GAUSS™
Programming Language

Structures and SqpSolvemt

Version 1.0.0
November 12, 2003

Aptech Systems, Inc.
Maple Valley, WA

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

November 12, 2003

Chapter 1

Basic Structures

1.0.1 Structure Definition

The syntax for a structure definition is as follows

```
struct A {  /* list of members */ };
```

The list of members can include scalars, arrays, matrices, strings, string arrays, as well as other structures. As a type, scalars are unique to structures and don't otherwise exist.

For example, the following defines a structure containing the possible contents,

```
struct generic_example {

            scalar x;
            matrix y;
            string s1;
            string array s2
            struct other_example t;

              };
```

A useful convention is to put the structure definition into a file with an `.sdf` extension. Then for any command file or source code file that requires this definition put

```
#include filename.sdf
```

For example

```
#include example.sdf
```

These statements create structure definitions that persist until the workspace is cleared. They do not create structures, only structure type definitions. The next section describes how to create an instance of a structure.

1.0.2 Declaring an Instance

To use a structure it is necessary to declare an instance. The syntax for this is

```
struct structure_type  structure_name;
```

For example

```
#include example.sdf
struct generic_example p0;
```

1.0.3 Initializing an Instance

Members of structures are referenced using a "dot" syntax:

```
p0.x = rndn(20,3);
```

The same syntax applies when referred to on the right-hand side:

```
mn = meanc(p0.x);
```

Initialization of Global Structures

Global structures are initialized at compile time. Each member of the structure initialized according to the following schedule:

scalar 0, a scalar zero

matrix {}, a empty matrix with zero rows and zero columns

array 0, a one dimensional array set to zero

string "", a null string

string array "", a 1×1 string array set to null

If a global already exists in memory, it will not be reinitialized. It may be the case in your program that when it is rerun, the global variables may need to be reset to default values. That is, your program may depend on certain members of a structure being set to default values which are set to some other value later in the program. When you rerun this program you will want to reinitialize the global structure. To do this, make an assignment to at least one of the members. This can be made convenient by writing a procedure that declares a structure and initializes one of its members to a default value and then returns it. for example,

```
/* ds.src */

#include ds.sdf
proc dsCreate;
    struct DS d0;
    d0.dataMatrix = 0;
    retp(d0);
endp;
```

Calling this function after declaring an instance of the structure will ensure initialization to default values each time your program is run.

```
struct DS d0;
d0 = dsCreate;
```

Initializing Local Structures

Local structures, which are structures defined inside procedures, are initialized at the first assignment. The procedure may have been written in such a way that a subset of structures are used an any one call, and in that case time is saved by not initializing the unused structures. They will be initialized to default values only when the first assignment is made to one of its members.

1.0.4 Arrays of Structures

To create a matrix of instances use the **reshape** command:

```
#include ds.sdf
struct DS p0;
p0 = reshape(dsCreate,5,1);
```

This creates a 5 by 1 vector of instances of `option`, all of the with members initialized to default values.

When the instance members have been set to some other values, reshape will produce multiple copies of that instance set to those values.

Matrices or vectors of instances can also be created by concatenation.

```
#include trade.sdf
struct option p0,p1,p2;
p0 = optionCreate;
p1 = optionCreate;

p2 = p1 | p0;
```

1.0.5 Saving an Instance to the Disk

Instances and vectors or matrices of instances of structures can be saved in a file on the disk, and later loaded from a file on the disk. The syntax for saving an instance to the disk is

```
ret = savestruct(instance,filename);
```

The file on the disk will have an `.fsr` extension.

For example,

```
#include ds.sdf
struct DS p0;
p0 = reshape(dsCreate,2,1);
retc = saveStruct(p2,"p2");
```

This saves the vector of instances in a file called `p2.fsr`. `retc` will be zero if the save was successful and otherwise nonzero.

1.0.6 Loading an Instance from the Disk

The syntax for loading a file containing an instance or matrix of instances is

```
{ instance, retc } = loadstruct(file_name,structure_name);
```

For example,

```
#include trade.sdf;
struct DS p3;
{ p3, retc } = loadstruct("p2","ds");
```

1.1 Passing Structures to Procedures

Structures or members of structures can be passed to procedures. When a structure is passed as an argument to a procedure, it is passed by value. The structure becomes a local copy of the structure that was passed. The data in the structure is not duplicated unless the local copy of the structure has a new value assigned to one of its members.

Structure arguments must be declared in the procedure definition.

```
struct rectangle {
                  matrix ulx;
          matrix uly;
                matrix lrx;
          matrix lry;
         };

proc area(struct rectangle rect);
    retp( (rect.lrx - rect.ulx) .* (rect.uly - rect.lry) );
endp;
```

Local structures are defined using a **struct** statement inside the procedure definition.

```
 proc center(struct rectangle rect);
     struct rectangle cent;

cent.lrx = (rect.lrx - rect.ulx) / 2;
     cent.ulx = -cent.lrx;
cent.uly = (rect.uly - rect.lry) / 2;
     cent.lry = -cent.uly;
retp(cent);

 endp;
```

Chapter 2

Special Structures

There are three common types of structures that will be found in the **GAUSS** Run-Time Library and applications.

The DS and PV structures are defined in the **GAUSS** Run-Time Library. There definitions are found in `ds.sdf` and `pv.sdf` respectively in the `src` source code subdirectory.

Before structures many procedures in the Run-Time Library and all applications had global variables serving a variety of purposes such as setting and altering defaults. Currently these variables are being entered as members of "control" structures.

2.1 The DS Structure

The DS structure, or "data" structure, is a very simple structure. It contains a member for each **GAUSS** data type. The following is found in `ds.sdf`:

```
struct DS {
            scalar type;
            matrix dataMatrix;
            array  dataArray;
            string dname;
            string array vnames;
          };
```

This structure was designed for use by the various optimization functions in **GAUSS**, in particular, **sqpSolvemt**, as well as a set of gradient procedures, **gradmt**, **hessmt**, et al.

These procedures all require that the user provide a procedure computing a function (to be optimized or take the derivative of, etc.) which takes the DS structure as an argument. The Run-Time Library procedures such as **sqpSolvemt** take the DS structure as an argument and pass it on to the user-provided procedure without modification. Thus, the user can put into that structure whatever they might need as data in their procedure.

To initialize an instance of a DS structure, the procedure **dsCreate** is defined in `ds.src`.

```
#include ds.sdf
struct DS d0;
d0 = dsCreate;
```

2.2 The PV Structure

The PV structure, or "parameter vector" structure is used by various optimization, modelling, and gradient procedures, in particular, **sqpSolvemt**, for handling the parameter vector. The **GAUSS** Run-Time Library contains special functions that work with this structure. They are prefixed by "pv" and defined in `pv.src`. These functions store matrices and arrays with parameters in the structure, and retrieve various kinds of information about the parameters and parameter vector from it.

"Packing" into a PV Structure

The various procedures in the Run-Time Library and applications for optimization, modelling, derivatives, etc., all require a parameter vector. Parameters in complex models, however, often come in matrices of various types, and it has been the responsibility of the programmer to generate the parameter vector from the matrices and vice versa. The PV procedures make this problem much more convenient to solve.

The typical situation involves two parts, first, "packing" the parameters into the PV structure, which is then passed to the Run-Time Library procedure or application, and second "unpacking" the PV structure in the user-provided procedure for use in computing the objective function. For example, to pack parameters into PV structure,

```
#include sqpsolvemt.sdf

/* starting values */

b0 = 1;  /* constant in mean equation */
garch = { .1, .1 };  /* garch parameters */
arch = { .1, .1 };  /* arch parameters */
omega = .1 /* constant in variance equation */
```

```
    struct PV p0;

    p0 = pvPack(pvCreate,b0,"b0");
    p0 = pvPack(p0,garch,"garch");
    p0 = pvPack(p0,arch,"arch");
    p0 = pvPack(p0,omega,"omega");

    /* data */

    z = loadd("tseries");

    struct DS d0;
    d0.dataMatrix = z;
```

Next, in the user-provided procedure for computing the objective function, in this case minus the log-likelihood, the parameter vector is unpacked

```
proc ll(struct PV p0, struct DS d0);

    local b0,garch,arch,omega,p,q,h,u,vc,w;

    b0 = pvUnpack(p0,"b0");
    garch = pvUnpack(p0,"garch");
    arch = pvUnpack(p0,"arch");
    omega = pvUnpack(p0,"omega");

    p = rows(garch);
    q = rows(arch);

    u = d0.dataMatrix - b0;
    vc = moment(u,0)/rows(u);

    w = omega + (zeros(q,q) | shiftr((u.*ones(1,q))',seqa(q-1,-1,q))) * arch;

    h = recserar(w,vc*ones(p,1),garch);

    logl = -0.5 * ((u.*u)./h + ln(2*pi) + ln(h));
    retp(logl);

endp;
```

Masked Matrices

The **pvUnpack** function unpacks parameters into matrices or arrays for use in computations. The first argument is a **PV** structure containing the parameter vector. Sometimes the matrix or vector is partly parameters to be estimated (that is, a parameter to be entered in the parameter vector) and partly fixed parameters. To distinguish between estimated and fixed parameters, and additional argument is used in the packing function called a "mask" which is strictly conformable to the input matrix and the elements of which are set to 1 for an estimated parameter and 0 for a fixed parameter. For example,

```
p0 = pvPackm(p0,.1*eye(3),"theta",eye(3));
```

Here just the diagonal of a 3×3 matrix is added to the parameter vector.

When this matrix is unpacked the entire matrix is returned with current values of the parameters on the diagonal.

```
print pvUnpack(p0,"theta");

0.1000  0.0000  0.0000
0.0000  0.1000  0.0000
0.0000  0.0000  0.1000
```

Symmetric Matrices

Symmetric matrices are a special case because even if the entire matrix is to be estimated only the nonredundant portion is to be put into the parameter vector. Thus for them there are special procedures. For example,

```
vc = { 1 .6 .4, .6 1 .2, .4 .2 1 };
p0 = pvPacks(p0,vc,"vc");
```

There is also a procedure for masking in case only a subset of the nonredundant elements are to be included in the parameter vector:

```
vc = { 1 .6 .4, .6 1 .2, .4 .2 1 };
mask = { 1 1 0, 1 1 0, 0 0 1 };
p0 = pvPacksm(p0,vc,"vc",mask);
```

Fast Unpacking

When unpacking matrices using a matrix name, **pvUnpack** has to make a search through a list of names which is relatively time-consuming. This can be alleviated by

using an index rather than a name in unpacking. To do this, though, requires using a special pack procedure that establishes the index:

```
p0 = pvPacki(p0,b0,"b0",1);
p0 = pvPacki(p0,garch,"garch",2);
p0 = pvPacki(p0,arch,"arch",3);
p0 = pvPacki(p0,omega,"omega",4);
```

Now they may be unpacked using the index number

```
b0 = pvUnpack(p0,1);
garch = pvUnpack(p0,2);
arch = pvUnpack(p0,3);
omega = pvUnpack(p0,4);
```

When packed with an index number they may be unpacked either by index or by name, but unpacking by index is faster.

2.2.1 Miscellaneous PV Procedures

pvList

This procedure generates a list of the matrices or arrays packed into the structure.

```
p0 = pvPack(p0,b0,"b0");
p0 = pvPack(p0,garch,"garch");
p0 = pvPack(p0,arch,"arch");
p0 = pvPack(p0,omega,"omega");

print pvList(p0);

    b0
    garch
    arch
    omega
```

pvLength

This procedure returns the length of the parameter vector.

```
print pvLength(p0);

   6.0000
```

pvGetParNames

This procedure generates a list of parameter names:

```
print pvGetParNames(p0);

     b0[1,1]
     garch[1,1]
     garch[2,1]
     arch[1,1]
     arch[2,1]
     omega[1,1]
```

pvGetParVector

This procedure returns the parameter vector itself.

```
print pvGetParVector(p0);

     1.0000
     0.1000
     0.1000
     0.1000
     0.1000
     1.0000
```

pvPutParVector

This procedure replaces the parameter vector with the one in the argument:

```
newp = { 1.5, .2, .2, .3, .3, .8 };
p0 = pvPutParVector(newp);

print pvGetParVector(p0);

     1.5000
     0.2000
     0.2000
     0.3000
     0.3000
     0.8000
```

pvGetIndex

This procedure returns the indices in the parameter vector of the parameters in a matrix. These indices are useful when setting linear constraints or bounds in **sqpSolvemt**. Bounds for example, are set by specifying a $K \times 2$ matrix where K is the length of the parameter vector, and the first column are the lower bounds and the second the upper bounds. To set the bounds for a particular parameter then requires knowing where that parameter is in the parameter vector. This information can be found using **pvGetIndex**. For example,

```
/*
**  get indices of lambda parameters
**  in parameter vector
*/

lind = pvGetIndex(par0,"lambda");

/*
**  set bounds constraint matrix to unconstrained default
*/

c0.bounds = ones(pvLength(par0),1).*(-1e250~1e250);

/*
**  set bounds for lambda parameters to be positive

c0.bounds[lind,1] = zeros(rows(lind),1);
```

2.3 Control Structures

Another important class of structures is the "control" structure. Applications developed before structures were introduced into Gauss typically handled some program specifications by the use of global variables which had some disadvantages, in particular preventing the nesting of calls to procedures.

Currently, the purposes served by global variables are now served by the use of a control structure. For example for **sqpSolvemt**,

```
struct sqpSolvemtControl {
            matrix A;
            matrix B;
            matrix C;
            matrix D;
```

```
    scalar eqProc;
    scalar ineqProc;
    matrix bounds;
    scalar gradProc;
    scalar hessProc;
    scalar maxIters;
    scalar dirTol;
    scalar CovType;
    scalar feasibleTest;
    scalar maxTries;
    scalar randRadius;
    scalar trustRadius;
    scalar seed;
    scalar output;
    scalar printIters;
    matrix weights;
};
```

The members of this structure determine optional behaviors of **sqpSolvemt**:

Chapter 3

sqpSolvemt

sqpSolvemt is a procedure in the **GAUSS** Run-Time Library that solves the general nonlinear programming problem using a Sequential Quadratic Programming descent method, that is, it solves

$$\min f(\theta)$$

subject to

$A\theta = B$	linear equality
$C\theta \geq D$	linear inequality
$H(\theta) = 0$	nonlinear equality
$G(\theta) \geq 0$	nonlinear inequality
$\theta_{lb} \leq \theta \leq \theta_{ub}$	bounds

The linear and bounds constraints are redundant with respect to the nonlinear constraints, but are treated separately for computational convenience.

The call to **sqpSolvemt** has four input arguments and one output argument:

```
out = SQPsolveMT(&fct,P,D,C);
```

3.0.1 Input Arguments

The first input argument is a pointer to the objective function to be minimized. The procedure computing this objective function has two arguments, a **PV** structure containing the start values, and a **DS** structure containing data, if any. For example,

```
proc fct(struct PV p0, struct DS d0);
```

```
        local y, x, b0, b, e, s;
        y = d0[1].dataMatrix;
        x = d0[2].dataMatrix;
        b0 = pvUnpack(p0,"constant");
        b = pvUnpack(p0,"coefficients");
        e = y - b0  - x * b;
        s = sqrt(e'e/rows(e));
        retp(-pdfn(e/s);
endp;
```

Note that this procedure returns a vector rather than a scalar. When the objective function is a properly defined log-likelihood, returning a vector of minus log-probabilities permits the calculation of a QML covariance matrix of the parameters.

The remaining input arguments are structures,

P a **PV** structure containing starting values of the parameters,

D a **DS** structure containing data, if any

C an **sqpSolvemtControl** structure

The **DS** structure is optional. **sqpSolvemt** passes this argument on to the user-provided procedure that `&fct` is pointing to without modification. If there is no data, a default structure can be passed to it.

sqpSolvemt Control Structure

A default **sqpSolvemtControl** structure can be passed in the fourth argument for an unconstrained problem. The members of this structure are as follows

A $M \times K$ matrix, linear equality constraint coefficients: $A\theta = B$ where p is a vector of the parameters.

B $M \times 1$ vector, linear equality constraint constants: $A\theta = B$ where p is a vector of the parameters.

C $M \times K$ matrix, linear inequality constraint coefficients: $C\theta >= D$ where p is a vector of the parameters.

D $M \times 1$ vector, linear inequality constraint constants: $C\theta >= D$ where p is a vector of the parameters.

eqProc scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input

arguments, instances of PV and DS structures, and one output argument, a vector of computed inequality constraints.

Default = {.}, i.e., no inequality procedure.

IneqProc scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, instances of PV and DS structures, and one output argument, a vector of computed inequality constraints.

Default = {.}, i.e., no inequality procedure.

Bounds 1×2 or $K \times 2$ matrix, bounds on parameters. If 1x2 all parameters have same bounds.

Default = { -1e256 1e256 }.

GradProc scalar, pointer to a procedure that computes the gradient of the function with respect to the parameters. When such a procedure has been provided, it has two input arguments, instances of PV and DS structures, and one output argument, the derivatives. If the function procedure returns a scalar, the gradient procedure returns a $1 \times K$ row vector of derivatives. If function procedure turns an $N \times 1$ vector, the gradient procedure returns an $N \times K$ matrix of derivatives.

This procedure may compute a subset of the derivatives. **sqpSolvemt** will compute numerical derivatives for all those elements set to missing values in the return vector or matrix.

Default = {.}, i.e., no gradient procedure has been provided.

HessProc scalar, pointer to a procedure that computes the Hessian, i.e., the matrix of second order partial derivatives of the function with respect to the parameters. When such a procedure has been provided, it has two input arguments, instances of PV and DS structures, and one output argument, a vector of computed inequality constraints. Default = {.}, i.e., Default = {.}, i.e., no Hessian procedure has been provided.

Whether the objective function procedure returns a scalar or vector, the Hessian procedure must return a $K \times K$ matrix. Elements set to missing values will be computed numerically by **sqpSolvemt**.

MaxIters scalar, maximum number of iterations. Default = 1e+5.

MaxTries scalar, maximum number of attemps in random search. Default = 100.

DirTol scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisifed sqpSolvemt exits the iterations.

CovType scalar, if 2, QML covariance matrix, else if 0, no covariance matrix is computed, else ML covariance matrix is computed. For a QML covariance matrix the objective function procedure must return an $N \times 1$ vector of minus log-probabilities.

FeasibleTest scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.

randRadius scalar, If zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = .001.

seed scalar, if nonzero, seeds random number generator for random search, otherwise time in seconds from midnight is used.

trustRadius scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = .001.

output scalar, if nonzero, results are printed. Default = 0.

PrintIters scalar, if nonzero, prints iteration information. Default = 0.

weights vector, weights for objective function returning a vector. Default = 1.

3.0.2 Output Argument

The single output argument is an **sqpSolvemtOut** structure. Its definition is

```
struct SQPsolveMTOut {
            struct PV par;
            scalar fct;
            struct SQPsolveMTLagrange lagr;
            scalar retcode;
            matrix moment;
            matrix hessian;
            matrix xproduct;
          };
```

The members of this structure are

par instance of a **PV** structure containing the parameter estimates are placed in the member matrix **par**.

fct scalar, function evaluated at final parameter estimates

lagr an instance of a SQPLagrange structure containing the Lagrangeans for the constraints. For an instance named lagr, the members are:

lagr.lineq $M \times 1$ vector, Lagrangeans of linear equality constraints,

lagr.nlineq $N \times 1$ vector, Lagrangeans of nonlinear equality constraints

lagr.linineq $P \times 1$ vector, Lagrangeans of linear inequality constraints

lagr.nlinineq $Q \times 1$ vector, Lagrangeans of nonlinear inequality constraints

lagr.bounds $K \times 2$ matrix, Lagrangeans of bounds

Whenever a constraint is active, its associated Lagrangean will be nonzero. For any constraint that is inactive throughout the iterations as well as at convergence, the corresponding Lagrangean matrix will be set to a scalar missing value.

retcode return code:

0 normal convergence

1 forced exit

2 maximum number of iterations exceeded

3 function calculation failed

4 gradient calculation failed

5 Hessian calculation failed

6 line search failed

7 error with constraints

8 function complex

9 feasible direction couldn't be found

3.0.3 Example

Define

$$Y = \Lambda\eta + \theta$$

where Λ is a $K \times L$ matrix of "loadings", η an $L \times 1$ vector of unobserved "latent" variables, and θ an $K \times 1$ vector of unobserved errors. Then

$$\Sigma = \Lambda\Phi\Lambda' + \Psi$$

where Φ is the $L \times L$ covariance matrix of the latent variables, and Ψ is the $K \times K$ covariance matrix of the errors.

The log-likelihood of the i-th observation is

$$logP(i) = -\frac{1}{2}[Kln(2\pi) + ln|\Sigma| + Y(i)\Sigma Y(i)']$$

Not all elements of Λ, Φ, and Ψ can be estimated. At least one element of each column of Λ must be fixed to 1, and Ψ is usually a diagonal matrix.

Constraints

To ensure a well-defined log-likelihood, constraints on the parameters are required to guarantee positive definite covariance matrices. To do this a procedure is written that returns the eigenvalues of Σ and Φ minus a small number. **sqpSolvemt** then finds parameters such that these eigenvalues are greater or equal to that small number.

3.0.4 The Command File

This command file can be found in the file `sqpfact.e` in the .
examples subdirectory.

```
#include sqpsolvemt.sdf

lambda = { 1.0  0.0,
           0.5  0.0,
           0.0  1.0,
           0.0  0.5 };

lmask  = {  0    0,
            1    0,
            0    0,
            0    1 };

phi = { 1.0  0.3,
        0.3  1.0 };

psi = { 0.6  0.0  0.0  0.0,
        0.0  0.6  0.0  0.0,
        0.0  0.0  0.6  0.0,
        0.0  0.0  0.0  0.6 };

tmask = {  1    0    0    0,
           0    1    0    0,
           0    0    1    0,
```

```
            0    0    0    1 };

struct PV par0;
par0 = pvCreate;
par0 = pvPackm(par0,lambda,"lambda",lmask);
par0 = pvPacks(par0,phi,"phi");
par0 = pvPacksm(par0,psi,"psi",tmask);


struct SQPsolveMTControl c0;
c0 = sqpSolveMTcontrolCreate;

lind = pvGetIndex(par0,"lambda"); /* get indices of lambda parameters */
                                  /* in parameter vector              */
tind = pvGetIndex(par0,"psi");  /* get indices of psi  parameters */
                                /* in parameter vector             */

c0.bounds = ones(pvLength(par0),1).*(-1e250~1e250);

c0.bounds[lind,1] = zeros(rows(lind),1);
c0.bounds[lind,2] = 10*ones(rows(lind),1);
c0.bounds[tind,1] = .001*ones(rows(tind),1);
c0.bounds[tind,2] = 100*ones(rows(tind),1);


c0.output = 1;
c0.printIters = 1;
c0.trustRadius = 1;
c0.ineqProc = &ineq;
c0.covType = 1;

struct DS d0;
d0 = dsCreate;
d0.dataMatrix = loadd("maxfact");

output file = sqpfact.out reset;


struct SQPsolveMTOut out0;
out0 = SQPsolveMT(&lpr,par0,d0,c0);

lambdahat = pvUnpack(out0.par,"lambda");
phihat = pvUnpack(out0.par,"phi");
psihat = pvUnpack(out0.par,"psi");
```

```
print "estimates";
print;
print "lambda" lambdahat;
print;
print "phi" phihat;
print;
print "psi" psihat;


struct PV stderr;
stderr = out0.par;

if not scalmiss(out0.moment);
    stderr = pvPutParVector(stderr,sqrt(diag(out0.moment)));

   lambdase = pvUnpack(stderr,"lambda");
   phise = pvUnpack(stderr,"phi");
   psise = pvUnpack(stderr,"psi");

   print "standard errors";
   print;
   print "lambda" lambdase;
   print;
   print "phi" phise;
   print;
   print "psi" psise;
endif;


output off;


proc lpr(struct PV par1, struct DS data1);
    local lambda,phi,psi,sigma,logl;

    lambda = pvUnpack(par1,"lambda");
    phi = pvUnpack(par1,"phi");
    psi = pvUnpack(par1,"psi");
    sigma = lambda*phi*lambda' + psi;

    logl = -lnpdfmvn(data1.dataMatrix,sigma);
    retp(logl);

endp;
```

3. **SQPSOLVEMT**

```
proc ineq(struct PV par1, struct DS data1);
    local lambda,phi,psi,sigma,e;

    lambda = pvUnpack(par1,"lambda");
    phi = pvUnpack(par1,"phi");
    psi = pvUnpack(par1,"psi");
    sigma = lambda*phi*lambda' + psi;
    e = eigh(sigma) - .001; /* eigenvalues of sigma */
    e = e | eigh(phi) - .001;   /* eigenvalues of phi */
    retp(e);

endp;
```