Algorithmic Derivatives 1.0

for GAUSS™ Mathematical and Statistical System

Aptech Systems, Inc.

Information in this document is subject to change without notice and does not represent a commitment on the part of Aptech Systems, Inc. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. The purchaser may make one copy of the software for backup purposes. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose other than the purchaser's personal use without the written permission of Aptech Systems, Inc.

©Copyright 2007-2010 by Aptech Systems, Inc., Black Diamond, WA.
All Rights Reserved.

GAUSS, **GAUSS Engine** and **GAUSS Light** are trademarks of Aptech Systems, Inc. Other trademarks are the property of their respective owners.

Part Number: `006000`
Version 1.0
Documentation Revision: 1860 May 17, 2010

Contents

1 Installation

1.1 UNIX/Linux/Mac 1-1
1.1.1 Download 1-1
1.1.2 CD 1-2
1.2 Windows 1-2
1.2.1 Download 1-2
1.2.2 CD 1-2
1.2.3 64-Bit Windows 1-3
1.3 Difference Between the UNIX and Windows Versions 1-3

2 Getting Started

2.0.1 README Files 2-2
2.0.2 Setup 2-2

3 Algorithmic Derivatives

3.1 Using Algorithmic Derivatives 3-1
3.2 Naming Conventions for Procedures with Several Arguments 3-4
3.3 Adding a Derivative Function 3-6
3.3.1 Calling Functions Returning Matrices with Dependent Columns 3-6
3.3.2 Calling Functions Returning Matrices with Independent Columns 3-7
3.4 Running the Test Example 3-9
3.5 Disallowed **GAUSS** Constructions 3-9
3.6 References 3-10

4 Algorithmic Derivatives Reference

AD 4-1

gradp1d 4-2
gradp4d 4-3
gradp4d_2_1 4-5
gradp4d_2_2 4-7

Index

1 Installation

1.1 UNIX/Linux/Mac

If you are unfamiliar with UNIX/Linux/Mac, see your system administrator or system documentation for information on the system commands referred to below.

1.1.1 Download

1. Copy the `.tar.gz` or `.zip` file to `/tmp`.
2. If the file has a `.tar.gz` extension, unzip it using `gunzip`. Otherwise skip to step 3.

   `gunzip app_`*appname*`_`*vernum*`.`*revnum*`_UNIX.tar.gz`
3. `cd` to your **GAUSS** or **GAUSS Engine** installation directory. We are assuming `/usr/local/gauss` in this case.

   ```
   cd /usr/local/gauss
   ```

4. Use `tar` or `unzip`, depending on the file name extension, to extract the file.

   `tar xvf /tmp/app_`*appname*`_`*vernum*`.`*revnum*`_UNIX.tar`

   – or –

   `unzip /tmp/app_`*appname*`_`*vernum*`.`*revnum*`_UNIX.zip`

1.1.2 CD

1. Insert the Apps CD into your machine's CD-ROM drive.
2. Open a terminal window.
3. `cd` to your current **GAUSS** or **GAUSS Engine** installation directory. We are assuming `/usr/local/gauss` in this case.

   ```
   cd /usr/local/gauss
   ```

4. Use `tar` or `unzip`, depending on the file name extensions, to extract the files found on the CD. For example:

   `tar xvf /cdrom/apps/app_`*appname*`_`*vernum*`.`*revnum*`_UNIX.tar`

   – or –

   `unzip /cdrom/apps/app_`*appname*`_`*vernum*`.`*revnum*`_UNIX.zip`

   However, note that the paths may be different on your machine.

1.2 Windows

1.2.1 Download

Unzip the `.zip` file into your **GAUSS** or **GAUSS Engine** installation directory.

1.2.2 CD

1. Insert the Apps CD into your machine's CD-ROM drive.

2. Unzip the `.zip` files found on the CD to your **GAUSS** or **GAUSS Engine** installation directory.

1.2.3 64-Bit Windows

If you have both the 64-bit version of **GAUSS** and the 32-bit Companion Edition installed on your machine, you need to install any **GAUSS** applications you own in both **GAUSS** installation directories.

1.3 Difference Between the UNIX and Windows Versions

- If the functions can be controlled during execution by entering keystrokes from the keyboard, it may be necessary to press ENTER after the keystroke in the UNIX version.

2 Getting Started

GAUSS 6.0.25+ is required to use these routines. See **_rtl_ver** in src/gauss.dec.

AD needs the Java Runtime Enviroment (JRE) V1.4.1 or a later version in order to run. If you do not already have JRE 1.4.1 installed, you can download it for free from Sun at http://java.sun.com/j2se/1.4.1/download.html Follow the instructions to install the JRE and add the bin directory containing the java.exe to your path; e.g., on a Windows machine:

```
path=%path%;C:\Program Files\Java\j2re1.4.1\bin
```

The **AD** version number is stored in a global variable:

_ad_ver 3×1 matrix, the first element contains the major version number, the second element the minor version number, and the third element the revision number.

If you call for technical support, you may be asked for the version of your copy of **AD**.

2.0.1 README Files

If there is a **README.ad** file, it contains any last minute information on the **AD** procedures. Please read it before using them.

2.0.2 Setup

Algorithmic Derivatives or **AD** is a program which takes a **GAUSS** procedure that computes a function and produces a **GAUSS** procedure for computing its derivative.

In order to use the procedures in **AD**, the **AD** library must be active. This is done by including **ad** in the **library** statement at the top of your program or command file:

```
library ad;
```

This enables **GAUSS** to find the **AD** procedures.

You will also need to include the **AD** structure definition file

```
#include ad.sdf;
```

at the top of the command file.

3 Algorithmic Derivatives

3.1 Using Algorithmic Derivatives

AD is a program for generating a **GAUSS** procedure to compute derivatives from a **GAUSS** procedure that computes a function value. If the input function procedure returns a scalar value given a $K \times 1$ input vector, the output derivative procedure computes a $1 \times K$ gradient. If the input function returns an $N \times 1$ vector given a $K \times 1$ input vector, the output derivative procedure computes an $N \times K$ Jacobian matrix.

First, copy the input function procedure to a separate file. Second, from the command line enter

```
ad file_name d_file_name
```

where `file_name` is the name of the file containing the input function procedure, and `d_file_name` is the name of the file containing the output derivative procedure.

If the input function procedure is named `fct`, the output derivative procedure has the name `d_fct` if the function procedure has a single argument. If the function procedure has two arguments, the derivative procedure is given the name `d_1_fct` where the addition to the prefix indicates that the derivative is with respect to the first argument.

For example, put the following function into a file called `lpr.fct`:

```
proc lpr(x,z);
    local s,m,u;
    s = x[4];
    m = z[.,2:4]*x[1:3,.];
    u = z[.,1] ./= 0;
    retp(u.*lnpdfmvn(z[.,1]-m,s) + (1-u).*(lncdfnc(m/sqrt(s))));
endp;
```

Then enter the following at the **GAUSS** command line

```
library ad;
ad lpr.fct d_lpr.fct;
```

If successful, the following is printed to the screen

```
java -jar d:\gauss6.0\src\gauss_ad.jar lpr.fct d_lpr.fct
```

and the derivative procedure is written to file named `d_lpr.fct`:

```
/* Version:1.0 - May 15, 2004 */
/* Generated from:lpr.fct */

/* Taking derivative with respect to argument 1 */
Proc(1)=d_1_lpr(x, z);
```

```
    Clearg _AD_fnValue;
        Local s, m, u;
        s =  x[(4)] ;
        Local _AD_t1;
        _AD_t1 =  x[(1):(3),.] ;
        m =  z[.,(2):(4)]  * _AD_t1;
        u =  z[.,(1)]  ./= 0;
        _AD_fnValue = (u .* lnpdfmvn( z[.,(1)]  - m, s)) + ((1 - u) .*
lncdfnc(m / sqrt(s)));
        /* retp(_AD_fnValue); */
        /* endp; */
        struct _ADS_optimum _AD_d__AD_t1 ,_AD_d_x ,_AD_d_s ,_AD_d_m
,_AD_d__AD_fnValue;
        /* _AD_d__AD_t1 = 0; _AD_d_s = 0; _AD_d_m = 0;  */
    _AD_d__AD_fnValue = _ADP_d_x_dx(_AD_fnValue);
    _AD_d_s = _ADP_DtimesD(_AD_d__AD_fnValue,
_ADP_DplusD(_ADP_DtimesD(_ADP_d_xplusy_dx(u .* lnpdfmvn( z[.,(1)]  - m, s),
(1 - u) .* lncdfnc(m / sqrt(s))), _ADP_DtimesD(_ADP_d_ydotx_dx(u, lnpdfmvn(
 z[.,(1)]  - m, s)), _ADP_DtimesD(_ADP_internal(d_2_lnpdfmvn( z[.,(1)]  - m,
s)), _ADP_d_x_dx(s)))), _ADP_DtimesD(_ADP_d_yplusx_dx(u .* lnpdfmvn(
 z[.,(1)]  - m, s), (1 - u) .* lncdfnc(m / sqrt(s))),
_ADP_DtimesD(_ADP_d_ydotx_dx(1 - u, lncdfnc(m / sqrt(s))),
_ADP_DtimesD(_ADP_d_lncdfnc(m / sqrt(s)), _ADP_DtimesD(_ADP_d_ydivx_dx(m,
sqrt(s)), _ADP_DtimesD(_ADP_d_sqrt(s), _ADP_d_x_dx(s)))))));
    _AD_d_m = _ADP_DtimesD(_AD_d__AD_fnValue,
_ADP_DplusD(_ADP_DtimesD(_ADP_d_xplusy_dx(u .* lnpdfmvn( z[.,(1)]  - m, s),
(1 - u) .* lncdfnc(m / sqrt(s))), _ADP_DtimesD(_ADP_d_ydotx_dx(u, lnpdfmvn(
 z[.,(1)]  - m, s)), _ADP_DtimesD(_ADP_internal(d_1_lnpdfmvn( z[.,(1)]  - m,
s)), _ADP_DtimesD(_ADP_d_yminusx_dx( z[.,(1)] , m), _ADP_d_x_dx(m)))),
_ADP_DtimesD(_ADP_d_yplusx_dx(u .* lnpdfmvn( z[.,(1)]  - m, s), (1 - u) .*
lncdfnc(m / sqrt(s))), _ADP_DtimesD(_ADP_d_ydotx_dx(1 - u, lncdfnc(m / sqrt(s)
)), _ADP_DtimesD(_ADP_d_lncdfnc(m / sqrt(s)), _ADP_DtimesD(_ADP_d_xdivy_dx(m,
sqrt(s)), _ADP_d_x_dx(m)))))));
    /* u =  z[.,(1)]  ./= 0; */
    _AD_d__AD_t1 = _ADP_DtimesD(_AD_d_m, _ADP_DtimesD(_ADP_d_yx_dx(
 z[.,(2):(4)] , _AD_t1), _ADP_d_x_dx(_AD_t1)));
    Local _AD_sr_x, _AD_sc_x;
    _AD_sr_x = _ADP_seqaMatrixRows(x);
    _AD_sc_x = _ADP_seqaMatrixCols(x);
    _AD_d_x = _ADP_DtimesD(_AD_d__AD_t1, _ADP_d_x2Idx_dx(x,
 _AD_sr_x[(1):(3)] ,  _AD_sc_x[0] ));
    Local _AD_s_x;
```

```
    _AD_s_x = _ADP_seqaMatrix(x);
    _AD_d_x = _ADP_DplusD(_ADP_DtimesD(_AD_d_s, _ADP_d_xIdx_dx(x,
 _AD_s_x[(4)] )), _AD_d_x);
    retp(_ADP_external(_AD_d_x));
endp;
```

If there is a syntax error in the input function procedure, the following is written to the screen

```
java -jar d:\gauss6.0\src\gauss_ad.jar lpr.fct d_lpr.fct
Command 'java -jar d:\gauss6.0\src\gauss_ad.jar cmlad3.fct d_lpr.fct' exit status 1
```

the **exit status 1** indicating that an error has occurred. The output file then contains the reason for the error:

```
/* Version:1.0 - May 15, 2004 */
/* Generated from:lpr.fct */

/* Taking derivative with respect to argument 1 */

proc lpr(x,z);
    local s,m,u;
    s = x[4];
    m = z[.,2:4]*x[1:3,.];
    u = z[.,1] ./= 0;
    retp(u.*lnpdfmvn(z[.,1]-m,s) + (1-u).*(lncdfnc(m/sqrt(s)));
Error: lpr.fct:12:63: expecting ')', found ';'
```

3.2 Naming Conventions for Procedures with Several Arguments

For a function procedure with a single argument,

```
proc fct(x);
  /* code */
endp;
```

in a file called, for example, `fct.src` with a single argument, the following

```
ad fct.src d_fct.src
```

produces a derivative procedure

```
proc d_fct(x);
 /* code */
endp;
```

in the file `d_fct.src` with the same single argument.

For a function procedure with two arguments,

```
proc fct(x,y);
  /* code */
endp;
```

produces a derivative procedure

```
proc d_1_fct(x);
 /* code */
endp;
```

where the "`_1_`" indicates the derivative is taken with respect to the first argument.

By default, the derivative is with respect to the first argument. To produce the derivative with respect to the second argument, add a "`_2_`" to the name of the file that will contain the derivative procedure. For example,

```
ad fct.src d_2_fct.src
```

The derivative procedure will then have the name

```
proc d_2_fct(b,x);
 /* code */
endp;
```

3.3 Adding a Derivative Function

The function procedure may contain calls to **GAUSS** functions that have not yet been included in **AD**, or it may contain calls to functions you have written. **AD** will need to know how to compute the derivatives of these functions before being able to produce the derivative procedure. This section describes several methods for doing this.

3.3.1 Calling Functions Returning Matrices with Dependent Columns

The derivative of the called function must be computed numerically. Add two procedures to the `ad.src` file in the `src` subdirectory:

```
proc _ADP_utility_userfct(x);
    retp(userfct(x));
endp;

proc d_userfct(x);
    retp(gradp4d(&_ADP_utility_userfct,x));
endp;
```

where `userfct` is the name of the called function. For example, for the **GAUSS** **`invpd`** function,

```
proc _ADP_utility_invpd(x);
    retp(invpd(x));
endp;

proc d_invpd(x);
    retp(gradp4d(&_ADP_utility_invpd,x));
endp;
```

3.3.2 Calling Functions Returning Matrices with Independent Columns

Most functions, for example, the **GAUSS** `log` function, return matrices that are independent. Their derivatives can be provided either numerically or analytically.

Analytical

For example, the following computes the derivatives for the `log` function. For your own function change "log" below to the name of your function, substitute the calculation of the derivative for the appropriate line, and add these procedures to the `ad.src` file:

```
proc(1) = d_log(x);
    retp(_ADP_external(_ADP_d_log(x)));
endp;

proc(1) = _ADP_d_log(x);
local xCols,xRows;
xCols = cols(x);
xRows = rows(x);
x = 1 ./ (ln(10) .* vec(x));
    retp(_ADP_putDiag(xCols|xCols|xRows|xRows,x));
endp;
```

Note that the input matrix is "vec-ed" after the number of rows and columns have been recorded.

Numerical for User-Provided Called Function

`gradp1d` is a function provided in **AD** for computing the derivative of a function returning a matrix with independent columns. Substitute your own called function name for "userfct".

```
proc(1) = d_userfct(x);
    retp(_ADP_external(_ADP_d_userfct(x)));
endp;

proc(1) = _ADP_d_userfct(x);
    local xCols,xRows;
    xCols = cols(x);
    xRows = rows(x);
    x = gradp1d(&userfct,x);
    retp(_ADP_putDiag(xCols|xCols|xRows|xRows,x));
endp;
```

Numerical for GAUSS Called Function

In order to handle a **GAUSS** function, a wrapper function needs to be written.

```
proc(1) = d_log(x);
    retp(_ADP_external(_ADP_d_log(x)));
endp;

proc _ADP_utility_log(x);
    retp(log(x));
endp;

proc(1) = _ADP_d_log(x);
    local xCols,xRows;
    xCols = cols(x);
    xRows = rows(x);
```

```
    x = gradp1d(&_ADP_utility_log,x);
    retp(_ADP_putDiag(xCols|xCols|xRows|xRows,x));
endp;
```

3.4 Running the Test Example

The `example_procs` subdirectory has a number of files containing function procedures (for example, `test1.src`). When run in **GAUSS**, the example file `test.e` generates files containing derivative procedures using the files with the function procedures (for example, `d_test1.src`).

Additionally, the example file `d_test.e` tests the accuracy of the resulting derivative procedures. Thus after running `test.e`, run `d_test.e` and an accuracy report is printed to the screen.

3.5 Disallowed GAUSS Constructions

The following **GAUSS** language constructions are not allowed in the input procedure

Label: statement

CLEARG

DLLCALL

FORMAT

IF

ELSEIF

ELSE

ENDIF

FOR

ENDFOR

DO

WHILE

UNTIL

ENDO

BREAK

CONTINUE

GOTO

GOSUB

3.6 References

1. Griewank, Andreas, *Principles and Techniques of Algorithmic Differentiation*, **SIAM**, 2000.

Algorithmic Derivatives Reference 4

AD

PURPOSE Generates a procedure for computing derivatives from a procedure that computes a function.

LIBRARY **ad**

FORMAT ad *infile_name outfile_name*

INPUT *infile_name* string, name of file containing procedure computing function

outfile_name string, name of file into which the derivative procedure is to be put

EXAMPLE

```
library ad
ad fct.src d_fct.src
```

gradp1d

PURPOSE Computes the gradient vector defined in a procedure. Single-sided (forward difference) gradients are computed.

LIBRARY **ad**

FORMAT **g = gradp1d(&*fct*,*x*)**

INPUT

&*fct* a pointer to a procedure that evaluates a function given *x*

```
proc fct(x);
    /* function evaluation here */
   retp(result);
endp;
```

This function must return a vector or a matrix with independent columns.

x $K \times 1$ vector, values at which to evaluate the function

OUTPUT

g $M \times 1$ vector, derivatives of function evaluated at *x* where *M* is the number of columns of the matrix returned by *fct*.

EXAMPLE

```
proc myfunc(x);
   retp(lngamma(x));
endp;

x0 = { 0.1 0.2,
       0.4 0.5 };
```

```
gradp1d(&myfunc,x0);

     -10.4238
      -2.5614
      -5.2890
      -1.9635
```

SEE ALSO **gradp4d**, **gradp4d_2_1**, **gradp4d_2_2**, **gradp**, **hessp**

gradp4d

PURPOSE Computes the gradient vector or matrix (Jacobian) of a matrix-valued function defined in a procedure. Single-sided (forward difference) gradients are computed.

LIBRARY **ad**

FORMAT **g = gradp4d(&*fct*,*x*)**

INPUT **&*fct*** a pointer to a procedure that evaluates a function given x

```
proc fct(x);
    /* function evaluation here */
   retp(result);
endp;
```

x $K \times J$ vector, values at which to evaluate the function

OUTPUT *g* scalar, $1 \times K$ vector, $Q \times K$ matrix, $L \times Q \times K$ array or $P \times L \times Q \times K$ array, derivatives of function evaluated at x

If x is a $K \times 1$ vector and $fct(x)$ is a 1×1 scalar, the result g is row vector $[1, K]$ of gradients

If x is a $K \times 1$ vector and $fct(x)$ is an $N \times 1$ vector, the result g is matrix $[N, K]$ of cross gradients

If x is a matrix $K \times J$ and $fct(x)$ is an $N \times 1$ vector, the result g is 3D matrix $[J, N, K]$

If x is a matrix $K \times J$ and $fct(x)$ is a matrix $N \times M$, the result g is 4D matrix $[M, J, N, K]$

REMARKS **gradp4d** will return a row for every row that is returned by *fct*. For instance, if *fct* returns a 1×1 result, then **gradp4d** will return a $1 \times K$ row vector. This allows the same function to be used where N is the number of rows in the result returned by *fct*. Thus, for instance, **gradp4d** can be used to compute the Jacobian matrix of a set of equations.

EXAMPLE

```
proc myfunc(x);
   retp(x*x');
endp;

x0 = { 0.1 0.2 0.3,
       0.4 0.5 0.6 };

gradp4d(&myfunc,x0);

   Plane [1,1,.,.]

    0.20  0.00
    0.40  0.10

   Plane [1,2,.,.]

    0.40  0.00
    0.50  0.20

   Plane [1,3,.,.]

    0.60  0.00
    0.60  0.30
```

```
Plane [2,1,.,.]

 0.40  0.10
 0.00  0.80

Plane [2,2,.,.]

 0.50  0.20
 0.00  1.00

Plane [2,3,.,.]

 0.60  0.30
 0.00  1.20
```

SEE ALSO **gradp1d**, **gradp4d_2_1**, **gradp4d_2_2**, **gradp**, **hessp**

gradp4d_2_1

PURPOSE Computes 4-dimensional numerical derivatives.

LIBRARY **ad**

FORMAT **g = gradp4d_2_1(&*fct*,*x*,*y*)**

INPUT **&*fct*** a pointer to a procedure that evaluates a function given *x* and *y*

```
proc fct(x,y);
    /* function evaluation here */
   retp(result);
```

```
endp;
```

x $K \times L$ matrix, values at which to evaluate the function

y $M \times N$ matrix

OUTPUT *g* scalar, $1 \times K$ vector, $Q \times K$ matrix, $L \times Q \times K$ array or $P \times L \times Q \times K$ array, derivatives of function evaluated at *x*

If *x* is a $K \times 1$ vector and *fct*(*x*,*y*) is a 1×1 scalar, the result *g* is row vector $[1, K]$ of gradients

If *x* is a $K \times 1$ vector and *fct*(*x*,*y*) is an $N \times 1$ vector, the result *g* is matrix $[N, K]$ of cross gradients

If *x* is a matrix $K \times J$ and *fct*(*x*,*y*) is an $N \times 1$ vector, the result *g* is 3D matrix $[J, N, K]$

If *x* is a matrix $K \times J$ and *fct*(*x*,*y*) is a matrix $N \times M$, the result *g* is 4D matrix $[M, J, N, K]$

REMARKS **gradp4D_2_1** will return a row for every row that is returned by *fct*. For instance, if *fct* returns a 1×1 result, then **gradp4D_2_1** will return a $1 \times K$ row vector. This allows the same function to be used where *N* is the number of rows in the result returned by *fct*. Thus, for instance, **gradp4D_2_1** can be used to compute the Jacobian matrix of a set of equations.

EXAMPLE

```
proc myfunc(x,y);
    retp(x * y);
endp;

x0 = { 0.1 0.2 0.3,
       0.4 0.5 0.6 };
y  = { 1 4,2 5,3 6 };

gradp4d_2_1(&myfunc,x0,y);

   Plane [1,1,.,.]
```

```
 1.00  0.00
 0.00  1.00

Plane [1,2,.,.]

 2.00  0.00
 0.00  2.00

Plane [1,3,.,.]

 3.00  0.00
 0.00  3.00

Plane [2,1,.,.]

 4.00  0.00
 0.00  4.00

Plane [2,2,.,.]

 5.00  0.00
 0.00  5.00

Plane [2,3,.,.]

 6.00  0.00
 0.00  6.00
```

SEE ALSO **gradp4d_2_2**, **gradp4d**, **gradp**, **hessp**

gradp4d_2_2

PURPOSE Computes 4-dimensional numerical derivatives.

gradp4d_2_2

LIBRARY **ad**

FORMAT **g = gradp4d_2_2(&*fct*,*x*,*y*)**

INPUT **&*fct*** a pointer to a procedure that evaluates a function given *x* and *y*

```
proc fct(x,y);
    /* function evaluation here */
   retp(result);
endp;
```

x $M \times N$ matrix

y $K \times L$ matrix, values at which to evaluate the function

OUTPUT *g* scalar, $1 \times K$ vector, $Q \times K$ matrix, $L \times Q \times K$ array or $P \times L \times Q \times K$ array, derivatives of function evaluated at *y*

If *y* is a $K \times 1$ vector and *fct*(*x*,*y*) is a 1×1 scalar, the result *g* is row vector $[1, K]$ of gradients

If *y* is a $K \times 1$ vector and *fct*(*x*,*y*) is an $N \times 1$ vector, the result *g* is matrix $[N, K]$ of cross-gradients

If *y* is a matrix $K \times J$ and *fct*(*x*,*y*) is an $N \times 1$ vector, the result *g* is 3D matrix $[J, N, K]$

If *y* is a matrix $K \times J$ and *fct*(*x*,*y*) is a matrix $N \times M$, the result *g* is 4D matrix $[M, J, N, K]$

REMARKS **gradp4D_2_2** will return a row for every row that is returned by *fct*. For instance, if *fct* returns a 1×1 result, then **gradp4D_2_2** will return a $1 \times K$ row vector. This allows the same function to be used where *N* is the number of rows in the result returned by *fct*. Thus, for instance, **gradp4D_2_2** can be used to compute the Jacobian matrix of a set of equations.

EXAMPLE

```
proc myfunc(x,y);
    retp(x * y);
endp;

x = { 0.1 0.2 0.3,
      0.4 0.5 0.6 };
y0  = { 1 4,2 5,3 6 };

gradp4d_2_2(&myfunc,x,y0);

   Plane [1,1,.,.]

    0.10  0.20  0.30
    0.40  0.50  0.60

   Plane [1,2,.,.]

    0.00  0.00  0.00
    0.00  0.00  0.00

   Plane [2,1,.,.]

    0.00  0.00  0.00
    0.00  0.00  0.00

   Plane [2,2,.,.]

    0.10  0.20  0.30
    0.40  0.50  0.60
```

SEE ALSO **gradp4D_2_1**, **gradp4d**, **gradp**, **hessp**

Index

AD, 4-1

D

disallowed statements, 3-9

G

gradp1d, 3-8, 4-2
gradp4d, 4-3
gradp4d_2_1, 4-5
gradp4d_2_2, 4-7

I

Installation, 1-1

U

UNIX, 1-3
UNIX/Linux/Mac, 1-1

W

Windows, 1-2, 1-3