

NAG Library Function Document

nag_kalman_sqrt_filt_cov_var (g13eac)

1 Purpose

nag_kalman_sqrt_filt_cov_var (g13eac) performs a combined measurement and time update of one iteration of the time-varying Kalman filter. The method employed for this update is the square root covariance filter with the system matrices in their original form.

2 Specification

```
#include <nag.h>
#include <naggl3.h>

void nag_kalman_sqrt_filt_cov_var (Integer n, Integer m, Integer p,
    double s[], Integer tds, const double a[], Integer tda,
    const double b[], Integer tdb, const double q[], Integer tdq,
    const double c[], Integer tdc, const double r[], Integer tdr,
    double k[], Integer tdk, double h[], Integer tdh, double tol,
    NagError *fail)
```

3 Description

For the state space system defined by

$$\begin{aligned} X_{i+1} &= A_i X_i + B_i W_i & \text{var}(W_i) &= Q_i \\ Y_i &= C_i X_i + V_i & \text{var}(V_i) &= R_i \end{aligned}$$

the estimate of X_i given observations Y_1 to Y_{i-1} is denoted by $\hat{X}_{i|i-1}$ with $\text{var}(\hat{X}_{i|i-1}) = P_{i|i-1} = S_i S_i^T$. nag_kalman_sqrt_filt_cov_var (g13eac) performs one recursion of the square root covariance filter algorithm, summarised as follows:

$$\begin{pmatrix} R_i^{1/2} & C_i S_i & 0 \\ 0 & A_i S_i & B_i Q_i^{1/2} \end{pmatrix} U = \begin{pmatrix} H_i^{1/2} & 0 & 0 \\ G_i & S_{i+1} & 0 \end{pmatrix}$$

(Pre-array) (Post-array)

where U is an orthogonal transformation triangularizing the pre-array. The triangularisation is carried out via Householder transformations exploiting the zero pattern in the pre-array. The measurement-update for the estimated state vector X is

$$\hat{X}_{i|i} = \hat{X}_{i|i-1} - K_i [C_i \hat{X}_{i|i-1} - Y_i] \quad (1)$$

where K_i is the Kalman gain matrix, whilst the time-update for X is

$$\hat{X}_{i+1|i} = A_i \hat{X}_{i|i} + D_i U_i \quad (2)$$

where $D_i U_i$ represents any deterministic control used. The relationship between the Kalman gain matrix K_i and G_i is given by

$$A_i K_i = G_i \left(H_i^{1/2} \right)^{-1}$$

The function returns the product of the matrices A_i and K_i represented as AK_i , and the state covariance matrix $P_{i|i-1}$ factorized as $P_{i|i-1} = S_i S_i^T$ (see the g13 Chapter Introduction for more information concerning the covariance filter).

4 References

Anderson B D O and Moore J B (1979) *Optimal Filtering* Prentice–Hall

Harvey A C and Phillips G D A (1979) Maximum likelihood estimation of regression models with autoregressive — moving average disturbances *Biometrika* **66** 49–58

Vanbegin M, van Dooren P and Verhaegen M H G (1989) Algorithm 675: FORTRAN subroutines for computing the square root covariance filter and square root information filter in dense or Hessenberg forms *ACM Trans. Math. Software* **15** 243–256

Verhaegen M H G and van Dooren P (1986) Numerical aspects of different Kalman filter implementations *IEEE Trans. Auto. Contr.* **AC-31** 907–917

Wei W W S (1990) *Time Series Analysis: Univariate and Multivariate Methods* Addison–Wesley

5 Arguments

- 1: **n** – Integer *Input*
On entry: the actual state dimension, n , i.e., the order of the matrices S_i and A_i .
Constraint: $n \geq 1$.
- 2: **m** – Integer *Input*
On entry: the actual input dimension, m , i.e., the order of the matrix $Q_i^{1/2}$.
Constraint: $m \geq 1$.
- 3: **p** – Integer *Input*
On entry: the actual output dimension, p , i.e., the order of the matrix $R_i^{1/2}$.
Constraint: $p \geq 1$.
- 4: **s[n × tds]** – double *Input/Output*
Note: the (i, j) th element of the matrix S is stored in $s[(i - 1) \times \mathbf{tds} + j - 1]$.
On entry: the leading n by n lower triangular part of this array must contain S_i , the left Cholesky factor of the state covariance matrix $P_{i|i-1}$.
On exit: the leading n by n lower triangular part of this array contains S_{i+1} , the left Cholesky factor of the state covariance matrix $P_{i+1|i}$.
- 5: **tds** – Integer *Input*
On entry: the stride separating matrix column elements in the array **s**.
Constraint: $\mathbf{tds} \geq n$.
- 6: **a[n × tda]** – const double *Input*
Note: the (i, j) th element of the matrix A is stored in $a[(i - 1) \times \mathbf{tda} + j - 1]$.
On entry: the leading n by n part of this array must contain A_i , the state transition matrix of the discrete system.
- 7: **tda** – Integer *Input*
On entry: the stride separating matrix column elements in the array **a**.
Constraint: $\mathbf{tda} \geq n$.

- 8: **b**[**n** × **tdb**] – const double *Input*
Note: the (i, j) th element of the matrix B is stored in **b**[($i - 1$) × **tdb** + $j - 1$].
On entry: if **q** is not **NULL** then the leading n by m part of this array must contain the matrix B_i , otherwise if **q** is **NULL** then the leading n by m part of the array must contain the matrix $B_i Q_i^{1/2}$. B_i is the input weight matrix and Q_i is the noise covariance matrix.
- 9: **tdb** – Integer *Input*
On entry: the stride separating matrix column elements in the array **b**.
Constraint: **tdb** ≥ **m**.
- 10: **q**[**m** × **tdq**] – const double *Input*
Note: the (i, j) th element of the matrix Q is stored in **q**[($i - 1$) × **tdq** + $j - 1$].
On entry: if the noise covariance matrix is to be supplied separately from the input weight matrix then the leading m by m lower triangular part of this array must contain $Q_i^{1/2}$, the left Cholesky factor of the input process noise covariance matrix. If the noise covariance matrix is to be input with the weight matrix as $B_i Q_i^{1/2}$ then the array **q** must be set to **NULL**.
- 11: **tdq** – Integer *Input*
On entry: the stride separating matrix column elements in the array **q**.
Constraint: **tdq** ≥ **m** if **q** is defined.
- 12: **c**[**p** × **tdc**] – const double *Input*
Note: the (i, j) th element of the matrix C is stored in **c**[($i - 1$) × **tdc** + $j - 1$].
On entry: the leading p by n part of this array must contain C_i , the output weight matrix of the discrete system.
- 13: **tdc** – Integer *Input*
On entry: the stride separating matrix column elements in the array **c**.
Constraint: **tdc** ≥ **n**.
- 14: **r**[**p** × **tdr**] – const double *Input*
Note: the (i, j) th element of the matrix R is stored in **r**[($i - 1$) × **tdr** + $j - 1$].
On entry: the leading p by p lower triangular part of this array must contain $R_i^{1/2}$, the left Cholesky factor of the measurement noise covariance matrix.
- 15: **tdr** – Integer *Input*
On entry: the stride separating matrix column elements in the array **r**.
Constraint: **tdr** ≥ **p**.
- 16: **k**[**n** × **tdk**] – double *Output*
Note: the (i, j) th element of the matrix K is stored in **k**[($i - 1$) × **tdk** + $j - 1$].
On exit: if **k** is not **NULL** then the leading n by p part of **k** contains AK_i , the product of the Kalman filter gain matrix K_i with the state transition matrix A_i . If AK_i is not required then **k** must be set to **NULL**.

- 17: **tdk** – Integer *Input*
On entry: the stride separating matrix column elements in the array **k**.
Constraint: if **k** is not **NULL**, $\mathbf{tdk} \geq \mathbf{p}$
- 18: **h**[**p** × **tdh**] – double *Output*
Note: the (i, j) th element of the matrix H is stored in $\mathbf{h}[(i - 1) \times \mathbf{tdh} + j - 1]$.
On exit: if **k** is not **NULL** then the leading p by p lower triangular part of this array contains $H_i^{1/2}$. If **k** is **NULL** then **h** is not referenced and may be set to **NULL**.
- 19: **tdh** – Integer *Input*
On entry: the stride separating matrix column elements in the array **h**.
Constraint: if both **k** and **h** are not **NULL**, $\mathbf{tdh} \geq \mathbf{p}$
- 20: **tol** – double *Input*
On entry: if both **k** and **h** are not **NULL** then **tol** is used to test for near singularity of the matrix $H_i^{1/2}$. If you set **tol** to be less than $p^2\epsilon$ then the tolerance is taken as $p^2\epsilon$, where ϵ is the *machine precision*. Otherwise, **tol** need not be set by you.
- 21: **fail** – NagError * *Input/Output*
The NAG error argument (see Section 2.7 in How to Use the NAG Library and its Documentation).

6 Error Indicators and Warnings

NE_2_INT_ARG_LT

- On entry **tda** = $\langle value \rangle$ while **n** = $\langle value \rangle$. These arguments must satisfy $\mathbf{tda} \geq \mathbf{n}$.
On entry **tdb** = $\langle value \rangle$ while **m** = $\langle value \rangle$. These arguments must satisfy $\mathbf{tdb} \geq \mathbf{m}$.
On entry **tdc** = $\langle value \rangle$ while **n** = $\langle value \rangle$. These arguments must satisfy $\mathbf{tdc} \geq \mathbf{n}$.
On entry **tdh** = $\langle value \rangle$ while **p** = $\langle value \rangle$. These arguments must satisfy $\mathbf{tdh} \geq \mathbf{p}$.
On entry **tdk** = $\langle value \rangle$ while **p** = $\langle value \rangle$. These arguments must satisfy $\mathbf{tdk} \geq \mathbf{p}$.
On entry **tdq** = $\langle value \rangle$ while **m** = $\langle value \rangle$. These arguments must satisfy $\mathbf{tdq} \geq \mathbf{m}$.
On entry **tdr** = $\langle value \rangle$ while **p** = $\langle value \rangle$. These arguments must satisfy $\mathbf{tdr} \geq \mathbf{p}$.
On entry, **tds** = $\langle value \rangle$ while **n** = $\langle value \rangle$. These arguments must satisfy $\mathbf{tds} \geq \mathbf{n}$.

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_INT_ARG_LT

- On entry, **m** = $\langle value \rangle$.
Constraint: $\mathbf{m} \geq 1$.
On entry, **n** = $\langle value \rangle$.
Constraint: $\mathbf{n} \geq 1$.
On entry, **p** = $\langle value \rangle$.
Constraint: $\mathbf{p} \geq 1$.

NE_MAT_SINGULAR

The matrix $\text{sqrt}(H)$ is singular.

NE_NULL_ARRAY

Array **h** has null address.

7 Accuracy

The use of the square root algorithm improves the stability of the computations.

8 Parallelism and Performance

`nag_kalman_sqrt_filt_cov_var` (g13eac) is not threaded in any implementation.

9 Further Comments

The algorithm requires $\frac{7}{6}n^3 + n^2(\frac{5}{2}p + m) + n(\frac{1}{2}m^2 + p^2)$ operations and is backward stable (see Vanbegin *et al.* (1989)).

10 Example

For this function two examples are presented. There is a single example program for `nag_kalman_sqrt_filt_cov_var` (g13eac), with a main program and the code to solve the two example problems is given in the functions `ex1` and `ex2`.

Example 1 (ex1)

To apply three iterations of the Kalman filter (in square root covariance form) to the system (A_i, B_i, C_i) . The same data is used for all three iterative steps.

Example 2 (ex2)

In the second example 2000 terms of an ARMA(1,1) time series (with $\sigma^2 = 1.0, \theta = 0.9$ and $\phi = 0.4$) are generated using the function `nag_rand_arma` (g05phc). The Kalman filter and optimization function `nag_opt_nlp` (e04ucc) are then used to find the maximum likelihood estimate for the time series arguments θ and ϕ . The ARMA(1,1) time series is defined by

$$y_k = \phi y_{k-1} + \epsilon_k - \theta \epsilon_{k-1}$$

This has the following state space representation (Harvey and Phillips (1979))

$$\begin{aligned} x_k &= \begin{pmatrix} \phi & 1 \\ 0 & 0 \end{pmatrix} x_{k-1} + \begin{pmatrix} 1 \\ -\theta \end{pmatrix} \epsilon_k \\ y_k &= \begin{pmatrix} 1 & 0 \end{pmatrix} x_k \end{aligned}$$

where the state vector $x_k = \begin{pmatrix} y_k \\ -\theta \epsilon_k \end{pmatrix}$ and ϵ_k is uncorrelated white noise with zero mean and variance σ^2 , i.e.,

$$E[\epsilon_k] = 0, E[\epsilon_k \epsilon_k] = \sigma^2, E[y_k \epsilon_k] = \sigma^2 \text{ and } E[\epsilon_k \epsilon_{k-1}] = 0.$$

Since $\sigma^2 = 1$ we arrive at the following Kalman Filter matrices

$$\begin{aligned} A_k &= \begin{pmatrix} \phi & 1 \\ 0 & 0 \end{pmatrix}, B_k = \begin{pmatrix} 1 \\ -\theta \end{pmatrix} \\ C_k &= \begin{pmatrix} 1 & 0 \end{pmatrix}, Q_k = 0 \text{ and } R_k = 1. \end{aligned}$$

The initial estimates for the state vector, $x_{1|0}$, and state covariance matrix, $P_{1|0}$, are:

$$x_{1|0} = E[x_k] = 0 \text{ and } P_{1|0} = E[x_k x_k^T] = \begin{pmatrix} E[y_k y_k] & -\theta E[y_k \epsilon_k] \\ -\theta E[y_k \epsilon_k] & \theta^2 E[\epsilon_k \epsilon_k] \end{pmatrix}.$$

Since $E[y_k y_k] = \gamma_\circ = \frac{(1+\theta^2-2\phi\theta)\sigma^2}{(1-\phi^2)}$ (Wei (1990))

$$P_{1|0} = \begin{pmatrix} \gamma_\circ & -\theta \\ -\theta & \theta^2 \end{pmatrix}.$$

Using $P_{1|0} = S_{1|0} S_{1|0}^T$ gives an initial Cholesky ‘square root’ of

$$S_{1|0} = \begin{pmatrix} \sqrt{\gamma_\circ} & 0 \\ \frac{-\theta}{\sqrt{\gamma_\circ}} & \theta \sqrt{\frac{\gamma_\circ-1}{\gamma_\circ}} \end{pmatrix}.$$

10.1 Program Text

```

/* nag_kalman_sqrt_filt_cov_var (g13eac) Example Program.
 *
 * NAGPRODCODE Version.
 *
 * Copyright 2016 Numerical Algorithms Group
 *
 * Mark 26, 2016.
 *
 */

#include <nag.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <nag_stdlib.h>
#include <nage04.h>
#include <nagf06.h>
#include <nagg05.h>
#include <nagg13.h>
#include <nagx02.h>

#ifdef __cplusplus
extern "C"
{
#endif
    static void NAG_CALL objfun(Integer n, const double theta_phi[],
                                double *objf, double g[], Nag_Comm *comm);
#ifdef __cplusplus
}
#endif

static int ex1(void);
static int ex2(void);

int main(void)
{
    /* Integer scalar and array declarations */
    Integer exit_status_ex1 = 0;
    Integer exit_status_ex2 = 0;

    printf("nag_kalman_sqrt_filt_cov_var (g13eac) Example Program Results\n\n");

    /* Run example 1 */
    exit_status_ex1 = ex1();

    /* Run example 2 */
    exit_status_ex2 = ex2();

    return (exit_status_ex1 == 0 && exit_status_ex2 == 0) ? 0 : 1;
}

/* Start of the first example ... */
#define A(I, J) a[(I) *tda + J]
#define B(I, J) b[(I) *tdb + J]
#define C(I, J) c[(I) *tdc + J]

```

```

#define K(I, J) k[(I) *tdk + J]
#define Q(I, J) q[(I) *tdq + J]
#define R(I, J) r[(I) *tdr + J]
#define S(I, J) s[(I) *tds + J]

static int ex1()
{
    /* Integer scalar and array declarations */
    Integer exit_status = 0;
    Integer i, j, m, n, p, istep, tda, tdb, tdc, tdk, tdq, tdr, tds, tdh;

    /* Double scalar and array declarations */
    double tol;
    double *a = 0, *b = 0, *c = 0, *k = 0, *q = 0, *r = 0, *s = 0, *h = 0;

    /* NAG structures */
    NagError fail;

    /* Initialize the error structure */
    INIT_FAIL(fail);

    printf("Example 1\n");

    /* Skip the heading in the data file */
#ifdef _WIN32
    scanf_s("%*[\n]");
#else
    scanf("%*[\n]");
#endif

    /* Get the problem size */
#ifdef _WIN32
    scanf_s("%" NAG_IFMT "%" NAG_IFMT "%" NAG_IFMT "%lf", &n, &m, &p, &tol);
#else
    scanf("%" NAG_IFMT "%" NAG_IFMT "%" NAG_IFMT "%lf", &n, &m, &p, &tol);
#endif
    if (n < 1 || m < 1 || p < 1) {
        printf("Invalid n or m or p.\n");
        exit_status = 1;
        goto END;
    }

    tda = tdc = tds = n;
    tdb = tdq = m;
    tdk = tdr = tdh = p;

    /* Allocate arrays */
    if (!(a = NAG_ALLOC(n * tda, double)) ||
        !(b = NAG_ALLOC(n * tdb, double)) ||
        !(c = NAG_ALLOC(p * tdc, double)) ||
        !(k = NAG_ALLOC(n * tdk, double)) ||
        !(q = NAG_ALLOC(m * tdq, double)) ||
        !(r = NAG_ALLOC(p * tdr, double)) ||
        !(s = NAG_ALLOC(n * tds, double)) || !(h = NAG_ALLOC(n * tdh, double)))
    {
        printf("Allocation failure\n");
        exit_status = -1;
        goto END;
    }

    /* Read data */
    for (i = 0; i < n; ++i)
        for (j = 0; j < n; ++j)
#ifdef _WIN32
            scanf_s("%lf", &S(i, j));
#else
            scanf("%lf", &S(i, j));
#endif
    for (i = 0; i < n; ++i)
        for (j = 0; j < n; ++j)
#ifdef _WIN32

```

```

        scanf_s("%lf", &A(i, j));
#else
        scanf("%lf", &A(i, j));
#endif
        for (i = 0; i < n; ++i)
            for (j = 0; j < m; ++j)
#ifdef _WIN32
                scanf_s("%lf", &B(i, j));
#else
                scanf("%lf", &B(i, j));
#endif
            if (q) {
                for (i = 0; i < m; ++i)
                    for (j = 0; j < m; ++j)
#ifdef _WIN32
                        scanf_s("%lf", &Q(i, j));
#else
                        scanf("%lf", &Q(i, j));
#endif
            }
            for (i = 0; i < p; ++i)
                for (j = 0; j < n; ++j)
#ifdef _WIN32
                    scanf_s("%lf", &C(i, j));
#else
                    scanf("%lf", &C(i, j));
#endif
            for (i = 0; i < p; ++i)
                for (j = 0; j < p; ++j)
#ifdef _WIN32
                    scanf_s("%lf", &R(i, j));
#else
                    scanf("%lf", &R(i, j));
#endif
#endif

/* Perform three iterations of the Kalman filter recursion */
for (istep = 1; istep <= 3; ++istep) {
    /* Run the kalman filter */
    nag_kalman_sqrt_filt_cov_var(n, m, p, s, tds, a, tda, b, tdb, q, tdq,
                                c, tdc, r, tdr, k, tdk, h, tdh, tol, &fail);
    if (fail.code != NE_NOERROR) {
        printf("Error from nag_kalman_sqrt_filt_cov_var (g13eac).\n%s\n",
              fail.message);
        exit_status = 1;
        goto END;
    }
}

/* Print the results */
printf("\nThe square root of the state covariance matrix is\n\n");
for (i = 0; i < n; ++i) {
    for (j = 0; j < n; ++j)
        printf("%8.4f ", S(i, j));
    printf("\n");
}
if (k) {
    printf("\nThe matrix AK (the product of the Kalman gain\n");
    printf("matrix with the state transition matrix) is\n\n");
    for (i = 0; i < n; ++i) {
        for (j = 0; j < p; ++j)
            printf("%8.4f ", K(i, j));
        printf("\n");
    }
}
}

END:
NAG_FREE(a);
NAG_FREE(b);
NAG_FREE(c);
NAG_FREE(k);
NAG_FREE(q);

```



```

    NAG_FREE(r);
    NAG_FREE(s);
    NAG_FREE(h);

    return exit_status;
}

/* ... end of the first example */

/* Start of the second example ... */
#define NY 2000

/* This illustrates the use of the kalman filter to estimate the
   parameters of an ARMA(1,1) time series model.
   Note : theta_phi[0] contains theta (moving average coefficient), and
   theta_phi[1] contains phi (autoregressive coefficient)
*/
static int ex2(void)
{
    /* Integer scalar and array declarations */
    Integer exit_status = 0;
    Integer n, lr;
    Integer lstate;
    Integer *state = 0;

    /* Double scalar and array declarations */
    double a[1], sy[NY];
    double *theta_phi = 0, *g = 0, *bl = 0, *bu = 0, *r = 0;
    double objf, var;

    /* NAG structures and data types */
    Nag_Comm comm;
    Nag_EO4_Opt options;
    NagError fail;
    Nag_ModeRNG mode;

    /* Choose the base generator */
    Nag_BaseRNG genid = Nag_Basic;
    Integer subid = 0;

    /* Set the seed */
    Integer seed[] = { 1762543 };
    Integer lseed = 1;

    /* Set the autoregressive coefficients for the (randomly generated)
       time series */
    Integer ip = 1;
    double sphi[] = { 0.4 };

    /* Set the moving average coefficients for the (randomly generated)
       time series */
    Integer iq = 1;
    double stheta[] = { 0.9 };

    /* Number of terms in the (randomly generated) time series */
    Integer nterms = NY;

    /* Mean and variance of the (randomly generated) time series */
    double mean = 0.0, vara = 1.0;

    /* Initialize the error structure */
    INIT_FAIL(fail);

    printf("\n\nExample 2\n\n");

    /* Get the length of the state array */
    lstate = -1;
    nag_rand_init_repeatabe(genid, subid, seed, lseed, state, &lstate, &fail);
    if (fail.code != NE_NOERROR) {
        printf("Error from nag_rand_init_repeatabe (g05kfc).\n%s\n",
            fail.message);
    }
}

```

```

    exit_status = 1;
    goto END;
}

/* Calculate the size of the reference vector */
lr = (ip > iq + 1) ? ip : iq + 1;
lr += ip + iq + 6;

/* Allocate arrays */
if (!(state = NAG_ALLOC(lstate, Integer)) || !(r = NAG_ALLOC(lr, double)))
{
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
}

/* Initialize the generator to a repeatable sequence */
nag_rand_init_repeatable(genid, subid, seed, lseed, state, &lstate, &fail);
if (fail.code != NE_NOERROR) {
    printf("Error from nag_rand_init_repeatable (g05kfc).\n%s\n",
        fail.message);
    exit_status = 1;
    goto END;
}

/* Generate a time series with nterms terms */
mode = Nag_InitializeAndGenerate;
nag_rand_arma(mode, nterms, mean, ip, sphl, iq, stheta, vara, r, lr, state,
    &var, sy, &fail);
if (fail.code != NE_NOERROR) {
    printf("Error from nag_rand_arma (g05phc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}

/* Number of parameters to estimate */
n = ip + iq;

/* Allocate arrays */
if (!(theta_phi = NAG_ALLOC(n, double)) ||
    !(g = NAG_ALLOC(n, double)) ||
    !(bl = NAG_ALLOC(n, double)) || !(bu = NAG_ALLOC(n, double)))
{
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
}

/* Make an initial guess of the parameters */
theta_phi[0] = 0.6;
theta_phi[1] = 0.4;

/* Set the bounds */
bl[0] = -1.0;
bu[0] = 1.0;
bl[1] = -1.0;
bu[1] = 1.0;
comm.user = &sy[0];

/* nag_opt_init (e04xxc).
 * Initialization function for option setting
 */
nag_opt_init(&options);
#ifdef _WIN32
    strcpy_s(options.outfile, (unsigned)_countof(options.outfile), "stdout");
#else
    strcpy(options.outfile, "stdout");
#endif
    options.print_level = Nag_NoPrint;
    options.list = Nag_FALSE;

```

```

options.obj_deriv = Nag_FALSE;
options.con_deriv = Nag_FALSE;
options.print_deriv = Nag_D_NoPrint;

/* nag_opt_nlp (e04ucc).
 * Minimization with nonlinear constraints using a sequential QP method.
 */
nag_opt_nlp(n, 0, 0, a, 1, bl, bu, objfun, NULLFN, theta_phi, &objf,
            g, &options, &comm, &fail);
if (fail.code != NE_NOERROR && fail.code != NW_KT_CONDITIONS) {
    printf("Error from nag_opt_nlp (e04ucc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}

/* nag_opt_free (e04xzc).
 * Memory freeing function for use with option setting
 */
nag_opt_free(&options, "all", &fail);
if (fail.code != NE_NOERROR) {
    printf("Error from nag_opt_free (e04xzc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}

/* Display the results */
printf("The estimates are : theta = %7.3f, phi = %7.3f \n",
       theta_phi[0], theta_phi[1]);

END:
    NAG_FREE(state);
    NAG_FREE(theta_phi);
    NAG_FREE(g);
    NAG_FREE(bl);
    NAG_FREE(bu);
    NAG_FREE(r);

    return exit_status;
}

/* Define objective function used in the non-linear optimization routine ... */
static void NAG_CALL objfun(Integer n, const double theta_phi[], double *objf,
                           double g[], Nag_Comm *comm)
{
    /* Routine to evaluate objective function. */

    Integer ione = 1, itwo = 2, k, m1 = 1, n1 = 2, nsteps = NY, nsum = 0, p1 =
        1;
    double a[2][2], ak[2][1], b[2][1], c[1][2], h[1][1], hs, k11, logdet = 0.0;
    double phi, q[1][1], r[1][1];
    double s[2][2], ss = 0.0, temp1, temp2, theta, tol = 0.0;
    double v, xp[2], *y;
    NagError fail;

    /* Initialize the error structure */
    INIT_FAIL(fail);

    y = comm->user;
    /* The expectation of the mean of an ARMA(1,1) is 0.0 */
    xp[0] = 0.0;
    xp[1] = 0.0;
    q[0][0] = 1.0;

    c[0][0] = 1.0;
    c[0][1] = 0.0;

    /* There is no measurement noise */
    r[0][0] = 0.0;
    theta = theta_phi[0];
    phi = theta_phi[1];

```

```

b[0][0] = 1.0;
b[1][0] = -theta;

a[0][0] = phi;
a[1][0] = 0.0;
a[0][1] = 1.0;
a[1][1] = 0.0;

/* set value for Cholesky factor of state covariance matrix */
temp1 = 1.0 + (theta * theta) - (2.0 * theta * phi);
temp2 = 1.0 - (phi * phi);
k11 = temp1 / temp2;
s[0][0] = sqrt(k11);
s[0][1] = 0.0;
s[1][0] = -theta / s[0][0];
s[1][1] = theta * sqrt(1.0 - (1.0 / k11));

/* iterate kalman filter for number of observations */
for (k = 1; k <= nsteps; ++k) {
  /* nag_kalman_sqrt_filt_cov_var (g13eac).
   * One iteration step of the time-varying Kalman filter
   * recursion using the square root covariance implementation
   */
  nag_kalman_sqrt_filt_cov_var(n1, m1, p1, &s[0][0], itwo, &a[0][0], itwo,
                               &b[0][0], ione, &q[0][0], ione, &c[0][0],
                               itwo, &r[0][0], ione, &ak[0][0], ione,
                               &h[0][0], ione, tol, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_kalman_sqrt_filt_cov_var (g13eac).\n%s\n",
           fail.message);
    *objf = 0.0;
    return;
  }

  v = y[k - 1] - c[0][0] * xp[0];
  hs = h[0][0] * h[0][0];
  logdet = logdet + log(hs);
  ss = ss + (v * v / hs);
  nsum = nsum + 1;

  xp[0] = a[0][0] * xp[0] + a[0][1] * xp[1] + ak[0][0] * v;
  xp[1] = ak[1][0] * v;
}
*objf = nsum * log(ss / nsum) + logdet;
}

/* ... end of the objective function definition */
/* ... end of the second example */

```

10.2 Program Data

None.

10.3 Program Results

nag_kalman_sqrt_filt_cov_var (g13eac) Example Program Results

Example 1

The square root of the state covariance matrix is

```

-1.2936  0.0000  0.0000  0.0000
-1.1382 -0.2579  0.0000  0.0000
-0.9622 -0.1529  0.2974  0.0000
-1.3076  0.0936  0.4508 -0.4897

```

The matrix AK (the product of the Kalman gain matrix with the state transition matrix) is

0.3638	0.9469
0.3532	0.8179
0.2471	0.5542
0.1982	0.6471

Example 2

The estimates are : $\theta = 0.898$, $\phi = 0.406$
