NAG Library Function Document

nag_kalman_sqrt_filt_info_var (g13ecc)

1 Purpose

nag_kalman_sqrt_filt_info_var (g13ecc) 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 information filter with the system matrices in their original form.

2 Specification

```
#include <nag.h>
#include <nagg13.h>
void nag_kalman_sqrt_filt_info_var (Integer n, Integer m, Integer p,
     Nag_ab_input inp_ab, double t[], Integer tdt, const double ainv[],
     Integer tda, const double b[], Integer tdb, 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

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

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

$$U_1 \quad \underbrace{\begin{pmatrix} Q_i^{-1/2} & 0 & Q_i^{-1/2}\bar{w}_i \\ S_i^{-1}A_i^{-1}B_i & S_i^{-1}A_i^{-1} & S_i^{-1}\hat{X}_{i|i} \\ 0 & R_{i+1}^{-1/2}C_{i+1} & R_{i+1}^{-1/2}Y_{i+1} \end{pmatrix}}_{\text{(Pre-array)}} = \underbrace{\begin{pmatrix} F_{i+1}^{-1/2} & * & * \\ 0 & S_{i+1}^{-1} & \xi_{i+1|i+1} \\ 0 & 0 & E_{i+1} \end{pmatrix}}_{\text{(Post-array)}}$$

where U_1 is an orthogonal transformation triangularizing the pre-array. The triangularization is done entirely via Householder transformations exploiting the zero pattern of the pre-array. The term $\bar{w}_i$ 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 triangular).

The new state filtered state estimate is computed via

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

The function returns S_{i+1}^{-1} and $\hat{X}_{i+1|i+1}$ (see the g13 Chapter Introduction 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

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_i^{-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 $R_{i+1}^{-1/2}$.

Constraint: **p** ≥ 1.

4: **inp_ab** – Nag_ab_input *Input*

On entry: indicates how the matrix B_i is to be passed to the function.

inp_ab = Nag_ab_prod
Array **b** must contain the product $A_i^{-1}B_i$.

inp_ab = Nag_ab_sep
Then array **b** must contain B_i.

5: **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 $P_{i+1|i+1}$.

6: **tdt** – Integer *Input*

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

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

7: **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 A_i^{-1} the inverse of the state transition matrix.

8: **tda** – Integer *Input*

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

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

9: **b**[**n** $\times$ **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: the leading n by m part of this array must contain B_i (if **inp_ab** = Nag_ab_sep) or its product with A_i^{-1} (if **inp_ab** = Nag_ab_prod).

10: **tdb** – Integer *Input*

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

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

11: **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 measurement 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 $R_{i+1}^{-1/2}$, 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 (see below) then the array **rinv** must be set to **NULL**.

12: **tdr** – Integer *Input*

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

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

13: **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 **rinv** has been set to **NULL** then the leading p by n part of this array must contain C_{i+1}, the output weight matrix (or its product with $R_{i+1}^{-1/2}$) of the discrete system at instant $i+1$.

14: **tdc** – Integer *Input*

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

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

15: **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.

16: **tdq** – Integer *Input*

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

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

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

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

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

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

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

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

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

20: **tol** – double *Input*

On entry: **tol** is used to test for near singularity of the matrix S_{i+1}^{-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***.

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, **tda** = ⟨*value*⟩ while **n** = ⟨*value*⟩. These arguments must satisfy **tda** ≥ **n**.

On entry, **tdb** = ⟨*value*⟩ while **m** = ⟨*value*⟩. These arguments must satisfy **tdb** ≥ **m**.

On entry, **tdc** = ⟨*value*⟩ while **n** = ⟨*value*⟩. These arguments must satisfy **tdc** ≥ **n**.

On entry, **tdq** = ⟨*value*⟩ while **m** = ⟨*value*⟩. These arguments must satisfy **tdq** ≥ **m**.

On entry, **tdr** = ⟨*value*⟩ while **p** = ⟨*value*⟩. These arguments must satisfy **tdr** ≥ **p**.

On entry, **tdt** = ⟨*value*⟩ while **n** = ⟨*value*⟩. These arguments must satisfy **tdt** ≥ **n**.

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_BAD_PARAM

On entry, argument **inp_ab** had an illegal value.

NE_INT_ARG_LT

On entry, **m** = ⟨*value*⟩.
Constraint: **m** ≥ 1.

On entry, **n** = ⟨*value*⟩.
Constraint: **n** ≥ 1.

On entry, **p** = ⟨*value*⟩.
Constraint: **p** ≥ 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

Not applicable.

9 Further Comments

The algorithm requires approximately $\frac{7}{6}n^3 + n^2\left(\frac{7}{2}m + p\right) + n\left(\frac{1}{2}p^2 + m^2\right)$ operations and is backward stable (see Verhaegen and van Dooren (1986)).

10 Example

To apply three iterations of the Kalman filter (in square root information form) to the system $\left(A_i^{-1}, A_i^{-1}B_i, C_{i+1}\right)$. The same data is used for all three iterative steps.

10.1 Program Text

```
/* nag_kalman_sqrt_filt_info_var (g13ecc) Example Program.
 *
 * Copyright 1993 Numerical Algorithms Group
 *
 * Mark 3, 1993
 * Mark 8 revised, 2004.
 */

#include <nag.h>
#include <stdio.h>
#include <nag_stdlib.h>
#include <nagg13.h>

#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 B(I, J)    b[(I) *tdb + J]
#define C(I, J)    c[(I) *tdc + J]
int main(void)
{

  Integer      exit_status = 0, i, istep, j, m, n, p, tdainv, tdb, tdc, tdqinv,
               tdrinv;
  Integer      tdt;
  Nag_ab_input inp_ab;
  double       *ainv = 0, *b = 0, *c = 0, *qinv = 0, *rinv = 0, *rinvy = 0;
  double       *t = 0, tol, *x = 0, *z = 0;
  NagError     fail;

  INIT_FAIL(fail);

  printf(
         "nag_kalman_sqrt_filt_info_var (g13ecc) Example Program Results\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 (!(ainv = NAG_ALLOC(n*n, double)) ||
          !(qinv = NAG_ALLOC(m*m, double)) ||
          !(rinv = NAG_ALLOC(p*p, double)) ||
          !(t = NAG_ALLOC(n*n, double)) ||
          !(b = 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;
      tdb = m;
      tdc = n;
    }
  else
    {
      printf("Invalid n or m or p.\n");
      exit_status = 1;
      return exit_status;
    }
  inp_ab = Nag_ab_prod;

  /* Read data */
  for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j)
      scanf("%lf", &AINV(i, j));
  for (i = 0; i < p; ++i)
    for (j = 0; j < n; ++j)
      scanf("%lf", &C(i, j));
  if (rinv)
    for (i = 0; i < p; ++i)
      for (j = 0; j < p; ++j)
        scanf("%lf", &RINV(i, j));
  for (i = 0; i < n; ++i)
    for (j = 0; j < m; ++j)
      scanf("%lf", &B(i, j));
  for (i = 0; i < m; ++i)
    for (j = 0; j < m; ++j)
      scanf("%lf", &QINV(i, j));
  for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j)
      scanf("%lf", &T(i, j));
  for (j = 0; j < m; ++j)
    scanf("%lf", &z[j]);
  for (j = 0; j < n; ++j)
    scanf("%lf", &x[j]);
  for (j = 0; j < p; ++j)
    scanf("%lf", &rinvy[j]);

  /* Perform three iterations of the (Kalman) filter recursion
     (in square root information form). */
  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, b,
                                  tdb, 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;
    }

  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("%ld  ", i);
      printf("  %8.4f  \n", x[i]);
    }

 END:
  NAG_FREE(ainv);
  NAG_FREE(qinv);
  NAG_FREE(rinv);
  NAG_FREE(t);
  NAG_FREE(b);
  NAG_FREE(c);
  NAG_FREE(x);
  NAG_FREE(z);
  NAG_FREE(rinvy);
  return exit_status;
}
```

10.2 Program Data

```
nag_kalman_sqrt_filt_info_var (g13ecc) Example Program Data
   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  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
```

10.3 Program Results

```
nag_kalman_sqrt_filt_info_var (g13ecc) Example Program Results

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

  0.6897   0.7721   0.7079   0.6102
  0.0000  -0.3363  -0.2252  -0.2642
  0.0000   0.0000  -0.1650   0.0319
  0.0000   0.0000   0.0000   0.3708

The components of the estimated filtered state are

k       x(k)
0     -0.7125
1     -1.8324
2      1.7500
3      1.5854
```