

# GAUSS<sup>TM</sup> Programming Language

## Working with Arrays

Version 1.0.0  
November 19, 2003

Aptech Systems, Inc.  
Maple Valley, WA

The information in this workbook is subject to change without notice and does not represent a commitment on the part of Aptech Systems. The manual and the accompanying software are provided under the terms of a license agreement or non-disclosure agreement. The software may be used and copied only according to the terms of the agreement. No part of this manual may be reproduced, transmitted, transcribed, stored in any retrieval system, or translated into any language by any means without the prior written permission of:

Aptech Systems, Inc.  
23804 SE Kent-Kangley Road  
Maple Valley, WA 98038

Copyright ©2003 by Aptech Systems, Inc.  
All Rights Reserved.

November 19, 2003

GAUSS is a trademark of Aptech Systems, Inc.

# Chapter 1

## Initializing Arrays

The use of N-dimensional arrays in **GAUSS** is an additional tool for reducing development time and increasing execution speed of programs. There are multiple ways of handling N-dimensional arrays and using them to solve problems, and these ways sometimes have implications for a trade-off between speed of execution and development time. We will try to make this clear in this article.

The term arrays specifically refers to N-dimensional arrays and must not be confused with matrices. Matrices and arrays are distinct types even if in fact they contain identical information. Functions will be described below for conversion from one to the other.

There are six basic ways of creating an array depending how the contents are specified

**areshape** creates array from specified matrix

**aconcat** creates array from matrices and arrays

**aeye** creates array of identity matrices

**arrayinit** allocates array filled with specified scalar value

**arrayalloc** allocates array with no specified contents

### 1.0.1 areshape

**areshape** is an additional method for creating an array with specified contents. **arrayinit** creates an array filled with a selected scalar value, but **areshape** will do the same but with a matrix. For example given a matrix, **areshape** will create an array containing multiple versions of that matrix:

```
x = reshape(seqa(1,1,4),2,2);
ord = 3 | 2 | 2;
a = areshape(x,ord);
print a;
```

```
Plane [1,..]
      1.0000      2.0000
      3.0000      4.0000

Plane [2,..]
      1.0000      2.0000
      3.0000      4.0000

Plane [3,..]
      1.0000      2.0000
      3.0000      4.0000
```

#### Reading Data from the Disk into an Array

**areshape** is a very fast way to re-dimension a matrix or array already in memory. For example, suppose we have a **GAUSS** data set containing panel data and that it's small enough to be read in all at once,

```
panel = areshape(load("panel"),5|100|10);

mn = amean(panel,2); /* 5x1x10 array of means of each panel */
mm = moment(panel,0); /* 5x10x10 array of moments of each panel */

/*
** vc is a 5x10x10 array of
```

## 1. INITIALIZING ARRAYS

```
** covariance matrices
*/

vc = mm / 100 - amult(atranspose(mn,1|3|2),mn);
```

$x$  is a  $5 \times 100 \times 10$  array, and in this context is 5 panels of 100 cases measured on 10 variables.

### Inserting Random Numbers into Arrays

A random array of any dimension or size can be very quickly created using **areshape**. Thus for a  $10 \times 10 \times 5 \times 3$  array:

```
ord = { 10, 10, 5, 3 };
y = areshape(rndu(prodc(ord),1),ord);
```

The quick and dirty method above uses the linear congruential generator which is fast but doesn't have the properties required for serious Monte Carlo work. For series simulation you will need to use the KM generator:

```
sd0 = 345678;
ord = { 10, 10, 5, 3 };
{ z,sd0 } = rndKMu(prodc(ord),1,sd0);
y = areshape(z,ord);
```

### Expanding a Matrix into an Array Vector of Matrices

For computing the log-likelihood of a variance components model of panel data, it is necessary to expand a  $T \times T$  matrix into an  $NT \times T$  array of these matrices. This is easily accomplished using **areshape**. For example,

```
m = { 1.0  0.3  0.2,
      0.3  1.0  0.1,
      0.2  0.1  1.0 };

r = areshape(m,3|3|3);
```

## 1. INITIALIZING ARRAYS

```
print r;

Plane [1,..]

    1.0000    0.30000    0.20000
    0.30000    1.0000    0.10000
    0.20000    0.10000    1.0000

Plane [2,..]

    1.0000    0.30000    0.20000
    0.30000    1.0000    0.10000
    0.20000    0.10000    1.0000

Plane [3,..]

    1.0000    0.30000    0.20000
    0.30000    1.0000    0.10000
    0.20000    0.10000    1.0000
```

### 1.0.2 aconcat

**aconcat** creates arrays from conformable sets of matrices or arrays. With this function contents are completely specified by the user. This example tries three concatenations, one along each dimension:

```
rndseed 345678;
x1 = rndn(2,2);
x2 = arrayinit(2|2,1);

/*
** along the first dimension or rows
*/
a = aconcat(x1,x2,1);
print a;

    -0.4300    -0.2878    1.0000    1.0000
    -0.1327    -0.0573    1.0000    1.0000
```

## 1. INITIALIZING ARRAYS

```
/*  
** along the second dimension or columns  
*/
```

```
a = aconcat(x1,x2,2);  
print a;
```

```
    -0.4300    -0.2878  
    -0.1327    -0.0573  
     1.0000     1.0000  
     1.0000     1.0000
```

```
/*  
** along the third dimension  
*/
```

```
a = aconcat(x1,x2,3);  
print a;
```

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

```
    -0.4300    -0.2878  
    -0.1327    -0.0573
```

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

```
     1.0000     1.0000  
     1.0000     1.0000
```

### 1.0.3 aeye

**aeye** creates an array in which the principle diagonal of the two trailing dimensions are set to one. For example,

```
ord = 2 | 3 | 3;  
a = aeye(ord);  
print a;
```

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

## 1. INITIALIZING ARRAYS

```
1.00000 0.00000 0.00000
0.00000 1.00000 0.00000
0.00000 0.00000 1.00000

Plane [2,.,.]

1.00000 0.00000 0.00000
0.00000 1.00000 0.00000
0.00000 0.00000 1.00000
```

### 1.0.4 arrayinit

**arrayinit** creates an array with all elements set to a specified value. For example

```
ord = 3 | 2 | 3;
a = arrayinit(ord,1);
print a;

plane [1,.,.]

1.0000 1.0000 1.0000
1.0000 1.0000 1.0000

Plane [2,.,.]

1.0000 1.0000 1.0000
1.0000 1.0000 1.0000

Plane [3,.,.]

1.0000 1.0000 1.0000
1.0000 1.0000 1.0000
```

### 1.0.5 arrayalloc

**arrayalloc** creates an array with specified number and size of dimensions without setting elements to any values. This requires a vector specifying the order of the array. The length of the vector determines the number of dimensions, and each element

## 1. INITIALIZING ARRAYS

determines the size of the corresponding dimensions. The array will then have to be filled using any of several methods described later in this article.

For example to allocate a  $2 \times 2 \times 3$  array,

```
rndseed 345678;
ord = 3 | 2 | 2;
a = arrayalloc(ord,0);

for i(1,ord[1],1);
    a[i,.,.] = rndn(2,3);
endfor;
print a;
```

Plane [1,.,.]

-0.4300	-0.2878	-0.1327
-0.0573	-1.2900	0.2467

Plane [2,.,.]

-1.4249	-0.0796	1.2693
-0.7530	-1.7906	-0.6103

Plane [3,.,.]

1.2586	-0.4773	0.7044
-1.2544	0.5002	0.3559

The second argument in the call to **arrayalloc** specifies whether the created array is real or complex. **arrayinit** creates only real arrays.

## 1. *INITIALIZING ARRAYS*

## Chapter 2

# Assigning Arrays

There are three methods for assignment to an array

**index operator**

**putArray** puts a subarray into an N-dimensional array and returns the result

**setArray** sets a subarray of an N-dimensional array in place

And there are several ways to extract parts of arrays:

**index operator** the same method as matrices generalized to arrays

**getArray** gets a subarray from an array

**getMatrix** gets a matrix from an array

**getMatrix4D** gets a matrix from a 4-dimensional array

**getScalar3D** gets a scalar from a 3-dimensional array

**getScalar4D** gets a scalar from a 4-dimensional array

The index operator is the slowest way to extract parts of arrays. The specialized functions are the fastest when the circumstances are appropriate for their use.

### 2.0.6 index operator

The index operator will put a subarray into an array in a manner analogous to the use of index operators on matrices.

```
a = arrayinit(3|2|2,0);
b = arrayinit(3|1|2,1);
a[.,2,.] = b;

print a;

Plane [1,.,.]
      0.00000      0.00000
      1.0000      1.0000

Plane [2,.,.]
      0.00000      0.00000
      1.0000      1.0000

Plane [3,.,.]
      0.00000      0.00000
      1.0000      1.0000
```

As this example illustrates, the assignment doesn't have to be contiguous. **putMatrix** and **setMatrix** require a contiguous assignment, but for that reason they are faster.

The right hand side of the assignment can also be a matrix.

```
a[1,.,.] = rndn(2,2);
print a;

Plane [1,.,.]
      -1.7906502      -0.61038103
```

## 2. ASSIGNING ARRAYS

```
1.2586160    -0.47736360  
  
Plane [2,.,.]  
0.00000    0.00000  
1.0000    1.0000  
  
Plane [3,.,.]  
0.00000    0.00000  
1.0000    1.0000
```

The index operator will extract an array from a subarray in a manner analogous to the use of index operators on matrices.

```
a = areshape(seqa(1,1,12),3|2|2);  
b = a[:,1,.];
```

```
print a;
```

```
Plane [1,.,.]  
1.0000    2.0000  
3.0000    4.0000  
  
Plane [2,.,.]  
5.0000    6.0000  
7.0000    8.0000  
  
Plane [3,.,.]  
9.0000    10.000  
11.000    12.000
```

```
print b;
```

```
Plane [1,.,.]  
1.0000    2.0000  
  
Plane [2,.,.]  
5.0000    6.0000
```

```
Plane [3,.,.]
      9.0000    10.000
```

It is important to note that the result is always an array even if it's a scalar value.

```
c = a[1,1,1];
print c;

Plane [1,.,.]
      1.0000
```

If you require a matrix result, and if the result has one or two dimensions, use **arraytomat** to convert to a matrix, or use **getMatrix**, **getMatrix3D**, or **getMatrix4D**. Or if the result is a scalar, use **getScalar3D** or **getScalar4D**.

### 2.0.7 getArray

**getArray** is an additional method for extracting arrays.

```
a = areshape(seqa(1,1,12),3|2|2);
b = getarray(a,2|1);

print a;

Plane [1,.,.]
      1.0000    2.0000
      3.0000    4.0000

Plane [2,.,.]
      5.0000    6.0000
      7.0000    8.0000

Plane [3,.,.]
      9.0000    10.000
      11.000    12.000
```

## 2. ASSIGNING ARRAYS

```
print b;
      5.0000      6.0000
```

**getArray** can only extract a *contiguous* part of an array. To get non-contiguous parts you must use the index operator.

### 2.0.8 getMatrix

If the result is one or two dimensions, **getMatrix** returns a portion of an array converted to a matrix. **getMatrix** is about 20 percent faster than the index operator.

```
a = areshape(seqa(1,1,12),3|2|2);
b = getMatrix(a,2);
print b;
      5.0000      6.0000
      7.0000      8.0000
```

### 2.0.9 getMatrix4D

This is a specialized version of **getMatrix** for 4-dimensional arrays. It behaves just like **getMatrix** but is dramatically faster for that type of array. The following illustrates the difference in timing,

```
a = arrayinit(100|100|10|10,1);

t0 = date;
for i(1,100,1);
  for j(1,100,1);
    b = a[i,j,..];
  endfor;
endfor;
t1 = date;
e1 = ethsec(t0,t1);
print e1;
```

```

print;

t2=date;
for i(1,100,1);
    for j(1,100,1);
        b = getMatrix4d(a,i,j);
    endfor;
endfor;
t3 = date;
e2 = ethsec(t2,t3);
print e2;
print;
print ftostrC(100*((e1-e2)/e1),"percent difference - %6.21f%%");

13.000000

5.000000

percent difference - 61.54%

```

### 2.0.10 getScalar3D, getScalar4D

These are specialized versions of **getMatrix** for retrieving scalar elements of 3-dimensional and 4-dimensional arrays respectively. They behave just like **getMatrix** with scalar results but are much faster. For example,

```

a = arrayinit(100|10|10,1);

t0 = date;
for i(1,100,1);
    for j(1,10,1);
        for k(1,10,1);
            b = a[i,j,k];
        endfor;
    endfor;
endfor;
t1 = date;
e1 = ethsec(t0,t1);
print e1;
print;

```

## 2. ASSIGNING ARRAYS

```
t2=date;
for i(1,100,1);
  for j(1,10,1);
    for k(1,10,1);
      b = getscalar3d(a,i,j,k);
    endfor;
  endfor;
endfor;
t3 = date;
e2 = ethsec(t2,t3);
print e2;
print;
print ftostrC(100*((e1-e2)/e1),"percent difference - %6.21f%%");

      7.0000000

      2.0000000

percent difference - 71.43%
```

### 2.0.11 putArray

**putArray** enters a subarray, matrix, or scalar into an N-dimensional array and returns the result in an array. This function is much faster than the index operator but it requires the part of the array being assigned to be contiguous.

```
a = arrayinit(3|2|2,3);
b = putarray(a,2,eye(2));
print b;

Plane [1,..]

      3.0000      3.0000
      3.0000      3.0000

Plane [2,..]

      1.0000      0.0000
      0.0000      1.0000
```

```

Plane [3,..]
      3.0000      3.0000
      3.0000      3.0000

```

### 2.0.12 setArray

`setArray` enters a subarray, matrix or scalar into an N-dimensional array in place.

```

a = arrayinit(3|2|2,3);
setarray a,2,eye(2);
print b;

```

```

Plane [1,..]
      3.0000      3.0000
      3.0000      3.0000

```

```

Plane [2,..]
      1.0000      0.00000
      0.00000     1.0000

```

```

Plane [3,..]
      3.0000      3.0000
      3.0000      3.0000

```

## 2.1 Looping with Arrays

When working with arrays, for loops and do loops may be used in the usual way. In the following let  $Y$  be an  $N \times 1 \times L$  array of  $L$  time series,  $X$  an  $N \times 1 \times K$  array of  $K$  independent variables,  $B$  a  $K \times L$  matrix of regression coefficients,  $\mathbf{phi}$  a  $P \times L \times L$  array of garch coefficients,  $\mathbf{theta}$  a  $Q \times L \times L$  array of arch coefficients, and  $\mathbf{Omega}$  a  $L \times L$  symmetric matrix of constants. The log-likelihood for a multivariate garch BEKK model can be computed using the index operator:

## 2. ASSIGNING ARRAYS

```
yord = getOrders(Y);
xord = getOrders(X);
gord = getOrders(phi);
aord = getOrders(theta);

N = yord[1]; /* No. of observations */
L = yord[3]; /* No. of time series */
K = xord[3]; /* No. of independent variables in mean equation */
P = gord[1]; /* order of garch parameters */
Q = aord[1]; /* order of arch parameters */

r = maxc(P|Q);

E = Y - amult(X,areshape(B,N|K|L));
sigma = areshape(omega,N|L|L);

for i(r+1,N,1);

    for j(1,Q,1);
        W = amult(theta[j,..],atranspose(E[i-j,..],1|3|2));
        sigma[i,..] = sigma[i,..] + amult(W,atranspose(W,1|3|2));
    endfor;

    for j(1,P,1);
        sigma[i,..] = sigma[i,..] +
            amult(amult(phi[j,..],sigma[i-j,..]),phi[j,..]);
    endfor;

endfor;

sigmai = invpd(sigma);
lndet = ln(det(sigma));

lnl = -0.5*( L*(N-R)*asum(ln(det(sigmai)),1) + asum(amult(amult(E,sigmai),atranspose(E,1|3
```

Instead of index operators, the above computation can be done using **getArray** and **setArray**:

```
yord = getOrders(Y);
xord = getOrders(X);
gord = getOrders(phi);
aord = getOrders(theta);

N = yord[1]; /* No. of observations */
L = yord[3]; /* No. of time series */
```

## 2. ASSIGNING ARRAYS

```
K = xord[3]; /* No. of independent variables in mean equation */
P = gord[1]; /* order of garch parameters */
Q = aord[1]; /* order of arch parameters */

r = maxc(P|Q);

E = Y - amult(X,areshape(B,N|K|L));
sigma = areshape(omega,N|L|L);

for i(r+1,N,1);

    for j(1,Q,1);
        W = amult(getArray(theta,j),atranspose(getArray(E,i-j),2|1));
        setarray sigma,i,getArray(sigma,i)+amult(W,atranspose(W,2|1));
    endfor;

    for j(1,P,1);
        setarray sigma,i,getArray(sigma,i)+areshape(amult(amult(getArray(phi,j),getArr
    endfor;

endfor;

sigma_i = invpd(sigma);
lndet = ln(det(sigma));

lnl = -0.5*( L*(N-R)*asum(ln(det(sigma_i)),1) + asum(amult(amult(E,sigma_i),atranspose(E
```

Putting the two code fragments above into loops that called them a hundred times and measuring the time produced the following results:

```
index operator 2.604 seconds
getArray,setArray 1.092 seconds
```

Thus the **getArray** and **setArray** method is more than twice as fast.

### 2.1.1 loopnextindex

Several keyword functions are available in **GAUSS** for looping with arrays. The problem in the previous section for example can be written using these functions rather than with **for** loops:

## 2. ASSIGNING ARRAYS

```
sigind = r + 1;
sigloop:

    sig0ind = sigind[1];
    thetainsd = 1;
    thetaloo:

        sig0ind = sig0ind - 1;

        W = amult(getArray(theta, thetainsd), atranspose(getArray(E, sig0ind), 2|1));
        setarray sigma, sigind, getArray(sigma, sigind)+amult(W, atranspose(W, 2|1));

    loopnextindex thetaloo, thetainsd, aord;

sig0ind = sigind;
phiind = 1;
philoo:

    sig0ind[1] = sig0ind[1] - 1;
    setarray sigma, sigind, getArray(sigma, sigind)+
        areshape(amult(amult(getArray(phi, phiind),
            getArray(sigma, sig0ind)), getArray(phi, phiind)), 3|3);

    loopnextindex philoo, phiind, gord;

loopnextindex sigloop, sigind, sigord;
```

The **loopnextindex** function in this isn't faster than the **for** loop used in the previous section primarily because the code is looping only through the first dimension in each loop. The advantages of **loopnextindex**, **previousindex**, **nextindex** and **walkindex** are when the code is looping through the higher dimensions of a highly dimensioned array. In this case looping through an array can be very complicated and difficult to manage using **for** loops and **loopnextindex** can be faster and more useful.

The next example compares two ways of extracting a subarray from a 5-dimensional array.

```
ord = 3|3|3|3|3;

a = areshape(seqa(1, 1, prodc(ord)), ord);
b = eye(3);
```

## 2. ASSIGNING ARRAYS

```
for i(1,3,1);
  for j(1,3,1);
    for k(1,3,1);
      setarray a,i|j|k,b;
    endfor;
  endfor;
endfor;

ind = { 1,1,1 };
loopi:

    setarray a,ind,b;

loopnextindex loopi,ind,ord;
```

Calling each loop 10,000 times and measuring the time each takes we get

*for loop* 1.171 seconds

*loopnextindex* .321 seconds

In other words, **loopnextindex** is about four times faster, a very significant difference.

## Chapter 3

# Miscellaneous Array Functions

### 3.0.2 atranspose

This function changes the order of the dimensions. For example,

```
a = areshape(seqa(1,1,12),2|3|2);
print a;
```

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

```
    1.0000    2.0000
    3.0000    4.0000
    5.0000    6.0000
```

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

```
    7.0000    8.0000
    9.0000   10.0000
   11.0000   12.0000
```

```
/*
** swap 2nd and 3rd dimension
```

### 3. MISCELLANEOUS ARRAY FUNCTIONS

```
*/  
  
print atranspose(a,1|3|2);  
  
Plane [1,...]  
  
      1.0000      3.0000      5.0000  
      2.0000      4.0000      6.0000  
  
Plane [2,...]  
  
      7.0000      9.0000      11.000  
      8.0000     10.000      12.000  
  
/*  
** swap 1st and 3rd dimension  
*/  
  
print atranspose(a,3|2|1);  
  
Plane [1,...]  
  
      1.0000      7.0000  
      3.0000      9.0000  
      5.0000     11.000  
  
Plane [2,...]  
  
      2.0000      8.0000  
      4.0000     10.000  
      6.0000     12.000  
  
/*  
** move 3rd into the front  
*/  
  
print atranspose(a,3|1|2);  
  
Plane [1,...]  
  
      1.0000  3.0000  5.0000  
      7.0000  9.0000 11.000
```

### 3. MISCELLANEOUS ARRAY FUNCTIONS

```
Plane [2,...]
      2.0000   4.0000   6.0000
      8.0000  10.0000  12.0000
```

#### 3.0.3 amult

This function performs a matrix multiplication on the last two trailing dimensions of an array. The leading dimensions must be strictly conformable, and the last two trailing dimensions must be conformable in the matrix product sense. For example,

```
a = areshape(seqa(1,1,12),2|3|2);
b = areshape(seqa(1,1,16),2|2|4);
c = amult(a,b);
print a;
```

```
Plane [1,...]
      1.0000   2.0000
      3.0000   4.0000
      5.0000   6.0000
```

```
Plane [2,...]
      7.0000   8.0000
      9.0000  10.0000
     11.0000  12.0000
print b;
```

```
Plane [1,...]
      1.0000   2.0000   3.0000   4.0000
      5.0000   6.0000   7.0000   8.0000
```

```
Plane [2,...]
      9.0000   10.0000  11.0000  12.0000
     13.0000  14.0000  15.0000  16.0000
print c;
```

### 3. MISCELLANEOUS ARRAY FUNCTIONS

```
Plane [1,...]
      11.000    14.000    17.000    20.000
      23.000    30.000    37.000    44.000
      35.000    46.000    57.000    68.000
```

```
Plane [2,...]
      167.00    182.00    197.00    212.00
      211.00    230.00    249.00    268.00
      255.00    278.00    301.00    324.00
```

Suppose we have a **matrix** of data sets, a  $2 \times 2$  **matrix** of  $100 \times 5$  data sets which we've stored in a  $2 \times 2 \times 100 \times 5$  array called **x**. The moment matrices of these data sets can easily and quickly be computed using **atranspose** and **amult**:

```
vc = amult(atranspose(x,1|2|4|3),x);
```

#### 3.0.4 amean, amin, amax

These functions compute the means, minimums, and maximums respectively across a dimension of an array. The size of the selected dimension of resulting array is shrunk to one and contains the means, minimums or maximums depending on the function called. For example,

```
a = areshape(seqa(1,1,12),2|3|2);
print a;
```

```
Plane [1,...]
      1.0000    2.0000
      3.0000    4.0000
      5.0000    6.0000
```

```
Plane [2,...]
      7.0000    8.0000
```

### 3. MISCELLANEOUS ARRAY FUNCTIONS

```
          9.0000      10.000
          11.000     12.000
```

```
/*
** compute means along third dimension
*/
```

```
print amean(a,3);
```

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

```
          4.0000      5.0000
          6.0000      7.0000
          8.0000      9.0000
```

```
/*
** print means along the second dimension, i.e.,
** down the columns
*/
```

```
print amean(a,2);
```

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

```
          3.0000      4.0000
```

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

```
          9.0000     10.000
```

```
/*
** print the minimums down the columns
*/
```

```
print amin(a,2);
```

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

```
          1.0000      2.0000
```

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

### 3. MISCELLANEOUS ARRAY FUNCTIONS

```
7.0000      8.0000

/*
** print the maximums along the third dimension
*/

print amax(a,3);

Plane [1,...]

7.0000      8.0000
9.0000     10.000
11.000     12.000
```

#### 3.0.5 getDims

This function returns the number of dimensions of an array.

```
a = arrayinit(4|4|5|2,0);
print getdims(a);

4.00
```

#### 3.0.6 getOrders

This function returns the sizes of each dimension of an array. The length of the vector returned by **getOrders** is the dimension of the array.

```
a = arrayinit(4|4|5|2,0);
print getOrders(a);

4.00
4.00
5.00
2.00
```

### 3. MISCELLANEOUS ARRAY FUNCTIONS

#### 3.0.7 arraytomat

This function converts an array with two or fewer dimension to a matrix.

```
a = arrayinit(2|2,0);
b = arraytomat(a);
type(a);
    21.000
type(b);
    6.0000
```

#### 3.0.8 mattoarray

This function converts a matrix to an array.

```
b = rndn(2,2);
a = mattoarray(b);
type(b);
    6.0000

type(a);
    21.000
```

### 3. MISCELLANEOUS ARRAY FUNCTIONS

## Chapter 4

# Using Arrays with GAUSS functions

Many of the **GAUSS** functions have been re-designed to work with arrays. There are two general approaches to this implementation. There are exceptions however, and you are urged to refer to the documentation if you are not sure how a particular **GAUSS** function handles array input.

In the first approach, the function returns an element by element result that is strictly conformable to the input. For example, **cdfnc** returns an array of identical size and shape to the input array:

```
a = areshape(seqa(-2, .5, 12), 2|3|2);  
b = cdfnc(a);  
print b;
```

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

0.9772	0.9331
0.8413	0.6914
0.5000	0.3085

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

#### 4. USING ARRAYS WITH GAUSS FUNCTIONS

```
0.1586    0.0668
0.0227    0.0062
0.0013    0.0002
```

In the second approach, which applies generally to **GAUSS** matrix functions, the function operates on the matrix defined by the last two trailing dimensions of the array. Thus, given a  $5 \times 10 \times 3$  array, **moment** returns a  $5 \times 3 \times 3$  array of five moment matrices computed from the five  $10 \times 3$  matrices in the input array.

Only the last two trailing dimensions matter, i.e., given a  $2 \times 3 \times 4 \times 5 \times 10 \times 6$  array, **moment** returns a  $2 \times 3 \times 4 \times 5 \times 6 \times 6$  array of moment matrices.

For example, in the following the result is a  $2 \times 3$  array of  $3 \times 1$  vectors of singular values of a  $2 \times 3$  array of  $6 \times 3$  matrices.

```
a = areshape(seqa(1,1,108),2|3|6|3);
b=svds(a);
print b;
```

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

```
45.894532
1.6407053
1.2063156e-015
```

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

```
118.72909
0.63421188
5.8652600e-015
```

```
Plane [1,3,..]
```

```
194.29063
0.38756064
1.7162751e-014
```

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

```
270.30524
0.27857175
```

#### 4. USING ARRAYS WITH GAUSS FUNCTIONS

```
1.9012118e-014

Plane [2,2,..]

    346.47504
    0.21732995
    1.4501098e-014

Plane [2,3,..]

    422.71618
    0.17813229
    1.6612287e-014
```

It might be tempting to conclude from this example that in general **GAUSS** function's behavior on the last two trailing dimensions of an array is strictly analogous as the **GAUSS** function's behavior on a matrix. This may be true with some of the functions but not all. For example, the **GAUSS meanc** function returns a column result for matrix input. However, the behavior for the **GAUSS amean** function is not analogous. This function takes a second argument that specified on which dimension the mean is to be taken. That dimension is then collapsed to a size of 1. Thus,

```
a = areshape(seqa(1,1,24),2|3|4);
print a;

Plane [1,..]

    1.000  2.000  3.000  4.000
    5.000  6.000  7.000  8.000
    9.000 10.000 11.000 12.000

Plane [2,..]

    13.000 14.000 15.000 16.000
    17.000 18.000 19.000 20.000
    21.000 22.000 23.000 24.000

/*
** means computed across rows
*/
```

#### 4. USING ARRAYS WITH GAUSS FUNCTIONS

```
b = amean(a,1);
print b;

Plane [1,..]

2.500
6.500
10.500

Plane [2,..]

14.500
18.500
22.500

/*
** means computed down columns
*/

c = amean(a,2);
print c;

Plane [1,..]
5.000 6.000 7.000 8.000

Plane [2,..]
17.000 18.000 19.000 20.000

/*
** means computed along 3rd dimension
*/

d = amean(a,3);
print d;

Plane [1,..]

7.000 8.000 9.000 10.000
11.000 12.000 13.000 14.000
15.000 16.000 17.000 18.000
```

## Chapter 5

# A Panel Data Model

Suppose we have  $N$  cases observed at  $T$  times. Let  $y_{it}$  be the an observation on a dependent variable for the  $i$ -th case at time  $t$ ,  $X_{it}$  an observation of  $k$  independent variables for the  $i$ -th case at time  $t$ ,  $B$ , a  $K \times 1$  vector of coefficients. Then

$$y_{it} = X_{it}B + \mu_i + \epsilon_{it}$$

is a variance components model where  $\mu_i$  is a random error term uncorrelated with  $\epsilon_{it}$  but which is correlated within cases. This implies an  $NT \times NT$  residual moment matrix that is block diagonal with  $N T \times T$  moment matrices with the following form

$$\begin{bmatrix} \sigma_\mu^2 + \sigma_\epsilon^2 & \sigma_\mu^2 & \cdots & \sigma_\mu^2 \\ \sigma_\mu^2 & \sigma_\mu^2 + \sigma_\epsilon^2 & \cdots & \sigma_\mu^2 \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_\mu^2 & \sigma_\mu^2 & \cdots & \sigma_\mu^2 + \sigma_\epsilon^2 \end{bmatrix}$$

The log-likelihood for this model is

$$\ln L = -0.5(NT \ln(2\pi) - \ln|\Omega| + (Y - XB)' \Omega^{-1} (Y - XB))$$

where  $\Omega$  is the block-diagonal moment matrix of the residuals.

### Computing the Log-likelihood

Using **GAUSS** arrays we can compute the log-likelihood of this model without resorting to do loops. Let **Y** be a  $100 \times 3 \times 1$  array of observations on the dependent variable, and **X** a  $100 \times 3 \times 5$  array of observations on the independent variables. Further let **B** be a  $5 \times 1$  vector of coefficients, and **sigu** and **sige** be the residual variances of  $\mu$  and  $\epsilon$  respectively. Then in explicit steps we compute

```

N = 100;
T = 3;
K = 5;
sigma = sigu * ones(T,T) + sige * eye(T); /* TxT sigma */
sigmai = invpd(sigma); /* sigma inverse */
lndet = N*ln(detl);
E = Y - amult(X,areshape(B,N|K|1)); /* residuals */
Omegai = areshape(sigmai,N|T|T); /* diagonal blocks stacked */
/* in a vector array */
R1 = amult(atranspose(E,1|3|2),Omegai); /* E'Omegai */
R2 = amult(R1,E); /* R1*E */

lnL = -0.5*(N*T*ln(2*pi) - lndet + asum(R2,3)); /* log-likelihood */

```

All of this can be made more efficient by nesting statements which eliminates copying of temporary intervening arrays to local arrays. It is also useful to add a check for the positive definiteness of **sigma**.

```

N = 100;
T = 3;
K = 5;
const = -0.5*N*T*ln(2*pi);
oldt = trapchk(1);
trap 1,1;
sigmai = invpd(sigu*ones(T,T)+sige*eye(T));
trap oldt,1;
if not scalmiss(sigmai);
    E = Y - amult(X,areshape(B,N|K|1));

    lnL = const + 0.5*N*ln(detl) -
        0.5*asum(amult(amult(atranspose(E,1|3|2),areshape(sigmai,N|T|T)),E),3);
else;
    lnL = error(0);
endif;

```

## Chapter 6

# Appendix

This is an incomplete List special functions for working with arrays. Many **GAUSS** functions have been modified to handle arrays and are not Listed here. For example **cdfnc** computes the complement of the Normal cdf for each element of an array just as it would for a matrix. See the documentation for these **GAUSS** functions for information about their behavior with arrays.

**aconcat** Concatenates conformable matrices and arrays in a user-specified dimension.

**aeeye** creates array of identity matrices

**amax** computes the maximum elements across a dimension of an array

**amean** computes the mean along one dimension of an array

**amin** computes the minimum elements across a dimension of an array

**amult** performs a matrix multiplication on the last two trailing dimensions of an array

**areshape** Reshapes a scalar, matrix, or array into an array of user-specified size.

**arrayalloc** Creates an N-dimensional array with unspecified contents.

**arrayinit** Creates an N-dimensional array with a specified fill value.

- arraytomat** Changes an array to type matrix.
- asum** computes the sum across one dimension of an array
- atranspose** Transposes an N-dimensional array.
- getarray** Gets a contiguous subarray from an N-dimensional array.
- getdims** Gets the number of dimensions in an array.
- getmatrix** Gets a contiguous matrix from an N-dimensional array.
- getmatrix4D** Gets a contiguous matrix from a 4-dimensional array.
- getorders** Gets the vector of orders corresponding to an array.
- getscalar3D** Gets a scalar from a 3-dimensional array.
- getscalar4D** Gets a scalar form a 4-dimensional array.
- loopnextindex** Increments an index vector to the next logical index and jumps to the specified label if the index did not wrap to the beginning.
- mattoarray** Changes a matrix to a type array.
- nextindex** Returns the index of the next element or subarray in an array.
- previousindex** Returns the index of the previous element or subarray in an array.
- putarray** Puts a contiguous subarray into an N-dimensional array and returns the resulting array.
- setarray** Sets a contiguous subarray of an N-dimensional array.
- walkindex** Walks the index of an array forward or backward through a specified dimension.