NAG Library Function Document

nag_kalman_sqrt_filt_cov_invar (g13ebc)

1 Purpose

nag_kalman_sqrt_filt_cov_invar (g13ebc) performs a combined measurement and time update of one iteration of the time-invariant Kalman filter. The method employed for this update is the square root covariance filter with the system matrices transformed into condensed observer Hessenberg form.

2 Specification

```
#include <nag.h>
#include <nagg13.h>
void nag_kalman_sqrt_filt_cov_invar (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

$$\begin{array}{lcll} X_{i+1} & = & AX_i + BW_i & \text{var}\,(W_i) = Q_i \\ Y_i & = & CX_i + V_i & \text{var}\,(V_i) = R_i \end{array}$$

the estimate of X_i given observations Y_1 to Y_{i-1} is denoted by $\hat{X}_{i|i-1}$, with $\text{var}\left(\hat{X}_{i|i-1}\right) = P_{i|i-1} = S_i S_i^{\mathrm{T}}$ (where A, B and C are time invariant). The function performs one recursion of the square root covariance filter algorithm, summarised as follows:

$$\underset{\text{(Pre-array)}}{\begin{pmatrix} R_i^{1/2} & 0 & CS_i \\ 0 & BQ_i^{1/2} & AS_i \end{pmatrix}} \quad U_1 = \quad \underset{\text{(Post-array)}}{\begin{pmatrix} H_i^{1/2} & 0 & 0 \\ G_i & S_{i+1} & 0 \end{pmatrix}}$$

where U_1 is an orthogonal transformation triangularizing the pre-array, and the matrix pair (A, C) is in lower observer Hessenberg form. The triangularization is carried out via Householder transformations exploiting the zero pattern of the pre-array. An example of the pre-array is given below (where $n = 6, p = 2$ and $m = 3$):

$$\left(\begin{array}{ccccccccccc} x & & & & & x & & & & & \\ x & x & & & & x & x & & & & \\ & & x & x & x & x & x & x & & & \\ \hline & & x & x & x & x & x & x & x & & \\ & & x & x & x & x & x & x & x & x & \\ & & x & x & x & x & x & x & x & x & x \\ & & x & x & x & x & x & x & x & x & x \\ \hline & & x & x & x & x & x & x & x & x & x \end{array}\right)$$

The measurement-update for the estimated state vector X is

$$\hat{X}_{i|i} = \hat{X}_{i|i-1} - K_i\left[C\hat{X}_{i|i-1} - Y_i\right]$$

whilst the time-update for X is

$$\hat{X}_{i+1|i} = A\hat{X}_{i|i} + D_iU_i$$

where D_iU_i represents any deterministic control used. The relationship between the Kalman gain matrix K_i and G_i is

$$AK_i = G_i\left(H_i^{1/2}\right)$$

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

4 References

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

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

van Dooren P and Verhaegen M H G (1988) Condensed forms for efficient time-invariant Kalman filtering *SIAM J. Sci. Stat. Comput.* **9** 516–530

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

5 Arguments

1: **n** – Integer *Input*

On entry: the actual state dimension, n, i.e., the order of the matrices S_i and A.

Constraint: **n** ≥ 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** ≥ 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** ≥ 1.

4: **s**[**n** $\times$ **tds**] – double *Input/Output*

Note: the (i,j)th element of the matrix S is stored in **s**$[(i-1) \times$ **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: **tds** $\geq$ **n**.

6: **a**[**n** × **tda**] – const double *Input*

Note: the (i,j)th element of the matrix A is stored in **a**$[(i-1) \times$ **tda** $+ j - 1]$.

On entry: the leading n by n part of this array must contain the lower observer Hessenberg matrix UAU^{T}. Where A is the state transition matrix of the discrete system and U is the unitary transformation generated by the function nag_trans_hessenberg_observer (g13ewc).

7: **tda** – Integer *Input*

On entry: the stride separating matrix column elements in the array **a**.

Constraint: **tda** $\geq$ **n**.

8: **b**[**n** × **tdb**] – const double *Input*

Note: the (i,j)th element of the matrix B is stored in **b**$[(i-1) \times$ **tdb** $+ j - 1]$.

On entry: if **q** is not **NULL** then the leading n by m part of this array must contain the matrix UB, otherwise (if **q** is **NULL** then the leading n by m part of the array must contain the matrix $UBQ_i^{1/2}$. B is the input weight matrix, Q_i is the noise covariance matrix and U is the same unitary transformation used for defining array arguments **a** and **c**.

9: **tdb** – Integer *Input*

On entry: the stride separating matrix column elements in the array **b**.

Constraint: **tdb** $\geq$ **m**.

10: **q**[**m** × **tdq**] – const double *Input*

Note: the (i,j)th element of the matrix Q is stored in **q**$[(i-1) \times$ **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 process noise covariance matrix. If the noise covariance matrix is to be input with the weight matrix as $BQ_i^{1/2}$ 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** $\geq$ **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**$[(i-1) \times$ **tdc** $+ j - 1]$.

On entry: the leading p by n part of this array must contain the lower observer Hessenberg matrix CU^{T}. Where C is the output weight matrix of the discrete system and U is the unitary transformation matrix generated by the function nag_trans_hessenberg_observer (g13ewc).

13: **tdc** – Integer *Input*

On entry: the stride separating matrix column elements in the array **c**.

Constraint: **tdc** $\geq$ **n**.

14: **r**[**p** × **tdr**] – const double *Input*

Note: the (i,j)th element of the matrix R is stored in **r**$[(i-1) \times$ **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** $\geq$ **p**.

16: **k**[**n** $\times$ **tdk**] – double *Output*

Note: the (i,j)th element of the matrix K is stored in **k**$[(i-1) \times$ **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: **tdk** $\geq$ **p** if **k** is defined.

18: **h**[**p** $\times$ **tdh**] – double *Output*

Note: the (i,j)th element of the matrix H is stored in **h**$[(i-1) \times$ **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: **tdh** $\geq$ **p** if **k** and **h** are defined.

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 3.6 in the Essential Introduction).

6 Error Indicators and Warnings

NE_2_INT_ARG_LT

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

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_INT_ARG_LT

On entry, **m** $= \langle value \rangle$.
Constraint: **m** ≥ 1.

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

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

NE_MAT_SINGULAR

The matrix 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

Not applicable.

9 Further Comments

The algorithm requires $\frac{1}{6}n^3 + n^2\left(\frac{3}{2}p + m\right) + 2np^2 + \frac{2}{3}p^3$ operations and is backward stable (see Verhaegen *et al*).

10 Example

For this function two examples are presented. There is a single example program for nag_kalman_sqrt_filt_cov_invar (g13ebc), 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 time-invariant system (A, B, C) supplied in lower observer Hessenberg form.

Example 2 (ex2)

To apply three iterations of the Kalman filter (in square root covariance form) to the general time-invariant system (A, B, C). The use of the time-varying Kalman function nag_kalman_sqrt_filt_cov_var (g13eac) is compared with that of the time-invariant function nag_kalman_sqrt_filt_cov_invar (g13ebc). The same original data is used by both functions but additional transformations are required before it can be supplied to nag_kalman_sqrt_filt_cov_invar (g13ebc). It can be seen that (after the appropriate back-transformations on the output of nag_kalman_sqrt_filt_cov_invar (g13ebc)) the results of both nag_kalman_sqrt_filt_cov_var (g13eac) and nag_kalman_sqrt_filt_cov_invar (g13ebc) are the same.

10.1 Program Text

```
/* nag_kalman_sqrt_filt_cov_invar (g13ebc) Example Program.
 *
 * Copyright 1993 Numerical Algorithms Group
 *
 * Mark 3, 1993
 * Mark 7, revised, 2001.
 * Mark 8 revised, 2004.
 *
 */

#include <nag.h>
#include <stdio.h>
#include <nag_stdlib.h>
```

```
#include <nagf03.h>
#include <nagf06.h>
#include <nagf16.h>
#include <nagg13.h>

typedef enum { read, print }   ioflag;

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

int main(void)
{
  Integer  exit_status_ex1 = 0;
  Integer  exit_status_ex2 = 0;

  printf("nag_kalman_sqrt_filt_cov_invar (g13ebc) Example Program "
         "Results\n\n");

  /* Skip the heading in the data file  */
  scanf(" %*[^\n] ");

  exit_status_ex1 = ex1();
  exit_status_ex2 = ex2();

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

#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]
#define H(I, J) h[(I) *tdh + J]

static int ex1()
{ /* simple example (matrices A and C are supplied in lower observer
     Hessenberg form) */
  Integer  exit_status = 0, i, istep, j, m, n, p, tda, tdb, tdc, tdh, tdk, tdq;
  Integer  tdr, tds;
  NagError fail;
  double   *a = 0, *b = 0, *c = 0, *h = 0, *k = 0, *q = 0, *r = 0, *s = 0, tol;

  INIT_FAIL(fail);

  /* Skip the heading in the data file  */
  scanf(" %*[^\n]");

  printf("Example 1\n");
  scanf("%ld%ld%ld%lf", &n, &m, &p, &tol);
  if (n >= 1 && m >= 1 && p >= 1)
    {
      if (!(a = NAG_ALLOC(n*n, double)) ||
          !(b = NAG_ALLOC(n*m, double)) ||
          !(c = NAG_ALLOC(p*n, double)) ||
          !(k = NAG_ALLOC(n*p, double)) ||
          !(q = NAG_ALLOC(m*m, double)) ||
          !(r = NAG_ALLOC(p*p, double)) ||
          !(s = NAG_ALLOC(n*n, double)) ||
          !(h = NAG_ALLOC(n*p, double)))
        {
          printf("Allocation failure\n");
          exit_status = -1;
          goto END;
        }
      tda = n;
      tdb = m;
      tdc = n;
      tdk = p;
      tdq = m;
```

```
      tdr = p;
      tds = n;
      tdh = p;
    }
  else
    {
      printf("Invalid n or m or p.\n");
      exit_status = 1;
      return exit_status;
    }

  /* Read data */
  for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j)
      scanf("%lf", &S(i, j));
  for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j)
      scanf("%lf", &A(i, j));
  for (i = 0; i < n; ++i)
    for (j = 0; j < m; ++j)
      scanf("%lf", &B(i, j));

  if (q)
    {
      for (i = 0; i < m; ++i)
        for (j = 0; j < m; ++j)
          scanf("%lf", &Q(i, j));
    }
  for (i = 0; i < p; ++i)
    for (j = 0; j < n; ++j)
      scanf("%lf", &C(i, j));
  for (i = 0; i < p; ++i)
    for (j = 0; j < p; ++j)
      scanf("%lf", &R(i, j));

  /* Perform three iterations of the Kalman filter recursion  */
  for (istep = 1; istep <= 3; ++istep)
    /* nag_kalman_sqrt_filt_cov_invar (g13ebc).
     * One iteration step of the time-invariant Kalman filter
     * recursion using the square root covariance implementation
     * with (AC) in lower observer Hessenberg form
     */
    nag_kalman_sqrt_filt_cov_invar(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_invar (g13ebc).\n%s\n",
              fail.message);
      exit_status = 1;
      goto END;
    }

  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;
}

static void mat_io(Integer n, Integer m, double mat[], Integer tdmat,
                   ioflag flag, const char *message);


#define KE(I, J)    ke[(I) *tdke + J]
#define KF(I, J)    kf[(I) *tdkf + J]
#define UB(I, J)    ub[(I) *tdub + J]
#define RWORK(I, J) rwork[(I) *tdrwork + J]
#define SF(I, J)    sf[(I) *tdsf + J]
#define SE(I, J)    se[(I) *tdse + J]
#define PF(I, J)    pf[(I) *tdpf + J]
#define PE(I, J)    pe[(I) *tdpe + J]
#define UAUT(I, J)  uaut[(I) *tduaut + J]
#define CUT(I, J)   cut[(I) *tdcut + J]
#define U(I, J)     u[(I) *tdu + J]

static int ex2()
{ /* more general example which requires the data to be transformed. The
     results produced by nag_kalman_sqrt_filt_cov_var (g13eac) and
     nag_kalman_sqrt_filt_cov_invar (g13ebc) are compared */
  Integer          dete, exit_status = 0, i, ione = 1, istep, j, m, n, p, tda,
                   tdb;
  Integer          tdc, tdcut, tdh, tdke, tdkf, tdpe, tdpf, tdq, tdr, tdrwork,
                   tdse;
  Integer          tdsf, tdu, tduaut, tdub;
  NagError         fail;
  Nag_ObserverForm reduceto = Nag_LH_Observer;
  double           *a = 0, *b = 0, *c = 0, *cut = 0, detf, *diag = 0, *h = 0;
  double           *ke = 0, *kf = 0, one = 1.0, *pe = 0, *pf = 0, *q = 0;
  double           *r = 0, *rwork = 0, *se = 0, *sf = 0, tol, *u = 0;
  double           *uaut = 0, *ub = 0, zero = 0.0;

  INIT_FAIL(fail);

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

  /* skip the heading in the data file */
  scanf(" %*[^\n]");
  scanf("%ld%ld%ld%lf", &n, &m, &p, &tol);
  if (n >= 1 && m >= 1 && p >= 1)
    {
      if (!(a = NAG_ALLOC(n*n, double)) ||
          !(b = NAG_ALLOC(n*m, double)) ||
          !(c = NAG_ALLOC(p*n, double)) ||
          !(ke = NAG_ALLOC(n*p, double)) ||
          !(kf = NAG_ALLOC(n*p, double)) ||
          !(ub = NAG_ALLOC(n*m, double)) ||
          !(q = NAG_ALLOC(m*m, double)) ||
          !(r = NAG_ALLOC(p*p, double)) ||
          !(rwork = NAG_ALLOC(n*n, double)) ||
          !(sf = NAG_ALLOC(n*n, double)) ||
          !(se = NAG_ALLOC(n*n, double)) ||
          !(h = NAG_ALLOC(n*p, double)) ||
          !(pf = NAG_ALLOC(n*n, double)) ||
          !(pe = NAG_ALLOC(n*n, double)) ||
          !(uaut = NAG_ALLOC(n*n, double)) ||
          !(cut = NAG_ALLOC(p*n, double)) ||
          !(u = NAG_ALLOC(n*n, double)) ||
          !(diag = NAG_ALLOC(n, double)))
```

```
      {
        printf("Allocation failure\n");
        exit_status = -1;
        goto END;
      }
    tda = n;
    tdb = m;
    tdc = n;
    tdke = p;
    tdkf = p;
    tdub = m;
    tdq = m;
    tdr = p;
    tdrwork = n;
    tdsf = n;
    tdse = n;
    tdh = p;
    tdpf = n;
    tdpe = n;
    tduaut = n;
    tdcut = n;
    tdu = n;
  }
else
  {
    printf("Invalid n or m or p.\n");
    exit_status = 1;
    return exit_status;
  }
mat_io(n, n, se, tdse, read, "");
mat_io(n, n, a, tda, read, "");
mat_io(n, m, b, tdb, read, "");
if (q)
  mat_io(m, m, q, tdq, read, "");
mat_io(p, n, c, tdc, read, "");
mat_io(p, p, r, tdr, read, "");
for (i = 0; i < n; ++i)
  {
    for (j = 0; j < n; ++j)
      {
        if (i < p)
          CUT(i, j) = C(i, j);
        SF(i, j) = SE(i, j);
        UAUT(i, j) = A(i, j);
        U(i, j) = zero;
      }
    U(i, i) = one;
  }
/* Set up the matrix pair (A,C) in the lower observer hessenberg form */
/* nag_trans_hessenberg_observer (g13ewc).
 * Unitary state-space transformation to reduce (AC) to
 * lower or upper observer Hessenberg form
 */
nag_trans_hessenberg_observer(n, p, reduceto, uaut, tduaut, cut, tdcut,
                              u, tdu, &fail);
if (fail.code != NE_NOERROR)
  {
    printf("Error from nag_trans_hessenberg_observer (g13ewc).\n%s\n",
            fail.message);
    exit_status = 1;
    goto END;
  }
for (j = 0; j < m; ++j)
  for (i = 0; i < n; ++i)
    UB(i, j) = f06eac(n, &U(i, 0), ione, &B(0, j), tdb);

/* Generate noise covariance matrices PE and PF = U * PE * U' */
nag_dgemm(Nag_RowMajor, Nag_NoTrans, Nag_Trans, n, n, n, one, se, tdse,
          se, tdse, zero, pe, tdpe, &fail);
nag_dgemm(Nag_RowMajor, Nag_NoTrans, Nag_Trans, n, n, n, one, pe, tdpe,
          u, tdu, zero, rwork, tdrwork, &fail);
```

```
  nag_dgemm(Nag_RowMajor, Nag_NoTrans, Nag_NoTrans, n, n, n, one, u, tdu,
            rwork, tdrwork, zero, pf, tdpf, &fail);

  /* Now find the lower triangular (left) cholesky factor of PF. */
  /* nag_real_cholesky (f03aec).
   * LL^T factorization and determinant of real symmetric
   * positive-definite matrix
   */
  nag_real_cholesky(n, pf, tdpf, diag, &detf, &dete, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_real_cholesky (f03aec).\n%s\n",
              fail.message);
      exit_status = 1;
      goto END;
    }
  for (i = 0; i < n; ++i)
    {
      SF(i, i) = one/diag[i];
      for (j = 0; j < i; ++j)
        SF(i, j) = PF(i, j);
    }
  /* Perform three steps of the Kalman filter recursion */
  for (istep = 1; istep <= 3; ++istep)
    {
      /* 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(n, m, p, se, tdse, a, tda, b, tdb, q,
                                   tdq, c, tdc, r, tdr, ke, tdke, 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;
        }
      /* nag_kalman_sqrt_filt_cov_invar (g13ebc), see above. */
      nag_kalman_sqrt_filt_cov_invar(n, m, p, sf, tdsf, uaut, tduaut, ub, tdub,
                                     q, tdq, cut, tdcut, r, tdr, kf, tdkf, h,
                                     tdh, tol, &fail);
      if (fail.code != NE_NOERROR)
        {
          printf("Error from nag_kalman_sqrt_filt_cov_invar (g13ebc).\n%s\n",
                  fail.message);
          exit_status = 1;
          goto END;
        }
    }
  nag_dgemm(Nag_RowMajor, Nag_NoTrans, Nag_Trans, n, n, n, one, se, tdse,
            se, tdse, zero, pe, tdpe, &fail);
  nag_dgemm(Nag_RowMajor, Nag_NoTrans, Nag_Trans, n, n, n, one, sf, tdsf,
            sf, tdsf, zero, pf, tdpf, &fail);
  mat_io(n, n, pe, tdpe, print, "Covariance matrix PE from "
         "nag_kalman_sqrt_filt_cov_var (g13eac) is\n");
  mat_io(n, n, pf, tdpf, print, "Covariance matrix PF from "
         "nag_kalman_sqrt_filt_cov_invar (g13ebc) is\n");

  /* Calculate PF = U' * PF * U */
  nag_dgemm(Nag_RowMajor, Nag_NoTrans, Nag_NoTrans, n, n, n, one, pf, tdpf,
            u, tdu, zero, rwork, tdrwork, &fail);
  nag_dgemm(Nag_RowMajor, Nag_Trans, Nag_NoTrans, n, n, n, one, u, tdu,
            rwork, tdrwork, zero, pf, tdpf, &fail);
  mat_io(n, n, pf, tdpf, print, "Matrix U' * PF * U is \n");
  mat_io(n, p, ke, tdke, print,
         "The matrix KE from nag_kalman_sqrt_filt_cov_var (g13eac) is\n");
  mat_io(n, p, kf, tdkf, print,
         "The matrix KF from nag_kalman_sqrt_filt_cov_invar (g13ebc) is\n");
```

```
  /* calculate U' * K */
  nag_dgemm(Nag_RowMajor, Nag_Trans, Nag_NoTrans, n, p, n, one, u, tdu,
            kf, tdkf, zero, rwork, tdrwork, &fail);
  mat_io(n, p, rwork, tdrwork, print, "U' * KF is\n");

 END:
  NAG_FREE(a);
  NAG_FREE(b);
  NAG_FREE(c);
  NAG_FREE(ke);
  NAG_FREE(kf);
  NAG_FREE(ub);
  NAG_FREE(q);
  NAG_FREE(r);
  NAG_FREE(rwork);
  NAG_FREE(sf);
  NAG_FREE(se);
  NAG_FREE(h);
  NAG_FREE(pf);
  NAG_FREE(pe);
  NAG_FREE(uaut);
  NAG_FREE(cut);
  NAG_FREE(u);
  NAG_FREE(diag);

  return exit_status;
}


static void mat_io(Integer n, Integer m, double mat[], Integer tdmat,
                   ioflag flag, const char *message)
{
  Integer i, j;
#define MAT(I, J) mat[((I) -1)*tdmat + (J) -1]
  if (flag == print) printf("%s \n", message);
  for (i = 1; i <= n; ++i)
    {
      for (j = 1; j <= m; ++j)
        {
          if (flag == read) scanf("%lf", &MAT(i, j));
          if (flag == print) printf("%8.4f ", MAT(i, j));
        }
      if (flag == print) printf("\n");
    }
  if (flag == print) printf("\n");
} /* mat_io */
```

10.2 Program Data

```
nag_kalman_sqrt_filt_cov_invar (g13ebc) Example Program Data
Example 1
   4     2     2   0.0
   0.0000  0.0000  0.0000  0.0000
   0.0000  0.0000  0.0000  0.0000
   0.0000  0.0000  0.0000  0.0000
   0.0000  0.0000  0.0000  0.0000
   0.2113  0.8497  0.7263  0.0000
   0.7560  0.6857  0.1985  0.6525
   0.0002  0.8782  0.5442  0.3076
   0.3303  0.0683  0.2320  0.9329
   0.5618  0.5042
   0.5896  0.3493
   0.6853  0.3873
   0.8906  0.9222
   1.0000  0.0000
   0.0000  1.0000
   0.3616  0.0000  0.0000  0.0000
   0.2922  0.4826  0.0000  0.0000
   0.9488  0.0000
   0.3760  0.7340
```

```
Example 2
   4     2     2     0.0
   1.0000  0.0000  0.0000  0.0000
   0.8400  0.9010  0.0000  0.0000
   0.3000  0.7001  0.8300  0.0000
   0.5000  0.2300  0.1100  0.4303
   0.2113  0.8497  0.7263  0.8833
   0.7560  0.6857  0.1985  0.6525
   0.0002  0.8782  0.5442  0.3076
   0.3303  0.0683  0.2320  0.9329
   0.5618  0.5042
   0.5896  0.3493
   0.6853  0.3873
   0.8906  0.9222
   1.0000  0.0000
   0.0000  1.0000
   0.3616  0.5664  0.5015  0.2693
   0.2922  0.4826  0.4368  0.6325
   0.9488  0.0000
   0.3760  0.7340
```

10.3 Program Results

```
nag_kalman_sqrt_filt_cov_invar (g13ebc) Example Program Results

Example 1

The square root of the state covariance matrix is

 -1.7223   0.0000   0.0000   0.0000
 -2.1073   0.5467   0.0000   0.0000
 -1.7649   0.1412  -0.1710   0.0000
 -1.8291   0.2058  -0.1497   0.7760

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

 -0.2135   1.6649
 -0.2345   2.1442
 -0.2147   1.7069
 -0.1345   1.4777

Example 2

Covariance matrix PE from nag_kalman_sqrt_filt_cov_var (g13eac) is

  1.6761   1.4744   1.2519   1.6852
  1.4744   1.3646   1.1367   1.4651
  1.2519   1.1367   1.0668   1.3445
  1.6852   1.4651   1.3445   2.2045

Covariance matrix PF from nag_kalman_sqrt_filt_cov_invar (g13ebc) is

  5.0635  -1.5512   0.0231   1.1756
 -1.5512   0.8503  -0.0492  -0.3631
  0.0231  -0.0492   0.0648  -0.0217
  1.1756  -0.3631  -0.0217   0.3336

Matrix U' * PF * U is

  1.6761   1.4744   1.2519   1.6852
  1.4744   1.3646   1.1367   1.4651
  1.2519   1.1367   1.0668   1.3445
  1.6852   1.4651   1.3445   2.2045

The matrix KE from nag_kalman_sqrt_filt_cov_var (g13eac) is

  0.3699   0.9447
  0.3526   0.8199
  0.2783   0.5375
```

```
  0.1588   0.6704

The matrix KF from nag_kalman_sqrt_filt_cov_invar (g13ebc) is

 -0.5857  -1.4263
 -0.0280   0.2239
  0.0170   0.1200
 -0.1405  -0.4519

U' * KF is

  0.3699   0.9447
  0.3526   0.8199
  0.2783   0.5375
  0.1588   0.6704
```