NAG Library Function Document

nag_kalman_sqrt_filt_info_invar (g13edc)

1 Purpose

nag_kalman_sqrt_filt_info_invar (g13edc) 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 information filter with the system matrices in condensed controller Hessenberg form.

2 Specification

```
#include <nag.h>
#include <nagg13.h>
```

```
void nag_kalman_sqrt_filt_info_invar (Integer n, Integer m, Integer p,
     double t[], Integer tdt, const double ainv[], Integer tda,
     const double ainvb[], Integer tdai, const double rinv[], Integer tdr,
     const double c[], Integer tdc, const double qinv[], Integer tdq,
     double x[], const double rinvy[], const double z[], double tol,
     NagError *fail)
```

3 Description

For the state space system defined by

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

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

$$U_1 \underset{\text{(Pre-array)}}{\begin{pmatrix} Q_i^{-1/2} & 0 & Q_i^{-1/2}\bar{w} \\ 0 & R_i^{-1/2}C & R^{-1/2}Y_{i+1} \\ S_i^{-1}A^{-1}B & S_i^{-1}A^{-1} & S_i^{-1}X_{i|i} \end{pmatrix}} = \underset{\text{(Post-array)}}{\begin{pmatrix} F_{i+1}^{-1/2} & * & * \\ 0 & S_{i+1}^{-1} & \xi_{i+1|i+1} \\ 0 & 0 & E_{i+1} \end{pmatrix}}$$

where U_1 is an orthogonal transformation triangularizing the pre-array, and the matrix pair $\left(A^{-1}, A^{-1}B\right)$ is in upper controller Hessenberg form. The triangularization is done entirely via Householder transformations exploiting the zero pattern of the pre-array. An example of the pre-array is given below (where $n = 6$, $m = 2$ and $p = 3$):

$$\left(\begin{array}{ccc|cccccc} x & x & & & & & & & x \\ & x & & & & & & & x \\ & & x & x & x & x & x & x & x \\ \hline x & x & x & x & x & x & x & x & x \\ & x & x & x & x & x & x & x & x \\ & & x & x & x & x & x & x & x \\ & & & x & x & x & x & x & x \\ \hline & & & & x & x & x & x & x \\ \hline & & & & & x & x & x & x \\ \hline \end{array}\right)$$

The term $\bar{w}$ is the mean process noise, and E_{i+1} is the estimated error at instant $i + 1$. The inverse of the state covariance matrix $P_{i|i}$ is factored as follows

$$P_{i|i}^{-1} = \left(S_i^{-1}\right)^{\mathrm{T}} S_i^{-1}$$

where $P_{i|i} = S_i S_i^{\mathrm{T}}$ (S_i is lower). The new state filtered state estimate is computed via

$$\hat{X}_{i+1|i+1} = S_{i+1}^{-1} \xi_{i+1|i+1}$$

The function returns S_{i+1}^{-1} and, optionally, $\hat{X}_{i+1|i+1}$ (see the Introduction to Chapter g13 for more information concerning the information 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^{-1} and A^{-1}.

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: **t**[**n** × **tdt**] – double *Input/Output*

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

On entry: the leading n by n upper triangular part of this array must contain S_i^{-1} the square root of the inverse of the state covariance matrix $P_{i|i}$.

On exit: the leading n by n upper triangular part of this array contains S_{i+1}^{-1}, the square root of the inverse of the of the state covariance matrix $P_{i+1|i+1}$.

5: **tdt** – Integer *Input*

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

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

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

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

On entry: the leading n by n part of this array must contain the upper controller Hessenberg matrix $UA^{-1}U^{\mathrm{T}}$. Where A^{-1} is the inverse of the state transition matrix, and U is the unitary matrix generated by the function nag_trans_hessenberg_controller (g13exc).

7: **tda** – Integer *Input*

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

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

8: **ainvb**[**n** $\times$ **tdai**] – const double *Input*

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

On entry: the leading n by m part of this array must contain the upper controller Hessenberg matrix $UA^{-1}B$. Where A^{-1} is the inverse of the transition matrix, B is the input weight matrix B, and U is the unitary transformation generated by the function nag_trans_hessenberg_controller (g13exc).

9: **tdai** – Integer *Input*

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

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

10: **rinv**[**p** $\times$ **tdr**] – const double *Input*

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

On entry: if the noise covariance matrix is to be supplied separately from the output weight matrix, then the leading p by p upper triangular part of this array must contain $R^{-1/2}$ the right Cholesky factor of the inverse of the measurement noise covariance matrix. If this information is not to be input separately from the output weight matrix **c** then the array **rinv** must be set to **NULL**.

11: **tdr** – Integer *Input*

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

Constraint: **tdr** $\geq$ **p** if **rinv** is defined.

12: **c**[**p** $\times$ **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: if the array argument **rinv** (above) has been defined then the leading p by n part of this array must contain the matrix CU^{T}, otherwise (if **rinv** is **NULL** then the leading p by n part of the array must contain the matrix $R_i^{-1/2}CU^{\mathrm{T}}$. C is the output weight matrix, R_i is the noise covariance matrix and U is the same unitary transformation used for defining array arguments **ainv** and **ainvb**.

13: **tdc** – Integer *Input*

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

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

14: **qinv**[**m** $\times$ **tdq**] – const double *Input*

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

On entry: the leading m by m upper triangular part of this array must contain $Q_i^{-1/2}$ the right Cholesky factor of the inverse of the process noise covariance matrix.

15: **tdq** – Integer *Input*

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

Constraint: $\mathbf{tdq} \geq \mathbf{m}$.

16: **x**[**n**] – double *Input/Output*

On entry: this array must contain the estimated state $\hat{X}_{i|i}$

On exit: this array contains the estimated state $\hat{X}_{i+1|i+1}$.

17: **rinvy**[**p**] – const double *Input*

On entry: this array must contain $R_i^{-1/2} Y_{i+1}$, the product of the upper triangular matrix $R_i^{-1/2}$ and the measured output vector Y_{i+1}.

18: **z**[**m**] – const double *Input*

On entry: this array must contain $\bar{w}$, the mean value of the state process noise.

19: **tol** – double *Input*

On entry: **tol** is used to test for near singularity of the matrix S_{i+1}. If you set **tol** to be less than $n^2 \times \epsilon$ then the tolerance is taken as $n^2 \times \epsilon$, where ϵ is the ***machine precision***.

20: **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, $\mathbf{tdt} = \langle value \rangle$ while $\mathbf{n} = \langle value \rangle$. These arguments must satisfy $\mathbf{tdt} \geq \mathbf{n}$.
On entry $\mathbf{tda} = \langle value \rangle$ while $\mathbf{n} = \langle value \rangle$. These arguments must satisfy $\mathbf{tda} \geq \mathbf{n}$.
On entry $\mathbf{tdai} = \langle value \rangle$ while $\mathbf{m} = \langle value \rangle$. These arguments must satisfy $\mathbf{tdai} \geq \mathbf{m}$.
On entry $\mathbf{tdc} = \langle value \rangle$ while $\mathbf{n} = \langle value \rangle$. These arguments must satisfy $\mathbf{tdc} \geq \mathbf{n}$.
On entry $\mathbf{tdq} = \langle value \rangle$ while $\mathbf{m} = \langle value \rangle$. These arguments must satisfy $\mathbf{tdq} \geq \mathbf{m}$.
On entry $\mathbf{tdr} = \langle value \rangle$ while $\mathbf{p} = \langle value \rangle$. These arguments must satisfy $\mathbf{tdr} \geq \mathbf{p}$.

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_INT_ARG_LT

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

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 inverse(S) is singular.

7 Accuracy

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

8 Parallelism and Performance

nag_kalman_sqrt_filt_info_invar (g13edc) is not threaded in any implementation.

9 Further Comments

The algorithm requires approximately $\frac{1}{6}n^3 + n^2\left(\frac{3}{2}m + p\right) + 2nm^2 + \frac{2}{3}m^3$ operations and is backward stable (see Verhaegen and van Dooren (1986)).

10 Example

For this function two examples are presented. There is a single example program for nag_kalman_sqrt_filt_info_invar (g13edc), 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 information form) to the time-invariant system $\left(A^{-1}, A^{-1}B, C\right)$ supplied in upper controller Hessenberg form.

Example 2 ex2)

To apply three iterations of the Kalman filter (in square root information form) to the general time-invariant system $\left(A^{-1}, A^{-1}B, C\right)$. The use of the time-varying Kalman function nag_kalman_sqrt_filt_info_var (g13ecc) is compared with that of the time-invariant function nag_kalman_sqrt_filt_info_invar (g13edc). The same original data is used by both functions but additional transformations are required before it can be supplied to nag_kalman_sqrt_filt_info_invar (g13edc). It can be seen that (after the appropriate back-transformations on the output of nag_kalman_sqrt_filt_info_invar (g13edc)) the results of both nag_kalman_sqrt_filt_info_var (g13ecc) and nag_kalman_sqrt_filt_info_invar (g13edc) are in agreeement.

10.1 Program Text

```
/* nag_kalman_sqrt_filt_info_invar (g13edc) Example Program.
 *
 * NAGPRODCODE Version.
 *
 * Copyright 2016 Numerical Algorithms Group.
 *
 * Mark 26, 2016.
 */

#include <nag.h>
#include <stdio.h>
#include <nag_stdlib.h>
#include <nagf07.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;
  /* Skip the heading in the data file   */
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  printf("nag_kalman_sqrt_filt_info_invar (g13edc) Example Program "
         "Results\n\n");

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

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

#define AINV(I, J)  ainv[(I) *tdainv + J]
#define QINV(I, J)  qinv[(I) *tdqinv + J]
#define RINV(I, J)  rinv[(I) *tdrinv + J]
#define T(I, J)     t[(I) *tdt + J]
#define AINVB(I, J) ainvb[(I) *tdainvb + J]
#define C(I, J)     c[(I) *tdc + J]

static int ex1(void)
{
  Integer exit_status = 0, i, istep, j, m, n, p, tdainv, tdainvb, tdc, tdqinv;
  Integer tdrinv, tdt;
  double *ainv = 0, *ainvb = 0, *c = 0, *qinv = 0, *rinv = 0, *rinvy = 0;
  double *t = 0, tol, *x = 0, *z = 0;

  /* Nag Types */
  NagError fail;

  INIT_FAIL(fail);

  printf("Example 1\n");

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

#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) {
    if (!(ainv = NAG_ALLOC(n * n, double)) ||
        !(qinv = NAG_ALLOC(m * m, double)) ||
        !(rinv = NAG_ALLOC(p * p, double)) ||
        !(t = NAG_ALLOC(n * n, double)) ||
        !(ainvb = NAG_ALLOC(n * m, double)) ||
        !(c = NAG_ALLOC(p * n, double)) ||
        !(x = NAG_ALLOC(n, double)) ||
        !(z = NAG_ALLOC(m, double)) || !(rinvy = NAG_ALLOC(p, double)))
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }
    tdainv = n;
    tdqinv = m;
    tdrinv = p;
    tdt = n;
    tdainvb = m;
    tdc = n;
  }
  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)
#ifdef _WIN32
      scanf_s("%lf", &AINV(i, j));
#else
      scanf("%lf", &AINV(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
  if (rinv)
    for (i = 0; i < p; ++i)
      for (j = 0; j < p; ++j)
#ifdef _WIN32
        scanf_s("%lf", &RINV(i, j));
#else
        scanf("%lf", &RINV(i, j));
#endif
  for (i = 0; i < n; ++i)
    for (j = 0; j < m; ++j)
#ifdef _WIN32
      scanf_s("%lf", &AINVB(i, j));
#else
      scanf("%lf", &AINVB(i, j));
#endif
  for (i = 0; i < m; ++i)
    for (j = 0; j < m; ++j)
#ifdef _WIN32
      scanf_s("%lf", &QINV(i, j));
#else
      scanf("%lf", &QINV(i, j));
#endif
  for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j)
#ifdef _WIN32
      scanf_s("%lf", &T(i, j));
#else
      scanf("%lf", &T(i, j));
#endif
  for (j = 0; j < m; ++j)
#ifdef _WIN32
    scanf_s("%lf", &z[j]);
#else
    scanf("%lf", &z[j]);
#endif
  for (j = 0; j < n; ++j)
#ifdef _WIN32
    scanf_s("%lf", &x[j]);
#else
    scanf("%lf", &x[j]);
#endif
  for (j = 0; j < p; ++j)
#ifdef _WIN32
    scanf_s("%lf", &rinvy[j]);
#else
    scanf("%lf", &rinvy[j]);
#endif

  /* Perform three iterations of the Kalman filter recursion  */

  for (istep = 1; istep <= 3; ++istep)
```

```
    /* nag_kalman_sqrt_filt_info_invar (g13edc).
     * One iteration step of the time-invariant Kalman filter
     * recursion using the square root information
     * implementation with (A^(-1)(A^(-1)B)) in upper
     * controller Hessenberg form
     */
    nag_kalman_sqrt_filt_info_invar(n, m, p, t, tdt, ainv,
                                    tdainv, ainvb, tdainvb, rinv,
                                    tdrinv, c, tdc, qinv,
                                    tdqinv, x, rinvy, z, tol, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_kalman_sqrt_filt_info_invar (g13edc).\n%s\n",
           fail.message);
    exit_status = 1;
    goto END;
  }
  printf("\nThe inverse of the 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 ", T(i, j));
    printf("\n");
  }
  printf("\nThe components of the estimated filtered state are\n\n");
  printf("   k       x(k)  \n");
  for (i = 0; i < n; ++i) {
    printf("   %" NAG_IFMT "  ", i);
    printf("  %8.4f  \n", x[i]);
  }

END:
  NAG_FREE(ainv);
  NAG_FREE(qinv);
  NAG_FREE(rinv);
  NAG_FREE(t);
  NAG_FREE(ainvb);
  NAG_FREE(c);
  NAG_FREE(x);
  NAG_FREE(z);
  NAG_FREE(rinvy);

  return exit_status;
}

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

#define AINV(I, J)   ainv[(I) *tdainv + J]
#define AINVB(I, J)  ainvb[(I) *tdainvb + J]
#define AINVU(I, J)  ainvu[(I) *tdainvu + J]
#define AINVBU(I, J) ainvbu[(I) *tdainvbu + J]
#define TU(I, J)     tu[(I) *tdtu + J]
#define IH(I, J)     ih[(I) *tdih + J]
#define U(I, J)      u[(I) *tdu + J]

static int ex2(void)
{
  Integer exit_status = 0, i, ione = 1, istep, j, m, n, p, tdainv, tdainvb;
  Integer tdainvbu, tdainvu, tdc, tdcu, tdig, tdih, tdqinv, tdrinv, tdrwork;
  Integer tdt, tdtu, tdu;
  Nag_ControllerForm reduceto = Nag_UH_Controller;
  Nag_ab_input inp_ab = Nag_ab_prod;
  double *ainv = 0, *ainvb = 0, *ainvbu = 0, *ainvu = 0, *c = 0;
  double *cu = 0, *ig = 0, *ih = 0, one = 1.0;
  double *qinv = 0, *rinv = 0, *rinvy = 0, *rwork = 0, *t = 0;
  double tol, *tu = 0, *u = 0, *ux = 0, *x = 0, *z = 0, zero = 0.0;

  /* Nag Types */
  NagError fail;
  INIT_FAIL(fail);

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

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

#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) {
    if (!(ainv = NAG_ALLOC(n * n, double)) ||
        !(ainvb = NAG_ALLOC(n * m, double)) ||
        !(c = NAG_ALLOC(p * n, double)) ||
        !(ainvu = NAG_ALLOC(n * n, double)) ||
        !(ainvbu = NAG_ALLOC(n * m, double)) ||
        !(qinv = NAG_ALLOC(m * m, double)) ||
        !(rinv = NAG_ALLOC(p * p, double)) ||
        !(t = NAG_ALLOC(n * n, double)) ||
        !(x = NAG_ALLOC(n, double)) ||
        !(z = NAG_ALLOC(n, double)) ||
        !(rwork = NAG_ALLOC(n * n, double)) ||
        !(tu = NAG_ALLOC(n * n, double)) ||
        !(rinvy = NAG_ALLOC(p, double)) ||
        !(ig = NAG_ALLOC(n * n, double)) ||
        !(ih = NAG_ALLOC(n * n, double)) ||
        !(cu = NAG_ALLOC(p * n, double)) ||
        !(u = NAG_ALLOC(n * n, double)) || !(ux = NAG_ALLOC(n, double)))
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }
    tdainv = n;
    tdainvb = m;
    tdc = n;
    tdainvu = n;
    tdainvbu = m;
    tdqinv = m;
    tdrinv = p;
    tdt = n;
    tdrwork = n;
    tdtu = n;
    tdig = n;
    tdih = n;
    tdcu = n;
    tdu = n;
  }
  else {
    printf("Invalid n or m or p.\n");
    exit_status = 1;
    return exit_status;
  }

  /* Read data */
  mat_io(n, n, ainv, tdainv, read, "");
  mat_io(p, n, c, tdc, read, "");
  if (rinv)
    mat_io(p, p, rinv, tdrinv, read, "");
  mat_io(n, m, ainvb, tdainvb, read, "");
  mat_io(m, m, qinv, tdqinv, read, "");
  mat_io(n, n, t, tdt, read, "");
  for (j = 0; j < m; ++j)
#ifdef _WIN32
    scanf_s("%lf", &z[j]);
```

```
#else
    scanf("%lf", &z[j]);
#endif
  for (j = 0; j < n; ++j)
#ifdef _WIN32
    scanf_s("%lf", &x[j]);
#else
    scanf("%lf", &x[j]);
#endif
  for (j = 0; j < p; ++j)
#ifdef _WIN32
    scanf_s("%lf", &rinvy[j]);
#else
    scanf("%lf", &rinvy[j]);
#endif

  for (i = 0; i < n; ++i) { /* Initialize the identity matrix u */
    for (j = 0; j < n; ++j)
      U(i, j) = zero;
    U(i, i) = one;
  }

  /* Copy the arrays ainv[] and ainvb[] into ainvu[] and ainvbu[] */
  for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j)
      AINVU(j, i) = AINV(j, i);
  for (j = 0; j < m; ++j)
    for (i = 0; i < n; ++i)
      AINVBU(i, j) = AINVB(i, j);

  /* Transform (ainvu[],ainvbu[]) to reduceto controller Hessenberg form */
  /* nag_trans_hessenberg_controller (g13exc).
   * Unitary state-space transformation to reduce (BA) to
   * lower or upper controller Hessenberg form
   */
  nag_trans_hessenberg_controller(n, m, reduceto, ainvu, tdainvu, ainvbu,
                                  tdainvbu, u, tdu, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_trans_hessenberg_controller (g13exc).\n%s\n",
           fail.message);
    exit_status = 1;
    goto END;
  }

  /* Calculate the matrix cu = c*u'    */
  nag_dgemm(Nag_RowMajor, Nag_NoTrans, Nag_Trans, p, n, n, one, c, tdc,
            u, tdu, zero, cu, tdcu, &fail);

  /* Calculate the vector ux = u*x     */
  nag_dgemv(Nag_RowMajor, Nag_NoTrans, n, n, one, u, tdu, x, ione,
            zero, ux, ione, &fail);

  /* Form the information matrices ih = u*ig*u' and ig = t'*t     */
  nag_dgemm(Nag_RowMajor, Nag_Trans, Nag_NoTrans, n, n, n, one, t, tdt,
            t, tdt, zero, ig, tdig, &fail);
  nag_dgemm(Nag_RowMajor, Nag_NoTrans, Nag_Trans, n, n, n, one, ig, tdig,
            u, tdu, zero, rwork, tdrwork, &fail);
  nag_dgemm(Nag_RowMajor, Nag_NoTrans, Nag_NoTrans, n, n, n, one, u, tdu,
            rwork, tdrwork, zero, ih, tdih, &fail);

  /* Now find the triangular (right) Cholesky factor of ih */
  /* nag_dpotrf (f07fdc).
   * Cholesky factorization of real symmetric positive definite matrix.
   */
  nag_dpotrf(Nag_RowMajor, Nag_Lower, n, ih, tdih, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_dpotrf (f07fdc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }
  for (i = 0; i < n; ++i) {
    TU(i, i) = IH(i, i);
    for (j = 0; j < i; ++j) {
      TU(j, i) = IH(i, j);
      TU(i, j) = zero;
    }
  }

  /* Do three iterations of the Kalman filter recursion  */
  for (istep = 1; istep <= 3; ++istep) {
    /* nag_kalman_sqrt_filt_info_var (g13ecc).
     * One iteration step of the time-varying Kalman filter
     * recursion using the square root information
     * implementation
     */
    nag_kalman_sqrt_filt_info_var(n, m, p, inp_ab, t, tdt, ainv,
                                  tdainv, ainvb, tdainvb, rinv, tdrinv,
                                  c, tdc, qinv, tdqinv, x,
                                  rinvy, z, tol, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_kalman_sqrt_filt_info_var (g13ecc).\n%s\n",
             fail.message);
      exit_status = 1;
      goto END;
    }
    /* nag_kalman_sqrt_filt_info_invar (g13edc), see above. */
    nag_kalman_sqrt_filt_info_invar(n, m, p, tu, tdtu, ainvu, tdainvu,
                                    ainvbu, tdainvbu, rinv, tdrinv, cu,
                                    tdcu, qinv, tdqinv, ux, rinvy,
                                    z, tol, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_kalman_sqrt_filt_info_invar (g13edc).\n%s\n",
             fail.message);
      exit_status = 1;
      goto END;
    }
  }

  /* Print Results */
  printf("\nResults from nag_kalman_sqrt_filt_info_var (g13ecc) \n\n");
  /* let ig = t' * t */
  nag_dgemm(Nag_RowMajor, Nag_Trans, Nag_NoTrans, n, n, n, one, t, tdt,
            t, tdt, zero, ig, tdig, &fail);
  mat_io(n, n, ig, tdig, print, "The information matrix ig is\n");
  printf("\nThe components of the estimated filtered state are\n\n");
  printf("  k       x(k)  \n");
  for (i = 0; i < n; ++i)
    printf("  %" NAG_IFMT "   %8.4f \n", i, x[i]);
  printf("\nResults from nag_kalman_sqrt_filt_info_invar (g13edc) \n\n");
  /* let ih = tu' * tu */
  nag_dgemm(Nag_RowMajor, Nag_Trans, Nag_NoTrans, n, n, n, one, tu, tdtu,
            tu, tdtu, zero, ih, tdih, &fail);
  mat_io(n, n, ih, tdih, print, "The information matrix ih is\n");

  /* Calculate ih = u'*ih*u   */
  nag_dgemm(Nag_RowMajor, Nag_NoTrans, Nag_NoTrans, n, n, n, one, ih, tdih,
            u, tdu, zero, rwork, tdrwork, &fail);
  nag_dgemm(Nag_RowMajor, Nag_Trans, Nag_NoTrans, n, n, n, one, u, tdu,
            rwork, tdrwork, zero, ih, tdih, &fail);
  mat_io(n, n, ih, tdih, print, "\nThe matrix u' * ih * u is\n");

  /* Calculate x = u' * ux */
  nag_dgemv(Nag_RowMajor, Nag_Trans, n, n, one, u, tdu, ux, ione,
            zero, x, ione, &fail);
  printf("\nThe components of the estimated filtered state are \n\n");
  printf("  k       x(k)  \n");
  for (i = 0; i < n; ++i)
    printf("  %" NAG_IFMT "   %8.4f \n", i, x[i]);

END:
  NAG_FREE(ainv);
  NAG_FREE(ainvb);
  NAG_FREE(c);
  NAG_FREE(ainvu);
  NAG_FREE(ainvbu);
  NAG_FREE(qinv);
  NAG_FREE(rinv);
  NAG_FREE(t);
  NAG_FREE(x);
  NAG_FREE(z);
  NAG_FREE(rwork);
  NAG_FREE(tu);
  NAG_FREE(rinvy);
  NAG_FREE(ig);
  NAG_FREE(ih);
  NAG_FREE(cu);
  NAG_FREE(u);
  NAG_FREE(ux);

  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) {
#ifdef _WIN32
      if (flag == read)
        scanf_s("%lf", &MAT(i, j));
#else
      if (flag == read)
        scanf("%lf", &MAT(i, j));
#endif
      if (flag == print)
        printf("%8.4f ", MAT(i, j));
    }
    if (flag == print)
      printf("\n");
  }
} /* mat_io */
```

10.2 Program Data

```
nag_kalman_sqrt_filt_info_invar (g13edc) Example Program Data
Example 1
   4     2     2   0.0
   0.2113  0.7560  0.0002  0.3303
   0.8497  0.6857  0.8782  0.0683
   0.7263  0.1985  0.5442  0.2320
   0.0000  0.6525  0.3076  0.9329
   0.3616  0.5664  0.5015  0.2693
   0.2922  0.4826  0.4368  0.6325
   1.0000  0.0000
   0.0000  1.0000
  -0.8805  1.3257
   0.0000  0.5207
   0.0000  0.0000
   0.0000  0.0000
   1.1159  0.2305
   0.0000  0.6597
   1.0000  0.0000  0.0000  0.0000
   0.0000  1.0000  0.0000  0.0000
   0.0000  0.0000  1.0000  0.0000
   0.0000  0.0000  0.0000  1.0000
   0.0019
   0.5075
   0.4076
   0.8408
   0.5017
   0.9128
   0.2129
   0.5591
Example 2
   4     2     2    0.0
   0.2113  0.7560  0.0002  0.3303
   0.8497  0.6857  0.8782  0.0683
   0.7263  0.1985  0.5442  0.2320
   0.8833  0.6525  0.3076  0.9329
   0.3616  0.5664  0.5015  0.2693
   0.2922  0.4826  0.4368  0.6325
   1.0000  0.0000
   0.0000  1.0000
  -0.8805  1.3257
   2.1039  0.5207
  -0.6075  1.0386
  -0.8531  1.1688
   1.1159  0.2305
   0.0000  0.6597
   1.0000  2.1000  0.1400   0.0000
   0.0000  0.6010  2.8000  -1.3400
   0.0000  0.0000  1.3000  -0.8000
   0.0000  0.0000  0.0000   1.4100
   0.0019
   0.5075
   0.4076
   0.8408
   0.5017
   0.9128
   0.2129
   0.5591
```

10.3 Program Results

```
nag_kalman_sqrt_filt_info_invar (g13edc) Example Program Results

Example 1

The inverse of the square root of the state covariance matrix is

 -0.8731  -1.1461  -1.0260  -0.8901
  0.0000  -0.2763  -0.1929  -0.3763
  0.0000   0.0000  -0.1110  -0.1051
  0.0000   0.0000   0.0000   0.3120

The components of the estimated filtered state are

   k       x(k)
   0     -2.0688
   1     -0.7814
   2      2.2181
   3      0.9298


Example 2

Results from nag_kalman_sqrt_filt_info_var (g13ecc)

The information matrix ig is

  0.4661   0.5290   0.4826   0.4134
  0.5290   0.7196   0.6158   0.5657
  0.4826   0.6158   0.5781   0.4776
  0.4134   0.5657   0.4776   0.5825

The components of the estimated filtered state are

  k       x(k)
  0    -0.8369
  1    -1.4649
  2     1.4877
  3     1.5276

Results from nag_kalman_sqrt_filt_info_invar (g13edc)

The information matrix ih is

  0.0399  -0.0805  -0.0137  -0.0174
 -0.0805   2.1143   0.2453   0.0406
 -0.0137   0.2453   0.0770  -0.0294
 -0.0174   0.0406  -0.0294   0.1151

The matrix u' * ih * u is

  0.4661   0.5290   0.4826   0.4134
  0.5290   0.7196   0.6158   0.5657
  0.4826   0.6158   0.5781   0.4776
  0.4134   0.5657   0.4776   0.5825

The components of the estimated filtered state are

  k       x(k)
  0    -0.8369
  1    -1.4649
  2     1.4877
  3     1.5276
```