achieved and 0 otherwise.

`moptimize_result_iterationlog(`*M*`)` returns a real colvector containing the values of the objective function at the end of each iteration. Up to the last 20 iterations are returned, one to a row.

`moptimize_result_errorcode(`*M*`)` returns the real scalar containing the error code from the most recently run optimization or function evaluation. The error code is 0 if there are no errors. This function is useful only after `_moptimize()` or `_moptimize_evaluate()` because the nonunderscore versions aborted with error if there were problems.

`moptimize_result_errortext(`*M*`)` returns a string scalar containing the error text corresponding to `moptimize_result_errorcode()`.

`moptimize_result_returncode(`*M*`)` returns a real scalar containing the Stata return code corresponding to `moptimize_result_errorcode()`.

The following error codes and their corresponding Stata return codes are for `moptimize()` only. To see other error codes and their corresponding Stata return codes, see [M-5] **optimize()**.

Error code	Return code	Error text
400	1400	could not find feasible values
401	491	Stata program evaluator returned an error
402	198	views are required when the evaluator is a Stata program
403	198	Stata program evaluators require a touse variable

Stata evaluators

The following function is useful only when your evaluator is a Stata program instead of a Mata function.

`moptimize_ado_cleanup(`*M*`)` removes all the global macros with the `ML_` prefix. A temporary weight variable is also dropped if weights were specified.

Advanced functions

These functions are not really advanced, they are just seldomly used.

`moptimize_init_verbose(`*M*`, {"on"|"off"})` specifies whether error messages are to be displayed. The default is `"on"`.

`moptimize_init_evaluations(`*M*`, {"off"|"on"})` specifies whether the system is to count the number of times the evaluator is called. The default is `"off"`.

`moptimize_result_evaluations(`*M*`)` returns a 1×3 real rowvector containing the number of times the evaluator was called, assuming `moptimize_init_evaluations()` was set on. Contents are the number of times called for the purposes of 1) calculating the objective function, 2) calculating the objective function and its first derivative, and 3) calculating the objective function and its first and second derivatives. If `moptimize_init_evaluations()` was set off, returned is (0,0,0).

Syntax of evaluators

An *evaluator* is a program you write that calculates the value of the function being optimized and optionally calculates the function's first and second derivatives. The evaluator you write is called by the `moptimize()` functions.

There are five styles in which the evaluator can be written, known as types `lf`, `d`, `lf*`, `gf`, and `q`. *evaluatortype*, optionally specified in `moptimize_init_evaluatortype()`, specifies the style in which the evaluator is written. Allowed values are

(*Continued on next page*)

evaluatortype	description
`"lf"`	*function*`()` returns $N \times 1$ colvector value
`"d0"`	*function*`()` returns scalar value
`"d1"`	same as `"d0"` and returns gradient rowvector
`"d2"`	same as `"d1"` and returns Hessian matrix
`"d1debug"`	same as `"d1"` but checks gradient
`"d2debug"`	same as `"d2"` but checks gradient and Hessian
`"lf0"`	*function*`()` returns $N \times 1$ colvector value
`"lf1"`	same as `"lf0"` and return equation-level score matrix
`"lf2"`	same as `"lf1"` and returns Hessian matrix
`"lf1debug"`	same as `"lf1"` but checks gradient
`"lf2debug"`	same as `"lf2"` but checks gradient and Hessian
`"gf0"`	*function*`()` returns $N \times 1$ colvector value
`"gf1"`	same as `"gf0"` and returns score matrix
`"gf2"`	same as `"gf1"` and returns Hessian matrix
`"gf1debug"`	same as `"gf1"` but checks gradient
`"gf2debug"`	same as `"gf2"` but checks gradient and Hessian
`"q0"`	*function*`()` returns colvector value
`"q1"`	same as `"q0"` and returns score matrix
`"q1debug"`	same as `"q1"` but checks gradient

The default is `"lf"` if not set.
`"q"` evaluators are used with technique `"gn"`.
Returned gradients are $1 \times K$ rowvectors.
Returned Hessians are $K \times K$ matrices.

Examples of each of the evaluator types are outlined below.

You must tell `moptimize()` the identity and type of your evaluator, which you do by using the `moptimize_init_evaluator()` and `moptimize_init_evaluatortype()` functions.

`moptimize_init_evaluator(`*M*`, &`*functionname*`())` sets the identity of the evaluator function that you write in Mata.

`moptimize_init_evaluator(`*M*`, "`*programname*`")` sets the identity of the evaluator program that you write in Stata.

`moptimize_init_evaluatortype(`*M*`,` *evaluatortype*`)` informs `moptimize()` of the style of evaluator you have written. *evaluatortype* is a string scalar from the table above. The default is `"lf"`.

`moptimize_init_negH(`*M*`,` { `"off"` | `"on"` }`)` sets whether the evaluator you have written returns H or $-H$, the Hessian or the negative of the Hessian, if it returns a Hessian at all. This is for backward compatibility with prior versions of Stata's `ml` command (see [R] **ml**). Modern evaluators return H. The default is `"off"`.

Syntax of type lf evaluators

`lfeval(`*M*`,` *b*`,` $\overline{fv}$`):`

inputs:

M:	problem definition
b:	coefficient vector

outputs:

fv:	$N \times 1$, $N = \#$ of observations

Notes:

1. The objective function is $f() =$ `colsum(`*fv*`)`.
2. In the case where $f()$ is a log-likelihood function, the values of the log likelihood must be summable over the observations.
3. For use with any technique except `gn`.
4. May be used with robust, clustering, and survey.
5. Returns *fv* containing missing (*fv* = `.`) if evaluation is not possible.

Syntax of type d evaluators

`deval(`*M*`, todo,` *b*`,` $\overline{fv}$`,` $\overline{g}$`,` $\overline{H}$`):`

inputs:

M:	problem definition
todo:	real scalar containing 0, 1, or 2
b:	coefficient vector

outputs:

fv:	real scalar
g:	$1 \times K$, gradients, $K = \#$ of coefficients
H:	$K \times K$, Hessian

Notes:

1. The objective function is $f() = fv$.
2. For use with any log-likelihood function, or any function.
3. For use with any technique except `gn` and `bhhh`.
4. Cannot be used with robust, clustering, or survey.
5. *deval*`()` must always fill in *fv*, and fill in *g* if *todo* ≥ 1, and fill in *H* if *todo* $= 2$. For type `d0`, *todo* will always be 0. For type `d1` and `d1debug`, *todo* will be 0 or 1. For type `d2` and `d2debug`, *todo* will be 0, 1, or 2.
6. Returns *fv* = `.` if evaluation is not possible.

Syntax of type lf* evaluators

`lfeval(`*M*`,` *todo*`,` *b*`,` $\overline{fv}$`,` $\overline{S}$`,` $\overline{H}$`):`

inputs:

M:	problem definition
todo:	real scalar containing 0, 1, or 2
b:	coefficient vector

outputs:

fv:	$N \times 1$, $N = \#$ of observations
S:	$N \times m$, scores, $m = \#$ of equations (parameters)
H:	$K \times K$, Hessian, $K = \#$ of coefficients

Notes:

1. The objective function is $f() =$ `colsum(`*fv*`)`.
2. Type `lf*` is a variation of type `lf` that allows the user to supply analytic derivatives. Although `lf*` could be used with an arbitrary function, it is intended for use when $f()$ is a log-likelihood function and the log-likelihood values are summable over the observations.
3. For use with any technique except `gn`.
4. May be used with robust, clustering, and survey.
5. Always returns *fv*, returns *S* if *todo* ≥ 1, and returns *H* if *todo* $= 2$. For type `lf0`, *todo* will always be 0. For type `lf1` and `lf1debug`, *todo* will be 0 or 1. For type `lf2` and `lf2debug`, *todo* will be 0, 1, or 2.
6. Returns *fv* containing missing (*fv* = `.`) if evaluation is not possible.

Syntax of type gf evaluators

`gfeval(`*M*`,` *todo*`,` *b*`,` $\overline{fv}$`,` $\overline{S}$`,` $\overline{H}$`):`

inputs:

M:	problem definition
todo:	real scalar containing 0, 1, or 2
b:	coefficient vector

outputs:

fv:	$L \times 1$, values, $L = \#$ of independent elements
S:	$L \times K$, scores, $K = \#$ of coefficients
H:	$K \times K$, Hessian

Notes:

1. The objective function is $f() =$ `colsum(`*fv*`)`.
2. Type `gf` is a variation on type `lf*` that relaxes the requirement that the log-likelihood function be summable over the observations. `gf` is especially useful for fitting panel-data models with technique `bhhh`. Then L is the number of panels.
3. For use with any technique except `gn`.
4. May be used with robust, clustering, and survey.
5. Always returns *fv*, returns S if *todo* ≥ 1, and returns H if *todo* $= 2$. For type `gf0`, *todo* will always be 0. For type `gf1` and `gf1debug`, *todo* will be 0 or 1. For type `gf2` and `gf2debug`, *todo* will be 0, 1, or 2.
6. Returns *fv* = `.` if evaluation is not possible.

Syntax of type q evaluators

```
qeval(M, todo, b, r̄, S̄)
```

inputs:

M:	problem definition
todo:	real scalar containing 0 or 1
b:	coefficient vector

outputs:

r:	$L \times 1$ of independent elements
S:	$L \times m$, scores, $m = \#$ of parameters

Notes:

1. Type `q` is for quadratic optimization. The objective function is $f() = r'Wr$, where r is returned by *qeval*`()` and W has been previously set by using `moptimize_init_gnweightmatrix()`, described below.
2. For use only with techniques `gn` and `nm`.
3. Always returns r and returns S if *todo* $= 1$. For type `q0`, *todo* will always be 0. For type `q1` and `q1debug`, *todo* will be 0 or 1. There is no type `q2`.
4. Returns r containing missing, or $r =$ `.` if evaluation is not possible.

Use `moptimize_init_gnweightmatrix()` during initialization to set matrix W.

`moptimize_init_gnweightmatrix(`*M*`,` *W*`)` sets real matrix W: $L \times L$, which is used only by type `q` evaluators. The objective function is $r'Wr$. If W is not set and if observation weights w are set by using `moptimize_init_weight()`, then $W = \text{diag}(w)$. If w is not set, then W is the identity matrix.

`moptimize()` does not produce a robust VCE when you set W with `moptimize_init_gnweight()`.

Passing extra information to evaluators

In addition to the arguments the evaluator receives, you may arrange that extra information be sent to the evaluator. Specify the extra information to be sent by using `moptimize_init_userinfo()`.

`moptimize_init_userinfo(`*M*`,` *l*`,` *Z*`)` specifies that the *l*th piece of extra information is *Z*. *l* is a real scalar. The first piece of extra information should be 1; the second piece, 2; and so on. *Z* can be anything. No copy of *Z* is made.

`moptimize_init_nuserinfo(`*M*`,` *n_user*`)` specifies the total number of extra pieces of information to be sent. Setting *n_user* is optional; it will be automatically determined from the `moptimize_init_userinfo()` calls you issue.

Inside your evaluator, you access the information by using `moptimize_util_userinfo()`.

`moptimize_util_userinfo(`*M*`,` *l*`)` returns the *Z* set by `moptimize_init_userinfo()`.

Utility functions

There are various utility functions that are helpful in writing evaluators and in processing results returned by the `moptimize_result_*()` functions.

The first set of utility functions are useful in writing evaluators, and the first set return results that all evaluators need.

`moptimize_util_depvar(`*M*`,` *j*`)` returns an $Nj \times 1$ colvector containing the values of the *j*th dependent variable, the values set by `moptimize_init_depvar(`*M*`,` *j*`, ...)`.

`moptimize_util_xb(`*M*`,` *b*`,` *i*`)` returns the $Ni \times 1$ colvector containing the value of the *i*th parameter, which is usually $Xi \times bi'$ `:+` *b0i*, but might be as complicated as $Xi \times bi' + oi + \ln(ti)$ `:+` *b0i*.

Once the inputs of an evaluator have been processed, the following functions assist in making the calculations required of evaluators.

`moptimize_util_sum(`*M*`,` *v*`)` returns the "sum" of colvector *v*. This function is for use in evaluators that require you to return an overall objective function value rather than observation-by-observation results. Usually, `moptimize_util_sum()` returns `sum(`*v*`)`, but in cases where you have specified a weight by using `moptimize_init_weight()` or there is an implied weight due to use of `moptimize_init_svy()`, the appropriately weighted sum is returned. Use `moptimize_util_sum()` to sum log-likelihood values.

`moptimize_util_vecsum(`*M*`,` *i*`,` *s*`,` *value*`)` is like `moptimize_util_sum()`, but for use with gradients. The gradient is defined as the vector of partial derivatives of $f()$ with respect to the coefficients *bi*. Some evaluator types require that your evaluator be able to return this vector. Nonetheless, it is usually easier to write your evaluator in terms of parameters rather than coefficients, and this function handles the mapping of parameter gradients to the required coefficient gradients.

Input *s* is an $Ni \times 1$ colvector containing df/dpi for each observation. df/dpi is the partial derivative of the objective function, but with respect to the *i*th parameter rather than the *i*th set of coefficients. `moptimize_util_vecsum()` takes *s* and returns the $1 \times (ki + ci)$ summed gradient. Also weights, if any, are factored into the calculation.

If you have more than one equation, you will need to call `moptimize_util_vecsum()` *m* times, once for each equation, and then concatenate the individual results into one vector.

value plays no role in `moptimize_util_vecsum()`'s calculations. *value*, however, should be specified as the result obtained from `moptimize_util_sum()`. If that is inconvenient, make *value* any nonmissing value. If the calculation from parameter space to vector space cannot be performed, or if your original parameter space derivatives have any missing values, *value* will be changed to missing. Remember, when a calculation cannot be made, the evaluator is to return a missing value for the objective function. Thus storing the value of the objective function in *value* ensures that your evaluator will return missing if it is supposed to.

`moptimize_util_matsum(`*M*`,` *i*`,` *i2*`,` *s*`,` *value*`)` is similar to `moptimize_util_vecsum()`, but for Hessians (matrix of second derivatives).

Input *s* is an *Ni* × 1 colvector containing $d^2f/dpidpi2$ for each observation. `moptimize_util_matsum()` returns the $(ki + ci) \times (ki2 + ci2)$ summed Hessian. Also weights, if any, are factored into the calculation.

If you have $m > 1$ equations, you will need to call `moptimize_util_matsum()` $m \times (m+1)/2$ times and then join the results into one symmetric matrix.

value plays no role in the calculation and works the same way it does in `moptimize_util_vecsum()`.

`moptimize_util_matbysum()` is an added helper for making `moptimize_util_matsum()` calculations in cases where you have panel data and the log-likelihood function's values exists only at the panel level. `moptimize_util_matbysum(`*M*`,` *i*`,` *a*`,` *b*`,` *value*`)` is for making diagonal calculations and `moptimize_util_matbysum(`*M*`,` *i*`,` *i2*`,` *a*`,` *b*`,` *c*`,` *value*`)` is for making off-diagonal calculations.

This is an advanced topic; see Gould, Pitblado, and Sribney (2006, 117–119) for a full description of it. In applying the chain rule to translate results from parameter space to coefficient space, `moptimize_util_matsum()` can be used to make some of the calculations, and `moptimize_util_matbysum()` can be used to make the rest. *value* plays no role and works just as it did in the other helper functions. `moptimize_util_matbysum()` is for use sometimes when *by* has been set, which is done via `moptimize_init_by(`*M*`,` *by*`)`. `moptimize_util_matbysum()` is never required unless *by* has been set.

The formula implemented in `moptimize_util_matbysum(`*M*`,` *i*`,` *a*`,` *b*`,` *value*`)` is

$$\sum_{j=1}^{N}\left(\sum_{t=1}^{T_j} a_{jt}\right)\left(\sum_{t=1}^{T_j} b_{jt}\mathbf{x}'_{1jt}\right)\left(\sum_{t=1}^{T_j} b_{jt}\mathbf{x}_{1jt}\right)$$

The formula implemented in `moptimize_util_matbysum(`*M*`,` *i*`,` *i2*`,` *a*`,` *b*`,` *c*`,` *value*`)` is

$$\sum_{j=1}^{N}\left(\sum_{t=1}^{T_j} a_{jt}\right)\left(\sum_{t=1}^{T_j} b_{jt}\mathbf{x}'_{1jt}\right)\left(\sum_{t=1}^{T_j} c_{jt}\mathbf{x}_{2jt}\right)$$

`moptimize_util_by()` returns a pointer to the vector of group identifiers that were set using `moptimize_init_by()`. This vector can be used with `panelsetup()` to perform panel level calculations.

The other utility functions are useful inside or outside of evaluators. One of the more useful is `moptimize_util_eq_indices()`, which allows two or three arguments.

`moptimize_util_eq_indices(`*M*`,` *i*`)` returns a 1×2 vector that can be used with range subscripts to extract the portion relevant for the *i*th equation from any $1 \times K$ vector, that is, from any vector conformable with the full coefficient vector.

`moptimize_util_eq_indices(`*M*`,` *i*`,` *i2*`)` returns a 2×2 matrix that can be used with range subscripts to exact the portion relevant for the *i*th and *i2*th equations from any $K \times K$ matrix, that is, from any matrix with rows and columns conformable with the full variance matrix.

For instance, let `b` be the $1 \times K$ full coefficient vector, perhaps obtained by being passed into an evaluator, or perhaps obtained from `b` = `moptimize_result_coefs(`*M*`)`. Then `b[|moptimize_util_eq_indices(`*M*`,` *i*`)|]` is the $1 \times (ki + ci)$ vector of coefficients for the *i*th equation.

Let `V` be the $K \times K$ full variance matrix obtained by `V` = `moptimize_result_V(`*M*`)`. Then `V[|moptimize_util_eq_indices(`*M*`,` *i*`,` *i*`)|]` is the $(ki + ci) \times (ki + ci)$ variance matrix for the *i*th equation. `V[|moptimize_util_eq_indices(`*M*`,` *i*`,` *j*`)|]` is the $(ki + ci) \times (kj + cj)$ covariance matrix between the *i*th and *j*th equations.

Finally, there is one more utility function that may help when you become confused: `mopti-mize_query()`.

`moptimize_query(`*M*`)` displays in readable form everything you have set via the `mopti-mize_init_*()` functions, along with the status of the system.

Description

The `moptimize()` functions find coefficients $(\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_m)$ that maximize or minimize $f(\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_m)$, where $\mathbf{p}_i = \mathbf{X}_i \times \mathbf{b}'_i$, a linear combination of $\mathbf{b}_i$ and the data. The user of `moptimize()` writes a Mata function or Stata program to evaluate $f(\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_m)$. The data can be in Mata matrices or in the Stata dataset currently residing in memory.

`moptimize()` is especially useful for obtaining solutions for maximum likelihood models, minimum chi-squared models, minimum squared-residual models, and the like.

Remarks

Remarks are presented under the following headings:

Relationship of moptimize( ) to Stata's ml and to Mata's optimize( )

`moptimize()` is Mata's and Stata's premier optimization routine. This is the routine used by most of the official optimization-based estimators implemented in Stata.

That said, Stata's `ml` command—see [R] **ml**—provides most of the capabilities of Mata's `moptimize()`, and `ml` is easier to use. In fact, `ml` uses `moptimize()` to perform the optimization, and `ml` amounts to little more than a shell providing a friendlier interface. If you have a maximum likelihood model you wish to fit, we recommend you use `ml` instead of `moptimize()`. Use `moptimize()` when you need or want to work in the Mata environment, or when you wish to implement a specialized system for fitting a class of models.

Also make note of Mata's `optimize()` function; see [M-5] **optimize( )**. `moptimize()` finds coefficients $(\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_m)$ that maximize or minimize $f(\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_m)$, where $\mathbf{p}_i = \mathbf{X}_i \times \mathbf{b}_i$. `optimize()` handles a simplified form of the problem, namely, finding constant $(p_1, p_2, \ldots, p_m)$ that maximizes or minimizes $f()$. `moptimize()` is the appropriate routine for fitting a Weibull model, but if all you need to estimate are the fixed parameters of the Weibull distribution for some population, `moptimize()` is overkill and `optimize()` will prove easier to use.

These three routines are all related. Stata's `ml` uses `moptimize()` to do the numerical work. `moptimize()`, in turn, uses `optimize()` to perform certain calculations, including the search for parameters. There is nothing inferior about `optimize()` except that it cannot efficiently deal with models in which parameters are given by linear combinations of coefficients and data.

Mathematical statement of the moptimize( ) problem

We mathematically describe the problem `moptimize()` solves not merely to fix notation and ease communication, but also because there is a one-to-one correspondence between the mathematical notation and the `moptimize*()` functions. Simply writing your problem in the following notation makes obvious the `moptimize*()` functions you need and what their arguments should be.

In what follows, we are going to simplify the mathematics a little. For instance, we are about to claim $\mathbf{p}_i = \mathbf{X}_i \times \mathbf{b}_i$ `:+` c_i, when in the syntax section, you will see that $\mathbf{p}_i = \mathbf{X}_i \times \mathbf{b}_i + \mathbf{o}_i + \ln(\mathbf{t}_i)$ `:+` c_i. Here we omit $\mathbf{o}_i$ and $\ln(\mathbf{t}_i)$ because they are seldom used. We will omit some other details, too. The statement of the problem under *Syntax*, above, is the full and accurate statement. We will also use typefaces a little differently. In the syntax section, we use italics following programming convention. In what follows, we will use boldface for matrices and vectors, and italics for scalars so that you can follow the math more easily. So in this section, we will write $\mathbf{b}_i$, whereas under syntax we would write *bi*; regardless of typeface, they mean the same thing.

Function `moptimize()` finds coefficients

$$\mathbf{b} = ((\mathbf{b}_1, c_1), (\mathbf{b}_2, c_2), \ldots, (\mathbf{b}_m, c_m))$$

where

$$\begin{array}{llll} \mathbf{b}_1\text{: } 1 \times k_1, & \mathbf{b}_2\text{: } 1 \times k_2, & \ldots, & \mathbf{b}_m\text{: } 1 \times k_m \\ c_1\text{: } 1 \times 1, & c_2\text{: } 1 \times 1, & \ldots, & c_m\text{: } 1 \times 1 \end{array}$$

that maximize or minimize function

$$f(\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_m;\;\; \mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_D)$$

where

$$\begin{array}{ll}
\mathbf{p}_1 = \mathbf{X}_1 \times \mathbf{b}_1' \texttt{ :+ } c_1, & \mathbf{X}_1 : N_1 \times k_1 \\
\mathbf{p}_2 = \mathbf{X}_2 \times \mathbf{b}_2' \texttt{ :+ } c_2, & \mathbf{X}_2 : N_2 \times k_2 \\
. & \\
. & \\
\mathbf{p}_m = \mathbf{X}_m \times \mathbf{b}_m' \texttt{ :+ } c_m, & \mathbf{X}_m : N_m \times k_m
\end{array}$$

and where $\mathbf{y}_1$, $\mathbf{y}_2$, …, $\mathbf{y}_D$ are of arbitrary dimension.

Usually, $N_1 = N_2 = \cdots = N_m$, and the model is said to be fit on data of *N* observations. Similarly, column vectors $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_D$ are usually called dependent variables, and each is also of *N* observations.

As an example, let's write the maximum likelihood estimator for linear regression in the above notation. We begin by stating the problem in the usual way, but in Mata-ish notation:

Given data $\mathbf{y}$: $N \times 1$ and $\mathbf{X}$: $N \times k$, obtain $((\mathbf{b}, c), s^2)$ to fit

$$\mathbf{y} = \mathbf{X} \times \mathbf{b}' \texttt{ :+ } c + \mathbf{u}$$

where the elements of **u** are distributed $N(0, s^2)$. The log-likelihood function is

$$\ln L = \sum_j \texttt{ln(normalden(}\mathbf{y}_j - (\mathbf{X}_j \times \mathbf{b}' \texttt{ :+ } c)\texttt{, 0, sqrt(}s^2\texttt{)))}$$

where `normalden(`x`,` *mean*`,` *sd*`)` returns the density at *x* of the Gaussian normal with the specified mean and standard deviation; see [M-5] **normal()**.

The above is a two-parameter or, equivalently, two-equation model in `moptimize()` jargon. There may be many coefficients, but the likelihood function can be written in terms of two parameters, namely $\mathbf{p}_1 = \mathbf{X} \times \mathbf{b}' \texttt{ :+ } c$ and $\mathbf{p}_2 = s^2$. Here is the problem stated in the `moptimize()` notation:

Find coefficients

$$\mathbf{b} = ((\mathbf{b}_1, c_1), (c_2))$$

where

$$\begin{array}{ll}
\mathbf{b}_1\text{: } 1 \times k & \\
c_1\text{: } 1 \times 1, & c_2\text{: } 1 \times 1
\end{array}$$

that maximize

$$f(\mathbf{p}_1, \mathbf{p}_2;\ \mathbf{y}) = \sum \texttt{ln(normalden(}\mathbf{y} - \mathbf{p}_1\texttt{,0,sqrt(}\mathbf{p}_2\texttt{))}$$

where

$$\begin{array}{ll}
\mathbf{p}_1 = \mathbf{X} \times \mathbf{b}_1' \texttt{ :+ } c_1, & \mathbf{X} : N \times k \\
\mathbf{p}_2 = c_2 &
\end{array}$$

and where *y* is $N \times 1$.

Notice that, in this notation, the regression coefficients $(\mathbf{b}_1, c_1)$ play a secondary role, namely, to determine $\mathbf{p}_1$. That is, the function, $f()$, to be optimized—a log-likelihood function here—is written in terms of $\mathbf{p}_1$ and $\mathbf{p}_2$. The program you will write to evaluate $f()$ will be written in terms of $\mathbf{p}_1$ and $\mathbf{p}_2$, thus abstracting from the particular regression model being fit. Whether the regression is mpg on weight or log income on age, education, and experience, your program to calculate $f()$ will remain unchanged. All that will change are the definitions of $\mathbf{y}$ and $\mathbf{X}$, which you will communicate to `moptimize()` separately.

There is another advantage to this arrangement. We can trivially generalize linear regression without writing new code. Note that the variance s^2 is given by $\mathbf{p}_2$, and currently, we have $\mathbf{p}_2 = c_2$, that is, a constant. `moptimize()` allows parameters to be constant, but it equally allows them to be given by a linear combination. Thus rather than defining $\mathbf{p}_2 = c_2$, we could define $\mathbf{p}_2 = \mathbf{X}_2 \times \mathbf{b}_2' \texttt{ :+ } c_2$. If we did that, we would have a second linear equation that allowed the variance to vary observation by observation. As far as `moptimize()` is concerned, that problem is the same as the original problem.

Filling in moptimize() from the mathematical statement

The mathematical statement of our sample problem is the following:

Find coefficients

$$\mathbf{b} = ((\mathbf{b}_1, c_1), (c_2))$$

$$\begin{aligned} \mathbf{b}_1&: 1 \times k \\ c_1&: 1 \times 1, \qquad\qquad c_2 : 1 \times 1 \end{aligned}$$

that maximize

$$f(\mathbf{p}_1, \mathbf{p}_2; \mathbf{y}) = \sum \texttt{ln(normalden(}\mathbf{y} - \mathbf{p}_1\texttt{,0,sqrt(}\mathbf{p}_2\texttt{))}$$

where

$$\begin{aligned} \mathbf{p}_1 &= \mathbf{X} \times \mathbf{b}_1' \texttt{ :+ } c_1, \qquad\qquad \mathbf{X} : N \times k \\ \mathbf{p}_2 &= c_2 \end{aligned}$$

and where y is $N \times 1$.

The corresponding code to perform the optimization is

```
. sysuse auto
. mata:
: function linregeval(transmorphic M, real rowvector b,
                      real colvector lnf)
{
          real colvector  p1, p2
          real colvector  y1
          p1 = moptimize_util_xb(M, b, 1)
          p2 = moptimize_util_xb(M, b, 2)
          y1 = moptimize_util_depvar(M, 1)
          lnf = ln(normalden(y1:-p1, 0, sqrt(p2)))
}
: M = moptimize_init()
: moptimize_init_evaluator(M, &linregeval())
: moptimize_init_depvar(M, 1, "mpg")
: moptimize_init_eq_indepvars(M, 1, "weight foreign")
: moptimize_init_eq_indepvars(M, 2, "")
: moptimize(M)
: moptimize_result_display(M)
```

Here is the result of running the above code:

```
. sysuse auto
(1978 Automobile Data)
. mata:
------------------------------------------------- mata (type end to exit) -----
: function linregeval(transmorphic M, real rowvector b, real colvector lnf)
> {
>          real colvector  p1, p2
>          real colvector  y1
>
>          p1 = moptimize_util_xb(M, b, 1)
>          p2 = moptimize_util_xb(M, b, 2)
>          y1 = moptimize_util_depvar(M, 1)
>
>          lnf = ln(normalden(y1:-p1, 0, sqrt(p2)))
> }
: M = moptimize_init()
: moptimize_init_evaluator(M, &linregeval())
: moptimize_init_depvar(M, 1, "mpg")
: moptimize_init_eq_indepvars(M, 1, "weight foreign")
: moptimize_init_eq_indepvars(M, 2, "")
: moptimize(M)
initial:       f(p) =     -<inf>  (could not be evaluated)
feasible:      f(p) = -12949.708
rescale:       f(p) = -243.04355
rescale eq:    f(p) = -236.58999
Iteration 0:   f(p) = -236.58999  (not concave)
Iteration 1:   f(p) = -227.46735
Iteration 2:   f(p) = -205.73496  (backed up)
Iteration 3:   f(p) = -195.72762
Iteration 4:   f(p) = -194.20885
Iteration 5:   f(p) = -194.18313
Iteration 6:   f(p) = -194.18306
Iteration 7:   f(p) = -194.18306
: moptimize_result_display(M)
                                                Number of obs   =         74

------------------------------------------------------------------------------
         mpg |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
eq1          |
      weight |  -.0065879   .0006241   -10.56   0.000     -.007811   -.0053647
     foreign |  -1.650029   1.053958    -1.57   0.117    -3.715749    .4156903
       _cons |    41.6797   2.121197    19.65   0.000     37.52223    45.83717
-------------+----------------------------------------------------------------
eq2          |
       _cons |   11.13746   1.830987     6.08   0.000      7.54879    14.72613
------------------------------------------------------------------------------
```

The type lf evaluator

Let's now interpret the code we wrote, which was

```
: function linregeval(transmorphic M, real rowvector b,
                      real colvector lnf)
{
        real colvector  p1, p2
        real colvector  y1

        p1 = moptimize_util_xb(M, b, 1)
        p2 = moptimize_util_xb(M, b, 2)
        y1 = moptimize_util_depvar(M, 1)

        lnf = ln(normalden(y1:-p1, 0, sqrt(p2)))
}
: M = moptimize_init()
: moptimize_init_evaluator(M, &linregeval())
: moptimize_init_depvar(M, 1, "mpg")
: moptimize_init_eq_indepvars(M, 1, "weight foreign")
: moptimize_init_eq_indepvars(M, 2, "")
: moptimize(M)
: moptimize_result_display(M)
```

We first defined the function to evaluate our likelihood function—we named the function `linregeval()`. The name was of our choosing. After that, we began an optimization problem by typing `M = moptimize_init()`, described the problem with `moptimize_init_*()` functions, performed the optimization by typing `moptimize()`, and displayed results by using `moptimize_result_display()`.

Function linregeval() is an example of a type lf evaluator. There are several different evaluator types, including d0, d1, d2, through q1. Of all of them, type lf is the easiest to use and is the one moptimize() uses unless we tell it differently. What makes lf easy is that we need only calculate the likelihood function; we are not required to calculate its derivatives. A description of lf appears under the heading *Syntax of type lf evaluators* under *Syntax* above.

In the syntax diagrams, you will see that type lf evaluators receive three arguments, *M*, *b*, and *fv*, although in linregeval(), we decided to call them M, b, and lnf. The first two arguments are inputs, and your evaluator is expected to fill in the third argument with observation-by-observation values of the log-likelihood function.

The input arguments are M and b. M is the problem handle, which we have not explained yet. Basically, all evaluators receive M as the first argument and are expected to pass M to any moptimize*() subroutines that they call. M in fact contains all the details of the optimization problem. The second argument, b, is the entire coefficient vector, which in the linregeval() case will be all the coefficients of our regression, the constant (intercept), and the variance of the residual. Those details are unimportant. Instead, your evaluator will pass *M* and *b* to moptimize() utility programs that will give you back what you need.

Using one of those utilities is the first action our linregeval() evaluator performs:

```
p1 = moptimize_util_xb(M, b, 1)
```

That returns observation-by-observation values of the first parameter, namely, $X \times b_1$:+ c_1. moptimize_util_xb(x, b, 1) returns the first parameter because the last argument specified is 1. We obtained the second parameter similarly:

```
p2 = moptimize_util_xb(M, b, 2)
```

To evaluate the likelihood function, we also need the dependent variable. Another moptimize*() utility returns that to us:

```
y1 = moptimize_util_depvar(M, 1)
```

Having p1, p2, and y1, we are ready to fill in the log-likelihood values:

```
lnf = ln(normalden(y1:-p1, 0, sqrt(p2)))
```

For a type lf evaluator, you are to return observation-by-observation values of the log-likelihood function; moptimize() itself will sum them to obtain the overall log likelihood. That is exactly what the line lnf = ln(normalden(y1:-p1, 0, sqrt(p2))) did. Note that y1 is $N \times 1$, p1 is $N \times 1$, and p2 is $N \times 1$, so the lnf result we calculate is also $N \times 1$. Some of the other evaluator types are expected to return a scalar equal to the overall value of the function.

With the evaluator defined, we can estimate a linear regression by typing

```
: M = moptimize_init()
: moptimize_init_evaluator(M, &linregeval())
: moptimize_init_depvar(M, 1, "mpg")
: moptimize_init_eq_indepvars(M, 1, "weight foreign")
: moptimize_init_eq_indepvars(M, 2, "")
: moptimize(M)
: moptimize_result_display(M)
```

All estimation problems begin with

```
        M = moptimize_init()
```

The returned value M is called a problem handle, and from that point on, you will pass M to every other moptimize() function you call. M contains the details of your problem. If you were to list *M*, you would see something like

```
        : M
          0x15369a
```

0×15369a is in fact the address where all those details are stored. Exactly how M works does not matter, but it is important that you understand what M is. M is your problem. In a more complicated problem, you might need to perform nested optimizations. You might have one optimization problem, and right in the middle of it, even right in the middle of evaluating its log-likelihood function, you might need to set up and solve another optimization problem. You can do that. The first problem you would set up as M1 = moptimize_init(). The second problem you would set up as M2 = moptimize_init(). moptimize() would not confuse the two problems because it would know to which problem you were referring by whether you used M1 or M2 as the argument of the moptimize() functions. As another example, you might have one optimization problem, M = moptimize_init(), and halfway through it, decide you want to try something wild. You could code M2 = M, thus making a copy of the problem, use the moptimize*() functions with M2, and all the while your original problem would remain undisturbed.

Having obtained a problem handle, that is, having coded M = moptimize_init(), you now need to fill in the details of your problem. You do that with the moptimize_init_*() functions. The order in which you do this does not matter. We first informed moptimize() of the identity of the evaluator function:

```
        : moptimize_init_evaluator(M, &linregeval())
```

We must also inform moptimize() as to the type of evaluator function linregeval() is, which we could do by coding

```
        : moptimize_init_evaluatortype(M, "lf")
```

We did not bother, however, because type lf is the default.

After that, we need to inform moptimize() as to the identity of the dependent variables:

```
        : moptimize_init_depvar(M, 1, "mpg")
```

Dependent variables play no special role in moptimize(); they are merely something that are remembered so that they can be passed to the evaluator function that we write. One problem might have no dependent variables and another might have lots of them. moptimize_init_depvar(*M*, *i*, *y*)'s second argument specifies which dependent variable is being set. There is no requirement that the number of dependent variables match the number of equations. In the linear regression case, we have one dependent variable and two equations.

Next we set the independent variables, or equivalently, the mapping of coefficients, into parameters. When we code

```
        : moptimize_init_eq_indepvars(M, 1, "weight foreign")
```

we are stating that there is a parameter, $\mathbf{p}_1 = \mathbf{X}_1 \times \mathbf{b}_1$:+ c_1, and that $\mathbf{X}_1 = (\texttt{weight}, \texttt{foreign})$. Thus $\mathbf{b}_1$ contains two coefficients, that is, $\mathbf{p}_1 = (\texttt{weight}, \texttt{foreign}) \times (b_{11}, b_{12})'$:+ c_1. Actually, we have not yet specified whether there is a constant, c_1, on the end, but if we do not specify otherwise, the constant will be included. If we want to suppress the constant, after coding `moptimize_init_eq_indepvars(M, 1, "weight foreign")`, we would code `moptimize_init_eq_cons(M, 1, "off")`. The `1` says first equation, and the `"off"` says to turn the constant off.

As an aside, we coded `moptimize_init_eq_indepvars(M, 1, "weight foreign")` and so specified that the independent variables were the Stata variables `weight` and `foreign`, but the independent variables do not have to be in Stata. If we had a 74×2 matrix named `data` in Mata that we wanted to use, we would have coded `moptimize_init_eq_indepvars(M, 1, data)`.

To define the second parameter, we code

```
        : moptimize_init_eq_indepvars(M, 2, "")
```

Thus we are stating that there is a parameter, $\mathbf{p}_2 = \mathbf{X}_2 \times \mathbf{b}_2$ `:+` c_2, and that $\mathbf{X}_2$ does not exist, leaving $\mathbf{p}_2 = c_2$, meaning that the second parameter is a constant.

Our problem defined, we code

```
        : moptimize(M)
```

to obtain the solution, and we code

```
        : moptimize_result_display(M)
```

to see the results. There are many `moptimize_result_*()` functions for use after the solution is obtained.

The type d, lf*, gf, and q evaluators

Above we wrote our evaluator function in the style of type `lf`. `moptimize()` provides four other evaluator types—called types `d`, `lf*`, `gf`, and `q`—and each have their uses.

Using type `lf` above, we were required to calculate the observation-by-observation log likelihoods and that was all. Using another type of evaluator, say, type `d`, we are required to calculate the overall log likelihood, and optionally, its first derivatives, and optionally, its second derivatives. The corresponding evaluator types are called `d0`, `d1`, and `d2`. Type `d` is better than type `lf` because if we do calculate the derivatives, then `moptimize()` can execute more quickly and it can produce a slightly more accurate result (more accurate because numerical derivatives are not involved). These speed and accuracy gains justify type `d1` and `d2`, but what about type `d0`? For many optimization problems, type `d0` is redundant and amounts to nothing more than a slight variation on type `lf`. In these cases, type `d0`'s justification is that if we want to write a type `d1` or type `d2` evaluator, then it is usually easiest to start by writing a type `d0` evaluator. Make that work, and then add to the code to convert our type `d0` evaluator into a type `d1` evaluator; make that work, and then, if we are going all the way to type `d2`, add the code to convert our type `d1` evaluator into a type `d2` evaluator.

For other optimization problems, however, there is a substantive reason for type `d0`'s existence. Type `lf` requires observation-by-observation values of the log-likelihood function, and for some likelihood functions, those simply do not exist. Think of a panel-data model. There may be observations within each of the panels, but there is no corresponding log-likelihood value for each of them. The log-likelihood function is defined only across the entire panel. Type `lf` cannot handle problems like that. Type `d0` can.

That makes type d0 seem to be a strict improvement on type lf. Type d0 can handle any problem that type lf can handle, and it can handle other problems to boot. Where both can handle the problem, the only extra work to use type d0 is that we must sum the individual values we produce, and that is not difficult. Type lf, however, has other advantages. If you write a type lf evaluator, then without writing another line of code, you can obtain the robust estimates of variance, adjust for clustering, account for survey design effects, and more. moptimize() can do all that because it has the results of an observation-by-observation calculation. moptimize() can break into the assembly of those observation-by-observation results and modify how that is done. moptimize() cannot do that for d0.

So there are advantages both ways.

Another provided evaluator type is type lf*. Type lf* is a variation on type lf. It also comes in the subflavors lf0, lf1, and lf2. Type lf* allows you to make observation-level derivative calculations, which means that results can be obtained more quickly and more accurately. Type lf* is designed to always work where lf is appropriate, which means panel-data estimators are excluded. In return, it provides all the ancillary features provided by type lf, meaning that robust standard errors, clustering, and survey-data adjustments are available. You write the evaluator function in a slightly different style when you use type lf* rather than type d.

Type gf is a variation on type lf* that relaxes the requirement that the log-likelihood function be summable over the observations. Thus type gf can work with panel-data models and resurrect the features of robust standard errors, clustering, and survey-data adjustments. Type gf evaluators, however, are more difficult to write than type lf* evaluators.

Type q is for the special case of quadratic optimization. You either need it, and then only type q will do, or you do not.

Example using type d

Let's return to our linear regression maximum-likelihood estimator. To remind you, this is a two-parameter model, and the log-likelihood function is

$$f(\mathbf{p}_1, \mathbf{p}_2; \mathbf{y}) = \sum \texttt{ln(normalden(}\mathbf{y} - \mathbf{p}_1\texttt{,0,sqrt(}\mathbf{p}_2))$$

This time, however, we are going to parameterize the variance parameter $\mathbf{p}_2$ as the log of the standard deviation, so we will write

$$f(\mathbf{p}_1, \mathbf{p}_2; \mathbf{y}) = \sum \texttt{ln(normalden(}\mathbf{y} - \mathbf{p}_1\texttt{,0,exp(}\mathbf{p}_2\texttt{))}$$

It does not make any conceptual difference which parameterization we use, but the log parameterization converges a little more quickly, and the derivatives are easier, too. We are going to implement a type d2 evaluator for this function. To save you from pulling out pencil and paper, let's tell you the derivatives:

$$\begin{aligned} df/d\mathbf{p}_1 &= \mathbf{z}\texttt{:/}\mathbf{s} \\ df/d\mathbf{p}_2 &= \mathbf{z}\texttt{:\^{}2:-1} \end{aligned}$$

$$\begin{aligned} d^2f/d\mathbf{p}_1^2 &= -1\texttt{:/}\mathbf{s}\texttt{:\^{}2} \\ d^2f/d\mathbf{p}2^2 &= -2 \times \mathbf{z}\texttt{:\^{}2} \\ d^2f/d\mathbf{p}d\mathbf{p}_2 &= -2 \times \mathbf{z}\texttt{:/}\mathbf{s} \end{aligned}$$

where

$$\mathbf{z} = (\mathbf{y} : -\mathbf{p}_1) :/ \mathbf{s}$$

$$\mathbf{s} = \exp(\mathbf{p}_2)$$

The d2 evaluator function for this problem is

```
function linregevald2(transmorphic M, real scalar todo,
                      real rowvector b, fv, g, H)
{
        y1  = moptimize_calc_depvar(M, 1)
        p1  = moptimize_calc_xb(M, b, 1)
        p2  = moptimize_calc_xb(M, b, 2)
        s   = exp(p2)
        z   = (y1:-p1):/s
        fv  = moptimize_util_sum(M, ln(normalden(y1:-p1, 0, s)))
        if (todo>=1) {
                s1  = z:/s
                s2  = z:^2 :- 1
                g1  = moptimize_util_vecsum(M, 1, s1, fv)
                g2  = moptimize_util_vecsum(M, 2, s2, fv)
                g   = (g1, g2)
                if (todo==2) {
                        h11 = -1:/s:^2
                        h22 = -2*z:^2
                        h12 = -2*z:/s
                        H11 = moptimize_util_matsum(M, 1,1, h11, fv)
                        H22 = moptimize_util_matsum(M, 2,2, h22, fv)
                        H12 = moptimize_util_matsum(M, 1,2, h12, fv)
                        H   = (H11, H12 \ H12', H22)
                }
        }
}
```

The code to fit a model of mpg on weight and foreign reads

```
: M = moptimize_init()
: moptimize_init_evaluator(M, &linregevald2())
: moptimize_init_evaluatortype(M, "d2")
: moptimize_init_depvar(M, 1, "mpg")
: moptimize_init_eq_indepvars(M, 1, "weight foreign")
: moptimize_init_eq_indepvars(M, 2, "")
: moptimize(M)
: moptimize_result_display(M)
```

By the way, function linregevald2() will work not only with type d2, but also with types d1 and d0. Our function has the code to calculate first and second derivatives, but if we use type d1, todo will never be 2 and the second derivative part of our code will be ignored. If we use type d0, todo will never be 1 or 2 and so the first and second derivative parts of our code will be ignored. You could delete the unnecessary blocks if you wanted.

It is also worth trying the above code with types d1debug and d2debug. Type d1debug is like d1; the second derivative code will not be used. Also type d1debug will almost ignore the first derivative code. Our program will be asked to make the calculation, but `moptimize()` will not use the results except to report a comparison of the derivatives we calculate with numerically calculated derivatives. That way, we can check that our program is right. Once we have done that, we move to type `d2debug`, which will check our second-derivative calculation.

Example using type lf*

The `lf2` evaluator function for the linear-regression problem is almost identical to the type `d2` evaluator. It differs in that rather than return the summed log likelhood, we return the observation-level log likelihoods. And rather than return the gradient vector, we return the equation-level scores that when used with the chain-rule can be summed to produce the gradient. The conversion from `d2` to `lf2` was possible because of the observation-by-observation nature of the linear-regression problem; if the evaluator was not going to be implemented as `lf`, it always should have been implemented as `lf1` or `lf2` instead of `d1` or `d2`. In the `d2` evaluator above, we went to extra work—summing the scores—the result of which was to eliminate `moptimize()` features such as being able to automatically adjust for clusters and survey data. In a more appropriate type `d` problem—a problem for which a type `lf*` evaluator could not have been implemented—those scores never would have been available in the first place.

The `lf2` evaluator is

```
        function linregevallf2(transmorphic M, real scalar todo,
                                real rowvector b, fv, S, H)
        {
                y1  = moptimize_calc_depvar(M, 1)
                p1  = moptimize_calc_xb(M, b, 1)
                p2  = moptimize_calc_xb(M, b, 2)
                s   = exp(p2)
                z   = (y1:-p1):/s
                fv  = ln(normalden(y1:-p1, 0, s))
                if (todo>=1) {
                        s1  = z:/s
                        s2  = z:^2 :- 1
                        S   = (s1, s2)
                        if (todo==2) {
                                h11 = -1:/s:^2
                                h22 = -2*z:^2
                                h12 = -2*z:/s
                                mis = 0
                                H11 = moptimize_util_matsum(M, 1,1, h11, mis)
                                H22 = moptimize_util_matsum(M, 2,2, h22, mis)
                                H12 = moptimize_util_matsum(M, 1,2, h12, mis)
                                H   = (H11, H12 \ H12', H22)
                        }
                }
        }
```

The code to fit a model of mpg on weight and foreign reads nearly identically to the code we used in the type d2 case. We must specify the name of our type lf2 evaluator and specify that it is type lf2:

```
: M = moptimize_init()
: moptimize_init_evaluator(M, &linregevallf2())
: moptimize_init_evaluatortype(M, "lf2")
: moptimize_init_depvar(M, 1, "mpg")
: moptimize_init_eq_indepvars(M, 1, "weight foreign")
: moptimize_init_eq_indepvars(M, 2, "")
: moptimize(M)
: moptimize_result_display(M)
```

Conformability

See *Syntax* above.

Diagnostics

All functions abort with error when used incorrectly.

moptimize() aborts with error if it runs into numerical difficulties. _moptimize() does not; it instead returns a nonzero error code.

The moptimize_result*() functions abort with error if they run into numerical difficulties when called after moptimize() or moptimize_evaluate(). They do not abort when run after _moptimize() or _moptimize_evaluate(). They instead return a properly dimensioned missing result and set moptimize_result_errorcode() and moptimize_result_errortext().

Reference

Gould, W. W., J. S. Pitblado, and W. M. Sribney. 2006. *Maximum Likelihood Estimation with Stata*. 3rd ed. College Station, TX: Stata Press.

Also see

[M-5] **optimize( )** — Function optimization

[M-4] **mathematical** — Important mathematical functions

[M-4] **statistical** — Statistical functions