NAG Library Function Document

nag_kalman_unscented_state (g13ekc)

1 Purpose

nag_kalman_unscented_state (g13ekc) applies the Unscented Kalman Filter (UKF) to a nonlinear state space model, with additive noise.

nag_kalman_unscented_state (g13ekc) uses direct communication for evaluating the nonlinear functionals of the state space model.

2 Specification

```
#include <nag.h>
#include <nagg13.h>
void nag_kalman_unscented_state (Integer mx, Integer my, const double y[],
     const double lx[], const double ly[],
     void (*f)(Integer mx, Integer n, const double xt[], double fxt[],
          Nag_Comm *comm, Integer *info),
     void (*h)(Integer mx, Integer my, Integer n, const double yt[],
          double hyt[], Nag_Comm *comm, Integer *info),
     double x[], double st[], Nag_Comm *comm, NagError *fail)
```

3 Description

nag_kalman_unscented_state (g13ekc) applies the Unscented Kalman Filter (UKF), as described in Julier and Uhlmann (1997b) to a nonlinear state space model, with additive noise, which, at time t, can be described by:

$$\begin{aligned} x_{t+1} &= F(x_t) + v_t \\ y_t &= H(x_t) + u_t \end{aligned}$$

where x_t represents the unobserved state vector of length m_x and y_t the observed measurement vector of length m_y. The process noise is denoted v_t, which is assumed to have mean zero and covariance structure Σ_x, and the measurement noise by u_t, which is assumed to have mean zero and covariance structure Σ_y.

3.1 Unscented Kalman Filter Algorithm

Given $\hat{x}_0$, an initial estimate of the state and P_0 and initial estimate of the state covariance matrix, the UKF can be described as follows:

(a) Generate a set of sigma points (see Section 3.2):

$$\mathcal{X}_t = \left[\hat{x}_{t-1} \quad \hat{x}_{t-1} + \gamma\sqrt{P_{t-1}} \quad \hat{x}_{t-1} - \gamma\sqrt{P_{t-1}}\right] \tag{1}$$

(b) Evaluate the known model function F:

$$\mathcal{F}_t = F(\mathcal{X}_t) \tag{2}$$

The function F is assumed to accept the $m_x \times n$ matrix, $\mathcal{X}_t$ and return an $m_x \times n$ matrix, $\mathcal{F}_t$. The columns of both $\mathcal{X}_t$ and $\mathcal{F}_t$ correspond to different possible states. The notation $\mathcal{F}_{t,i}$ is used to denote the ith column of $\mathcal{F}_t$, hence the result of applying F to the ith possible state.

(c) Time Update:

$$\hat{x}_t = \sum_{i=1}^{n} W_i^m \mathcal{F}_{t,i} \tag{3}$$

$$P_t = \sum_{i=1}^{n} W_i^c \left(\mathcal{F}_{t,i} - \hat{x}_t\right)\left(\mathcal{F}_{t,i} - \hat{x}_t\right)^{\mathrm{T}} + \Sigma_x \tag{4}$$

(d) Redraw another set of sigma points (see Section 3.2):

$$\mathcal{Y}_t = \left[\hat{x}_t \quad \hat{x}_t + \gamma\sqrt{P_t} \quad \hat{x}_t - \gamma\sqrt{P_t}\right] \tag{5}$$

(e) Evaluate the known model function H:

$$\mathcal{H}_t = H\left(\mathcal{Y}_t\right) \tag{6}$$

The function H is assumed to accept the $m_x \times n$ matrix, $\mathcal{Y}_t$ and return an $m_y \times n$ matrix, $\mathcal{H}_t$. The columns of both $\mathcal{Y}_t$ and $\mathcal{H}_t$ correspond to different possible states. As above $\mathcal{H}_{t,i}$ is used to denote the ith column of $\mathcal{H}_t$.

(f) Measurement Update:

$$\hat{y}_t = \sum_{i=1}^{n} W_i^m \mathcal{H}_{t,i} \tag{7}$$

$$P_{yy_t} = \sum_{i=1}^{n} W_i^c \left(\mathcal{H}_{t,i} - \hat{y}_t\right)\left(\mathcal{H}_{t,i} - \hat{y}_t\right)^{\mathrm{T}} + \Sigma_y \tag{8}$$

$$P_{xy_t} = \sum_{i=1}^{n} W_i^c \left(\mathcal{F}_{t,i} - \hat{x}_t\right)\left(\mathcal{H}_{t,i} - \hat{y}_t\right)^{\mathrm{T}} \tag{9}$$

$$\mathcal{K}_t = P_{xy_t} P_{yy_t}^{-1} \tag{10}$$

$$\hat{x}_t = \hat{x}_t + \mathcal{K}_t(y_t - \hat{y}_t) \tag{11}$$

$$P_t = P_t - \mathcal{K}_t P_{yy_t} \mathcal{K}_t^{\mathrm{T}} \tag{12}$$

Here $\mathcal{K}_t$ is the Kalman gain matrix, $\hat{x}_t$ is the estimated state vector at time t and P_t the corresponding covariance matrix. Rather than implementing the standard UKF as stated above nag_kalman_unscen ted_state (g13ekc) uses the square-root form described in the Haykin (2001).

3.2 Sigma Points

A nonlinear state space model involves propagating a vector of random variables through a nonlinear system and we are interested in what happens to the mean and covariance matrix of those variables. Rather than trying to directly propagate the mean and covariance matrix, the UKF uses a set of carefully chosen sample points, referred to as sigma points, and propagates these through the system of interest. An estimate of the propagated mean and covariance matrix is then obtained via the weighted sample mean and covariance matrix.

For a vector of m random variables, x, with mean μ and covariance matrix Σ, the sigma points are usually constructed as:

$$\mathcal{X}_t = \left[\mu \quad \mu + \gamma\sqrt{\Sigma} \quad \mu - \gamma\sqrt{\Sigma}\right]$$

When calculating the weighted sample mean and covariance matrix two sets of weights are required, one used when calculating the weighted sample mean, denoted W^m and one used when calculated the weighted sample covariance matrix, denoted W^c. The weights and multiplier, γ, are constructed as follows:

$$\begin{aligned}
\lambda &= \alpha^2(L+\kappa) - L \\
\gamma &= \sqrt{L+\lambda} \\
W_{\mathrm{i}}^m &= \begin{cases} \frac{\lambda}{L+\lambda} & i = 1 \\ \frac{1}{2(L+\lambda)} & i = 2,3,\ldots,2L+1 \end{cases} \\
W_{\mathrm{i}}^c &= \begin{cases} \frac{\lambda}{L+\lambda} + 1 - \alpha^2 + \beta & i = 1 \\ \frac{1}{2(L+\lambda)} & i = 2,3,\ldots,2L+1 \end{cases}
\end{aligned}$$

where, usually $L = m$ and α, β and κ are constants. The total number of sigma points, n, is given by $2L+1$. The constant α is usually set to somewhere in the range $10^{-4} \leq \alpha \leq 1$ and for a Gaussian distribution, the optimal values of κ and β are $3 - L$ and 2 respectively.

The constants, κ, α and β are given by $\kappa = 3 - m_x$, $\alpha = 1.0$ and $\beta = 2$. If more control is required over the construction of the sigma points then the reverse communication function, nag_kalman_unscented_ state_revcom (g13ejc), can be used instead.

4 References

Haykin S (2001) *Kalman Filtering and Neural Networks* John Wiley and Sons

Julier S J (2002) The scaled unscented transformation *Proceedings of the 2002 American Control Conference (Volume 6)* 4555–4559

Julier S J and Uhlmann J K (1997a) A consistent, debiased method for converting between polar and Cartesian coordinate systems *Proceedings of AeroSense97, International Society for Optics and Phonotonics* 110–121

Julier S J and Uhlmann J K (1997b) A new extension of the Kalman Filter to nonlinear systems *International Symposium for Aerospace/Defense, Sensing, Simulation and Controls (Volume 3)* **26**

5 Arguments

1: **mx** – Integer *Input*

On entry: m_x, the number of state variables.

Constraint: **mx** ≥ 1.

2: **my** – Integer *Input*

On entry: m_y, the number of observed variables.

Constraint: **my** ≥ 1.

3: **y**[**my**] – const double *Input*

On entry: y_t, the observed data at the current time point.

4: **lx**[**mx** $\times$ **mx**] – const double *Input*

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

On entry: L_x, such that $L_x L_x^{\mathrm{T}} = \Sigma_x$, i.e., the lower triangular part of a Cholesky decomposition of the process noise covariance structure. Only the lower triangular part of the matrix stored in **lx** is referenced.

If Σ_x is time dependent, then the value supplied should be for time t.

5: **ly**[**my** $\times$ **my**] – const double *Input*

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

On entry: L_y, such that $L_y L_y^{\mathrm{T}} = \Sigma_y$, i.e., the lower triangular part of a Cholesky decomposition of the observation noise covariance structure. Only the lower triangular part of the matrix stored in **ly** is referenced.

If Σ_y is time dependent, then the value supplied should be for time t.

6: **f** – function, supplied by the user *External Function*

The state function, F as described in (b).

The specification of **f** is:

```
void f (Integer mx, Integer n, const double xt[], double fxt[],
     Nag_Comm *comm, Integer *info)
```

1: **mx** – Integer *Input*

On entry: m_x, the number of state variables.

2: **n** – Integer *Input*

On entry: n, the number of sigma points.

3: **xt**[**mx** × **n**] – const double *Input*

On entry: X_t, the sigma points generated in (a). For the jth sigma point, the value for the ith parameter is held in **xt**$[(j-1) \times$ **mx** $+ i - 1]$, for $i = 1, 2, \ldots, m_x$ and $j = 1, 2, \ldots, n$.

4: **fxt**[**mx** × **n**] – double *Output*

On exit: $F(X_t)$.

For the jth sigma point the value for the ith parameter should be held in **fxt**$[(j-1) \times$ **mx** $+ i - 1]$, for $i = 1, 2, \ldots, m_x$ and $j = 1, 2, \ldots, n$.

5: **comm** – Nag_Comm *

Pointer to structure of type Nag_Comm; the following members are relevant to **f**.

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_kalman_unscented_state (g13ekc) you may allocate memory and initialize these pointers with various quantities for use by **f** when called from nag_kalman_unscented_state (g13ekc) (see Section 2.3.1.1 in How to Use the NAG Library and its Documentation).

6: **info** – Integer * *Input/Output*

On entry: **info** $= 0$.

On exit: set **info** to a nonzero value if you wish nag_kalman_unscented_state (g13ekc) to terminate with **fail.code** = NE_USER_STOP.

7: **h** – function, supplied by the user *External Function*

The measurement function, H as described in (e).

The specification of **h** is:

```
void h (Integer mx, Integer my, Integer n, const double yt[],
     double hyt[], Nag_Comm *comm, Integer *info)
```

1: **mx** – Integer *Input*

On entry: m_x, the number of state variables.

2: **my** – Integer *Input*

On entry: m_y, the number of observed variables.

3: **n** – Integer *Input*

On entry: n, the number of sigma points.

4: **yt**[**mx** × **n**] – const double *Input*

On entry: Y_t, the sigma points generated in (d). For the jth sigma point, the value for the ith parameter is held in **yt**$[(j-1) \times$ **mx** $+ i - 1]$, for $i = 1, 2, \ldots, m_x$ and $j = 1, 2, \ldots, n$, where m_x is the number of state variables and n is the number of sigma points.

5: **hyt**[**my** × **n**] – double *Output*

On exit: $H(Y_t)$.

For the jth sigma point the value for the ith parameter should be held in **hyt**$[(j-1) \times$ **my** $+ i - 1]$, for $i = 1, 2, \ldots, m_y$ and $j = 1, 2, \ldots, n$.

6: **comm** – Nag_Comm *

Pointer to structure of type Nag_Comm; the following members are relevant to **h**.

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_kalman_unscented_state (g13ekc) you may allocate memory and initialize these pointers with various quantities for use by **h** when called from nag_kalman_unscented_state (g13ekc) (see Section 2.3.1.1 in How to Use the NAG Library and its Documentation).

7: **info** – Integer * *Input/Output*

On entry: **info** $= 0$.

On exit: set **info** to a nonzero value if you wish nag_kalman_unscented_state (g13ekc) to terminate with **fail.code** = NE_USER_STOP.

8: **x**[*dim*] – double *Input/Output*

On entry: $\hat{x}_{t-1}$ the state vector for the previous time point.

On exit: $\hat{x}_t$ the updated state vector.

9: **st**[**mx** × **mx**] – double *Input/Output*

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

On entry: S_t, such that $S_{t-1}S_{t-1}^{\mathrm{T}} = P_{t-1}$, i.e., the lower triangular part of a Cholesky decomposition of the state covariance matrix at the previous time point. Only the lower triangular part of the matrix stored in **st** is referenced.

On exit: S_t, the lower triangular part of a Cholesky factorization of the updated state covariance matrix.

10: **comm** – Nag_Comm *

The NAG communication argument (see Section 2.3.1.1 in How to Use the NAG Library and its Documentation).

11: **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_ALLOC_FAIL

Dynamic memory allocation failed.
See Section 2.3.1.2 in How to Use the NAG Library and its Documentation for further information.

NE_BAD_PARAM

On entry, argument ⟨*value*⟩ had an illegal value.

NE_INT

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

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

NE_INTERNAL_ERROR

An internal error has occurred in this function. Check the function call and any array sizes. If the call is correct then please contact NAG for assistance.

An unexpected error has been triggered by this function. Please contact NAG.
See Section 2.7.6 in How to Use the NAG Library and its Documentation for further information.

NE_MAT_NOT_POS_DEF

A weight was negative and it was not possible to downdate the Cholesky factorization.

Unable to calculate the Cholesky factorization of the updated state covariance matrix.

Unable to calculate the Kalman gain matrix.

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 2.7.5 in How to Use the NAG Library and its Documentation for further information.

NE_USER_STOP

User requested termination in **f**.

User requested termination in **h**.

7 Accuracy

Not applicable.

8 Parallelism and Performance

nag_kalman_unscented_state (g13ekc) makes calls to BLAS and/or LAPACK routines, which may be threaded within the vendor library used by this implementation. Consult the documentation for the vendor library for further information.

Please consult the x06 Chapter Introduction for information on how to control and interrogate the OpenMP environment used within this function. Please also consult the Users' Note for your implementation for any additional implementation-specific information.

9 Further Comments

None.

10 Example

This example implements the following nonlinear state space model, with the state vector x and state update function F given by:

$$\begin{aligned} m_x &= 3 \\ x_{t+1} &= \begin{pmatrix} \xi_{t+1} & \eta_{t+1} & \theta_{t+1} \end{pmatrix}^{\mathrm{T}} \\ &= F(x_t) + v_t \\ &= x_t + \begin{pmatrix} \cos\theta_t & -\sin\theta_t & 0 \\ \sin\theta_t & \cos\theta_t & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0.5r & 0.5r \\ 0 & 0 \\ r/d & -r/d \end{pmatrix} \begin{pmatrix} \phi_{Rt} \\ \phi_{Lt} \end{pmatrix} + v_t \end{aligned}$$

where r and d are known constants and ϕ_{Rt} and ϕ_{Lt} are time-dependent knowns. The measurement vector y and measurement function H is given by:

$$\begin{aligned} m_y &= 2 \\ y_t &= (\delta_t, \alpha_t)^{\mathrm{T}} \\ &= H(x_t) + u_t \\ &= \begin{pmatrix} \Delta - \xi_t \cos A - \eta_t \sin A \\ \theta_t - A \end{pmatrix} + u_t \end{aligned}$$

where A and Δ are known constants. The initial values, x_0 and P_0, are given by

$$x_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \quad P_0 = \begin{pmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.1 \end{pmatrix}$$

and the Cholesky factorizations of the error covariance matrices, L_x and L_x by

$$L_{\mathrm{x}} = \begin{pmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.1 \end{pmatrix}, \quad L_{\mathrm{y}} = \begin{pmatrix} 0.01 & 0 \\ 0 & 0.01 \end{pmatrix}$$

10.1 Program Text

```
/* nag_kalman_unscented_state (g13ekc) Example Program.
 *
 * NAGPRODCODE Version.
 *
 * Copyright 2016 Numerical Algorithms Group.
 *
 * Mark 26, 2016.
 */

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

```
#include <nagg13.h>
#include <nagx01.h>

#define LY(I,J) ly[(J) * my + (I)]
#define LX(I,J) lx[(J) * mx + (I)]
#define ST(I,J) st[(J) * mx + (I)]
#define XT(I,J) xt[(J) * mx + (I)]
#define FXT(I,J) fxt[(J) * mx + (I)]
#define YT(I,J) yt[(J) * mx + (I)]
#define HYT(I,J) hyt[(J) * my + (I)]

typedef struct g13_problem_data
{
  double delta, a, r, d;
  double phi_rt, phi_lt;
} g13_problem_data;

#ifdef __cplusplus
extern "C"
{
#endif
  static void NAG_CALL f(Integer mx, Integer n, const double *xt,
                         double *fxt, Nag_Comm *comm, Integer *info);
  static void NAG_CALL h(Integer mx, Integer my, Integer n, const double *yt,
                         double *hyt, Nag_Comm *comm, Integer *info);
#ifdef __cplusplus
}
#endif

static void read_problem_dat(Integer t, Nag_Comm *comm);

int main(void)
{
  /* Integer scalar and array declarations */
  Integer mx = 3, my = 2;
  Integer i, ntime, t, j;
  Integer exit_status = 0;

  /* NAG structures and types */
  NagError fail;
  Nag_Comm comm;

  /* Double scalar and array declarations */
  double *lx = 0, *ly = 0, *st = 0, *x = 0, *y = 0;

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

  printf("nag_kalman_unscented_state (g13ekc) Example Program Results\n\n");

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

  if (!(lx = NAG_ALLOC(mx * mx, double)) ||
      !(ly = NAG_ALLOC(my * my, double)) ||
      !(x = NAG_ALLOC(mx, double)) ||
      !(st = NAG_ALLOC(mx * mx, double)) || !(y = NAG_ALLOC(my, double)))
  {
    printf("Allocation failure\n");
    exit_status = -1;
    goto END2;
  }

  /* Read in the Cholesky factorization of the covariance matrix for the
     process noise */
  for (i = 0; i < mx; i++) {
    for (j = 0; j <= i; j++) {
```

```
#ifdef _WIN32
      scanf_s("%lf", &LX(i, j));
#else
      scanf("%lf", &LX(i, j));
#endif
    }
#ifdef _WIN32
    scanf_s("%*[^\n] ");
#else
    scanf("%*[^\n] ");
#endif
  }

  /* Read in the Cholesky factorization of the covariance matrix for the
     observation noise */
  for (i = 0; i < my; i++) {
    for (j = 0; j <= i; j++) {
#ifdef _WIN32
      scanf_s("%lf", &LY(i, j));
#else
      scanf("%lf", &LY(i, j));
#endif
    }
#ifdef _WIN32
    scanf_s("%*[^\n] ");
#else
    scanf("%*[^\n] ");
#endif
  }

  /* Read in the initial state vector */
  for (i = 0; i < mx; i++) {
#ifdef _WIN32
    scanf_s("%lf", &x[i]);
#else
    scanf("%lf", &x[i]);
#endif
  }
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  /* Read in the Cholesky factorization of the initial state covariance
     matrix */
  for (i = 0; i < mx; i++) {
    for (j = 0; j <= i; j++) {
#ifdef _WIN32
      scanf_s("%lf", &ST(i, j));
#else
      scanf("%lf", &ST(i, j));
#endif
    }
#ifdef _WIN32
    scanf_s("%*[^\n] ");
#else
    scanf("%*[^\n] ");
#endif
  }

  /* Read in the number of time points to run the system for */
#ifdef _WIN32
  scanf_s("%" NAG_IFMT "%*[^\n] ", &ntime);
#else
  scanf("%" NAG_IFMT "%*[^\n] ", &ntime);
#endif

  /* Read in any problem specific data that is constant */
  read_problem_dat(0, &comm);
  /* Title for first set of output */
  printf("   Time  ");
  for (i = 0; i < (11 * mx - 16) / 2; i++)
    putchar(' ');
  printf("Estimate of State\n ");
  for (i = 0; i < 7 + 11 * mx; i++)
    putchar('-');
  printf("\n");

  /* Loop over each time point */
  for (t = 0; t < ntime; t++) {

    /* Read in any problem specific data that is time dependent */
    read_problem_dat(t + 1, &comm);

    /* Read in the observed data for time t */
    for (i = 0; i < my; i++) {
#ifdef _WIN32
      scanf_s("%lf", &y[i]);
#else
      scanf("%lf", &y[i]);
#endif
    }
#ifdef _WIN32
    scanf_s("%*[^\n] ");
#else
    scanf("%*[^\n] ");
#endif

    /* Call Unscented Kalman Filter routine (g13ekc) */
    nag_kalman_unscented_state(mx, my, y, lx, ly, f, h, x, st, &comm, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_kalman_unscented_state (g13ekc).\n%s\n",
             fail.message);
      exit_status = 1;
      goto END;
    }

    /* Display the some of the current state estimate */
    printf(" %3" NAG_IFMT "    ", t + 1);
    for (i = 0; i < mx; i++) {
      printf(" %10.3f", x[i]);
    }
    printf("\n");
  }

  printf("\n");
  printf("Estimate of Cholesky Factorization of the State\n");
  printf("Covariance Matrix at the Last Time Point\n");
  for (i = 0; i < mx; i++) {
    for (j = 0; j <= i; j++) {
      printf(" %10.2e", ST(i, j));
    }
    printf("\n");
  }

END:

  /* clean up any memory allocated in comm */
  read_problem_dat(-1, &comm);

END2:
  NAG_FREE(lx);
  NAG_FREE(ly);
  NAG_FREE(st);
  NAG_FREE(x);
  NAG_FREE(y);

  return (exit_status);
}
```

```
static void NAG_CALL f(Integer mx, Integer n, const double *xt,
                       double *fxt, Nag_Comm *comm, Integer *info)
{
  double t1, t3;
  Integer i;
  g13_problem_data *pdat;

  /* Cast the void point in comm back to point at the data structure */
  pdat = (g13_problem_data *) comm->p;

  t1 = 0.5 * pdat->r * (pdat->phi_rt + pdat->phi_lt);
  t3 = (pdat->r / pdat->d) * (pdat->phi_rt - pdat->phi_lt);

  for (i = 0; i < n; i++) {
    FXT(0, i) = XT(0, i) + cos(XT(2, i)) * t1;
    FXT(1, i) = XT(1, i) + sin(XT(2, i)) * t1;
    FXT(2, i) = XT(2, i) + t3;
  }
  /* Set info nonzero to terminate execution for any reason. */
  *info = 0;
}

static void NAG_CALL h(Integer mx, Integer my, Integer n, const double *yt,
                       double *hyt, Nag_Comm *comm, Integer *info)
{
  Integer i;
  g13_problem_data *pdat;

  /* Cast the void point in comm back to point at the data structure */
  pdat = (g13_problem_data *) comm->p;

  for (i = 0; i < n; i++) {
    HYT(0, i) =
           pdat->delta - YT(0, i) * cos(pdat->a) - YT(1, i) * sin(pdat->a);
    HYT(1, i) = YT(2, i) - pdat->a;

    /* Make sure that the theta is in the same range as the observed data,
       which in this case is [0, 2*pi) */
    if (HYT(1, i) < 0.0)
      HYT(1, i) += 2 * X01AAC;
  }
  /* Set info nonzero to terminate execution for any reason. */
  *info = 0;
}

static void read_problem_dat(Integer t, Nag_Comm *comm)
{
  /* Read in any data specific to the f and h functions */
  Integer tt;
  g13_problem_data *pdat;

  if (t == 0) {
    /* Allocate some memory to hold the data */
    pdat = NAG_ALLOC(1, g13_problem_data);

    /* Read in the data that is constant across all time points */
#ifdef _WIN32
    scanf_s("%lf%lf%lf%lf%*[^\n] ", &(pdat->r), &(pdat->d), &(pdat->delta),
            &(pdat->a));
#else
    scanf("%lf%lf%lf%lf%*[^\n] ", &(pdat->r), &(pdat->d), &(pdat->delta),
          &(pdat->a));
#endif
    /* Set the void pointer in comm to point to the data structure */
    comm->p = (void *) pdat;

  }
  else if (t > 0) {
    /* Cast the void point in comm back to point at the data structure */
    pdat = (g13_problem_data *) comm->p;
    /* Read in data for time point t */
#ifdef _WIN32
    scanf_s("%" NAG_IFMT "%lf%lf%*[^\n] ", &tt, &(pdat->phi_rt),
            &(pdat->phi_lt));
#else
    scanf("%" NAG_IFMT "%lf%lf%*[^\n] ", &tt, &(pdat->phi_rt),
          &(pdat->phi_lt));
#endif
    if (tt != t) {
      /* Sanity check */
      printf("Expected to read in data for time point %" NAG_IFMT "\n", t);
      printf("Data that was read in was for time point %" NAG_IFMT "\n", tt);
    }

  }
  else {
    /* Clean up */

    /* Cast the void point in comm back to point at the data structure */
    pdat = (g13_problem_data *) comm->p;

    if (pdat)
      NAG_FREE(pdat);
  }
}
```

10.2 Program Data

```
nag_kalman_unscented_state (g13ekc) Example Program Data
0.1
0.0 0.1
0.0 0.0 0.1           :: End of lx
0.01
0.0 0.01              :: End of ly
0.0 0.0 0.0           :: Initial value for x
0.1
0.0 0.1
0.0 0.0 0.1           :: End of initial value for st
15                    :: Number of time points
3.0 4.0 5.814  0.464  :: r, d, Delta, A
 1    0.4    0.1
      5.262  5.923
 2    0.4    0.1
      4.347  5.783
 3    0.4    0.1
      3.818  6.181
 4    0.4    0.1
      2.706  0.085
 5    0.4    0.1
      1.878  0.442
 6    0.4    0.1
      0.684  0.836
 7    0.4    0.1
      0.752  1.300
 8    0.4    0.1
      0.464  1.700
 9    0.4    0.1
      0.597  1.781
10    0.4    0.1
      0.842  2.040
11    0.4    0.1
      1.412  2.286
12    0.4    0.1
      1.527  2.820
13    0.4    0.1
      2.399  3.147
14    0.4    0.1
      2.661  3.569
15    0.4    0.1
      3.327  3.659    :: t, phi_rt, phi_lt, y = (delta_t, alpha_a)
```

10.3 Program Results

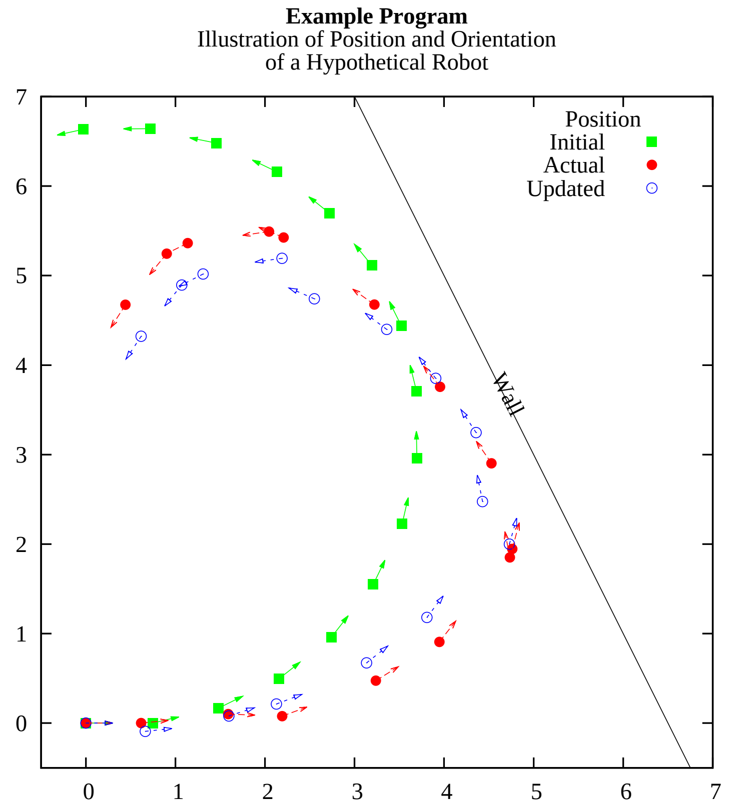
```
nag_kalman_unscented_state (g13ekc) Example Program Results

   Time          Estimate of State
 ---------------------------------------
   1          0.664     -0.092      0.104
   2          1.598      0.081      0.314
   3          2.128      0.213      0.378
   4          3.134      0.674      0.660
   5          3.809      1.181      0.906
   6          4.730      2.000      1.298
   7          4.429      2.474      1.762
   8          4.357      3.246      2.162
   9          3.907      3.852      2.246
  10          3.360      4.398      2.504
  11          2.552      4.741      2.750
  12          2.191      5.193      3.281
  13          1.309      5.018      3.610
  14          1.071      4.894      4.031
  15          0.618      4.322      4.124

Estimate of Cholesky Factorization of the State
Covariance Matrix at the Last Time Point
   1.92e-01
  -3.82e-01   2.22e-02
   1.58e-06   2.23e-07   9.95e-03
```

The example described above can be thought of relating to the movement of a hypothetical robot. The unknown state, x, is the position of the robot (with respect to a reference frame) and facing, with (ξ, η) giving the x and y coordinates and θ the angle (with respect to the x-axis) that the robot is facing. The robot has two drive wheels, of radius r on an axle of length d. During time period t the right wheel is believed to rotate at a velocity of ϕ_{Rt} and the left at a velocity of ϕ_{Lt}. In this example, these velocities are fixed with $\phi_{Rt} = 0.4$ and $\phi_{Lt} = 0.1$. The state update function, F, calculates where the robot should be at each time point, given its previous position. However, in reality, there is some random fluctuation in the velocity of the wheels, for example, due to slippage. Therefore the actual position of the robot and the position given by equation F will differ.

In the area that the robot is moving there is a single wall. The position of the wall is known and defined by its distance, Δ, from the origin and its angle, A, from the x-axis. The robot has a sensor that is able to measure y, with δ being the distance to the wall and α the angle to the wall. The measurement function H gives the expected distance and angle to the wall if the robot's position is given by x_t. Therefore the state space model allows the robot to incorporate the sensor information to update the estimate of its position.


Example Program
Illustration of Position and Orientation
of a Hypothetical Robot
Position
Initial
Actual
Updated
Wall