KALMTOOL

for Use with MATLAB

State Estimation for Nonlinear Systems

Technical Report IMM-REP-2000-6

Magnus Nørgaard
Department of Mathematical Modelling & Department of Automation
Technical University of Denmark

April 25, 2000

RELEASE NOTES

This note contains important information on how the present toolbox is to be installed, and the conditions under which it may be used. Please read it carefully before use.

Before You Start Using the Toolbox

After installation, all toolbox functions will be located in one directory, and the files associated with the demonstration programs will be located in a subdirectory under this. The user should make a path to these directories with the MATLAB function **path**:

```
>> path(path,'/xx/.../xx/Kalmtool');
>> path(path,'/xx/.../xx/Kalmtool/Demo');
```

If the toolbox is going to be used on a regular basis it is recommended to include this statement in the file *startup.m*, which is invoked during the start of MATLAB.

If a C compiler is available, and the MATLAB **mex** command has been properly set up, the CMEX functions in the demonstration directory are compiled as follows:

```
>> cd /xx/xx/.../Kalmtool/Demo
>> demomex
```

This script invokes the **mex** command for each of the CMEX functions in the demonstration directory.

Conditions/Disclaimer

By using the toolbox the user agrees to all of the following:

Magnus Nørgaard
Department of Mathematical Modelling
Bldg. 321
Technical University of Denmark
DK-2800 Kgs. Lyngby
Denmark
pmn@iau.dtu.dk

Chapter 1

Introduction

This manual is a user's guide for the KALMTOOL toolbox; a MATLAB toolbox containing functions for state estimation for nonlinear systems. The toolbox contains the well-known *Extended Kalman Filter (EKF)* and two new filters called the *DD1 filter* and the *DD2 filter*.

The toolbox will run under MATLAB 5.3 and higher, and it is independent of other MATLAB toolboxes. All functions exist as *m-files* but for faster execution of the computationally intensive functions these have also been written in C and are provided as *CMEX*-functions. The user will have to add some problem specific C code and compile the functions with the **mex** command in order to run the CMEX-functions.

Generally, it will require only superficial knowledge about the C programming language to work with the MEX functions. However, a C compiler must of course be installed on the system, and the **mex** command in MATLAB properly set up (see the Application Interface Guide for MATLAB).

1.1 Why this Toolbox?

The purpose of this toolbox is to make implementations of the new DD1 and DD2 filters available for solving nonlinear state estimation problems and to enable a comparison with a conventional method like the extended Kalman filter.

So what's the deal with these filters? If you are familiar with the EKF, or one of its 'relatives', you will quickly find out about the advantage of the new filters. First of all, they are easier to use. Secondly, you can expect a similar (DD1) or better (DD2) performance than with the EKF. On the downside, the new filers tend to require more computations than the EKF.

It should be mentioned that the filters are implemented in a *very* general fashion. In most practical application this generality is not needed but since it is provided one should be aware of the fact that faster execution times could be obtained with more specialized software. In particular in the case of the DD1 and DD2 filters one will encounter excessive computation times for "simple" applications.

What exactly is different about the new filters? To make a short story long, we found that the extended Kalman filter was somewhat inconvenient to use in some of our applications. A small modification of the application sometimes had serious implications on the EKF implementation. Moreover, it was often difficult to implement. Our problem was that the EKF requires a linearization of the system model. Sometimes this is easy to find but sometimes it can be pretty hard. In any case, it makes things inflexible. If a small change is made in the model one has to work out a new set of derivatives. This is particularly inconvenient in model calibration where certain model parameters are temporarily included in the state vector and estimated simultaneously with the actual states.

So where can I read about these filters? So far three publications are available.

The short introduction:
M. Nørgaard, N.K. Poulsen, O. Ravn: *Easy and Accurate State Estimation for Nonlinear Systems*, 14th IFAC World Conference in Beijing, China, July 5-9, 1999, pp. 343–348.

An expanded version:
M. Nørgaard, N.K. Poulsen, O. Ravn: *New Developments in State Estimation for Nonlinear Systems*, to be published in Automatica.

The most thorough description:
M. Nørgaard, N.K. Poulsen, O. Ravn: *Advances in Derivative-Free State Estimation for Nonlinear Systems*, Technical Report IMM-REP-1998-15, Department of Mathematical Modelling, DTU, 1998 (revised Apr. 2000).

Chapter 2

User's Guide

2.1 What You Need to Figure Out First

In order to use filter routines you need the following prerequisites:

A state space model in the form:

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, v_k) \\ y_k &= g(x_k, w_k) \end{aligned}$$

The noise covariance matrices

$$\begin{aligned} Q &= \mathbf{E}\left\{v_k v_k^T\right\} \\ R &= \mathbf{E}\left\{w_k w_k^T\right\} \end{aligned}$$

Initial estimates of state and covariance matrix

$$\begin{aligned} \bar{x}_0 &= \mathbf{E}\{x_0\} \\ P_0 &= \mathbf{E}\left\{(x_0 - \bar{x}_0)(x_0 - \bar{x}_0)^T\right\} \end{aligned}$$

Input-output data sets

$$\begin{aligned} y &= \{y_0, y_1, y_2, \ldots\} \\ u &= \{u_0, u_1, u_2, \ldots\} \end{aligned}$$

For application of the extended Kalman filter you must also derive the linearized state and observation equations:

$$\begin{aligned} x_{k+1} &\approx f(\hat{x}_k, u_k, \bar{v}_k) + A(k)(x_k - \hat{x}_k) + F(k)(v_k - \bar{v}_k) \\ y_k &\approx g(\bar{x}_k, \bar{w}_k) + C(k)(x_k - \bar{x}_k) + G(k)(w_k - \bar{w}_k) \end{aligned}$$

where

$$\begin{aligned} A(k) &= \left.\frac{\partial f(x, u_k, \bar{v}_k)}{\partial x}\right|_{x=\hat{x}_k} \qquad F(k) = \left.\frac{\partial f(\hat{x}_k, u_k, v)}{\partial v}\right|_{v=\bar{v}_k} \\ C(k) &= \left.\frac{\partial g(x, \bar{w}_k)}{\partial x}\right|_{x=\bar{x}_k} \qquad G(k) = \left.\frac{\partial g(\bar{x}_k, w)}{\partial w}\right|_{w=\bar{w}_k}. \end{aligned}$$

When you have collected all this information you must specify the model in MATLAB functions. These functions must conform to a particular structure, which is discussed below

2.2 Writing the Equations in M-Functions

If you wish to run the DD1 or DD2 filter you must write two functions. One should contain the state equation and the other should contain the output equation (if you are working with more than one observation stream, you must write a function for each stream but more about that later). If you are going to use the EKF, it is necessary to write an additional function that specifies the linearization of the two equations.

2.2.1 The state equation

As an example, let us implement the state equation in a function called **myxfunc**. The necessary components are shown below (for a nonsense system!).

```
function xout=myxfunc(x,u,v)

% Make variables static
persistent mypar1 mypar2;

% Check if variables should be initialized
if nargin==1,
   mypar1 = x(1)*0.5 + x(2);
   mypar2 = 75*x(3);
   return
end

% A priori update of states
xout    = zeros(3,1);
```

```
xout(1) = x(1) + mypar2*cos(x(2)+u(1)*v(1));
xout(2) = x(3) + mypar2*cos(x(2)+u(2)*v(2));
xout(3) = mypar1*x(1) + v(3);
```

Dissection of the function

The header must always look like this: `function xout=myxfunc(x,u,v)`

The function and variable names are unimportant, but the function must always take 3 arguments and return one output. The arguments, which should be (column) vectors, are the current state estimate, control input, and process noise (in that order). The function should output the *a priori* state update. Argument 2 and 3 *must* be present even if there are no inputs or process noise.

By using the `persistent` declaration, a feature which was introduced in MATLAB 5.2, it is possible to maintain parameters from one function call to another. This is convenient as one can initialize certain parameters before the filtering.

There should always be an initialization section in the function. This must take the form

```
if nargin==1,
   .. do initialization stuff
   return
end
```

Before the actual filtering is performed, `myxfunc` will be invoked as `myxfunc(opt.init)`. Thus, by specifying parameter initializations in the variable `opt.init`, these parameters are passed to the function through the argument `x`. The initialization section must be included even if there are no such initializations. In this case it should just contain the `return` statement.

The last part of the function is the actual state update. If `x` is not also used as the return variable, make sure that the returned variable is a column vector.

2.2.2 The output equation

The output equation is written in an m-function with a similar format:

```
function y=myyfunc(x,w)
% Make variables static
persistent mypar3

% Check if variables should be initialized
if nargin==1
   mypar3=x(4);
   return
end

% Calculate output estimate
y = mypar3*x.*x+w;
```

The function should take two arguments: the state vector and the measurement noise vector. Apart from that, the function has the same structure as *xfunc*.

Linearization of the equations

When using the EKF it is necessary to write a separate function that contains the linearization of the two equations. The function must have the following format (the contents is still nonsense!)

```
function [M,N]=mylinfunc(x,u,vw,flag)
% Make variables static
persistent mypar1 mypar2;

% Check if variables should be initialized
if nargin==1,
   mypar1 = x(1)*0.5 + x(2);
   mypar2 = 75*x(3);
   A0 = diag([1 1]);
   F0 = zeros(2,2);
   C0 = [1 0];
   G0 = [];
   return
end
```

```
% Linearize state equation
if flag==0,
  M = A0;
  M(1,2) = mypar1*(sin(x(1)*u(1)) + x(2)*u(2));

  N = F0;
  N(1,1) = mypar2/x(2);
  N(2,2) = x(1)*x(2);

% Linearize output equation
elseif flag==1,
   M=C0;
   N=G0;
end
```

Dissection of the function

The header must always look like this: `function [M,N]=my_linfunc(x,u,vw,flag)`

The function and variable names are unimportant, but the function must always take 4 input arguments and return 2 outputs. The arguments, three (column) vectors and an integer, are the current state estimate, control input, process noise or measurement noise, and a flag (in that order). If `flag=0` then `vw` is process noise and the function should linearize the state equation and return the matrix A in `M` and the matrix F in `N`. If `flag=1` then `vw` is measurement noise and the function should linearize the output equation and return the C matrix in `M` and the G matrix in `N`.

Argument 2 and 3 *must* be present even if there are no inputs or no process/measurement noise.

2.3 Running the Filters

When input and observation data are available, and the appropriate m-functions have been written, it is straightforward to run the filters. The DD2 filter is invoked in the following way:

```
[xhat,Smat]=dd2('myxfunc','myyfunc',x0,P0,Q,R,u,y,tidx);
```

u is a matrix of inputs to the system. The first row contains the inputs applied at time k=0, the second row contains the inputs at time k=1, etc. If the system has no inputs, the empty matrix [ ] is passed.

y is a matrix of observations and tidx is a vector with time stamps for the observations in y. The first row in y contains the observations acquired at the sample number specified in the first element of tidx, the second row in y contains the observations acquired at the sample number specified in the second element of tidx, etc. If observations are available at all sampling instants, we simply have that tidx=0:size(u,1).

The state estimates are returned in the matrix xhat. The first row in xhat is the estimate at time k=0. If there is no observation update at that time, it will simply be the argument x0 that was passed to the filter function upon the call.

The matrix Smat is a matrix containing coefficients of the Cholesky factors of the estimation error covariance matrices. The format of this matrix will be further explained in the following section.

The DD1 filter is called in the exact same way. For the extended Kalman filter it is necessary to include the name of your m-function that performs the linearizations:

```
[xhat,Pmat]=ekf('myxfunc','myyfunc','mylinfunc',x0,P0,Q,R,u,y,tidx)
```

The coefficients of the covariance matrices (and not their Cholesky factor) are returned in the matrix Pmat. The format of this matrix will also be explained in the following section.

If the noise processes have mean values different from 0, these can of course be subtracted at the appropriate places in the functions myxfunc and myyfunc. They can also be passed to the filter functions through the optional data structure 'optpar'. For example,

```
[xhat,Smat]=dd2('myxfunc','myyfunc',x0,P0,Q,R,u,y,tidx,optpar);
```

optpar contains up to three fields: optpar.vmean is the mean of process noise vector and optpar.wmean is the mean of measurement noise vector. optpar.init can be included if there are any initial parameters for the user-written m-functions. optpar.init can be either a vector or a matrix. Upon call of the filter functions, `optpar.init` will be passed as the first argument to the user's functions in the initialization process prior to the actual filtering, e.g., `myxfunc(opt.init)`

2.4 Analyzing the Results

The performance of the filters can be evaluated with the function `kalmeval`. This function calculates the output estimates and compares these to the actual observations. Additionally, it plots the state estimates along with three times their (estimated) standard deviations. For the DD2 filter, the function is called as follows:

```
[yhat,RMS]=kalmeval('dd2','myyfunc',R,xhat,Smat,y,tidx,optpar)
```

and similarly for the DD1 filter. `RMS` is the RMS error between observations and predictions.

To evaluate the extended Kalman filter, the function is called as follows (notice that `Pmat` is passed instead of `Smat`):

```
[yhat,RMS]=kalmeval('ekf','myyfunc',R,xhat,Pmat,y,tidx,optpar)
```

If you would like to take a closer look at the covariance estimates, four functions are available to accommodate this. Two of the functions are used together with the DD1 and DD2 filters. These filters works on Cholesky factors of the covariance matrices and not on the covariance matrices themselves. If the covariance matrix is denoted P, the Cholesky factor, S, is an upper triangular matrix with the property

$$P = SS^T$$

For sample no. k, the elements of $S(k)$ are stored in row $k+1$ of the matrix, `Smat`:

$$S(k) = \begin{bmatrix} S_{11} & S_{12} & S_{13} \\ 0 & S_{22} & S_{23} \\ 0 & 0 & S_{33} \end{bmatrix} \quad \rightarrow \quad Smat = \begin{bmatrix} & \vdots & & & \vdots & \\ S_{11} & S_{12} & S_{13} & S_{22} & S_{23} & S_{33} \\ & \vdots & & & \vdots & \end{bmatrix}$$

With the function `smat2cov` it is possible to extract the covariance matrix at a specific sample number:

```
P = smat2cov(Smat,k+1)
```

extracts the covariance matrix at time k (recall that the estimates for `k=0` are stored in the first row).

The function `smat2var` uses `Smat` to calculate the variance of each state estimate, i.e., the diagonal of the covariance matrix:

```
V = smat2var(Smat)
```

The first column contains the variance estimates for the first state, etc.

To extract the covariances estimated by the extended Kalman filter, two similar functions are available. `mat2cov` takes `Pmat` as input and returns the covariance matrix at a specified sample number. The format of `Pmat` is shown below. If $P(k)$ is the covariance matrix at time k then row $k+1$ of `Pmat` is organized as follows:

$$P(k) = \begin{bmatrix} P_{11} & P_{12} & P_{13} \\ P_{12} & P_{22} & P_{23} \\ P_{13} & P_{23} & P_{33} \end{bmatrix} \quad \rightarrow \quad Pmat = \begin{bmatrix} & \vdots & & & \vdots & \\ P_{11} & P_{12} & P_{13} & P_{22} & P_{23} & P_{33} \\ & \vdots & & & \vdots & \end{bmatrix}$$

$P(k)$ is extracted by

```
P = mat2cov(Pmat,k+1)
```

The function `mat2var` works the same way as `smat2var` except that it takes `Pmat` as input.

2.5 How to Handle Multiple Observation Streams

In some applications one might receive the observations from different sensors, and it is often reasonable to assume that these are independent. In this case we can talk about *multiple observation streams*. If all observations are available at every sampling instant the regular filters can be used. If the observations in the different streams do not occur simultaneously, you should instead use the special versions of the filters that have been designed for this specific application. It is now assumed that the model has the form:

$$\begin{aligned} x_{k+1} &= f\left(x_k, u_k, v_k\right) \\ y_k^{(1)} &= g^{(1)}\left(x_k, w_k^{(1)}\right) \\ &\vdots \\ y_k^{(n)} &= g^{(n)}\left(x_k, w_k^{(n)}\right) \end{aligned}$$

An m-function for *each* output equation must be written, conforming to the format discussed previously. The call of the filter function (in this case the DD2 filter) looks pretty much the same as before. If there are two observation streams the call is

```
[xhat,Smat]=dd2m('myxfunc',{'myyfunc1','myyfunc2'},x0,P0,Q,{R1,R2},...
                                              u,{y,y2},{tidx1,tidx2})
```

The so-called *cell structure* is used two wrap the multiple instances of function names, measurement noise covariances, observation matrices, and time stamp vectors into common arguments.

Similar extensions are available for the DD1 filter (`dd1m`) and the EKF (`ekfm`).

Chapter 3

Working with the MEX-files

With a little extra effort one can experience a tremendous increase in execution speed by using the MEX alternatives to the m-functions. It requires some knowledge about the C programming language, but typically a superficial knowledge will be enough. The MEX alternative is only available for the DD1 and DD2 filters as the increase in execution speed is particularly pronounced for these.

Whereas before one had to write m-functions containing state and output equations, in the MEX case it is necessary to write similar functions in C. As a prototyping stage it is always a good idea to start out in MATLAB. When things are working here you can "translate" the m-functions into C and use the MEX functions. Although it seems like having to do the work twice, overall it might save you time as in MATLAB it is relatively easy to make things right the first time. The MATLAB solution can then be used for debugging the C implementation.

It is recommended to complement the description provided in the following with a look in the demonstration functions located in the "Demo" subdirectory.

3.1 A Few Details You Should Know

The MEX files operate on a special matrix format for storage of vectors and matrices. In the C code a "matrix" will be pointer to a data structure. A new matrix is declared by:

```
matrix *M;
```

and later memory for the matrix is allocated with

```
M = mmake(rows,columns);
```

The data structure contains three variables: number of rows (`M->row`), number of columns (`M->col`) and a pointer to an array that contains pointers to the memory locations where each row is stored (the content of `M->mat[0]` points to the first element in the matrix).

Now things start sounding a little technical, but you do not really need to understand this completely. A number of C macros and functions are available to assist you when operating on matrices (and vectors):

macros	functionality
`rows = nof_rows(M);`	Number of rows in matrix.
`columns = nof_cols(M);`	Number of columns in matrix.
`len = vec_len(V);`	Length of vector (matrix with one row or column).
`value = get_val(M,r,c);`	Get element (r, c) from matrix.
`put_val(M,r,c,value);`	Insert 'value' in element (r, c).
`value = cvget(V,r)`	Get the rth element from a column vector.
`value = rvget(V,c)`	Get the cth element from a row vector.

function name	functionality
`M = mmake(rows,columns);`	Allocate matrix of the specified size.
`mfree(M);`	Free the memory allocated to the matrix.
`mprint(M);`	Display the matrix.
`minit(M);`	Initialize all matrix elements to 0.
`madd(A,B,C);`	Add two matrices, $A = B + C$.
`mset(A,B);`	Copy a matrix, $A = B$.

3.2 Step 1: Writing State and Output Equations in C

Like in the MATLAB case you must write separate functions for state equation and observation equation(s). Let us first take a look at the format for the state equation function.

3.2.1 State Equation

```
/* Function prototype */
```

```
 int myxfunc(matrix*, matrix*, matrix*, matrix*, int);

/* The state equation function */
 int myxfunc(matrix *xbar, matrix *xhat, matrix *u, matrix *v, int flag)
 {
   /* Variable declarations */
   int a, b, c;
   double d, e, f;
   static int h, i, j;
   static matrix *M, *N;

   /* Initializations */
   if (flag == -1){
      ... Initialize static variables
      return 0;

   /* Clean up */
   else if (flag == -2){
      ... free matrices that you might have allocated
      return 0;
   }

   /* Normal call of function */
   else{
      ... Perform state update (insert in xbar)
      return 0;
   }
}
```

Dissection of the function

Except for the function and variable names, which can be arbitrary, the function call must have exactly the above format. `xbar` is the *a priori* state estimate; i.e., the output of the function. `xhat` is the previous state estimate, `u` is the input (it must be included in the argument list, but it does not have to be used), and `v` is the process noise. All four arguments to the function are column vectors; i.e., matrices with one column.

The last argument, `flag`, is used for specifying whether the function is called in “initialization mode”, in “clean up mode”, or in “filtering mode”. The modes are explained below:

Initialization mode
The function is called once in this mode prior to the filtering. It is included because often it is useful to remember certain parameters, vectors, matrices, etc, from one call to another rather than having to recalculate them at every sample. In particular, this section is used for vector/matrix allocations (call of the `mmake` function). As `mmake` performs a dynamic memory allocation this is not something one should carry out at every sample. Variables that are to be remembered from call to call should be declared "static".

If you wish to pass certain parameters to the function upon filtering (in contrast to hard coding all the information), the filter function has an optional argument, which can be used for passing such parameters. This argument must be a matrix, and it will be passed to the function through the first argument (`xbar`) during initialization.

Clean up mode
The function will be called in this mode just before termination of the filter function. If you have dynamically allocated memory for matrices or arrays, this is place to free the memory. In principle MATLAB will do this for you, but in the manual they recommend that you do it yourself as this will be faster. Matrices allocated with `mmake` are deallocated with the `mfree` command.

Filtering mode
In this section of the function the actual state update is placed. The updated states are placed in the first vector in the argument list (`xbar`).

3.2.2 Observation Equation

The observation equation has a similar format:

```
/* Function prototype */
 int myyfunc(matrix*, matrix*, matrix*, int);

/* The state equation function */
 int myyfunc(matrix *ybar, matrix *xbar, matrix *w, int flag)
 {
   /* Variable declarations */
```

```
    /* Initializations */
    if (flag == -1){
       ... Initialize static variables
       return 0;

    /* Clean up */
    else if (flag == -2){
       ... free matrices that you might have allocated
       return 0;
    }

    /* Normal call of function */
    else{
       ... Calculate output estimate (insert in ybar)
       return 0;
    }
}
```

For applications with multiple observation streams the output function must be augmented to handle this. This is different from the MATLAB case where a separate function was written for each output stream.

```
    /* Normal call of function */
    else if (flag == 0){
       ... Calculate output estimate for stream 1
       return 0;
    }
    else if (flag == 1){
       ... Calculate output estimate for stream 2
       return 0;
    }
```

3.3 Step 2: Modifying the Template and Compile the Function

Before preparing the actual filtering you might want to check that the functions will in fact produce the expected results. To do this you must first place the functions described above in the same file (place both prototype declarations in the top). Next you must copy the test template file xytest.c located in the Kalmtool directory to a new name, e.g., my_xytest.c. Open the file in an editor. In the top of the file the are three "define" statements: KALMFILE, XFUNC, and YFUNC. After KALMFILE you write the name of the file containing your functions, after XFUNC you write the name of the state equation function, and after YFUNC you write the name of the output equation function:

```
#define KALMFILE "myfile.c"
#define XFUNC myxfunc
#define YFUNC myyfunc
```

The file is compiled by issuing the MATLAB command **mex**. The compilation depends on the platform you are working on and the compiler available:

```
>> mex my_xytest.c kalmlbbc.obj % PC/Windows, Borland (5.2)
>> mex my_xytest.c kalmlbwc.obj % PC/Windows, Watcom (11.0)
>> mex my_xytest.c kalmlbms.obj % PC/Windows, Microsoft (VS 6.0)
>> mex my_xytest.c kalmlblx.o   % PC/Linux
>> mex my_xytest.c kalmlbhp.o   % HP 9000/735
```

If the function compiled without problems you can now evaluate your functions with the statement:

```
>> [yout,xout] = my_xytest(x,u,ny,v,w,init);
```

x is a state vector, u is an input, ny is the number of outputs (the dimension of the output vector), v is a process noise vector, and w is an observation noise vector. init is a vector or matrix with possible parameters you might want to pass to the function for initialization purposes. Use [ ] if you do not wish to pass anything.

If your functions behave correctly your next move is to select a filter function. Four templates are available: dd1c.c dd1mc.c, dd2c.c, and dd2mc.c. Make a copy of the template corresponding to the filter function you wish to use, open the file in an editor and modify the define statements in the top of the file as explained above. The filter function can now be compiled:

```
>> mex myddfilter.c kalmlbbc.obj % PC/Windows, Borland (5.2)
>> mex myddfilter.c kalmlbwc.obj % PC/Windows, Watcom (11.0)
>> mex myddfilter.c kalmlbms.obj % PC/Windows, Microsoft (VS 6.0)
>> mex myddfilter.c kalmlblx.o   % PC/Linux
>> mex myddfilter.c kalmlbhp.o   % HP 9000/735
```

3.4 Step 3: Calling the Filter Routines

The MEX function is now ready to be called from MATLAB. The call is *almost* the same as when you call the m-function counterpart. The main difference is that you should not call the function with the covariance matrices, but with a root, S, for which $P = SS^T$ (not necessarily a Cholesky factor).

```
>> Sv = chol(Q)';
>> Sw = sqrtm(R);
>> [v,d] = eig(P0);
>> Sx0   = real(v*sqrt(d));

>> [xhat,Smat]=myddfilter(x0,Sx0,Sv,Sw,u,y,tidx,optidx);
```

For illustration purposes three different ways to factorize the covariance matrices have been shown above. You can use any which one you prefer.

To evaluate the result of the filtering you can use the MATLAB functions described previously.

Chapter 4

Two Examples

Finally it's time for some good clean family entertainment you can trust... Two demonstration examples are provided to illustrate how to work with the filters. In the first example the filters are applied to a real data set collected on an autonomous guided vehicle (AGV). The second example is a simulation study of a falling body; an often-used benchmark example for evaluation of nonlinear filter designs. Not all details will be given, and the user is encouraged to open the demos in an editor and take a look at their implementation. The implementations are by no means optimal; their purpose is to show the user different ways in which things can be implemented.

4.1 Pose Estimation and Calibration of an AGV

In this example we will consider a data set collected by an autonomous guided vehicle (a so-called AGV). The purpose is to estimate the position and orientation (*the pose*) of the vehicle relative to a pre-selected coordinate system in the room where the vehicle is located. The vehicle has three wheels. Two driving wheels in the front and a castor wheel in the back. The vehicle is equipped with wheel encoders for measuring the turning angle of each of the driving wheels, and a CCD-camera for detecting simple guide marks placed on the walls in the room. Moreover, a camera has been mounted to the ceiling and tracks two diodes placed on top of the vehicle.

We will work with a so-called *encoder* model of the vehicle. The encoder readings are used as inputs to the model, and the pose is the output:

$$x(k+1) \quad = \quad x(k) + s\cos\phi$$

$$
\begin{aligned}
y(k+1) &= y(k) + s\sin\phi \\
\theta(k+1) &= \theta(k) + 2t
\end{aligned}
$$

where

$$
\begin{aligned}
\phi &= \theta(k) + t \\
s &= \frac{\kappa}{2}\left[r_r u_1(k) + r_l u_2(k)\right] \\
t &= \frac{\kappa}{2B}\left[r_r u_1(k) - r_l u_2(k)\right]
\end{aligned}
$$

r_r and r_l are the wheel radii and B is the distance between the wheels. κ is a constant relating to the encoder gain (k_{enc} and the gear (N), $\kappa = \dfrac{1}{k_{enc}N}$.

The model is quite accurate but generally one will encounter a certain drift. The main reason for this is that wheel radii and distance between wheels are known only with a limited accuracy. Typically, the initial pose will not be known exactly either. As an absolute measure of the pose we therefore use the camera sensors. To demonstrate the regular filter functions only the camera in the ceiling is used. Subsequently these observations will be supplemented with the observations from the camera on board the vehicle to demonstrate how to handle multiple observation streams. The encoders are sampled with 40 msec intervals. The camera observations are not available with regular intervals as the time spent on image processing varies.

Because the camera in the ceiling has a relatively limited view, the data are recorded by controlling the vehicle with small jerks in a joystick.

We consider the joint state and parameter estimation problem. Along with the pose we estimate the wheel radii and distance between the wheels. The dimension of the state vector is therefore 6:

$$
\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \\ r_r \\ r_l \\ b \end{bmatrix} = \begin{bmatrix} \text{position i } x \text{ direction} \\ \text{position i } y \text{ direction} \\ \text{orientation} \\ \text{radius of right wheel} \\ \text{radius of left wheel} \\ \text{distance between wheels} \end{bmatrix}
$$

$$
x_1(k+1) = x_1(k) + s\cos\phi
$$

$$
\begin{aligned}
x_2(k+1) &= x_2(k) + s\sin\phi \\
x_3(k+1) &= x_3(k) + 2t \\
x_4(k+1) &= x_4(k) + v_3(k) \\
x_5(k+1) &= x_5(k) + v_4(k) \\
x_6(k+1) &= x_6(k) + v_5(k)
\end{aligned}
$$

where

$$
\begin{aligned}
\phi &= x_3(k) + t \\
s &= \frac{\kappa}{2}\left[x_4(k)\bar{u}_1(k) + x_5(k)\bar{u}_2(k)\right] \\
t &= \frac{\kappa}{2x_6(t)}\left[x_4(k)\bar{u}_1(k) - x_5(k)\bar{u}_2(k)\right] \\
\bar{u}_1(k) &= u_1(k) + v_1(k) \\
\bar{u}_2(k) &= u_2(k) + v_2(k)
\end{aligned}
$$

The image processing routine associated with the camera in the ceiling returns the pose; thus we have three observations:

$$
\begin{aligned}
y_1(k) &= x_1(k) + w_1(k) \\
y_2(k) &= x_2(k) + w_2(k) \\
y_3(k) &= x_3(k) + w_3(k)
\end{aligned}
$$

Open the file **agvdemo** located in the Demo subdirectory in an editor and take a look at the implementation and the calls of the different filter functions.

4.2 The Falling Body Benchmark Example

This famous benchmark has been considered in several publications. It will be a good idea to consult

M. Athans, R. P. Wishner, and A. B. Bertolini: *Suboptimal state estimation for continuous-time nonlinear systems from discrete noisy measurements.* IEEE Transactions on Automatic Control, AC-13(5):504–514, Oct. 1968.

for a description of the example. The fall of the body can be described by the two differential equations (4.1),(4.2). x_3 represents a ballistic coefficient

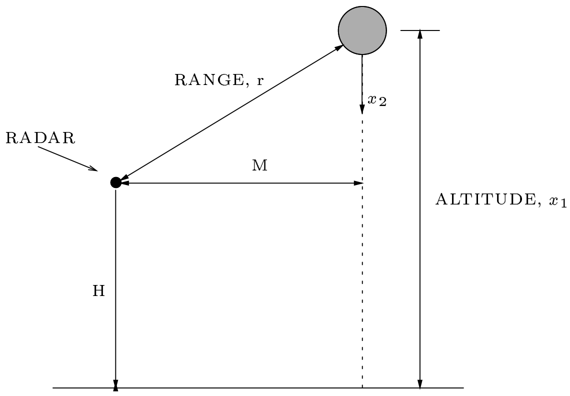

Figure 4.1: *Geometry of the vertically falling body problem.*

which we assume unknown and therefore wish to estimate simultaneously with the altitude and velocity of the body. The radar measures the range (r). The measurements appear with intervals of 1 second and are affected by additive white Gaussian noise.

The model is the following:

$$\dot{x}_1(t) = -x_2(t) \tag{4.1}$$
$$\dot{x}_2(t) = -e^{-\gamma x_1(t)} x_2(t)^2 x_3(t) \tag{4.2}$$
$$\dot{x}_3(t) = 0 \tag{4.3}$$
$$y_k = r_k + w_k = \sqrt{M^2 + (x_{1,k} - H)^2} + w_k \tag{4.4}$$

The model parameters are given by:

$$\begin{aligned} M &= 100,000 \text{ ft} \\ H &= 100,000 \text{ ft} \\ \gamma &= 5 \times 10^{-5} \\ E[w_k^2] &= 10^4 \text{ ft}^2 \end{aligned}$$

and the initial state of the system is

$$\begin{cases} x_{1,0} = 300,000 \text{ ft} \\ x_{2,0} = 20,000 \text{ ft/s} \\ x_{3,0} = 10^{-3} \end{cases}$$

Due to the nature of the problem it is common practice to employ a continuous-discrete filter implementation. The state equations (4.1)-(4.3) are integrated using a fourth order Runge-Kutta method with 64 steps taken between each observation. It is straightforward to implement continuous-discrete versions of the DD1 and DD2-filter as there is no process noise. The above mentioned paper describes the implementation of the EKF and the (modified Gaussian) second order filter for the considered application.

The following initialization of state estimates and covariance matrix is used:

$$\begin{cases} \hat{x}_{1,0} &= 300,000 \text{ ft} \\ \hat{x}_{2,0} &= 20,000 \text{ ft/s} \\ \hat{x}_{3,0} &= 3 \times 10^{-5} \end{cases}$$

$$\hat{P}(0) = \begin{bmatrix} 10^6 & 0 & 0 \\ 0 & 4 \times 10^6 & 0 \\ 0 & 0 & 10^{-4} \end{bmatrix}.$$

To enable a fair comparison of the estimates produced by each of the filters, the estimates are averaged across a Monte Carlo simulation consisting of 50 runs. Each run is carried out with a different noise sample.

Open the script **falldemo** in an editor to study the implementation and filter calls.

Chapter 5

Reference

Filter Functions	
dd1	DD1 filter.
dd1m	DD1 filter for systems with multiple observation streams.
dd2	DD2 filter.
dd2m	DD2 filter for systems with multiple observation streams.
ekf	Extended Kalman filter.
ekfm	Extended Kalman filter for systems with multiple observation streams.

MEX files	
dd1c	C-Mex counterpart to the 'dd1' function.
dd1mc	C-Mex counterpart to the 'dd1m' function.
dd2c	C-Mex counterpart to the 'dd2' function.
dd2mc	C-Mex counterpart to the 'dd2m' function.
kalmlb**	Object file that must be linked with the mex files.
xytest	Test C-functions before the filtering is performed.

Utilities	
kalmeval	Evaluate filter performance.
mat2cov	Extract covariance matrix from vector containing the upper triangular elements.
mat2var	Extract variance estimates from matrix containing covariance estimates.
smat2cov	restore covariance matrix from vector of Cholesky factor elements.
smat2var	Calculate variance estimate for each state from the Cholesky factored covariance matrices. covariance estimates.
triag	Triangularization with Householder transformation.

Demonstrations	
agvdemo	Position and orientation estimation and calibration of an AGV.
falldemo	Falling body example (a continuous-time example).
demomex	Generates MEX files to speed up the demonstrations.

dd1

Purpose

State estimation with the DD1 filter.

Synopsis

```
[xhat,Smat]=dd1(xfunc,yfunc,x0,P0,Q,R,u,y,tidx)
[xhat,Smat]=dd1(xfunc,yfunc,x0,P0,Q,R,u,y,tidx,optpar)
```

Description

dd1 uses the DD1 filter to estimate the states for a nonlinear system. The model of the system must be specified in the form:

$$\begin{aligned} x_{k+1} &= f\left(x_k, u_k, v_k\right) \\ y_k &= g\left(x_k, w_k\right) \end{aligned}$$

where x is the state vector, u is a possible input, and v and w are (white) noise sources. Each of the two equations must be written in an m-function.

The arguments to the **dd1** function are explained below:

`xfunc`	Name of file containing the state equation.
`yfunc`	Name of file containing the output equation.
`x0`	Initial state estimate.
`P0`	Initial covariance matrix (symmetric, nonnegative definite).
`Q,R`	Covariance matrices for v and w, respectively.
`u`	Input signal. Dimension is [samples × inputs]. Use [ ] if there is no input.
`y`	Output signal. Dimension is [observations × outputs].
`tidx`	Vector containing time stamps (in samples) for the observations in y.
`optpar`	Data structure containing initialization parameters (optional).

`optpar.vmean`:	Mean of process noise vector (default is 0).
`optpar.wmean`:	Mean of measurement noise vector (default is 0).
`optpar.init`:	Initial parameters for `xfunc`, `yfunc` (arbitrary format).

The function returns the state estimates and the Cholesky factored covariance matrices for the state estimation errors. If the final time is *k=samples*, `yhat` and `Smat` will have *samples+1* rows. The first row contains the initial estimates, i.e., the estimate at time *k=0*. The estimates will be *a posteriori* estimates at the sampling times for which an observation is available. At the remaining sampling times, the *a priori* estimates are provided.

dd1 works on Cholesky factors of the covariance matrices and not on the covariance matrices themselves. If the covariance matrix is denoted P, the Cholesky factor, S, is an upper triangular matrix with the property

$$P = SS^T$$

For sample no. k, the elements of $S(k)$ are stored in row $k+1$ of the returned matrix, `Smat`:

$$S(k) = \begin{bmatrix} S_{11} & S_{12} & S_{13} \\ 0 & S_{22} & S_{23} \\ 0 & 0 & S_{33} \end{bmatrix} \quad \rightarrow \quad Smat = \begin{bmatrix} & \vdots & & & \vdots & \\ S_{11} & S_{12} & S_{13} & S_{22} & S_{23} & S_{33} \\ & \vdots & & & \vdots & \end{bmatrix}$$

With the function **smat2cov** it is possible to extract the covariance matrix at a specified sample number.

The function **smat2var** uses `Smat` to calculate the variance of each state estimate, i.e., the diagonal of the covariance matrix.

In order to compare output estimates with observations of the output, use the function **kalmeval**.

dd1

How to write the m-functions

Each of the functions whose names are specified in `xfunc` and `yfunc` must have the appropriate structure. This is explained below:

xfunc

Assume `xfunc='myxfunc'`:

```
function xout=myxfunc(x,u,v)
% Make variables static
persistent mypar1 mypar2;

% Check if variables should be initialized
if nargin==1,
   mypar1 = x(1)*0.5 + x(2);
   mypar2 = 75*x(3);
   return
end

% A priori update of states
xout    = zeros(3,1);

xout(1) = x(1) + mypar2*cos(x(2)+u(1)*v(1));
xout(2) = x(3) + mypar2*cos(x(2)+u(2)*v(2));
xout(3) = mypar1*x(1) + v(3);
```

Dissection of the function

The header must always look like this: `function xout=myxfunc(x,u,v)`

The function and variable names are unimportant, but the function must always take 3 arguments and return one output. The arguments, which should be (column) vectors, are the current state estimate, control input, and process noise (in that order). The function should output the *a priori* state update. Argument 2 and 3 *must* be present even if there are no inputs or process noise.

By using the `persistent` declaration, a feature which was introduced in MATLAB 5.2, it is possible to maintain parameters from one call to another. This is convenient as one can initialize certain parameters before the filtering.

There should always be an initialization section in the function. This must take the form

```
if nargin==1,
   .. do initialization stuff
   return
end
```

Before the actual filtering is performed, `myxfunc` will be invoked as `myxfunc(opt.init)`. Thus, by specifying parameter initializations in `opt.init`, these parameters are passed to the function through the argument `x`. The section must be included even if there are no such intializations. In this case, it should just include the `return` statement.

The last part of the function is the actual state update. If `x` is not also used as the return variable, make sure that the returned variable is a column vector.

yfunc

The output equation is written in an m-function in a similar way:

```
function y=myyfunc(x,w)
% Make variables static
persistent mypar3

% Check if variables should be initialized
if nargin==1
   mypar3=x(4);
   return
end

% Calculate output estimate
y = mypar3*x.*x+w;
```

The function should take two arguments: the state vector and the measurement noise vector. Apart from that, the function has the same structure as *xfunc*.

dd1

Algorithm

The DD1 filter is based on first-order polynomial approximations of the nonlinear mappings. The approximations are derived by using a multivariable extension of Stirling's interpolation formula. The filter is described in:

M. Nørgaard, N.K. Poulsen, O. Ravn: *Easy and Accurate State Estimation for Nonlinear Systems*, 14th IFAC World Conference in Beijing, China, July 5-9, 1999, pp. 343–348.

and much more thoroughly in: M. Nørgaard, N.K. Poulsen, O. Ravn: *New Developments in State Estimation for Nonlinear Systems*, to be published in Automatica.

A similar, but slightly simpler, filter is described in:
Tor S. Schei: *A Finite-Difference Method for Linearization in Nonlinear Estimation Algorithms*, Automatica, Vol. 33, No. 11, 1997, pp. 2053–2058.

See Also

dd1m, **dd1c**

dd1m

Purpose

DD1 filtering for systems with multiple observation streams.

Synopsis

```
[xhat,Smat]=dd1m(xfunc,yfunc,x0,P0,Q,R,u,y,tidx)
[xhat,Smat]=dd1m(xfunc,yfunc,x0,P0,Q,R,u,y,tidx,optpar)
```

Description

dd1m uses the DD1 filter to estimate the states for a nonlinear system. The model of the system must be specified in the form:

$$
\begin{aligned}
x_{k+1} &= f\left(x_k, u_k, v_k\right) \\
y_k^{(1)} &= g^{(1)}\left(x_k, w_k^{(1)}\right) \\
&\vdots \\
y_k^{(n)} &= g^{(n)}\left(x_k, w_k^{(n)}\right)
\end{aligned}
$$

where x is the state vector, u is a possible input, and v and w are (white) noise sources. Each of the equations must be written in an m-function. The arguments to the **dd1m** function are explained below:

`xfunc`	Name of file containing the state equation.
`yfunc`	Cell structure containing in each cell the name of a file containing an output equation. Dimension is n
`x0`	Initial state estimate.
`P0`	Initial covariance matrix (symmetric, nonnegative definite).
`Q`	Covariance matrices for v.
`R`	Cell structure containing the covariance matrices for $w^{(1)} - w^{(n)}$.
`u`	Input signal. Dimension is [samples $\times$ inputs]. Use [ ] if there is no input.
`y`	Cell structure containing in each cell a matrix with output signals, $y^{(1)} - y^{(n)}$. Dimension of each cell is [observations-in-stream $\times$ outputs-in-stream].
`tidx`	Cell structure containing in each cell a vector of time stamps (in samples) for the corresponding observations.
`optpar`	Data structure containing initialization parameters (optional).

`optpar.vmean`:	Mean of process noise vector (default is 0).
`optpar.wmean`:	Cell structure with means of measurement noise vectors (def. 0).
`optpar.init`:	Initial parameters for the functions in `xfunc` and `yfunc`.

dd1m

The function returns the state estimates and the Cholesky factored covariance matrices for the state estimation errors. If the final time is *k=samples*, `yhat` and `Smat` will have *samples+1* rows. The first row contains the initial estimates, i.e., the estimate at time *k=0*. The estimates will be *a posteriori* estimates at the sampling times for which an observation is available. At the remaining sampling times, the *a priori* estimates are provided. See **dd1** for details on how to write the state and output equations in m-functions.

See Also

dd1, **dd1mc**

dd2

Purpose

State estimation with the DD2 filter.

Synopsis

```
[xhat,Smat]=dd2(xfunc,yfunc,x0,P0,Q,R,u,y,tidx)
[xhat,Smat]=dd2(xfunc,yfunc,x0,P0,Q,R,u,y,tidx,optpar)
```

Description

dd2 uses the DD2 filter to estimate the states for a nonlinear system. The model of the system must be specified in the form:

$$\begin{aligned} x_{k+1} &= f\left(x_k, u_k, v_k\right) \\ y_k &= g\left(x_k, w_k\right) \end{aligned}$$

where x is the state vector, u is a possible input, and v and w are (white) noise sources. Each of the two equations must be written in an m-function.

Read the section about the **dd1** function to see what the arguments for **dd2** are, and to see the format of the returned variables.

Algorithm

The DD2 filter is based on second-order polynomial approximations of the nonlinear mappings. The approximations are derived by using a multivariable extension of Stirling's interpolation formula. The filter is described in:

M. Nørgaard, N.K. Poulsen, O. Ravn: *Easy and Accurate State Estimation for Nonlinear Systems*, 14th IFAC World Conference in Beijing, China, July 5-9, 1999, pp. 343–348.

and much more thoroughly in: M. Nørgaard, N.K. Poulsen, O. Ravn: *New Developments in State Estimation for Nonlinear Systems*, to be published in Automatica.

See Also

dd1, **dd2m**, **dd2c**

dd2m

Purpose

DD2 filtering for systems with multiple observation streams.

Synopsis

```
[xhat,Smat]=dd2m(xfunc,yfunc,x0,P0,Q,R,u,y,tidx)
[xhat,Smat]=dd2m(xfunc,yfunc,x0,P0,Q,R,u,y,tidx,optpar)
```

Description

dd2m uses the DD2 filter to estimate the states for a nonlinear system. The model of the system must be specified in the form:

$$\begin{aligned} x_{k+1} &= f\left(x_k, u_k, v_k\right) \\ y_k^{(1)} &= g^{(1)}\left(x_k, w_k^{(1)}\right) \\ &\vdots \\ y_k^{(n)} &= g^{(n)}\left(x_k, w_k^{(n)}\right) \end{aligned}$$

where x is the state vector, u is a possible input, and v and w are (white) noise sources. Each of the equations must be written in an m-function. The arguments to the **dd2m** function are explained in the section covering the **dd1m** function.

See Also

dd1m, **dd2**, **dd2mc**

ekf

Purpose

State estimation with the extended Kalman filter (EKF).

Synopsis

```
[xhat,Pmat]=ekf(xfunc,yfunc,linfunc,x0,P0,Q,R,u,y,tidx)
[xhat,Pmat]=ekf(xfunc,yfunc,linfunc,x0,P0,Q,R,u,y,tidx,optpar)
```

Description

ekf uses the extended Kalman filter to estimate the states for a nonlinear system. The model of the system must be specified in the form:

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, v_k) \\ y_k &= g(x_k, w_k) \end{aligned}$$

where x is the state vector, u is a possible input, and v and w are (white) noise sources. Each of the two equations must be written in an m-function.

The arguments to the **ekf** function are explained below:

`xfunc`	Name of file containing the state equation.
`yfunc`	Name of file containing the output equation.
`linfunc`	Name of file containing the linearization procedures.
`x0`	Initial state estimate.
`P0`	Initial covariance matrix (symmetric, nonnegative definite).
`Q,R`	Covariance matrices for v and w, respectively.
`u`	Input signal. Dimension is [samples × inputs]. Use [ ] if there is no input.
`y`	Output signal. Dimension is [observations × outputs].
`tidx`	Vector containing time stamps (in samples) for the observations in `y`.
`optpar`	Data structure containing initialization parameters (optional).

`optpar.vmean`:	Mean of process noise vector (default is 0).
`optpar.wmean`:	Mean of measurement noise vector (default is 0).
`optpar.init`:	Initial parameters for `xfunc`, `yfunc` (arbitrary format).

The function returns the state estimates and covariance matrices for the state estimation errors. If the final time is *k=samples*, `yhat` and `Pmat` will have *samples+1* rows. The first row contains the initial estimates, i.e., the estimates at time *k=0.* The estimates will be *a posteriori* estimates at the sampling times for which an observation is available. At the remaining sampling times, the *a priori* estimates are provided.

In order to reduce the amount of memory required for storage, only the elements corresponding to the upper triangular part of the covariance matrices are stored in `Pmat` (a covariance matrix is always symmetric). If $P(k)$ is the covariance matrix at time k then row $k+1$ of `Pmat` is organized as follows:

$$P(k) = \begin{bmatrix} P_{11} & P_{12} & P_{13} \\ P_{12} & P_{22} & P_{23} \\ P_{13} & P_{23} & P_{33} \end{bmatrix} \quad \rightarrow \quad Pmat = \begin{bmatrix} & \vdots & & & \vdots & \\ P_{11} & P_{12} & P_{13} & P_{22} & P_{23} & P_{33} \\ & \vdots & & & \vdots & \end{bmatrix}$$

With the function **mat2cov** it is possible to extract the covariance matrix at a specified sample number.

The function **mat2var** extracts the variance estimates from `Pmat`, corresponding to the diagonal of each covariance matrix $P(k),\ k = 0, ..., N$.

In order to compare output estimates with observations of the output, use the function **kalmeval**.

How to write the m-functions

Each of the functions whose names are specified by `xfunc`, `yfunc`, and `linfunc` must have the appropriate structure. In the section covering the function **dd1** it is explained how to write the two former functions. Below, an example describing the structure of the linearization function is given (for a nonsense system). The following notation is used:

$$
\begin{aligned}
x_{k+1} &\approx f(\hat{x}_k, u_k, \bar{v}_k) + A(k)(x_k - \hat{x}_k) + F(k)(v_k - \bar{v}_k) \\
y_k &\approx g(\bar{x}_k, \bar{w}_k) + C(k)(x_k - \bar{x}_k) + G(k)(w_k - \bar{w}_k)
\end{aligned}
$$

where

$$
\begin{aligned}
A(k) &= \left.\frac{\partial f(x, u_k, \bar{v}_k)}{\partial x}\right|_{x=\hat{x}_k} \qquad F(k) = \left.\frac{\partial f(\hat{x}_k, u_k, v)}{\partial v}\right|_{v=\bar{v}_k} \\
C(k) &= \left.\frac{\partial g(x, \bar{w}_k)}{\partial x}\right|_{x=\bar{x}_k} \qquad G(k) = \left.\frac{\partial g(\bar{x}_k, w)}{\partial w}\right|_{w=\bar{w}_k}.
\end{aligned}
$$

linfunc

Assume `linfunc='my_linfunc'`:

```
function [M,N]=my_linfunc(x,u,vw,flag)
% Make variables static
persistent mypar1 mypar2;

% Check if variables should be initialized
if nargin==1,
   mypar1 = x(1)*0.5 + x(2);
   mypar2 = 75*x(3);
   A0 = diag([1 1]);
   F0 = zeros(2,2);
   C0 = [1 0];
   G0 = [];
   return
end
 .
 .
 .
```

```
.
.
% Linearize state equation
if flag==0,
  M = A0;
  M(1,2) = mypar1*(sin(x(1)*u(1)) + x(2)*u(2));

  N = F0;
  N(1,1) = mypar2/x(2);
  N(2,2) = x(1)*x(2);

% Linearize output equation
elseif flag==1,
   M=C0;
   N=G0;
end
```

Dissection of the function

The header must always look like this:
`function [M,N]=my_linfunc(x,u,vw,flag)`

The function and variable names are unimportant, but the function must always take 4 arguments and return two outputs. The arguments, three (column) vectors and an integer, are the current state estimate, control input, process noise or measurement noise, and a flag (in that order). If `flag=0` then `vw` is process noise and the function should linearize the state equation and return the matrix A in `M` and the matrix F in `N`. If `flag=1` then `vw` is measurement noise and the function should linearize the output equation and return the C in `M` and G in `N`.
Argument 2 and 3 *must* be present even if there are no inputs or no process/measurement noise.

By using the `persistent` declaration, a feature which was introduced in MATLAB 5.2, it is possible to maintain parameters from one call to another. This is convenient as one can initialize certain parameters before the filtering.

There should always be an initialization section in the function. This must take the form

```
if nargin==1,
   .. do initialization stuff
   return
end
```

Before the actual filtering is performed, `my_linfunc` will be invoked as `my_linfunc(opt.init)`. Thus, by specifying parameter initializations in `opt.init`, these parameters are passed to the function through the argument `x`. The section must be included even if there are no such intializations. In this case it should just include the `return` statement.

The remaining part of the function contains the actual linearizations. Notice that if the certain elements in the matrices are constant, one can set the constant elements in the initialization section of the function (A0, C0, etc). If one of the noise matrices (F, G) equals the identity matrix, it is recommended to set it to the empty matrix, `[ ]`, in which case one can reduce the number of computations performed in the filtering.

Algorithm

The extended Kalman filter is based on first-order Taylor approximations of the nonlinear mappings. The EKF is described in, e.g.,

M. S. Grewal & A. P. Andrews: *Kalman Filtering: Theory and Practice*, Prentice Hall, 1993.

F. L. Lewis: *Optimal Estimation*, John Wiley & Sons, 1986.

See Also

ekfm, **dd1**

ekfm

Purpose

Extended Kalman filtering for systems with multiple observation streams.

Synopsis

```
[xhat,Smat]=ekfm(xfunc,yfunc,linfunc,x0,P0,Q,R,u,y,tidx)
[xhat,Smat]=ekfm(xfunc,yfunc,linfunc,x0,P0,Q,R,u,y,tidx,optpar)
```

Description

ekfm uses the Extended Kalman Filter (EKF) to estimate the states for a nonlinear system. The model of the system must be specified in the form:

$$\begin{aligned} x_{k+1} &= f\left(x_k, u_k, v_k\right) \\ y_k^{(1)} &= g^{(1)}\left(x_k, w_k^{(1)}\right) \\ &\vdots \\ y_k^{(n)} &= g^{(n)}\left(x_k, w_k^{(n)}\right) \end{aligned}$$

where x is the state vector, u is a possible input, and v and w are (white) noise sources. Each of the equations must be written in an m-function. The arguments to the **ekfm** function are explained in the section covering the **dd1m** function. The difference from the call of **dd1m** is that it is necessary to write a file containing the linearizations. It was described under **ekf** how to do this. In the multi-stream case, the file must have a slightly different structure, though. An example of the structure is given below:

linfunc

Assume `linfunc='my_linfunc'`:

```
function [M,N]=my_linfunc(x,u,vw,flag)
% Make variables static
.
.

% Check if variables should be initialized
if nargin==1,
   .
   .
   return
end

% Linearize state equation
if flag==0,
  M = ..
  N = ..

% Linearize output equation 1
elseif flag==1,
   M=..
   N=..

% Linearize output equation 2
elseif flag==2,
   M=..
   N=..
end
```

Dissection of the function

The variable `flag` is used for pointing out which linearization to perform. If `flag=0` the linearization of the state equation should be returned. If `flag=1` the linearization of the first output equation should be returned. If `flag=2` the linearization of the second output equation should be returned, and so forth.

ekfm

See Also

dd1m, **ekf**.

kalmeval

Purpose

Evaluate filter performance.

Synopsis

```
[yhat,RMS]=kalmeval('method',yfunc,R,xhat,PS,y,tidx)
[yhat,RMS]=kalmeval('method',yfunc,R,xhat,PS,y,tidx,optpar)
```

Description

kalmeval estimates the output, y, based on the state estimates obtained from the filtering. The function plots the observed and estimated outputs as well as the state estimates along with 3 times their standard deviations.
The function assumes the (nonlinear) output equation

$$y_k \quad = \quad g\left(x_k, w_k\right)$$

is available, where x is the state vector and w is (white) measurement noise. The equation must be written in an m-function. The arguments to the **kalmeval** function are explained in the section covering the **dd1** function.

The arguments to the **kalmeval** function are explained below:

`method`	Filter method ('ekf', 'dd1', 'dd2', 'ekfm', 'dd1m', 'dd2m').
`yfunc`	Name of file containing the output equation.
`R`	Covariance matrix for the measurement noise. Only used if method='dd2' or 'dd2m'.
`xhat`	State estimates. Dimension is [samples+1 × states].
`PS`	Matrix where each row contains elements of (the upper triangular part of) the Cholesky factor of the covariance matrix (dd1, dd2, dd1m, dd2m) or the covariance matrix (ekf, ekfm). The dimension is [samples+1 × 0.5*states*(states+1)].
`y`	Output signal. Dimension is [observations × outputs].
`tidx`	Vector containing time stamps (in samples) for the observations in `y`.
`optpar`	Data structure containing initialization parameters (optional).

`optpar.vmean`:	Mean of process noise vector (default is 0).
`optpar.wmean`:	Mean of measurement noise vector (default is 0).
`optpar.init`:	Initial parameters for `xfunc`, `yfunc` (arbitrary format).

kalmeval

The section covering the **dd1** filter explains how to write the function `yfunc`. This section also explains the format of the matrix `PS`. See also **ekf**.
In case of multiple observation streams (dd1m, dd2m, ekfm), the arguments `yfile`, `R`, `y`, `tidx`, and `optpar.wmean` must be cell structures.

The function returns the output estimates through the argument `yhat`, which is a matrix of dimension [samples $\times$ outputs]. The output argument `RMS` is a vector containing the RMS error between observations and estimates of each output.

mat2cov

Purpose

Extract covariance matrix from vector of upper triangular elements.

Synopsis

`P = mat2cov(Pvec)` returns the (quadratic) covariance matrix when given a vector containing the upper triangular elements.

`P = mat2cov(Pmat,k)` extracts the kth row from the matrix of vectors, `Pmat`. The vectors of upper triangular elements must be organized row wise in `Pmat`.

Description

The matrix `Pmat` is an output argument from the functions **ekf** and **ekfm**. If $P(k)$ is the covariance matrix at time k then row $k+1$ of `Pmat` is organized as follows:

$$P(k) = \begin{bmatrix} P_{11} & P_{12} & P_{13} \\ P_{12} & P_{22} & P_{23} \\ P_{13} & P_{23} & P_{33} \end{bmatrix} \quad \rightarrow \quad Pmat = \begin{bmatrix} & \vdots & & & \vdots & \\ P_{11} & P_{12} & P_{13} & P_{22} & P_{23} & P_{33} \\ & \vdots & & & \vdots & \end{bmatrix}$$

The purpose of **mat2cov** is to extract the specified row from `Pmat` and restore the proper format of the covariance matrix.

See Also

ekf, **ekfm**, **mat2var**

mat2var

Purpose

Extract variance estimates from matrix of covariance estimates.

Synopsis

```
varmat = mat2var(Pmat)
```

Description

mat2var extracts the variance estimates (corresponding to the diagonals of the covariance matrices) from a matrix for which each row contains the upper triangular elements of a covariance matrix.

See Also

ekf, **ekfm**, **mat2cov**

smat2cov

Purpose

Restore covariance matrix from vector of Cholesky factor elements.

Synopsis

`P = smat2cov(Svec)` returns the (quadratic) covariance matrix when given a vector containing the (upper triangular) Cholesky factor elements.

`P = smat2cov(Smat,k)` extracts the kth row from the matrix of vectors, `Smat`. The vectors of Cholesky factor elements must be organized row wise in `Smat`.

Description

The matrix `Smat` is an output argument from the functions **dd1**, **dd2**, **dd1m**, and **dd2m**. If the covariance matrix is denoted P, the Cholesky factor S is an upper triangular matrix with the property

$$P = SS^T$$

For sample no. k, the elements of $S(k)$ are stored in row $k+1$ of the returned matrix, `Smat`:

$$S(k) = \begin{bmatrix} S_{11} & S_{12} & S_{13} \\ 0 & S_{22} & S_{23} \\ 0 & 0 & S_{33} \end{bmatrix} \quad \rightarrow \quad Smat = \begin{bmatrix} & \vdots & & & \vdots & \\ S_{11} & S_{12} & S_{13} & S_{22} & S_{23} & S_{33} \\ & \vdots & & & \vdots & \end{bmatrix}$$

The purpose of **smat2cov** is to extract the specified row from `Smat`, and restore the proper format of the covariance matrix.

See Also

dd1, **dd1m**, **dd2**, **dd2m**, **smat2var**

smat2var

Purpose

Calculate variance estimate for each state.

Synopsis

```
varmat = smat2var(Smat)
```

Description

smat2var returns a matrix where each column is the variance of a state estimate. `Smat` is a matrix where each row contains elements of (the upper triangular part of) the Cholesky factor of a covariance matrix.

See Also

dd1, **dd1m**, **dd2**, **dd2m**, **smat2cov**

triag

Purpose

Triangularization with Householder transformation.

Synopsis

```
S = triag(A)
```

Description

triag uses a Householder transformation on the rectangular matrix A to produce a square and upper triangular matrix S with the property $SS^T = AA^T$.

Bibliography