Title

stata.com

deriv( ) — Numerical derivatives

Syntax

```
                  D = deriv_init()

(varies)          deriv_init_evaluator(D [ , &function() ])
(varies)          deriv_init_evaluatortype(D [ , evaluatortype ])
(varies)          deriv_init_params(D [ , real rowvector parameters ])
(varies)          deriv_init_argument(D, real scalar k [ , X ])
(varies)          deriv_init_narguments(D [ , real scalar K ])
(varies)          deriv_init_weights(D [ , real colvector weights ])
(varies)          deriv_init_h(D [ , real rowvector h ])
(varies)          deriv_init_scale(D [ , real matrix scale ])
(varies)          deriv_init_bounds(D [ , real rowvector minmax ])
(varies)          deriv_init_search(D [ , search ])
(varies)          deriv_init_verbose(D [ , {"on" | "off"} ])

(varies)          deriv(D, {0 | 1 | 2})
real scalar       _deriv(D, {0 | 1 | 2})

real scalar       deriv_result_value(D)
real vector       deriv_result_values(D)
void              _deriv_result_values(D, v)
real rowvector    deriv_result_gradient(D)
void              _deriv_result_gradient(D, g)
real matrix       deriv_result_scores(D)
void              _deriv_result_scores(D, S)
real matrix       deriv_result_Jacobian(D)
void              _deriv_result_Jacobian(D, J)
real matrix       deriv_result_Hessian(D)
```

void	`_deriv_result_Hessian(`*D*`, `*H*`)`
real rowvector	`deriv_result_h(`*D*`)`
real matrix	`deriv_result_scale(`*D*`)`
real matrix	`deriv_result_delta(`*D*`)`
real scalar	`deriv_result_errorcode(`*D*`)`
string scalar	`deriv_result_errortext(`*D*`)`
real scalar	`deriv_result_returncode(`*D*`)`
void	`deriv_query(`*D*`)`

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

`transmorphic` *D*

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

evaluatortype	Description
`"d"`	*function*`()` returns *scalar* value
`"v"`	*function*`()` returns *colvector* value
`"t"`	*function*`()` returns *rowvector* value

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

and where *search* optionally specified in `deriv_init_search()` is

search	Description
`"interpolate"`	use linear and quadratic interpolation to search for an optimal delta
`"bracket"`	use a bracketed quadratic formula to search for an optimal delta
`"off"`	do not search for an optimal delta

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

Description

These functions compute derivatives of the real function $f(p)$ at the real parameter values p.

`deriv_init()` begins the definition of a problem and returns *D*, a problem-description handle set that contains default values.

The `deriv_init_*(`*D*`, ...)` functions then allow you to modify those defaults. You use these functions to describe your particular problem: to set the identity of function $f()$, to set parameter values, and the like.

`deriv(`*D*`, `*todo*`)` then computes derivatives depending upon the value of *todo*.

`deriv(`*D*`, 0)` returns the function value without computing derivatives.

`deriv(`*D*`, 1)` returns the first derivatives, also known as the gradient vector for scalar-valued functions (type `d` and `v`) or the Jacobian matrix for vector-valued functions (type `t`).

`deriv(`*D*`, 2)` returns the matrix of second derivatives, also known as the Hessian matrix; the gradient vector is also computed. This syntax is not allowed for type `t` evaluators.

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

Usually you would stop there. In other cases, you could compute derivatives at other parameter values:

`deriv_init_params(`*D*`, ` *p_alt*`)`

`deriv(`*D*`, ` *todo*`)`

Aside: The `deriv_init_*(`*D*`, ...)` 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 `deriv_init_params()` is

void `deriv_init_params(`*D*`, ` *real rowvector parameters*`)`

real rowvector `deriv_init_params(`*D*`)`

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

All the `deriv_init_*(`*D*`, ...)` functions work the same way.

Remarks and examples

stata.com

Remarks are presented under the following headings:

First example

The derivative functions may be used interactively.

Below we use the functions to compute $f'(x)$ at $x = 0$, where the function is

$$f(x) = \exp(-x^2 + x - 3)$$

```
: void myeval(x, y)
> {
>         y = exp(-x^2 + x - 3)
> }
: D = deriv_init()
: deriv_init_evaluator(D, &myeval())
: deriv_init_params(D, 0)
: dydx = deriv(D, 1)
: dydx
  .0497870683
: exp(-3)
  .0497870684
```

The derivative, given the above function, is $f'(x) = (-2\times x+1)\times\exp(-x^2+x-3)$, so $f'(0) = \exp(-3)$.

Notation and formulas

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

Differentiate $v = f(p)$ with respect to p, where

v: a scalar
p: $1 \times np$

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

g: $1 \times np$

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

H: $np \times np$

`deriv()` can also work with vector-valued functions. Here is the notation for vector-valued functions:

Differentiate $v = f(p)$ with respect to p, where

v: $1 \times nv$, a vector
p: $1 \times np$

The Jacobian matrix is $J = f'(p) = df/dp$, where

J: $nv \times np$

and where

$J[i,j] = dv[i]/dp[j]$

Second-order derivatives are not computed by `deriv()` when used with vector-valued functions.

`deriv()` uses the following formula for computing the numerical derivative of $f()$ at p

$$f'(p) = \frac{f(p+d) - f(p-d)}{2d}$$

where we refer to d as the delta used for computing numerical derivatives. To search for an optimal delta, we decompose d into two parts.

$$d = h \times scale$$

By default, h is a fixed value that depends on the parameter value.

$$h = \texttt{(abs(}p\texttt{)+1e-3)*1e-3}$$

`deriv()` searches for a value of *scale* that will result in an optimal numerical derivative, that is, one where d is as small as possible subject to the constraint that $f(x+d) - f(x-d)$ will be calculated to at least half the accuracy of a double-precision number. This is accomplished by searching for *scale* such that $f(x+d)$ and $f(x-d)$ fall between *v0* and *v1*, where

v0 = `(abs(`$f(x)$`)+1e-8)*1e-8`
v1 = `(abs(`$f(x)$`)+1e-7)*1e-7`

Use `deriv_init_h()` to change the default h values. Use `deriv_init_scale()` to change the default initial *scale* values. Use `deriv_init_bounds()` to change the default bounds (`1e-8`, `1e-7`) used for determining the optimal *scale*.

Type d evaluators

You must write an evaluator function to calculate $f()$ before you can use the derivative 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(x,  y)
{
        y = exp(-x^2 + x - 3)
}
```

All type `d` evaluators open the same way,

void evaluator(*x*, *y*)

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(*b*, *s*)

To avoid favoritism, we will write them as

void evaluator(*p*, *v*)

That is, we will think in terms of computing the derivative of $v = f(p)$ with respect to the elements of p.

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

---

void evaluator(*real rowvector p*, *v*)

where v is the value to be returned:

v: *real scalar*

evaluator() is to fill in v given the values in p.

evaluator() may return v = `.` if $f()$ cannot be evaluated at p.

Example of a type d evaluator

We wish to compute the gradient of the following function at $p_1 = 1$ and $p_2 = 2$:

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

Our numerical solution to the problem is

```
: void eval_d(p, v)
> {
>         v = exp(-p[1]^2 - p[2]^2 - p[1]*p[2] + p[1] - p[2] - 3)
> }
: D = deriv_init()
: deriv_init_evaluator(D, &eval_d())
```

```
        : deriv_init_params(D, (1,2))
        : grad = deriv(D, 1)
        : grad
                          1              2
            +-----------------------------+
          1 |  -.0000501051   -.0001002102 |
            +-----------------------------+
        : (-2*1 - 2 + 1)*exp(-1^2 - 2^2 - 1*2 + 1 - 2 - 3)
          -.0000501051
        : (-2*2 - 1 - 1)*exp(-1^2 - 2^2 - 1*2 + 1 - 2 - 3)
          -.0001002102
```

For this problem, the elements of the gradient function are given by the following formulas, and we see that deriv() computed values that are in agreement with the analytical results (to the number of significant digits being displayed).

$$\frac{dv}{dp_1} = (-2p_1 - p_2 + 1)\exp(-p_1^2 - p_2^2 - p_1p_2 + p_1 - p_2 - 3)$$
$$\frac{dv}{dp_2} = (-2p_2 - p_1 - 1)\exp(-p_1^2 - p_2^2 - p_1p_2 + p_1 - p_2 - 3)$$

Type v evaluators

In some statistical applications, you will find type v evaluators more convenient to code than type d evaluators.

In statistical applications, one tends to think of a dataset of values arranged in matrix X, the rows of which are observations. The 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 $f(p)$ from which we would like to compute derivatives.

Type v evaluators are for such cases.

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

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

The code outline for type v evaluators is the same as those for d evaluators. All that differs is that v, which is a *real scalar* in the d case, is now a *real colvector* in the v case.

User-defined arguments

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

You can pass extra arguments to evaluators. The first line of all evaluators, regardless of type, is

void evaluator(*p*, *v*)

If you code

deriv_init_argument(*D*, 1, *X*)

the first line becomes

void evaluator(p, X, v)

If you code

```
deriv_init_argument(D, 1, X)
deriv_init_argument(D, 2, Y)
```

the first line becomes

void evaluator(p, X, Y, v)

and so on, up to nine extra arguments. That is, you can specify extra arguments to be passed to your function.

Example of a type v evaluator

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 compute the gradient vector and Hessian matrix associated with the log likelihood at some values of those parameters alpha and beta (*a* and *b* in what follows). The formula for the density of the beta distribution is

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

In our type v solution to this problem, we compute the gradient and Hessian at $a = 0.5$ and $b = 2$.

```
: void lnbetaden_v(p, x, lnf)
> {
>         a = p[1]
>         b = p[2]
>         lnf = lngamma(a+b) :- lngamma(a) :- lngamma(b) :+
>                 (a-1)*log(x) :+ (b-1)*log(1:-x)
> }
: D = deriv_init()
: deriv_init_evaluator(D, &lnbetaden_v())
: deriv_init_evaluatortype(D, "v")
: deriv_init_params(D, (0.5, 2))
: deriv_init_argument(D, 1, x)                    ← important
: deriv(D, 2)
[symmetric]
                 1              2
    +-------------------------------+
  1 |   -116.4988089                |
  2 |    8.724410052   -1.715062542 |
    +-------------------------------+

: deriv_result_gradient(D)
                 1              2
    +-------------------------------+
  1 |    15.12578465   -1.701917722 |
    +-------------------------------+
```

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 `deriv_init_argument(D, 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 `"v"`.

Type t evaluators

Type `t` evaluators are for when you need to compute the Jacobian matrix from a vector-valued function.

Type `t` evaluators are different from type `v` evaluators in that the resulting vector of values should not be summed. One example is when the function $f()$ performs a nonlinear transformation from the domain of p to the domain of v.

Example of a type t evaluator

Let's compute the Jacobian matrix for the following transformation:

$$v_1 = p_1 + p_2$$

$$v_2 = p_1 - p_2$$

Here is our numerical solution, evaluating the Jacobian at $p = (0, 0)$:

```
: void eval_t1(p, v)
> {
>         v = J(1,2,.)
>         v[1] = p[1] + p[2]
>         v[2] = p[1] - p[2]
> }
: D = deriv_init()
: deriv_init_evaluator(D, &eval_t1())
: deriv_init_evaluatortype(D, "t")
: deriv_init_params(D, (0,0))
: deriv(D, 1)
[symmetric]
        1    2
    ┌──────────┐
  1 │   1      │
  2 │   1   -1 │
    └──────────┘
```

Now let's compute the Jacobian matrix for a less trivial transformation:

$$v_1 = p_1^2$$

$$v_2 = p_1 p_2$$

Here is our numerical solution, evaluating the Jacobian at $p = (1, 2)$:

```
: void eval_t2(p, v)
> {
>         v = J(1,2,.)
>         v[1] = p[1]^2
>         v[2] = p[1] * p[2]
> }
: D = deriv_init()
: deriv_init_evaluator(D, &eval_t2())
: deriv_init_evaluatortype(D, "t")
: deriv_init_params(D, (1,2))
: deriv(D, 1)
                  1              2
    ┌──────────────────────────────┐
  1 │   1.999999998              0 │
  2 │             2              1 │
    └──────────────────────────────┘
```

Functions

deriv_init()

transmorphic `deriv_init()`

`deriv_init()` is used to begin a derivative problem. Store the returned result in a variable name of your choosing; we have used *D* in this documentation. You pass *D* as the first argument to the other `deriv*()` functions.

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

The query form of `deriv_init_*()` can be used before or after calling `deriv()`.

deriv_init_evaluator() and deriv_init_evaluatortype()

void `deriv_init_evaluator(`*D*`,` *pointer(function) scalar fptr*`)`

void `deriv_init_evaluatortype(`*D*`,` *evaluatortype*`)`

pointer(function) scalar `deriv_init_evaluator(`*D*`)`

string scalar `deriv_init_evaluatortype(`*D*`)`

`deriv_init_evaluator(`*D*`,` *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 `deriv_init_evaluator(`*D*`, &myfcn())`.

`deriv_init_evaluatortype(`*D*`,` *evaluatortype*`)` specifies the capabilities of the function that has been set using `deriv_init_evaluator()`. Alternatives for *evaluatortype* are `"d"`, `"v"`, and `"t"`. The default is `"d"` if you do not invoke this function.

`deriv_init_evaluator(`*D*`)` returns a pointer to the function that has been set.

`deriv_init_evaluatortype(`*D*`)` returns the evaluator type currently set.

deriv_init_argument() and deriv_init_narguments()

void `deriv_init_argument(`*D*`,` *real scalar k*`,` *X*`)`

void `deriv_init_narguments(`*D*`,` *real scalar K*`)`

pointer scalar `deriv_init_argument(`*D*`,` *real scalar k*`)`

real scalar `deriv_init_narguments(`*D*`)`

`deriv_init_argument(`*D*`,` *k*`,` *X*`)` sets the *k*th extra argument of the evaluator function to be *X*. *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.

`deriv_init_narguments(`*D*`,` *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 when you use `deriv_init_argument()`.

`deriv_init_argument(`*D*`,` *k*`)` returns a pointer to the object that was previously set.

`deriv_init_narguments(`*D*`)` returns the number of extra arguments that were passed to the evaluator function.

deriv_init_weights()

void `deriv_init_weights(`*D*`,` *real colvector weights*`)`

pointer scalar `deriv_init_weights(`*D*`)`

`deriv_init_weights(`*D*`,` *weights*`)` sets the weights used with type `v` evaluators to produce the function value. By default, `deriv()` with a type `v` evaluator uses `colsum(`*v*`)` to compute the function value. With weights, `deriv()` uses `cross(`*weights*`,` *v*`)`. *weights* must be row conformable with the column vector returned by the evaluator.

`deriv_init_weights(`*D*`)` returns a pointer to the weight vector that was previously set.

deriv_init_params()

void	`deriv_init_params(`*D*`,` *real rowvector params*`)`
real rowvector	`deriv_init_params(`*D*`)`

`deriv_init_params(`*D*`,` *params*`)` sets the parameter values at which the derivatives will be computed. Use of this function is required.

`deriv_init_params(`*D*`)` returns the parameter values at which the derivatives were computed.

Advanced init functions

The rest of the `deriv_init_*()` functions provide finer control of the numerical derivative taker.

deriv_init_h(), . . . _scale(), . . . _bounds(), . . . _search()

void	`deriv_init_h(`*D*`,` *real rowvector h*`)`
void	`deriv_init_scale(`*D*`,` *real rowvector s*`)`
void	`deriv_init_bounds(`*D*`,` *real rowvector minmax*`)`
void	`deriv_init_search(`*D*`,` *search*`)`
real rowvector	`deriv_init_h(`*D*`)`
real rowvector	`deriv_init_scale(`*D*`)`
real rowvector	`deriv_init_bounds(`*D*`)`
string scalar	`deriv_init_search(`*D*`)`

`deriv_init_h(`*D*`,` *h*`)` sets the *h* values used to compute numerical derivatives.

`deriv_init_scale(`*D*`,` *s*`)` sets the starting scale values used to compute numerical derivatives.

`deriv_init_bounds(`*D*`,` *minmax*`)` sets the minimum and maximum values used to search for optimal scale values. The default is *minmax* = `(1e-8, 1e-7)`.

`deriv_init_search(`*D*`, "interpolate")` causes `deriv()` to use linear and quadratic interpolation to search for an optimal delta for computing the numerical derivatives. This is the default search method.

`deriv_init_search(`*D*`, "bracket")` causes `deriv()` to use a bracketed quadratic formula to search for an optimal delta for computing the numerical derivatives.

`deriv_init_search(`*D*`, "off")` prevents `deriv()` from searching for an optimal delta.

`deriv_init_h(`*D*`)` returns the user-specified *h* values.

`deriv_init_scale(`*D*`)` returns the user-specified starting scale values.

`deriv_init_bounds(`*D*`)` returns the user-specified search bounds.

`deriv_init_search(`*D*`)` returns the currently set search method.

deriv_init_verbose()

```
void          deriv_init_verbose(D, verbose)

string scalar deriv_init_verbose(D)
```

`deriv_init_verbose(`*D*`,` *verbose*`)` sets whether error messages that arise during the execution of `deriv()` or `_deriv()` are to be displayed. Setting *verbose* to `"on"` means that they are displayed; `"off"` means that they are not displayed. The default is `"on"`. Setting *verbose* to `"off"` is of interest only to users of `_deriv()`.

`deriv_init_verbose(`*D*`)` returns the current value of *verbose*.

deriv()

```
(varies) deriv(D, todo)
```

`deriv(`*D*`,` *todo*`)` invokes the derivative process. If something goes wrong, `deriv()` aborts with error.

> `deriv(`*D*`, 0)` returns the function value without computing derivatives.
>
> `deriv(`*D*`, 1)` returns the gradient vector; the Hessian matrix is not computed.
>
> `deriv(`*D*`, 2)` returns the Hessian matrix; the gradient vector is also computed.

Before you can invoke `deriv()`, you must have defined your evaluator function, *evaluator*`()`, and you must have set the parameter values at which `deriv()` is to compute derivatives:

```
D = deriv_init()

deriv_init_evaluator(D, &evaluator())

deriv_init_params(D, (...))
```

The above assumes that your evaluator function is type `d`. If your evaluator function type is `v` (that is, it returns a column vector of values instead of a scalar value), you will also have coded

```
deriv_init_evaluatortype(D, "v")
```

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

Once `deriv()` completes, you may use the `deriv_result_*()` functions. You may also continue to use the `deriv_init_*()` functions to access initial settings, and you may use them to change settings and recompute derivatives (that is, invoke `deriv()` again) if you wish.

_deriv()

```
real scalar _deriv(D, todo)
```

`_deriv(`*D*`)` performs the same actions as `deriv(`*D*`)` except that, rather than returning the requested derivatives, `_deriv()` returns a real scalar and, rather than aborting if numerical issues arise, `_deriv()` returns a nonzero value. `_deriv()` returns 0 if all went well. The returned value is called an error code.

`deriv()` returns the requested result. It can work that way because the numerical derivative calculation must have gone well. Had it not, `deriv()` would have aborted execution.

`_deriv()` returns an error code. If it is 0, the numerical derivative calculation went well, and you can obtain the gradient vector by using `deriv_result_gradient()`. If things did not go well, you can use the error code to diagnose what went wrong and take the appropriate action.

Thus `_deriv(`*D*`)` is an alternative to `deriv(`*D*`)`. Both functions do the same thing. The difference is what happens when there are numerical difficulties.

`deriv()` and `_deriv()` 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 $p+d$. If that happens while computing the derivative, `deriv()` and `_deriv()` will search for a better *d* for taking the derivative. `deriv()`, however, cannot tolerate that happening at *p* (the parameter values you set using `deriv_init_params())` because the function value must exist at the point when you want `deriv()` to compute the numerical derivative. `deriv()` 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.

`_deriv()`, however, does not stop execution. Rather than aborting, `_deriv()` returns a nonzero value to the caller, identifying what went wrong. The only exception is that `_deriv()` will return a zero value to the caller even when the evaluator function returns *v* equal to missing at *p*, allowing programmers to handle this special case without having to turn `deriv_init_verbose()` off.

Programmers implementing advanced systems will want to use `_deriv()` instead of `deriv()`. Everybody else should use `deriv()`.

Programmers using `_deriv()` will also be interested in the functions `deriv_init_verbose()`, `deriv_result_errorcode()`, `deriv_result_errortext()`, and `deriv_result_returncode()`.

The error codes returned by `_deriv()` are listed below, under the heading *deriv_result_errorcode( ), . . . _errortext( ), and . . . _returncode( ).*

deriv_result_value( )

real scalar `deriv_result_value(`*D*`)`

`deriv_result_value(`*D*`)` returns the value of *f*() evaluated at *p*.

deriv_result_values( ) and _deriv_result_values( )

real matrix `deriv_result_values(`*D*`)`

void `_deriv_result_values(`*D*`,` *v*`)`

`deriv_result_values(`*D*`)` returns the vector values returned by the evaluator. For type `v` evaluators, this is the column vector that sums to the value of *f*() evaluated at *p*. For type `t` evaluators, this is the rowvector returned by the evaluator.

`_deriv_result_values(`*D*`,` *v*`)` uses swap() (see [M-5] **swap( )**) to interchange *v* with the vector values stored in *D*. This destroys the vector values stored in *D*.

These functions should be called only with type `v` evaluators.

deriv_result_gradient() and _deriv_result_gradient()

real rowvector deriv_result_gradient(*D*)

void _deriv_result_gradient(*D*, *g*)

deriv_result_gradient(*D*) returns the gradient vector evaluated at *p*.

_deriv_result_gradient(*D*, *g*) uses swap() (see [M-5] **swap()**) to interchange *g* with the gradient vector stored in *D*. This destroys the gradient vector stored in *D*.

deriv_result_scores() and _deriv_result_scores()

real matrix deriv_result_scores(*D*)

void _deriv_result_scores(*D*, *S*)

deriv_result_scores(*D*) returns the matrix of the scores evaluated at *p*. The matrix of scores can be summed over the columns to produce the gradient vector.

_deriv_result_scores(*D*, *S*) uses swap() (see [M-5] **swap()**) to interchange *S* with the scores matrix stored in *D*. This destroys the scores matrix stored in *D*.

These functions should be called only with type v evaluators.

deriv_result_Jacobian() and _deriv_result_Jacobian()

real matrix deriv_result_Jacobian(*D*)

void _deriv_result_Jacobian(*D*, *J*)

deriv_result_Jacobian(*D*) returns the Jacobian matrix evaluated at *p*.

_deriv_result_Jacobian(*D*, *J*) uses swap() (see [M-5] **swap()**) to interchange *J* with the Jacobian matrix stored in *D*. This destroys the Jacobian matrix stored in *D*.

These functions should be called only with type t evaluators.

deriv_result_Hessian() and _deriv_result_Hessian()

real matrix deriv_result_Hessian(*D*)

void _deriv_result_Hessian(*D*, *H*)

deriv_result_Hessian(*D*) returns the Hessian matrix evaluated at *p*.

_deriv_result_Hessian(*D*, *H*) uses swap() (see [M-5] **swap()**) to interchange *H* with the Hessian matrix stored in *D*. This destroys the Hessian matrix stored in *D*.

These functions should not be called with type t evaluators.

deriv_result_h(), . . . _scale(), and . . . _delta()

real rowvector `deriv_result_h(`*D*`)`

real rowvector `deriv_result_scale(`*D*`)`

real rowvector `deriv_result_delta(`*D*`)`

`deriv_result_h(`*D*`)` returns the vector of *h* values that was used to compute the numerical derivatives.

`deriv_result_scale(`*D*`)` returns the vector of scale values that was used to compute the numerical derivatives.

`deriv_result_delta(`*D*`)` returns the vector of delta values used to compute the numerical derivatives.

deriv_result_errorcode(), . . . _errortext(), and . . . _returncode()

real scalar `deriv_result_errorcode(`*D*`)`

string scalar `deriv_result_errortext(`*D*`)`

real scalar `deriv_result_returncode(`*D*`)`

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

`deriv_result_errorcode(`*D*`)` returns the same error code as `_deriv()`. The value will be zero if there were no errors. The error codes are listed in the table directly below.

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

`deriv_result_returncode(`*D*`)` returns the Stata return code corresponding to the error code. The mapping is listed in the table directly below.

In advanced code, these functions might be used as

```
(void) _deriv(D, todo)
... if (ec = deriv_result_code(D)) {
        errprintf("{p}\n")
        errprintf("%s\n", deriv_result_errortext(D))
        errprintf("{p_end}\n")
        exit(deriv_result_returncode(D))
        /*NOTREACHED*/
}
```

The error codes and their corresponding Stata return codes are

Error code	Return code	Error text
1	198	invalid todo argument
2	111	evaluator function required
3	459	parameter values required
4	459	parameter values not feasible
5	459	could not calculate numerical derivatives—discontinuous region with missing values encountered
6	459	could not calculate numerical derivatives—flat or discontinuous region encountered
16	111	*function*() not found
17	459	Hessian calculations not allowed with type `t` evaluators

NOTE: Error 4 can occur only when evaluating $f()$ at the parameter values.
This error occurs only with `deriv()`.

deriv_query()

void `deriv_query(`*D*`)`

`deriv_query(`*D*`)` displays a report on the current `deriv_init_*()` values and some of the `deriv_result_*()` values. `deriv_query(`*D*`)` may be used before or after `deriv()`, and it is useful when using `deriv()` interactively or when debugging a program that calls `deriv()` or `_deriv()`.

Conformability

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

`deriv_init_params(`*D*`,` *params*`)`:

D:	*transmorphic*
params:	1 × *np*
result:	*void*

`deriv_init_params(`*D*`)`:

D:	*transmorphic*
result:	1 × *np*

`deriv_init_argument(`*D*`,` *k*`,` *X*`)`:

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

`deriv_init_weights(`*D*`,` *params*`)`:

D:	*transmorphic*
params:	*N* × 1
result:	*void*

deriv_init_h(*D*, *h*):

D:	*transmorphic*
h:	$1 \times np$
result:	*void*

deriv_init_h(*D*):

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

deriv_init_scale(*D*, *scale*):

D:	*transmorphic*
scale:	$1 \times np$ (type d and v evaluator)
	$nv \times np$ (type t evaluator)
void:	*void*

deriv_init_bounds(*D*, *minmax*):

D:	*transmorphic*
minmax:	1×2
result:	*void*

deriv_init_bounds(*D*):

D:	*transmorphic*
result:	$1 \times w$

deriv(*D*, 0):

D:	*transmorphic*
result:	1×1
	$1 \times nv$ (type t evaluator)

deriv(*D*, 1):

D:	*transmorphic*
result:	$1 \times np$
	$nv \times np$ (type t evaluator)

deriv(*D*, 2):

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

deriv_result_values(*D*):

D:	*transmorphic*
result:	$N \times 1$
	$1 \times nv$ (type t evaluator)
	$N \times 1$ (type v evaluator)

_deriv_result_values(*D*, *v*):

D:	*transmorphic*
v:	$N \times 1$
	$1 \times nv$ (type t evaluator)
	$N \times 1$ (type v evaluator)
result:	*void*

`deriv_result_gradient(`*D*`):`

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

`_deriv_result_gradient(`*D*`, `*g*`):`

D:	*transmorphic*
g:	$1 \times np$
result:	*void*

`deriv_result_scores(`*D*`):`

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

`_deriv_result_scores(`*D*`, `*S*`):`

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

`deriv_result_Jacobian(`*D*`):`

D:	*transmorphic*
result:	$nv \times np$

`_deriv_result_Jacobian(`*D*`, `*J*`):`

D:	*transmorphic*
J:	$nv \times np$
result:	*void*

`deriv_result_Hessian(`*D*`):`

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

`_deriv_result_Hessian(`*D*`, `*H*`):`

D:	*transmorphic*
H:	$np \times np$
result:	*void*

`deriv_result_h(`*D*`):`

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

`deriv_result_scale(`*D*`):`

D:	*transmorphic*
result:	$1 \times np$ (type `d` and `v` evaluator)
	$nv \times np$ (type `t` evaluator)

`deriv_result_delta(`*D*`):`

D:	*transmorphic*
result:	$1 \times np$ (type `d` and `v` evaluator)
	$nv \times np$ (type `t` evaluator)

Diagnostics

All functions abort with error when used incorrectly.

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

Methods and formulas

See sections 1.3.4 and 1.3.5 of Gould, Pitblado, and Poi (2010) for an overview of the methods and formulas `deriv()` uses to compute numerical derivatives.

Carl Gustav Jacob Jacobi (1804–1851) was born in Potsdam, Prussia (now Germany). A prodigy, at secondary school he read advanced mathematics texts such as Euler's *Introductio in analysin infinitorum*. Even at the University of Berlin, Jacobi studied on his own using the works of Lagrange and other leading mathematicians. His career included university posts in Berlin and Königsberg. Jacobi made major discoveries in analysis (especially elliptic functions and differential equations), algebra, geometry, mechanics, and astronomy. Expanding the theory of the determinant, which was previously defined and presented by Cauchy, Jacobi invented the functional determinant of the Jacobian matrix. Thus the Jacobian was born. It was described in his memoir, "De determinantibus functionalibus", which was published in 1841. Jacobi was also an inspiring teacher who made pioneering use of seminars. In the last years of his life, Jacobi suffered from much ill-health: he died in Berlin after catching influenza followed by smallpox.

References

Gould, W. W., J. S. Pitblado, and B. P. Poi. 2010. *Maximum Likelihood Estimation with Stata*. 4th ed. College Station, TX: Stata Press.

James, I. M. 2002. *Remarkable Mathematicians: From Euler to von Neumann*. Cambridge: Cambridge University Press.

Also see

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