Title

optimize( ) — Function optimization

Syntax

S = optimize_init()

(varies) optimize_init_which(*S* [, { "max" | "min" }])

(varies) optimize_init_evaluator(*S* [, &*function*()])

(varies) optimize_init_evaluatortype(*S* [, *evaluatortype*])

(varies) optimize_init_negH(*S*, { "off" | "on" })

(varies) optimize_init_params(*S* [, *real rowvector initialvalues*])

(varies) optimize_init_nmsimplexdeltas(*S* [, *real rowvector delta*])

(varies) optimize_init_argument(*S*, *real scalar k* [, *X*])

(varies) optimize_init_narguments(*S* [, *real scalar K*])

(varies) optimize_init_cluster(*S*, *c*)

(varies) optimize_init_colstripe(*S* [, stripe])

(varies) optimize_init_technique(*S* [, *technique*])

(varies) optimize_init_singularHmethod(*S* [, *singularHmethod*])

(varies) optimize_init_conv_maxiter(*S* [, *real scalar max*])

(varies) optimize_init_conv_warning(*S*, { "on" | "off" })

(varies) optimize_init_conv_ptol(*S* [, *real scalar ptol*])

(varies) optimize_init_conv_vtol(*S* [, *real scalar vtol*])

(varies) optimize_init_conv_nrtol(*S* [, *real scalar nrtol*])

(varies) optimize_init_conv_ignorenrtol(*S*, { "off" | "on" })

(varies) optimize_init_iterid(*S* [, *string scalar id*])

(varies) optimize_init_valueid(*S* [, *string scalar id*])

(varies) optimize_init_tracelevel(*S* [, *tracelevel*])

(varies)	`optimize_init_trace_dots(`*S*`,` { `"off"` | `"on"` }`)`
(varies)	`optimize_init_trace_value(`*S*`,` { `"on"` | `"off"` }`)`
(varies)	`optimize_init_trace_tol(`*S*`,` { `"off"` | `"on"` }`)`
(varies)	`optimize_init_trace_step(`*S*`,` { `"off"` | `"on"` }`)`
(varies)	`optimize_init_trace_paramdiffs(`*S*`,` { `"off"` | `"on"` }`)`
(varies)	`optimize_init_trace_params(`*S*`,` { `"off"` | `"on"` }`)`
(varies)	`optimize_init_trace_gradient(`*S*`,` { `"off"` | `"on"` }`)`
(varies)	`optimize_init_trace_Hessian(`*S*`,` { `"off"` | `"on"` }`)`
(varies)	`optimize_init_evaluations(`*S*`,` { `"off"` | `"on"` }`)`
(varies)	`optimize_init_constraints(`*S* [, *real matrix Cc*]`)`
(varies)	`optimize_init_verbose(`*S* [, *real scalar verbose*]`)`
real rowvector	`optimize(`*S*`)`
real scalar	`_optimize(`*S*`)`
void	`optimize_evaluate(`*S*`)`
real scalar	`_optimize_evaluate(`*S*`)`
real rowvector	`optimize_result_params(`*S*`)`
real scalar	`optimize_result_value(`*S*`)`
real scalar	`optimize_result_value0(`*S*`)`
real rowvector	`optimize_result_gradient(`*S*`)`
real matrix	`optimize_result_scores(`*S*`)`
real matrix	`optimize_result_Hessian(`*S*`)`
real matrix	`optimize_result_V(`*S*`)`
string scalar	`optimize_result_Vtype(`*S*`)`
real matrix	`optimize_result_V_oim(`*S*`)`
real matrix	`optimize_result_V_opg(`*S*`)`
real matrix	`optimize_result_V_robust(`*S*`)`
real scalar	`optimize_result_iterations(`*S*`)`
real scalar	`optimize_result_converged(`*S*`)`
real colvector	`optimize_result_iterationlog(`*S*`)`
real rowvector	`optimize_result_evaluations(`*S*`)`
real scalar	`optimize_result_errorcode(`*S*`)`
string scalar	`optimize_result_errortext(`*S*`)`
real scalar	`optimize_result_returncode(`*S*`)`

void `optimize_query(`*S*`)`

where *S*, if it is declared, should be declared

`transmorphic` *S*

and where *evaluatortype* optionally specified in `optimize_init_evaluatortype()` is

evaluatortype	description
`"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
`"gf0"`	*function*`()` returns *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

The default is `"d0"` if not set.

and where *technique* optionally specified in `optimize_init_technique()` is

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

The default is `"nr"`.

and where *singularHmethod* optionally specified in `optimize_init_singularHmethod()` is

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

The default is `"m-marquardt"` if not set;
`"hybrid"` is equivalent to `ml`'s `difficult` option; see [R] **ml**.

and where *tracelevel* optionally specified in `optimize_init_tracelevel()` is

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

The default is `"value"` if not set.

Description

These functions find parameter vector or scalar *p* such that function *f*(*p*) is a maximum or a minimum.

`optimize_init()` begins the definition of a problem and returns *S*, a problem-description handle set to contain default values.

The `optimize_init_*(`*S*`, ...)` functions then allow you to modify those defaults. You use these functions to describe your particular problem: to set whether you wish maximization or minimization, to set the identity of function *f*(), to set initial values, and the like.

`optimize(`*S*`)` then performs the optimization. `optimize()` returns *real rowvector p* containing the values of the parameters that produce a maximum or minimum.

The `optimize_result_*(`*S*`)` functions can then be used to access other values associated with the solution.

Usually you would stop there. In other cases, you could restart optimization by using the resulting parameter vector as new initial values, change the optimization technique, and restart the optimization:

```
optimize_init_param(S, optimize_result_param(S))
optimize_init_technique(S, "dfp")
optimize(S)
```

Aside: The `optimize_init_*(`*S*`, ...)` functions have two modes of operation. Each has an optional argument that you specify to set the value and that you omit to query the value. For instance, the full syntax of `optimize_init_params()` is

void `optimize_init_params(`*S*`,` *real rowvector initialvalues*`)`

real rowvector `optimize_init_params(`*S*`)`

The first syntax sets the initial values and returns nothing. The second syntax returns the previously set (or default, if not set) initial values.

All the `optimize_init_*(`*S*`, ...)` functions work the same way.

Remarks

Remarks are presented under the following headings:

First example

The optimization functions may be used interactively.

Below we use the functions to find the value of x that maximizes $y = \exp(-x^2 + x - 3)$:

```
: void myeval(todo, x,  y, g, H)
> {
>         y = exp(-x^2 + x - 3)
> }
note: argument todo unused
note: argument g unused
note: argument H unused
: S = optimize_init()
: optimize_init_evaluator(S, &myeval())
: optimize_init_params(S, 0)
: x = optimize(S)
Iteration 0:  f(p) = .04978707
Iteration 1:  f(p) = .04978708
Iteration 2:  f(p) = .06381186
Iteration 3:  f(p) = .06392786
Iteration 4:  f(p) = .06392786
: x
.5
```

Notation

We wrote the above in the way that mathematicians think, i.e., optimizing $y = f(x)$. Statisticians, on the other hand, think of optimizing $s = f(b)$. To avoid favoritism, we will write $v = f(p)$ and write the general problem with the following notation:

Maximize or minimize $v = f(p)$,

v: a scalar

p: $1 \times np$

subject to the constraint $Cp' = c$,

C: $nc \times np$ ($nc = 0$ if no constraints)
c: $nc \times 1$

where g, the gradient vector, is $g = f'(p) = df/dp$,

g: $1 \times np$

and H, the Hessian matrix, is $H = f''(p) = d^2f/dpdp'$

H: $np \times np$

Type d evaluators

You must write an evaluator function to calculate $f()$ before you can use the optimization functions. The example we showed above was of what is called a type `d` evaluator. Let's stay with that.

The evaluator function we wrote was

```
        void myeval(todo, x,  y, g, H)
        {
                y = exp(-x^2 + x - 3)
        }
```

All type `d` evaluators open the same way,

void evaluator(*todo*, *x*, *y*, *g*, *H*)

although what you name the arguments is up to you. We named the arguments the way that mathematicians think, although we could just as well have named them the way that statisticians think:

void evaluator(*todo*, *b*, *s*, *g*, *H*)

To avoid favoritism, we will write them as

void evaluator(*todo*, *p*, *v*, *g*, *H*)

i.e., we will think in terms of optimizing $v = f(p)$.

Here is the full definition of a type `d` evaluator:

---

void evaluator(*real scalar todo*, *real rowvector p*, *v*, *g*, *H*)

where *v*, *g*, and *H* are values to be returned:

v:	*real scalar*
g:	*real rowvector*
H:	*real matrix*

evaluator() is to fill in *v* given the values in *p* and optionally to fill in *g* and *H*, depending on the value of *todo*:

todo	Required action by *evaluator*()
0	calculate $v = f(p)$ and store in v
1	calculate $v = f(p)$ and $g = f'(p)$ and store in v and g
2	calculate $v = f(p)$, $g = f'(p)$, and $H = f''(p)$ and store in v, g, and H

evaluator() may return *v*=`.` if *f*() cannot be evaluated at *p*. Then *g* and *H* need not be filled in even if requested.

An evaluator does not have to be able to do all of this. In the first example, `myeval()` could handle only *todo* = 0. There are three types of type `d` evaluators:

`d` type	Capabilities expected of *evaluator*()
`d0`	can calculate $v = f(p)$
`d1`	can calculate $v = f(p)$ and $g = f'(p)$
`d2`	can calculate $v = f(p)$ and $g = f'(p)$ and $H = f''(p)$

myeval() is a type d0 evaluator. Type d0 evaluators are never asked to calculate g or H. Type d0 is the default type but, if we were worried that it was not, we could set the evaluator type before invoking optimize() by coding

```
optimize_init_evaluatortype(S, "d0")
```

Here are the code outlines of the three types of evaluators:

```
void d0_evaluator(todo, p, v, g, H)
{
        v = ...
}
```

```
void d1_evaluator(todo, p, v, g, H)
{
        v = ...
        if (todo>=1) {
                g = ...
        }
}
```

```
void d2_evaluator(todo, p, v, g, H)
{
        v = ...
        if (todo>=1) {
                g = ...
                if (todo==2) {
                        H = ...
                }
        }
}
```

Here is the code outline where there are three additional user arguments to the evaluator:

```
void d0_user3_eval(todo, p, u1, u2, u3, v, g, H)
{
        v = ...
}
```

Example of d0, d1, and d2

We wish to find the p_1 and p_2 corresponding to the maximum of

$$v = \exp(-p_1^2 - p_2^2 - p_1p_2 + p_1 - p_2 - 3)$$

A d0 solution to the problem would be

```
: void eval0(todo, p, v, g, H)
> {
>         v = exp(-p[1]^2 - p[2]^2 - p[1]*p[2] + p[1] - p[2] - 3)
> }
note: argument todo unused
note: argument g unused
note: argument h unused
: S = optimize_init()
: optimize_init_evaluator(S, &eval0())
: optimize_init_params(S, (0,0))
: p = optimize(S)
Iteration 0:  f(p) = .04978707  (not concave)
Iteration 1:  f(p) = .12513024
Iteration 2:  f(p) = .13495886
Iteration 3:  f(p) = .13533527
Iteration 4:  f(p) = .13533528
: p
        1    2
    +----------+
  1 |   1   -1 |
    +----------+
```

A d1 solution to the problem would be

```
: void eval1(todo, p, v, g, H)
> {
>         v = exp(-p[1]^2 - p[2]^2 - p[1]*p[2] + p[1] - p[2] - 3)
>         if (todo==1) {
>                 g[1] = (-2*p[1] - p[2] + 1)*v
>                 g[2] = (-2*p[2] - p[1] - 1)*v
>         }
> }
note: argument H unused
: S = optimize_init()
: optimize_init_evaluator(S, &eval1())
: optimize_init_evaluatortype(S, "d1")              ←   important
: optimize_init_params(S, (0,0))
: p = optimize(S)
Iteration 0:  f(p) = .04978707  (not concave)
Iteration 1:  f(p) = .12513026
Iteration 2:  f(p) = .13496887
Iteration 3:  f(p) = .13533527
Iteration 4:  f(p) = .13533528
: p
        1    2
    +----------+
  1 |   1   -1 |
    +----------+
```

The d1 solution is better than the d0 solution because it runs faster and usually is more accurate. Type d1 evaluators require more code, however, and deriving analytic derivatives is not always possible.

A d2 solution to the problem would be

```
: void eval2(todo, p, v, g, H)
> {
>         v = exp(-p[1]^2 - p[2]^2 - p[1]*p[2] + p[1] - p[2] - 3)
>         if (todo>=1) {
>                 g[1] = (-2*p[1] - p[2] + 1)*v
>                 g[2] = (-2*p[2] - p[1] - 1)*v
>                 if (todo==2) {
>                         H[1,1] = -2*v + (-2*p[1]-p[2]+1)*g[1]
>                         H[2,1] = -1*v + (-2*p[2]-p[1]-1)*g[1]
>                         H[2,2] = -2*v + (-2*p[2]-p[1]-1)*g[2]
>                         _makesymmetric(H)
>                 }
>         }
> }

: S = optimize_init()

: optimize_init_evaluator(S, &eval2())

: optimize_init_evaluatortype(S, "d2")                ←  important

: optimize_init_params(S, (0,0))

: p = optimize(S)
Iteration 0:  f(p) = .04978707  (not concave)
Iteration 1:  f(p) = .12513026
Iteration 2:  f(p) = .13496887
Iteration 3:  f(p) = .13533527
Iteration 4:  f(p) = .13533528

: p
        1    2
    +-----------+
  1 |   1   -1  |
    +-----------+
```

A d2 solution is best because it runs fastest and usually is the most accurate. Type d2 evaluators require the most code, and deriving analytic derivatives is not always possible.

In the d2 evaluator `eval2()`, note our use of `_makesymmetric()`. Type d2 evaluators are required to return *H* as a symmetric matrix; filling in just the lower or upper triangle is not sufficient. The easiest way to do that is to fill in the lower triangle and then use `_makesymmetric()` to reflect the lower off-diagonal elements; see [M-5] **makesymmetric( )**.

d1debug and d2debug

In addition to evaluator types `"d0"`, `"d1"`, and `"d2"` that are specified in `optimize_init_evaluatortype(`*S*`,` *evaluatortype*`)`, there are two more: `"d1debug"` and ”d2debug”. They assist in coding d1 and d2 evaluators.

In *Example of d0, d1, and d2* above, we admit that we did not correctly code the functions `eval1()` and `eval2()` at the outset, before you saw them. In both cases, that was because we had taken the derivatives incorrectly. The problem was not with our code but with our math. `d1debug` and `d2debug` helped us find the problem.

`d1debug` is an alternative to `d1`. When you code `optimize_init_evaluatortype(`*S*`, "d1debug")`, the derivatives you calculate are not taken seriously. Instead, `optimize()` calculates its own numerical derivatives and uses those. Each time `optimize()` does that, however, it compares your derivatives to the ones it calculated and gives you a report on how they differ. If you have coded correctly, they should not differ by much.

`d2debug` does the same thing, but for `d2` evaluators. When you code `optimize_init_evaluatortype(`*S*`, "d2debug")`, `optimize()` uses numerical derivatives but, each time, `optimize()` gives you a report on how much your results for the gradient and for the Hessian differ from the numerical calculations.

For each comparison, `optimize()` reports just one number: the `mreldif()` (see [M-5] **reldif( )**) between your results and the numerical ones. When you have done things right, gradient vectors will differ by approximately 1e–12 or less and Hessians will differ by 1e–7 or less.

When differences are large, you will want to see not only the summary comparison but also the full vectors and matrices so that you can compare your results element by element with those calculated numerically. Sometimes the error is in one element and not the others. To do this, set the trace level with `optimize_init_tracelevel(`*S*`,` *tracelevel*`)` before issuing `optimize()`. Code `optimize_init_tracelevel(`*S*`, "gradient")` to get a full report on the gradient comparison, or set `optimize_init_tracelevel(`*S*`, "hessian")` to get a full report on the gradient comparison and the Hessian comparison.

Type gf evaluators

In some statistical applications, you will find `gf0`, `gf1`, and `gf2` more convenient to code than `d0`, `d1`, and `d2`. The *gf* stands for general form.

In statistical applications, one tends to think of a dataset of values arranged in matrix *X*, the rows of which are observations. A function *h*(*p*, *X*`[`*i*`,.]`) can be calculated for each row separately, and it is the sum of those resulting values that forms the function *f(p)* that is to be maximized or minimized.

The `gf0`, `gf1`, and `gf2` methods are for such cases.

In a type `d0` evaluator, you return scalar $v = f(p)$.

In a type `gf0` evaluator, you return a column vector *v* such that `colsum(`*v*`)` $= f(p)$.

In a type `d1` evaluator, you return $v = f(p)$ and you return a row vector $g = f'(p)$.

In a type `gf1` evaluator, you return *v* such that `colsum(`*v*`)` $= f(p)$ and you return matrix *g* such that `colsum(`*g*`)` $= f'(p)$.

In a type `d2` evaluator, you return $v = f(p)$, $g = f'(p)$, and you return $H = f''(p)$.

In a type `gf2` evaluator, you return *v* such that `colsum(`*v*`)` $= f(p)$, *g* such that `colsum(`*g*`)` $= f'(p)$, and you return $H = f''(p)$. This is the same *H* returned for `d2`.

The code outline for type `gf` evaluators is the same as those for `d` evaluators. For instance, the outline for a `gf2` evaluator is

```
void gf2_evaluator(todo, p, v, g, H)
{
        v = ...
        if (todo>=1) {
                g = ...
                if (todo==2) {
                        H = ...
                }
        }
}
```

The above is the same as the outline for d2 evaluators. All that differs is that *v* and *g*, which were *real scalar* and *real rowvector* in the d2 case, are now *real colvector* and *real matrix* in the gf2 case. The same applies to gf1 and gf0.

The type gf evaluators arise in statistical applications and, in such applications, there are data; i.e., just knowing *p* is not sufficient to calculate *v*, *g*, and *H*. Actually, that same problem can arise when coding type d evaluators as well.

You can pass extra arguments to evaluators, whether they be d0, d1, or d2 or gf0, gf1, or gf2. The first line of all evaluators, regardless of style, is

```
void evaluator(todo, p, v, g, H)
```

If you code

```
optimize_init_argument(S, 1, X)
```

the first line becomes

```
void evaluator(todo, p, X, v, g, H)
```

If you code

```
optimize_init_argument(S, 1, X)
optimize_init_argument(S, 2, Y)
```

the first line becomes

```
void evaluator(todo, p, X, Y, v, g, H)
```

and so on, up to nine extra arguments. That is, you can specify extra arguments to be passed to your function. These extra arguments should be placed right after the parameter vector.

Example of gf0, gf1, and gf2

You have the following data:

```
: x
          1
     +-------+
   1 |  .35  |
   2 |  .29  |
   3 |   .3  |
   4 |   .3  |
   5 |  .65  |
   6 |  .56  |
   7 |  .37  |
   8 |  .16  |
   9 |  .26  |
  10 |  .19  |
     +-------+
```

You believe that the data are the result of a beta distribution process with fixed parameters alpha and beta and you wish to obtain the maximum likelihood estimates of alpha and beta (*a* and *b* in what follows). The formula for the density of the beta distribution is

$$\text{density}(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\, x^{a-1}\, (1-x)^{b-1}$$

The gf0 solution to this problem is

```
: void lnbetaden0(todo, p,  x,  lnf, S, H)
> {
>         a   = p[1]
>         b   = p[2]
>         lnf = lngamma(a+b) :- lngamma(a) :- lngamma(b) :+
>               (a-1)*log(x) :+ (b-1)*log(1:-x)
> }
note: argument todo unused
note: argument S unused
note: argument H unused
: S = optimize_init()
: optimize_init_evaluator(S, &lnbetaden0())
: optimize_init_evaluatortype(S, "gf0")
: optimize_init_params(S, (1,1))
: optimize_init_argument(S, 1, x)                    ←  important
: p = optimize(S)
Iteration 0:  f(p) =         0
Iteration 1:  f(p) = 5.7294728
Iteration 2:  f(p) = 5.7646641
Iteration 3:  f(p) = 5.7647122
Iteration 4:  f(p) = 5.7647122
: p
                 1             2
    +-----------------------------+
  1 |  3.714209592   7.014926315  |
    +-----------------------------+
```

Note the following:

1. Rather than calling the returned value v, we called it lnf. You can name the arguments as you please.
2. We arranged for an extra argument to be passed by coding `optimize_init_argument(S, 1, x)`. The extra argument is the vector x, which we listed previously for you. In our function, we received the argument as x, but we could have used a different name, just as we used lnf rather than v.
3. We set the evaluator type to "gf0".

This being a statistical problem, we should be interested not only in the estimates p but also in their variance. We can get this from the inverse of the negative Hessian, which is the observed information matrix:

```
: optimize_result_V_oim(S)
[symmetric]
                 1             2
    +-----------------------------+
  1 |  2.556301184                |
  2 |  4.498194785   9.716647065  |
    +-----------------------------+
```

The gf1 solution to this problem is

```
: void lnbetaden1(todo, p,  x,  lnf, S, H)
> {
>         a   = p[1]
>         b   = p[2]
>         lnf = lngamma(a+b) :- lngamma(a) :- lngamma(b) :+
>               (a-1)*log(x) :+ (b-1)*log(1:-x)
>          if (todo >= 1) {
>                S       = J(rows(x), 2, .)
>                S[.,1]  = log(x) :+ digamma(a+b) :- digamma(a)
>                S[.,2]  = log(1:-x) :+ digamma(a+b) :- digamma(b)
>         }
> }
note: argument H unused
: S = optimize_init()
: optimize_init_evaluator(S, &lnbetaden1())
: optimize_init_evaluatortype(S, "gf1")
: optimize_init_params(S, (1,1))
: optimize_init_argument(S, 1, x)
: p = optimize(S)
Iteration 0:  f(p) =         0
Iteration 1:  f(p) = 5.7297061
Iteration 2:  f(p) = 5.7641349
Iteration 3:  f(p) = 5.7647121
Iteration 4:  f(p) = 5.7647122
: p
                 1              2
    +-------------------------------+
  1 |  3.714209343    7.014925751  |
    +-------------------------------+

: optimize_result_V_oim(S)
[symmetric]
                 1              2
    +-------------------------------+
  1 |  2.556299425                 |
  2 |   4.49819212    9.716643068  |
    +-------------------------------+
```

Note the following:

1. We called the next-to-last argument of lnbetaden1() S rather than g in accordance with standard statistical jargon. What is being returned is in fact the observation-level scores, which sum to the gradient vector.

2. We called the next-to-last argument S even though that name conflicted with S outside the program, where S is the problem handle. Perhaps we should have renamed the outside S, but there is no confusion on Mata's part.

3. In our program we allocated *S* for ourselves: S = J(rows(x), 2, .). It is worth comparing this with the example of d1 in *Example of d0, d1, and d2*, where we did not need to allocate g. In d1, optimize() preallocates g for us. In gf1, optimize() cannot do this because it has no idea how many "observations" we have.

The gf2 solution to this problem is

```
: void lnbetaden2(todo, p,  x,  lnf, S, H)
> {
>         a   = p[1]
>         b   = p[2]
>         lnf = lngamma(a+b) :- lngamma(a) :- lngamma(b) :+
>               (a-1)*log(x) :+ (b-1)*log(1:-x)
>         if (todo >= 1) {
>                 S       = J(rows(x), 2, .)
>                 S[.,1]  = log(x) :+ digamma(a+b) :- digamma(a)
>                 S[.,2]  = log(1:-x) :+ digamma(a+b) :- digamma(b)
>                 if (todo==2) {
>                        n = rows(x)
>                        H[1,1] = n*(trigamma(a+b) - trigamma(a))
>                        H[2,1] = n*(trigamma(a+b))
>                        H[2,2] = n*(trigamma(a+b) - trigamma(b))
>                        _makesymmetric(H)
>                 }
>         }
> }
: S = optimize_init()
: optimize_init_evaluator(S, &lnbetaden2())
: optimize_init_evaluatortype(S, "gf2")
: optimize_init_params(S, (1,1))
: optimize_init_argument(S, 1, x)
: p = optimize(S)
Iteration 0:  f(p) =         0
Iteration 1:  f(p) = 5.7297061
Iteration 2:  f(p) = 5.7641349
Iteration 3:  f(p) = 5.7647121
Iteration 4:  f(p) = 5.7647122
: p
                 1             2
    +-----------------------------+
  1 |  3.714209343   7.014925751  |
    +-----------------------------+
: optimize_result_V_oim(S)
[symmetric]
                  1             2
    +-----------------------------+
  1 |  2.556299574                |
  2 |  4.498192412   9.716643651  |
    +-----------------------------+
```

Functions

optimize_init( )

transmorphic `optimize_init()`

`optimize_init()` is used to begin an optimization problem. Store the returned result in a variable name of your choosing; we have used *S* in this documentation. You pass *S* as the first argument to the other `optimize*()` functions.

`optimize_init()` sets all `optimize_init_*()` values to their defaults. You may use the query form of the `optimize_init_*()` to determine an individual default, or you can use `optimize_query()` to see them all.

The query form of `optimize_init_*()` can be used before or after optimization performed by `optimize()`.

optimize_init_which( )

void `optimize_init_which(`*S*`,` {`"max"` | `"min"`}`)`

string scalar `optimize_init_which(`*S*`)`

`optimize_init_which(`*S*`,` *which*`)` specifies whether `optimize()` is to perform maximization or minimization. The default is maximization if you do not invoke this function.

`optimize_init_which(`*S*`)` returns `"max"` or `"min"` according to which is currently set.

optimize_init_evaluator( ) and optimize_init_evaluatortype( )

void `optimize_init_evaluator(`*S*`,` *pointer(real function) scalar fptr*`)`

void `optimize_init_evaluatortype(`*S*`,` *evaluatortype*`)`

pointer(real function) scalar `optimize_init_evaluator(`*S*`)`

string scalar `optimize_init_evaluatortype(`*S*`)`

`optimize_init_evaluator(`*S*`,` *fptr*`)` specifies the function to be called to evaluate $f(p)$. Use of this function is required. If your function is named `myfcn()`, you code `optimize_init_evaluator(`*S*`, &myfcn())`.

`optimize_init_evaluatortype(`*S*`,` *evaluatortype*`)` specifies the capabilities of the function that has been set using `optimize_init_evaluator()`. Alternatives for *evaluatortype* are `"d0"`, `"d1"`, `"d2"`, `"d1debug"`, `"d2debug"`, `"gf0"`, `"gf1"`, `"gf2"`, `"gf1debug"`, and `"gf2debug"`. The default is `"d0"` if you do not invoke this function.

`optimize_init_evaluator(`*S*`)` returns a pointer to the function that has been set.

`optimize_init_evaluatortype(`*S*`)` returns the *evaluatortype* currently set.

optimize_init_negH( )

`optimize_init_negH(`*S*`,` { `"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"`.

optimize_init_params( )

void `optimize_init_params(`*S*`,` *real rowvector initialvalues*`)`

real rowvector `optimize_init_params(`*S*`)`

`optimize_init_params(`*S*`,` *initialvalues*`)` sets the values of p to be used at the start of the first iteration. Use of this function is required.

`optimize_init_params(`*S*`)` returns the initial values that will be (or were) used.

optimize_init_nmsimplexdeltas( )

void	`optimize_init_nmsimplexdeltas(`*S*`,` *real rowvector delta*`)`
real rowvector	`optimize_init_nmsimplexdeltas(`*S*`)`

`optimize_init_nmsimplexdeltas(`*S*`,` *delta*`)` sets the values of *delta* to be used, along with the initial parameters, to build the simplex required by technique `"nm"` (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*, which is set by `optimize_init_conv_ptol()`. The initial simplex will be $\{p, p + (d_1, 0), \ldots, 0, p + (0, d_2, 0, \ldots, 0), \ldots, p + (0, 0, \ldots, 0, d_k)\}$.

`optimize_init_nmsimplexdeltas(`*S*`)` returns the deltas that will be (or were) used.

optimize_init_argument( ) and optimize_init_narguments( )

void	`optimize_init_argument(`*S*`,` *real scalar k*`,` *X*`)`
void	`optimize_init_narguments(`*S*`,` *real scalar K*`)`
pointer scalar	`optimize_init_argument(`*S*`,` *real scalar k*`)`
real scalar	`optimize_init_narguments(`*S*`)`

`optimize_init_argument(`*S*`,` *k*`,` *X*`)` sets the *k*th extra argument of the evaluator function to be *X*, where *k* can only 1, 2, 3, ..., 9. *X* can be anything, including a view matrix or even a pointer to a function. No copy of *X* is made; it is a pointer to *X* that is stored, so any changes you make to *X* between setting it and *X* being used will be reflected in what is passed to the evaluator function.

`optimize_init_narguments(`*S*`,` *K*`)` sets the number of extra arguments to be passed to the evaluator function. This function is useless and included only for completeness. The number of extra arguments is automatically set as you use `optimize_init_argument()`.

`optimize_init_argument(`*S*`)` returns a pointer to the object that was previously set.

`optimize_init_nargs(`*S*`)` returns the number of extra arguments that are passed to the evaluator function.

optimize_init_cluster( )

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

optimize_init_colstripe( )

`optimize_init_colstripe(`*S* [`, stripe`]`)` sets the string matrix to be associated with the parameter vector. See `matrix colnames` in [P] **matrix rownames**.

optimize_init_technique( )

void	`optimize_init_technique(`*S*`,` *string scalar technique*`)`
string scalar	`optimize_init_technique(`*S*`)`

`optimize_init_technique(`*S*`,` *technique*`)` sets the optimization technique to be used. Current choices are

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

The default is `"nr"`.

`optimize_init_technique(`*S*`)` returns the technique currently set.

Aside: All techniques require `optimize_init_params()` be set. Technique `"nm"` also requires that `optimize_init_nmsimplexdeltas()` be set. Parameters (and delta) can be set before or after the technique is set.

You can switch between `"nr"`, `"dfp"`, `"bfgs"`, and `"bhhh"` by specifying two or more of them in a space-separated list. By default, `optimize()` 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 `optimize_init_technique(`*M*`, "bhhh 10 nr 1000")` requests that `optimize()` 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.

optimize_init_singularHmethod( )

void	`optimize_init_singularHmethod(`*S*`,` *string scalar method*`)`
string scalar	`optimize_init_singularHmethod(`*S*`)`

`optimize_init_singularHmethod(`*S*`,` *method*`)` specifies what the optimizer should do when, at an iteration step, it finds that *H* is singular. Current choices are

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

The default is `"m-marquardt"` if not set;
`"hybrid"` is equivalent to `ml`'s `difficult` option; see [R] **ml**.

`optimize_init_technique(`*S*`)` returns the *method* currently set.

optimize_init_conv_maxiter( )

void	`optimize_init_conv_maxiter(`*S*`,` *real scalar max*`)`
real scalar	`optimize_init_conv_maxiter(`*S*`)`

`optimize_init_conv_maxiter(`*S*, *max*`)` sets the maximum number of iterations to be performed before `optimization()` is stopped; results are posted to `optimize_result_*()` just as if convergence were achieved, but `optimize_result_converged()` is set to 0. The default *max* if not set is `c(maxiter)`, which is probably 16,000; type `creturn list` in Stata to determine the current default value.

`optimize_init_conv_maxiter(`*S*`)` returns the *max* currently set.

optimize_init_conv_warning( )

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

optimize_init_conv_ptol( ), . . . _vtol( ), . . . _nrtol( )

void	`optimize_init_conv_ptol(`*S*`,` *real scalar ptol*`)`
void	`optimize_init_conv_vtol(`*S*`,` *real scalar vtol*`)`
void	`optimize_init_conv_nrtol(`*S*`,` *real scalar nrtol*`)`
real scalar	`optimize_init_conv_ptol(`*S*`)`
real scalar	`optimize_init_conv_vtol(`*S*`)`
real scalar	`optimize_init_conv_nrtol(`*S*`)`

The two-argument form of these functions set the tolerances that control `optimize()`’s convergence criterion. `optimize()` performs iterations until the convergence criterion is met or until the number of iterations exceeds `optimize_init_conv_maxiter()`. When the convergence criterion is met, `optimize_result_converged()` is set to 1. The default values of *ptol*, *vtol*, and *nrtol* are 1e–6, 1e–7, and 1e–5, respectively.

The single-argument form of these functions return the current values of *ptol*, *vtol*, and *nrtol*.

Optimization criterion: In all cases except `optimize_init_technique(`*S*`)=="nm"`, i.e., in all cases except Nelder–Mead, i.e., in all cases of derivative-based maximization, the optimization criterion is defined as follows:

Define

C_ptol:	`mreldif(`*p*, *p_prior*`)` $< ptol$
C_vtol:	`reldif(`*v*, *v_prior*`)` $< vtol$
C_nrtol:	$g *$ `invsym(` $-H$`)` $* g' < nrtol$
C_concave:	$-H$ is positive semidefinite

The above definitions apply for maximization. For minimization, think of it as maximization of $-f(p)$. `optimize()` declares convergence when

$(C\_ptol \mid C\_vtol)$ & $C\_concave$ & C_nrtol

For `optimize_init_technique(`*S*`)=="nm"` (Nelder–Mead), the criterion is defined as follows:

Let *R* be the minimum and maximum values on the simplex and define

C_ptol: `mreldif(`vertices of *R*`)` < *ptol*

C_vtol: `reldif(`*R*`)` < *vtol*

`optimize()` declares successful convergence when

$C\_ptol \mid C\_vtol$

optimize_init_conv_ignorenrtol( )

`optimize_init_conv_ignorenrtol(`*S*`,` { `"off"` | `"on"` }`)` sets whether *C_nrtol* should simply be treated as true in all cases, which in effects removes the *nrtol* criterion from the convergence rule. The default is `"off"`.

optimize_init_iterid( )

void `optimize_init_iterid(`*S*`,` *string scalar id*`)`

string scalar `optimize_init_iterid(`*S*`)`

By default, `optimize()` shows an iteration log, a line of which looks like

```
Iteration 1: f(p) = 5.7641349
```

See *optimize_init_tracelevel()* below.

`optimize_init_iterid(`*S*`,` *id*`)` sets the string used to label the iteration in the iteration log. The default is `"Iteration"`.

`optimize_init_iterid(`*S*`)` returns the *id* currently in use.

optimize_init_valueid( )

void `optimize_init_valueid(`*S*`,` *string scalar id*`)`

string scalar `optimize_init_valueid(`*S*`)`

By default, `optimize()` shows an iteration log, a line of which looks like

```
Iteration 1: f(p) = 5.7641349
```

See *optimize_init_tracelevel()* below.

`optimize_init_valueid(`*S*`,` *id*`)` sets the string used to identify the value. The default is `"f(p)"`.

`optimize_init_valueid(`*S*`)` returns the *id* currently in use.

optimize_init_tracelevel( )

void `optimize_init_tracelevel(`*S*`,` *string scalar tracelevel*`)`

string scalar `optimize_init_tracelevel(`*S*`)` `optimize_init_tracelevel(`*S*`,` *tracelevel*`)` sets what is displayed in the iteration log. Allowed values of *tracelevel* are

tracelevel	To be displayed each iteration
`"none"`	nothing (suppress the log)
`"value"`	function value
`"tolerance"`	previous + convergence values
`"step"`	previous + stepping information
`"paramdiffs"`	previous + parameter relative differences
`"params"`	previous + parameter values
`"gradient"`	previous + gradient vector
`"hessian"`	previous + Hessian matrix

The default is `"value"` if not reset.

`optimize_init_tracelevel(`*S*`)` returns the value of *tracelevel* currently set.

optimize_init_trace_dots( ), . . . _value( ), . . . _tol( ), . . . _step( ), . . . _paramdiffs( ), . . . _params( ), . . . _gradient( ), . . . _Hessian( )

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

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

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

`optimize_init_trace_step(`*S*`,` { `"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"`.

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

`optimize_init_trace_params(`*S*`,` { `"off"` | `"on"` }`)` displays the parameters at the start of each iteration. The default is `"off"`.

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

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

optimize_init_evaluations( )

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

optimize_init_constraints()

```
void        optimize_init_constraints(S, real matrix Cc)

real matrix optimize_init_constraints(S)
```

nc linear constraints may be imposed on the *np* parameters in *p* according to $Cp' = c$, C: $nc \times np$ and c: $nc \times 1$. For instance, if there are four parameters and you wish to impose the single constraint $p_1 = p_2$, then $C = (1, -1, 0, 0)$ and $c = (0)$. If you wish to add the constraint $p_4 = 2$, then $C = (1, -1, 0, 0\backslash 0, 0, 0, 1)$ and $c = (0\backslash 2)$.

`optimize_init_constraints(`*S*`,` *Cc*`)` allows you to impose such constraints where $Cc = (C, c)$. Use of this function is optional. If no constraints have been set, then *Cc* is $0 \times (np + 1)$.

`optimize_init_constraints(`*S*`)` returns the current *Cc* matrix.

optimize_init_verbose()

```
void        optimize_init_verbose(S, real scalar verbose)

real scalar optimize_init_verbose(S)
```

`optimize_init_verbose(`*S*`,` *verbose*`)` sets whether error messages that arise during the execution of `optimize()` or `_optimize()` are to be displayed. *verbose*=1 means that they are; 0 means that they are not. The default is 1. Setting *verbose* to 0 is of interest only to users of `_optimize()`. If you wish to suppress all output, code

```
optimize_init_verbose(S, 0)
optimize_init_tracelevel(S, "none")
```

`optimize_init_verbose(`*S*`)` returns the current value of *verbose*.

optimize()

```
real rowvector optimize(S)
```

`optimize(`*S*`)` invokes the optimization process and returns the resulting parameter vector. If something goes wrong, `optimize()` aborts with error.

Before you can invoke `optimize()`, you must have defined your evaluator function *evaluator*`()` and you must have set initial values:

```
S = optimize_init()
optimize_init_evaluator(S, &evaluator())
optimize_init_params(S, (...))
```

The above assumes that your evaluator function is d0. Often you will also have coded

```
optimize_init_evaluatortype(S, "..."))
```

and you may have coded other `optimize_init_*()` functions as well.

Once optimize() completes, you may use the optimize_result_*() functions. You may also continue to use the optimize_init_*() functions to access initial settings, and you may use them to change settings and restart optimization (i.e., invoke optimize() again) if you wish. If you do that, you will usually want to use the resulting parameter values from the first round of optimization as initial values for the second. If so, do not forget to code

```
        optimize_init_params(S, optimize_result_params(S))
```

_optimize( )

```
        real scalar _optimize(S)
```

_optimize(*S*) performs the same actions as optimize(*S*) except that, rather than returning the resulting parameter vector, _optimize() returns a real scalar and, rather than aborting if numerical issues arise, _optimize() returns a nonzero value. _optimize() returns 0 if all went well. The returned value is called an error code.

optimize() returns the resulting parameter vector *p*. It can work that way because optimization must have gone well. Had it not, optimize() would have aborted execution.

_optimize() returns an error code. If it is 0, optimization went well and you can obtain the parameter vector by using optimize_result_param(). If optimization did not go well, you can use the error code to diagnose what went wrong and take the appropriate action.

Thus, _optimize(*S*) is an alternative to optimize(*S*). Both functions do the same thing. The difference is what happens when there are numerical difficulties.

optimize() and _optimize() work around most numerical difficulties. For instance, the evaluator function you write is allowed to return *v* equal to missing if it cannot calculate the *f*() at the current values of *p*. If that happens during optimization, optimize() and _optimize() will back up to the last value that worked and choose a different direction. optimize(), however, cannot tolerate that happening with the initial values of the parameters because optimize() has no value to back up to. optimize() issues an error message and aborts, meaning that execution is stopped. There can be advantages in that. The calling program need not include complicated code for such instances, figuring that stopping is good enough because a human will know to address the problem.

_optimize(), however, does not stop execution. Rather than aborting, _optimize() returns a nonzero value to the caller, identifying what went wrong.

Programmers implementing advanced systems will want to use _optimize() instead of optimize(). Everybody else should use optimize().

Programmers using _optimize() will also be interested in the functions

```
        optimize_init_verbose()
        optimize_result_errorcode()
        optimize_result_errortext()
        optimize_result_returncode()
```

If you perform optimization by using _optimize(), the behavior of all optimize_result_*() functions is altered. The usual behavior is that, if calculation is required and numerical problems arise, the functions abort with error. After _optimize(), however, a properly dimensioned missing result is returned and optimize_result_errorcode() and optimize_result_errortext() are set appropriately.

The error codes returned by `_optimize()` are listed under the heading *optimize_result_errorcode()* below.

optimize_evaluate( )

```
void optimize_evaluate(S)
```

`optimize_evaluate(S)` evaluates *f*() at `optimize_init_params()` and posts results to `optimize_result_*()` just as if optimization had been performed, meaning that all `optimize_result_*()` functions are available for use. `optimize_result_converged()` is set to 1.

The setup for running this function is the same as for running `optimize()`:

```
S = optimize_init()
optimize_init_evaluator(S, &evaluator())
optimize_init_params(S, (...))
```

Usually, you will have also coded

```
optimize_init_evaluatortype(S, ...))
```

The other `optimize_init_*()` settings do not matter.

_optimize_evaluate( )

```
real scalar _optimize_evaluate(S)
```

The relationship between `_optimize_evaluate()` and `optimize_evaluate()` is the same as that between `_optimize()` and `optimize()`; see *_optimize( )*.

`_optimize_evaluate()` returns an error code.

optimize_result_params( )

```
real rowvector optimize_result_params(S)
```

`optimize_result_params(S)` returns the resulting parameter values. These are the same values that were returned by `optimize()` itself. There is no computational cost to accessing the results, so rather than coding

```
p = optimize(S)
```

if you find it more convenient to code

```
(void) optimize(S)
...
p = optimize_result_params(S)
```

then do so.

optimize_result_value( ) and optimize_result_value0( )

real scalar `optimize_result_value(`*S*`)`

real scalar `optimize_result_value0(`*S*`)`

`optimize_result_value(`*S*`)` returns the value of *f*() evaluated at *p* equal to `optimize_result_param()`.

`optimize_result_value0(`*S*`)` returns the value of *f*() evaluated at *p* equal to `optimize_init_param()`.

These functions may be called regardless of the evaluator or technique used.

optimize_result_gradient( )

real rowvector `optimize_result_gradient(`*S*`)`

`optimize_result_gradient(`*S*`)` returns the value of the gradient vector evaluated at *p* equal to `optimize_result_param()`. This function may be called regardless of the evaluator or technique used.

optimize_result_scores( )

real matrix `optimize_result_scores(`*S*`)`

`optimize_result_scores(`*S*`)` returns the value of the scores evaluated at *p* equal to `optimize_result_param()`. This function may be called only if a type `gf` evaluator is used, but regardless of the technique used.

optimize_result_Hessian( )

real matrix `optimize_result_Hessian(`*S*`)`

`optimize_result_Hessian(`*S*`)` returns the value of the Hessian matrix evaluated at *p* equal to `optimize_result_param()`. This function may be called regardless of the evaluator or technique used.

optimize_result_V( ) and optimize_result_Vtype( )

real matrix `optimize_result_V(`*S*`)`

string scalar `optimize_result_Vtype(`*S*`)`

`optimize_result_V(`*S*`)` returns `optimize_result_V_oim(`*S*`)`
or `optimize_result_V_opg(`*S*`)`, depending on which is the natural conjugate for the optimization technique used. If there is no natural conjugate, `optimize_result_V_oim(`*S*`)` is returned.

`optimize_result_Vtype(`*S*`)` returns `"oim"` or `"opg"`.

optimize_result_V_oim( ), . . . _opg( ), . . . _robust( )

real matrix `optimize_result_V_oim(`*S*`)`

real matrix `optimize_result_V_opg(`*S*`)`

real matrix `optimize_result_V_robust(`*S*`)`

These functions return the variance matrix of *p* evaluated at *p* equal to `optimize_result_param()`. These functions are relevant only for maximization of log-likelihood functions but may be called in any context, including minimization.

`optimize_result_V_oim(`*S*`)` returns `invsym(`$-H$`)`, which is the variance matrix obtained from the observed information matrix. For minimization, returned is `invsym(`*H*`)`.

`optimize_result_V_opg(`*S*`)` returns `invsym(`$S'S$`)`, where *S* is the $N \times np$ matrix of scores. This is known as the variance matrix obtained from the outer product of the gradients. `optimize_result_V_opg()` is available only when the evaluator function is type `gf`, but regardless of the technique used.

`optimize_result_V_robust(`*S*`)` returns $H *$ `invsym(`$S'S$`)` $* H$, which is the robust estimate of variance, also known as the sandwich estimator of variance. `optimize_result_V_robust()` is available only when the evaluator function is type `gf`, but regardless of the technique used.

optimize_result_iterations( )

real scalar `optimize_result_iterations(`*S*`)`

`optimize_result_iterations(`*S*`)` returns the number of iterations used in obtaining results.

optimize_result_converged( )

real scalar `optimize_result_converged(`*S*`)`

`optimize_result_converged(`*S*`)` returns 1 if results converged and 0 otherwise.
See *optimize_init_conv_ptol()* for the definition of convergence.

optimize_result_iterationlog( )

real colvector `optimize_result_iterationlog(`*S*`)`

`optimize_result_iterationlog(`*S*`)` returns a column vector of the values of *f*() at the start of the final 20 iterations, or, if there were fewer, however many iterations there were. Returned vector is min(`optimize_result_iterations()`, 20) $\times$ 1.

optimize_result_evaluations( )

`optimize_result_evaluations(`*S*`)` returns a 1×3 real rowvector containing the number of times the evaluator was called, assuming `optimize_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 `optimize_init_evaluations()` was set to `off`, returned is $(0, 0, 0)$.

optimize_result_errorcode( ), . . . _errortext( ), and . . . _returncode( )

real scalar	`optimize_result_errorcode(`*S*`)`
string scalar	`optimize_result_errortext(`*S*`)`
real scalar	`optimize_result_returncode(`*S*`)`

These functions are for use after `_optimize()`.

`optimize_result_errorcode(`*S*`)` returns the error code of `_optimize()`, `_optimize_evaluate()`, or the last `optimize_result_*()` run after either of the first two functions. The value will be zero if there were no errors. The error codes are listed directly below.

`optimize_result_errortext(`*S*`)` returns a string containing the error message corresponding to the error code. If the error code is zero, the string will be `""`.

`optimize_result_returncode(`*S*`)` returns the Stata return code corresponding to the error code. The mapping is listed directly below.

In advanced code, these functions might be used as

```
(void) _optimize(S)
...
if (ec = optimize_result_code(S)) {
        errprintf("{p}\n")
        errprintf("%s\n", optimize_result_errortext(S))
        errprintf("{p_end}\n")
        exit(optimize_result_returncode(S))
        /*NOTREACHED*/
}
```

(*Continued on next page*)

The error codes and their corresponding Stata return codes are

Error code	Return code	Error text
1	1400	initial values not feasible
2	412	redundant or inconsistent constraints
3	430	missing values returned by evaluator
4	430	Hessian is not positive semidefinite *or* Hessian is not negative semidefinite
5	430	could not calculate numerical derivatives—discontinuous region with missing values encountered
6	430	could not calculate numerical derivatives—flat or discontinuous region encountered
7	430	could not calculate improvement—discontinuous region encountered
8	430	could not calculate improvement—flat region encountered
10	111	technique unknown
11	111	incompatible combination of techniques
12	111	singular H method unknown
13	198	matrix stripe invalid for parameter vector
14	198	negative convergence tolerance values are not allowed
15	503	invalid starting values
17	111	simplex delta required
18	3499	simplex delta not conformable with parameter vector
19	198	simplex delta value too small (must be greater than 10 × ptol)
20	198	evaluator type requires the nr technique
23	198	evaluator type not allowed with bhhh technique
24	111	evaluator functions required
25	198	starting values for parameters required
26	198	missing parameter values not allowed

NOTES: (1) Error 1 can occur only when evaluating *f*( ) at initial parameters.

(2) Error 2 can occur only if constraints are specified.

(3) Error 3 can occur only if the technique is `"nm"`.

(4) Error 9 can occur only if technique is `"bfgs"` or `"dfp"`.

optimize_query( )

void `optimize_query(`*S*`)`

`optimize_query(`*S*`)` displays a report on all `optimize_init_*()` and `optimize_result*()` values. `optimize_query()` may be used before or after `optimize()` and is useful when using `optimize()` interactively or when debugging a program that calls `optimize()` or `_optimize()`.

Conformability

All functions have 1×1 inputs and have 1×1 or *void* outputs except the following:

`optimize_init_params(`*S*`,` *initialvalues*`)`:

S:	*transmorphic*
initialvalues:	$1 \times np$
result:	*void*

`optimize_init_params(`*S*`)`:

S:	*transmorphic*
result:	$1 \times np$

`optimize_init_argument(`*S*`,` *k*`,` *X*`)`:

S:	*transmorphic*
k:	1×1
X:	*anything*
result:	*void*

`optimize_init_nmsimplexdeltas(`*S*`,` *delta*`)`:

S:	*transmorphic*
delta:	$1 \times np$
result:	*void*

`optimize_init_nmsimplexdeltas(`*S*`)`:

S:	*transmorphic*
result:	$1 \times np$

`optimize_init_constraints(`*S*`,` *Cc*`)`:

S:	*transmorphic*
Cc:	$nc \times (np + 1)$
result:	*void*

`optimize_init_constraints(`*S*`)`:

S:	*transmorphic*
result:	$nc \times (np + 1)$

`optimize(`*S*`)`:

S:	*transmorphic*
result:	$1 \times np$

`optimize_result_params(`*S*`)`:

S:	*transmorphic*
result:	$1 \times np$

`optimize_result_gradient(`*S*`)`, `optimize_result_evaluations(`*S*`)`:

S:	*transmorphic*
result:	$1 \times np$

`optimize_result_scores(`*S*`)`:

S:	*transmorphic*
result:	$N \times np$

`optimize_result_Hessian(`*S*`)`:

S:	*transmorphic*
result:	$np \times np$

`optimize_result_V(`*S*`)`, `optimize_result_V_oim(`*S*`)`, `optimize_result_V_opg(`*S*`)`, `optimize_result_V_robust(`*S*`)`:

S:	*transmorphic*
result:	$np \times np$

`optimize_result_iterationlog(`*S*`)`:

S:	*transmorphic*
result:	$L \times 1$, $L \leq 20$

For `optimize_init_cluster(`*S*`, `*c*`)` and `optimize_init_colstripe(`*S*`)`, see *Syntax* above.

Diagnostics

All functions abort with error when used incorrectly.

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

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

The `optimize_result_*()` functions abort with error if they run into numerical difficulties when called after `optimize()` or `optimize_evaluate()`. They do not abort when run after `_optimize()` or `_optimize_evaluate()`. They instead return a properly dimensioned missing result and set `optimize_result_errorcode()` and `optimize_result_errortext()`.

The formula $x_{i+1} = x_i - f(x_i)/f'(x_i)$ and its generalizations for solving $f(x) = 0$ (and its generalizations) are known variously as Newton's method or the Newton–Raphson method. The real history is more complicated than these names imply and has roots in the earlier work of Arabic algebraists and François Viète.

Newton's first formulation dating from about 1669 refers only to solution of polynomial equations and does not use calculus. In his *Philosophiae Naturalis Principia Mathematica*, first published in 1687, the method is used, but not obviously, to solve a nonpolynomial equation. Raphson's work, first published in 1690, also concerns polynomial equations, and proceeds algebraically without using calculus, but lays more stress on iterative calculation and so is closer to present ideas. It was not until 1740 that Thomas Simpson published a more general version explicitly formulated in calculus terms that was applied to both polynomial and nonpolynomial equations and to both single equations and systems of equations. Simpson's work was in turn overlooked in influential later accounts by Lagrange and Fourier, but his contribution also deserves recognition.

Isaac Newton (1643–1727) was an English mathematician, astronomer, physicist, natural philosopher, alchemist, theologian, biblical scholar, historian, politician and civil servant. He was born in Lincolnshire, according to the calendar then in use, in 1642, and studied there and at the University of Cambridge, where he was a fellow of Trinity College and elected Lucasian Professor in 1669. Newton demonstrated the generalized binomial theorem, did major work on power series, and deserves credit with Gottfried Leibniz for the development of calculus: during his lifetime and long afterward, that fact was obscured by a bitter and protracted priority dispute. He described universal gravitation and the laws of motion central to classical mechanics and showed that the motions of objects on Earth and beyond are subject to the same laws. Newton invented the reflecting telescope and developed a theory of color that was based on the fact that a prism splits white light into a visible spectrum. He also studied cooling and the speed of sound and proposed a theory of the origin of stars. Much of his later life was spent in London, including brief spells as member of Parliament and longer periods as master of the Mint and president of the Royal Society. He was knighted in 1705. Although undoubtedly one of the greatest mathematical and scientific geniuses of all time, Newton was also outstandingly contradictory, secretive, and quarrelsome.

Joseph Raphson (1648–1715) was an English or possibly Irish mathematician. No exact dates are known for his birth or death years. He appears to have been largely self-taught and was awarded a degree by the University of Cambridge after the publication of his most notable work, *Analysis Aequationum Universalis* (1690), and his election as a fellow of the Royal Society.

Thomas Simpson (1710–1761) was born in Market Bosworth, Leicestershire, England. He received little formal education and was self-taught in mathematics. Simpson moved to London and worked as a teacher in London coffee houses (as was De Moivre) and then at the Royal Military Academy at Woolwich. He published texts on calculus, astronomy, and probability and is best remembered for his work on interpolation and numerical methods of integration. However, what is now known as "Simpson's rule" was known earlier to Newton. He was also a fellow of the Royal Society.

References

Berndt, E. K., B. H. Hall, R. E. Hall, and J. A. Hausman. 1974. Estimation and inference in nonlinear structural models. *Annals of Economic and Social Measurement* 3/4: 653–665.

Davidon, W. C. 1959. Variable metric method for minimization. Technical Report ANL-5990, Argonne National Laboratory, U.S. Department of Energy, Argonne, IL.

Fletcher, R. 1970. A new approach to variable metric algorithms. *Computer Journal* 13: 317–322.

——. 1987. *Practical Methods of Optimization*. 2nd ed. New York: Wiley.

Fletcher, R., and M. J. D. Powell. 1963. A rapidly convergent descent method for minimization. *Computer Journal* 6: 163–168.

Gleick, J. 2003. *Isaac Newton*. New York: Pantheon.

Goldfarb, D. 1970. A family of variable-metric methods derived by variational means. *Mathematics of Computation* 24: 23–26.

Marquardt, D. W. 1963. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics* 11: 431–441.

Nelder, J. A., and R. Mead. 1965. A simplex method for function minimization. *Computer Journal* 7: 308–313.

Newton, I. 1671. *De methodis fluxionum et serierum infinitorum*. Translated by john colson as *the method of fluxions and infinite series* ed. London: Henry Wood Fall, 1736.

Raphson, J. 1690. *Analysis Aequationum Universalis*. Londioni: Prostant venales apud Abelem Swalle.

Shanno, D. F. 1970. Conditioning of quasi-Newton methods for function minimization. *Mathematics of Computation* 24: 647–656.

Westfall, R. S. 1980. *Never at Rest: A Biography of Isaac Newton*. Cambridge: Cambridge University Press.

Ypma, T. J. 1995. Historical development of the Newton–Raphson method. *SIAM Review* 37: 531–551.

Also see

[M-5] **moptimize( )** — Model optimization

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

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