CMAT Newsletter: January 2003

Wolfgang M. Hartmann

January 2003

1 General Remarks

Improved *Installation Instructions* were posted on this website. The Tcl/TK frontend was extended. Some new developments are discussed below.
One of my friends at the University of Heidelberg asked me if CMAT would be able to compute multivariate normal $\mathcal{N}(\mu, \Sigma)$ distributed random vectors for *large* but sparse covariance matrices $\Sigma \in \mathcal{R}^{n \times n}$, where *large* means about $n \in [10000, 20000]$. I did spend a few hours to look into this and wrote a small script which could be a useful illustration for others here too and is on the end of this newsletter.

2 Tcl/TK Frontend

A new version of the frontend file $cmat\backslash mytst\backslash cmat - fe.tcl$ was posted here on January 27, 2003. It corrects a font problem which was only with the first eight delivered copies.
Now in addition to the `CTRL c`, `CTRL v`, and `CTRL x` keys for copying and pasting inside the display the following keys can be used:

Page Up, Page Down for moving cursor and view one page up or down in the output.

Home, End for moving the cursor to the begin or end of the current line.

CTRL Home, CTRL End for moving the cursor (and view) to the first or last line of the output window.

The scroll bar provided this functionality already, but using the keys for scrolling back the output window may be preferred by some of the users.

3 New Developments

New developments in CMAT are described in more detail in a corresponding `newdev.pdf` file on this site. Here only some short comments.

3.1 Function `isoreg`

This function implements various methods of isotone regression . The method is also known as *optimal scaling* (see function `opscal()` in SAS/IML[**?**]) or *PAVA* (*Pool Adjacent Violators Algorithm*), see Miles (1959)[**?**], Kruskal (1964)[**?**]. Given n vectors x, predictor, y, response, and w weights, the n vector $\hat{y}$ is computed which minimizes the weighted least squares criterion

$$\min sum_{i=1}^{n} w_i(\hat{y}_i - y_i)^2$$

subject to the restriction that the rank order of the $haty_i$ values is the same as that of the given x_i values. The following modifications resp. extensions are implemented:

1. Kruskal's version A and B for treating ties. The preferred version A is default.
2. An extension of PAVA for nominal data.
3. The *running average algorithm* by Young (1975) which is slightly faster but in most cases only finds an approximate optimal solution.

Descriptions of all algorithms can be found in the authors habilitation (Hartmann, 1979, [**?**], pp. 207). The version in SAS/IML[**?**]), written by Warren Kuhfeld, does not permit the specification of weights and seems to have a problem with ties. The example in the SAS/IML[**?**]) Manual does not give an optimal solution with smallest $(\hat{y}_i - y_i)^2$.

3.2 Function `varsel`

Multiple response variable selection. A given data set a contains ny columns corresponding to a set of response variables and nx columns corresponding to a set of predictor variables. Subsets of x variables are selected so that they predict best (in some specific sense) the set of y variables using the linear model. The following algorithms are implemented:

1. forward selection of x variables
2. forward stepwise selection of x variables
3. backward selection
4. all subset combinations of x variables
5. K randomly generated subsets of x variables

The following criteria for selecting a *best* subset are implemented:

1. Residual Mean Square Error (MSE)
2. Coefficient of Multiple Determination (R Squared, R^2)

3. Adjusted R Squared
4. Mallows C_p
5. Akaike's Information Criterion (AIC)
6. Small-Sample Corrected Akaike's Information Criterion ($AICc$)
7. Schwarz's Bayesian Criterion (BIC)
8. Hannan and Quinn Information Criterion (HQ)
9. Small-Sample Corrected Hannan and Quinn Information Criterion (HQc)

See A.A. Al-Subaihi(2002) ([**?**]) or Miller ([**?**]) for more details. However, there are differences in the results when backward selection is used and it looks as if the Code at JSS has a problem.

4 Sampling in $\mathcal{N}(\mu, \Sigma)$

4.1 Functions `spmat()` and `mrand()`

The function `spmat` can be used to create an $n \times m$ matrix $\mathbf{A}$ when a set of row and column indices is given which defines the K nonzero entries of $\mathbf{A}$. The most simple form is *coordinate oriented* where K pairs (i, j) of the row and column indices of all nonzero entries are specified. A slightly more economic version is the row oriented specification, where the row index vector contains only n entries specifying the start indices of each row in the column index vector.
Since a covariance matrix is symmetric we specify here only the nonzero entries of the lower triangle. We must specify the number of rows and columns, since there may be zero rows or columns at the end.

```
nr = nc = 5;
rind1 = [ 1  2  3   4  4   5  5 ];
rind2 = [ 1  2  3   5      7     ];
cind  = [ 1  2  3   1  4   2  5  ];
vals  = [ 5. 5. 5.  1. 5.  2. 5. ];

a  = spmat(nr,nc,rind1,cind,vals,"coord");
c1 = (tri2sym)a;
print "A and C1", a, c1;
b  = spmat(nr,nc,rind2,cind,vals,"rowor");
c2 = (tri2sym)b;
print "B and C2", b, c2;
c3 = spmat(nr,nc,rind2,cind,vals,"rowsy");
print "Matrix C3", c3;
```

A and C1

L	1	2	3	4	5
1	5.00000	0	0	0	0
2	0.00000	5.00000	0	0	0
3	0.00000	0.00000	5.00000	0	0
4	1.00000	0.00000	0.00000	5.00000	0
5	0.00000	2.00000	0.00000	0.00000	5.00000

S	1	2	3	4	5
1	5.00000				
2	0.00000	5.00000			
3	0.00000	0.00000	5.00000		
4	1.00000	0.00000	0.00000	5.00000	
5	0.00000	2.00000	0.00000	0.00000	5.00000

B and C2

L	1	2	3	4	5
1	5.00000	0	0	0	0
2	0.00000	5.00000	0	0	0
3	0.00000	0.00000	5.00000	0	0
4	1.00000	0.00000	0.00000	5.00000	0
5	0.00000	2.00000	0.00000	0.00000	5.00000

S	1	2	3	4	5
1	5.00000				
2	0.00000	5.00000			
3	0.00000	0.00000	5.00000		
4	1.00000	0.00000	0.00000	5.00000	
5	0.00000	2.00000	0.00000	0.00000	5.00000

Matrix C3

S	1	2	3	4	5
1	5.00000				
2	0.00000	5.00000			
3	0.00000	0.00000	5.00000		
4	1.00000	0.00000	0.00000	5.00000	
5	0.00000	2.00000	0.00000	0.00000	5.00000

Now we create uniform random mean vector μ. Before calling `rand` or `mrand` we set the seed for the random generator. This makes the results unique. Otherwise the computer's clock time is used for the seed and the results are different. The result z of the `mrand` function with argument `"mnor"` is $\mathcal{N}(\mu, \Sigma)$ distributed.

```
srand(1);
mu = rand(nr,1);
z = mrand("mnor",mu,c1)‘;
print z;
```

```
   |         1         2         3         4         5
-------------------------------------------------------------
 1 |  -4.73467   1.93290   2.86649   0.83277   0.25110
```

If we want to create R random vectors $z \in \mathcal{N}(\mu, \Sigma)$ collected in a $R \times n$ matrix $\mathbf{Z}$ we can do this in a slow and a fast way. For a slow version we can use a `for` loop where Cholesky is done each time:

```
zmat = z;
for (i = 2; i <= 100; i++) {
  y = mrand("mnor",mu,c1)‘;
  zmat = zmat |> y;
}
nu = univar(zmat,"ari");
print "Input and sample mean vector", mu,nu;
cov = bivar(zmat,"cov");
print "Input and sample COV matrix", c1,cov;
```

```
Input and sample mean vector
   |         1
-------------
 1 |   0.18496
 2 |   0.97009
 3 |   0.39982
 4 |   0.25940
 5 |   0.92160

   |         1         2         3         4         5
-------------------------------------------------------------
 1 |  -0.08085   1.03579   0.51700   0.42361   1.14796

Input and sample COV matrix
 S |         1         2         3         4         5
-------------------------------------------------------------
 1 |   5.00000
 2 |   0.00000   5.00000
 3 |   0.00000   0.00000   5.00000
 4 |   1.00000   0.00000   0.00000   5.00000
 5 |   0.00000   2.00000   0.00000   0.00000   5.00000

 S |         1         2         3         4         5
-------------------------------------------------------------
```

```
 1 |   4.17940
 2 |   0.16524   5.11762
 3 |  -0.20095   0.26677   5.73816
 4 |   0.85488   1.28437   0.14584   5.71470
 5 |   0.20751   1.53842   0.25701   0.23071   4.02369
```

For a much faster version we use a $n \times R$ mean matrix $[\mu, \ldots, \mu]$ as second input argument of `mrand`. We use the Kronecker product with a vector ones to create a matrix with identical columns μ:

```
mumat = mu @ cons(1,1000,1.);
zmat = mrand("mnor",mumat,c1)';
nu = univar(zmat,"ari");
print "Input and sample mean vector", mu,nu;
cov = bivar(zmat,"cov");
print "Input and sample COV matrix", c1,cov;
```

```
Input and sample mean vector
   |         1
-------------
 1 |   0.18496
 2 |   0.97009
 3 |   0.39982
 4 |   0.25940
 5 |   0.92160

   |         1         2         3         4         5
-----------------------------------------------------------
 1 |   0.01727   0.91006   0.41082   0.12799   0.93299

Input and sample COV matrix
 S |         1         2         3         4         5
-----------------------------------------------------------
 1 |   5.00000
 2 |   0.00000   5.00000
 3 |   0.00000   0.00000   5.00000
 4 |   1.00000   0.00000   0.00000   5.00000
 5 |   0.00000   2.00000   0.00000   0.00000   5.00000

 S |         1         2         3         4         5
-----------------------------------------------------------
 1 |   4.76396
 2 |  -0.19142   4.91754
 3 |   0.22594  -0.11118   5.06682
 4 |   1.04382  -0.17613   0.00496   5.39699
 5 |  -0.11934   1.59986   0.02629   0.18066   4.73851
```

We verify that for $R = 1000$ replications the given mean vector and covariance matrix are more closely approximeted than it it was for $R = 100$ before. Unfortunately, the function `mrand` of the December 2002 release of CMAT was not well designed for large sparse covariance matrices. It did use a sparse Cholesky factorization, however, without any pivoting. Therefore, using the old implementation of `mrand` could generate a severe amount of nonzero fillin depending on the structure of nonzeros in the original covariance matrix. This has been corrected in the meantime. The new version will work with minimum degree pivoting. The next section describes an approach which could also be used with the December 2002 version of CMAT.

4.2 Do-it-yourself: Use Cholesky and Univariate Normal RAND

Assuming a sparse covariance matrix $\mathbf{\Sigma}$ and a mean vector μ are defined, maybe as described above. For computing the lower triangular Cholesky factor $\mathbf{L}$ we could use one of the two sparse Cholesky decomposition methods with minimum degree ordering:

"add" using the Amestoy-Davis-Duff ([**?**]) method

"eng" using the Ng & Peyton ([**?**]) method

Compute Cholesky factor $\mathbf{L}_1 =$ `cc1`:

```
< c1, pi1 > = chold(sigma,"add");
print c1, pi1;
cc1 = pi1 * c1 * c1‘ * pi1‘;
print "AmDaDu: Error pivoted", ssq(cc1 - sigma);
```

```
 Number Nonzeros (Method=Amestoy-Davis-Duff Minimum Degree): 8
              Time for Symbolic Factorization: 0
               Time for Numeric Factorization: 0

 L |         1         2         3         4         5
---------------------------------------------------------
 1 |   2.23607         0         0         0         0
 2 |   0.00000   2.23607         0         0         0
 3 |   0.00000   0.89443   2.04939         0         0
 4 |   0.00000   0.00000   0.00000   2.23607         0
 5 |   0.00000   0.00000   0.00000   0.44721   2.19089

   |   1   2   3   4   5
------------------------
 1 |   0   0   0   1   0
 2 |   0   1   0   0   0
 3 |   1   0   0   0   0
 4 |   0   0   0   0   1
 5 |   0   0   1   0   0
```

```
AmDaDu: Error pivoted 3.944e-030
```

Compute Cholesky factor $\mathbf{L}_2 =$ cc2:

```
  < c2, pi2 > = chold(sigma,"eng");
  print c2, pi2;
  cc2 = pi2 * c2 * c2‘ * pi2‘;
  print "Ng-Peyton-Liu: Error pivoted", ssq(cc2 - sigma);
```

```
 L |         1         2         3         4         5
-------------------------------------------------------
 1 |   2.23607         0         0         0         0
 2 |   0.00000   2.23607         0         0         0
 3 |   0.00000   0.89443   2.04939         0         0
 4 |   0.00000   0.00000   0.00000   2.23607         0
 5 |   0.00000   0.00000   0.00000   0.44721   2.19089

   |   1   2   3   4   5
------------------------
 1 |   0   0   0   0   1
 2 |   0   0   1   0   0
 3 |   1   0   0   0   0
 4 |   0   0   0   1   0
 5 |   0   1   0   0   0
```

```
Ng-Peyton-Liu: Error pivoted 3.944e-030
```

Some time ago we did compare the two methods in a small simulation study and found that the Amestoy-Davis-Duff method many times created slightly less fillin, but took slightly longer to compute than the fast Fortran implementation by Ng-Peyton. (My C code of the Amestoy-Davis-Duff method seems to be slower than Edmond's Fortran code.) However, for a specific matrix $\mathbf{A}$ as yours, the behavior maybe exactly the opposite.
To compute a vector z which is approximately $\mathcal{N}(\mu, \Sigma)$ distributed we have to compute $\mathbf{L}x + \mu$ where vector x is uniform normal $\mathcal{N}(0, 1)$.

```
  x = rand(nr,1,’g’,"nor");
  z1 = pi * c1 * x + mu;
  z2 = pi * c2 * x + mu;
  print "Should be the same:", z1, z2;
```

```
Should be the same:
   |         1
--------------
 1 |  -0.32703
 2 |   0.94723
 3 |  -0.07425
 4 |  -5.10947
 5 |   3.34691
```

```
   |         1
-------------
 1 |  -0.32703
 2 |   0.94723
 3 |  -0.07425
 4 |  -5.10947
 5 |   3.34691
```

For generating R random vectors $z \in \mathcal{N}(\mu, \Sigma)$ collected in a $R \times n$ matrix $\mathbf{Z}$ we can do this in a slow and a fast way. For a slow version we can use a `for` loop where each vector is computed once at each time:

```
  zmat = z1‘; pc = pi * c1;
  for (i = 2; i <= 1000; i++) {
    x = rand(nr,1,’g’,"nor");
    y = pc * x + mu;
    zmat = zmat |> y‘;
  }
  nu = univar(zmat,"ari");
  print "Input and sample mean vector", mu,nu;
  print mu,nu;
  cov = bivar(zmat,"cov");
  print "Input and sample COV matrix", sigma,cov;
```

```
Input and sample mean vector
   |         1
-------------
 1 |   0.18496
 2 |   0.97009
 3 |   0.39982
 4 |   0.25940
 5 |   0.92160

   |         1         2         3         4         5
-----------------------------------------------------
 1 |   0.19503   0.86880   0.44354   0.22714   0.95730

Input and sample COV matrix
 S |         1         2         3         4         5
-----------------------------------------------------
 1 |   5.00000
 2 |   0.00000   5.00000
 3 |   0.00000   0.00000   5.00000
 4 |   1.00000   0.00000   0.00000   5.00000
 5 |   0.00000   2.00000   0.00000   0.00000   5.00000

 S |         1         2         3         4         5
```

```
-------------------------------------------------
 1 |   4.77447
 2 |  -0.11385   5.04185
 3 |   0.20660   0.01580   5.66547
 4 |   1.09493   0.01435  -0.02033   4.84804
 5 |  -0.24929   2.02264   0.17657  -0.04911   5.04486
```

For a much faster version we use a $n \times R$ mean matrix $[\mu, \ldots, \mu]$ and $n \times R$ matrix $\mathbf{X}$ with uniform normal $\mathcal{N}(0, 1)$ distributed values. Again, we use the Kronecker product with a vector ones to create a matrix `mumat` with identical columns μ:

```
Input and sample mean vector
   |         1
-------------
 1 |   0.18496
 2 |   0.97009
 3 |   0.39982
 4 |   0.25940
 5 |   0.92160

   |         1         2         3         4         5
-----------------------------------------------------
 1 |   0.21963   0.85103   0.37769   0.30082   0.82342

Input and sample COV matrix
 S |         1         2         3         4         5
-----------------------------------------------------
 1 |   5.00000
 2 |   0.00000   5.00000
 3 |   0.00000   0.00000   5.00000
 4 |   1.00000   0.00000   0.00000   5.00000
 5 |   0.00000   2.00000   0.00000   0.00000   5.00000

 S |         1         2         3         4         5
-----------------------------------------------------
 1 |   4.92463
 2 |   0.07850   5.34932
 3 |   0.09516  -0.07955   4.91030
 4 |   0.88459   0.13079  -0.16772   5.17689
 5 |   0.26717   1.95758   0.00199   0.05116   4.96554
```

As we can see, the compiled matrix algebra is much faster than interpretung the `for` loop.

4.3 Large Scale Application

Now we create a much larger covariance matrix with $n = 5000$ by diagonal concatenation of the small sparse matrix above. We also determine the computer

time and print the attributes of the large covariance matrix.

```
nr = nc = 5;
rind = [ 1  2  3   5      7     ];
cind = [ 1  2  3   1  4   2  5  ];
vals = [ 5. 5. 5.  1. 5.  2. 5. ];
c = spmat(nr,nc,rind,cind,vals,"rowsy");
print "Matrix C=",c;
t1 = time("clo");
```

Instead of concatenating two thousand sparse 5×5 matrices which would take a long time we concatenate in three steps:

1. onehundred 5×5 matrices are concatenated to one 500×500 matrix,
2. ten 500×500 matrices are concatenated to one 5000×5000 matrix,
3. two 5000×5000 matrices are concatenated to one 10000×1000 matrix.

```
sigma = c;
for (i = 2; i <= 100; i++) sigma = sigma \> c;
/* from 500 to 5000 */
c = sigma;
for (i = 2; i <= 10; i++) sigma = sigma \> c;
/* from 5000 to 10000 */
sigma = sigma \> sigma;
dt1 = time("clo") - t1;
print "Attributes of Cov:"; attrib(sigma);
```

```
Matrix C=
 S |         1         2         3         4         5
-------------------------------------------------------
 1 |   5.00000
 2 |   0.00000   5.00000
 3 |   0.00000   0.00000   5.00000
 4 |   1.00000   0.00000   0.00000   5.00000
 5 |   0.00000   2.00000   0.00000   0.00000   5.00000

Attributes of Cov:

          ---------------------------------------
                  Table of Attributes
          ---------------------------------------
          Object Name      name            sigma
          Object Type      otyp       matrix_sym
          Data Type        dtyp             real
          Storage Type     styp      spar_packed
          Row Names        rnam                0
```

```
Column Names     cnam                0
Row Labels       rlab                0
Column Labels    clab                0
Number Rows      nrow            10000
Number Columns   ncol            10000
Lower Bandwidth  lbw                 3
Upper Bandwidth  ubw                 3
Size in Bytes    size           264240
String Length    slen                0
Number Strings   nstr                0
Number MissVals  nmis                0
Number NonzeroV  nzer            14000
Smallest Value   vmin                0
Largest Value    vmax                5
Frobenius Norm   nrm2                .
Recip Condition  rcond               .
Determinant      det                 .
Largest SingVal  svb                 .
Smallest SingV   svs                 .
Num Rank Estim   rnk                 .
---------------------------------------
```

Unfortunately, due to not enough core memory we can only compute $R = 100$ vectors z each with $n = 10000$ (nonzero) entries. First, we compute the Cholesky factor and then we apply to the $R = 100$ uniform normal vectors x:

```
srand(1);
n = nrow(sigma); mu = rand(n,1);
t2 = time("clo");
< lo, pi > = chold(sigma,"eng");
dt2 = time("clo") - t2;
t3 = time("clo");
pc = pi * lo;
R = 100;
x = rand(n,R,'g',"nor");
mumat = mu @ cons(1,R,1.);
zmat = pc * x + mumat;
dt3  = time("clo") - t3;
dt23 = dt2 + dt3;
print "Attributes of ZMAT:"; attrib(zmat);
print "Timings=",dt1,dt2,dt3;
```

These are the attributes of the 10000×100 matrix $\mathbf{Z}$:

```
Attributes of ZMAT:
          ---------------------------------------
                   Table of Attributes
```

```
--------------------------------------
Object Name      name              zmat
Object Type      otyp       matrix_gen
Data Type        dtyp             real
Storage Type     styp        dens_full
Row Names        rnam                0
Column Names     cnam                0
Row Labels       rlab                0
Column Labels    clab                0
Number Rows      nrow            10000
Number Columns   ncol              100
Lower Bandwidth  lbw              9999
Upper Bandwidth  ubw                99
Size in Bytes    size           8000240
String Length    slen                0
Number Strings   nstr                0
Number MissVals  nmis                0
Number NonzeroV  nzer          1000000
Smallest Value   vmin            -10.6
Largest Value    vmax            10.77
Frobenius Norm   nrm2                .
Recip Condition  rcond               .
Determinant      det                 .
Largest SingVal  svb                 .
Smallest SingV   svs                 .
Num Rank Estim   rnk                 .
--------------------------------------
```

```
Timings= 307.407 7.0160 128.359
```

This means, creating the large sparse covariance matrix took most of the time. Therefore, I will have to spend some time to improve the speed of sparse diagonal concatenation.