Adaptive Filtering and Change Detection

TOOLBOX

Fredrik Gustafsson

Purpose	Syntax

Simulation	y=simadfilter (z,nnn,adm,adg)
	y=simchange (z,nnn,jumptimes,TH)
	y=simresid (z,nnn,threc)
	[y,u,th]=simchannel (alphabet,nn,N,th)

Adaptive filtering	[thhat,epsi] =adfilter (z,nnn,adg,adm) [xhat] =adkalman (z,nnn,adg,options)
Change detection	[threc,jhat,thseg]=detect1 (z,nnn,options) [threc,jhat,thseg]=detect2 (z,nnn,options) [threc,jhat,thseg]=detectM (z,nnn,options) [threc,jumpsmc,jhat]=gibbs (z,nnn,options) [jhat,jtype,thseg]=multihyp (z,nnn,options) [jhat,thseg]=cpe (z,nnn,options) [jhat]=mlr (z,nnn,options) [xhat,jhat]=glr (z,nnn,options) [r]=faultdetect (z,nnn,options)
Equalization	[uhat]=viterbi (y,th,alphabet,lam,utrain)

Blind equalization	[threc,uhat]=blindeq (y,nnn,adg,adm,th0)
	[threc,uhat]=blindeqM (y,nnn,alphabet)
	ber=u2ber (u,uhat)
	m=openeye (bch,beq)

Model conversion	thseg=par2segm (TH,jumptimes,N)
	TH=segm2par (thseg)
	phi=z2phi (z,nnn)

Plot	segplot (z,jumptimes) thplot (th1,TH,jumptimes,th2) hypplot (XdetectM) radarplot (y,xhat,x)
Help	helpdetect
Demonstration	demodetect , reference , tutorial
Book examples	book , signal
GUI	guidetect
Real data sets	airbag, altdata, ash, bach, carspeed ceram, defibrator, eeg_human, eeg_rat ekg, equake, fricest, fuel, highway labmotor, nmt450, nmt900, nose, carpath photons, planepath, salesment, sfquake sheep, speech, tpi
Models	nnnplane , nnnDCm1

Contents

Command overview

Preface

This manual is the link between the MATLAB^{TM} toolbox

`http://www.sigmoid.se`

and the text book [10]

`http://www.comsys.isy.liu.se/books/adfilt`.

The tutorial describes how to make simple simulation studies and illustrates the different approaches to adaptive filtering and change detection as well as the supported model structures. Two chapters describe target tracking and equalization in digital communications, where dedicated functions are available. Chapter 6 shows some examples on how real data can be analysed. The examples are taken from the text book, and here accompanied with complete code. As a side mark, the toolbox contains a variety of real data from various applications useful for benchmark studies.

Chapter 7 presents some advanced material for those who want to make more customized applications, or real-time applications. Chapter 8 presents the Graphical User Interface (GUI).

The Reference chapter 9 explains each function in detail. Tables of functions grouped similar to the tutorial are given first. All examples given in the reference part are available in `reference` .

The theory behind the functions is thoroughly treated in the accompanying text book. Each of the toolbox algorithms corresponds to one or more algorithms in the textbook, and the reference part contains a precise reference to which algorithm is implemented. The examples in the text book are available in `book` . There is also a connection to the text book in *Signal Processing* (in swedish). The examples in that book are given in `signal` . Other examples are found in `demodetect` .

More information, related material and upgrades can be found at the toolbox homepage.

1 Preliminaries

This tutorial part is divided into sections of different application areas: Kalman filtering, adaptive filtering, change detection, equalization and target tracking. Each chapter can be seen as a mini-manual for its respective application, where a table of all relevant toolbox functions is given first, followed by examples. The tutorial examples are available in the toolbox as
`tutorial ('application')`.

1.1 The Adaptive Filtering and Change Detection Problems

The toolbox provides algorithms, plot facilities, analysis tools and design tools for adaptive filtering and change detection. Applications in these areas can be divided into the the following categories:

- *Surveillance* and *parameter tracking*. Classical surveillance problems consist in filtering of noisy measurements of physical variables as flows, temperatures, pressures etc. In model-based signal processing, *adaptive filtering* consists in tracking time-varying parameters. Monitoring physical parameters in the model is one application. Another one is *adaptive control*, which can be based on these parameters. Yet another example is *blind equalization*, where the model is used to equalize channel distortion.

- *State estimation*. The Kalman filter provides the best linear state estimate, and change detection support can be used to speed up the response after disturbances and abrupt changes. *State feedback*, such as *LQG* (*linear Gaussian control*), belongs to this area. *Navigation* and *target tracking* are two particular application examples.

- *Fault detection*. Faults can occur in almost all systems. Change detection here has the role to locate the fault occurence in time and to give a quick alarm. After the alarm, isolation is often needed to locate the faulty component. The combined task of *detection* and *isolation* is commonly refered to as *diagnosis*.

These problem areas are usually treated separately in literature. However, the tools for solving these problems have much in common and the same algorithms can be used. The main difference lies in the evaluation criteria. In surveillance the parameter estimate should be as close to the true value as possible, while in fault detection it is essential to get an alarm from the change detector as soon as possible after the fault, and at the same time receiving few false alarms. The design usually consists of the following steps:

1. Modeling the signal or system.
2. Implementing an algorithm.
3. Tuning the algorithm with respect to certain evaluation criteria, either using real or simulated data.

The toolbox provides a large number of algorithms suggested in literature, and many evaluation criteria. The novel concept of auto-tuning an adaptive algorithm is introduced. That is, an objective measure of performance is chosen and the design parameters are optimized with respect to this measure for each method and finally the best method is chosen. The output of a typical design session is a table summarizing the performance of different algorithms.

1.2 Data objects

To understand the syntax of the functions, a brief description of the data objects is motivated. The core object is the model. The toolbox accepts models in standard formats as state space models, ARX like structures (like `nn` in the System Identification TB), or a user-defined m-file. The first object in the list below is a generalization of these models.

`nnn` is the model structure used in simulation or filtering. It comprises the different filter structures `nn` in the System Identification Toolbox as well as linear regression models and state space models. `nnn` can also be a string, when it represents a user-written m-file defining the model.

`jumptimes` is a vector of time instants for abrupt changes. `jumphat` is an estimate of `jumptimes`. The number of jump times is denoted n.

`z` is the data vector, consisting of the inputs for simulation, and the outputs and inputs for estimation. The number of data is N.

`th` is a matrix of dimension $d \times N$ where the columns define the parameter vector at time t. `threc` is the recursive estimates and `thseg` stands for smoothed parameter estimate, which are piecewise constant.

`TH` with capital letters is as above, but the columns define the parameter vector for each segment (between the jump times). The n jumps define $n+1$ segments, so its size is $d \times (n+1)$ for parametric models. For state space models, its size is $d \times n$.

`x` is the state vector in a state space model, and `xhat` its estimate.

`sr` contains the design parameters of the *stopping rule*, see Section 23.

`dm` chooses the *distance measure*, see page 20.

1.3 Installation

The installation should be quite simple, since the core functions are plain text files. Assure that the directory `adfilt/` is in the MATLAB$^{\text{TM}}$ path. There are two sub directories. First, `adfilt/blockset/` contains C-code and Simulink blocks. If you want to use these, include the directory in the search path. Second, `adfilt/extras` contains mostly dummy functions in case some matlab toolboxes are missing. For instance, the GUI uses `leastsq` from the optimization toolbox at one instance, and if this function does not exist, the GUI cannot be started. For this reason, it is recommended that this directory should be put last in the path, for instance using `path(path,'/adfilt/extras');` Then the toolboxes you have installed, will be used.

To summarize:

1. Move the directory `adfilt` from the toolbox CD to anywhere on your hard disk, most logically to something like `c:/matlabR12/toolbox/adfilt`.
2. Add the following lines to the file `startup`:

```
disp('Adaptive filtering and change detection toolbox.')
disp('To get started, do ''help adfilt''')
disp(' ')
addpath('c:/matlabR12/toolbox/adfilt');
addpath('c:/matlabR12/toolbox/adfilt/blockset');
path(path,'c:/matlabR12/toolbox/adfilt/extras');
```

For upgrades, bug fixes and tips, check the home page `http://www.sigmoid.se`.

2 Adaptive filtering and change detection

Purpose	Syntax
Simulation	`y=simchange (z,nnn,jumptimes,TH)`

Change detection	`[threc,jhat,thseg]=detect1 (z,nnn,options)`
	`[threc,jhat,thseg]=detect2 (z,nnn,options)`
	`[threc,jhat,thseg] =detectM (z,nnn,options)`
	`[threc,jhat,thseg]=multihyp (z,nnn,options)`
	`[jhat,thseg]=cpe (z,nnn,options)`

Plot	`segplot (z,jumptimes)`
	`thplot (jumptimes,th0,threc,thseg)`

2.1 Introduction

The basic idea is that one function simulates the output of a specified signal model or system, and another one inverts it aiming at reconstructing its inputs. The principle is illustrated in Figure 2.1.

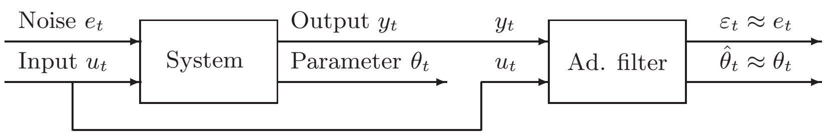


Figure 2.1 Adaptive filter structure.

A typical use of the functions could be

```
[y,th] = simadfilter([e u],nnn,adg,adm);
[thhat,lamhat,epsi] = adfilter([y u],nnn,adg,adm);
```

The simulation function `simadfilter` simulates a linear filter where the parameters change as a random walk. This random walk can be taylored to match the adaptive filters RLS, NLMS or the Kalman filter. For instance, one can simulate, in principle, a signal where NLMS with, say, step size 0.0014 is the optimal linear filter.

One can also use `simchange` to simulate signals with abruptly changing dynamics or disturbances, which can be used to test the tracking ability of the linear adaptive filters implemented in `adfilter`. In this latter case where the system undergoes abrupt changes, Figure 2.2 is more relevant:

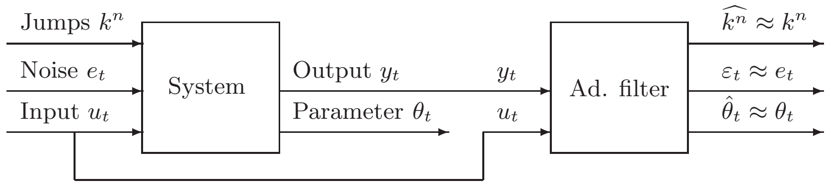


Figure 2.2 Adaptive filter structure for change detection.

The corresponding sequence of function calls may be:

```
[y,th] = simchange([e u],nnn,jumps,TH);
[thhat,lamhat,epsi] = adfilter([y u],nnn,adg,adm);
[thhat1,jumphat1,thseg1] = detect1([y u],nnn);
epsi1 = simresid([y u],nnn,thseg1);
```

The last call is necessary since the residuals using smoothed estimates (`thseg`) are not computed automatically.

2.2 Change detection principles

The change detectors `detect1`, `detect2`, and `detectM` can be applied anywhere where `adfilter` can be applied. The difference is a speed-up in tracking ability when an abrupt change is detected, and an explicit alarm output argument. `detect1` is based on whiteness residual tests from one adaptive filter (see Figure 2.3), `detect2` compares the results from two parallel filters, one slow and one fast (see Figure 2.4), and finally `detectM` is based on a multiple-model hypothesis test (see Figure 2.5).

`cpe` (Change Point Estimation) implements off-line algorithms proposed in the literature of mathematical statistics. `glr` (the Generalized Likelihood Ratio test) and `mlr` (the Generalized Likelihood Ratio test) are aimed at detecting state or sensor changes in state space models.
`faultdetect` implements algorithms found in the literature of fault/failure detection in the area of control theory.

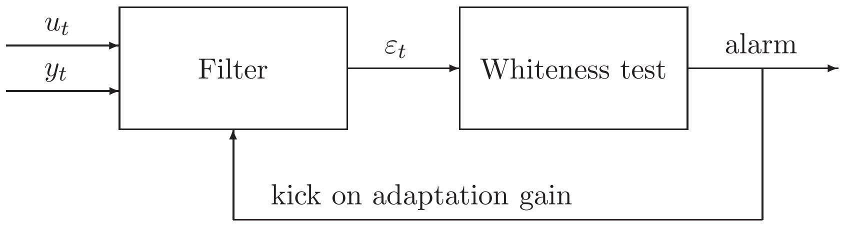


Figure 2.3 Idea of detection based on residual whiteness test. When non-white residuals are detected, the adaptation gain (forgetting factor, step size or the like) is momentarily increased.

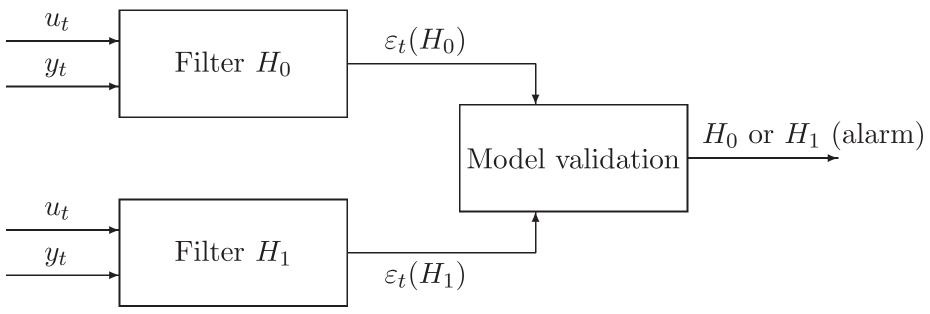


Figure 2.4 Idea of detection based on parallel filters. The slow filter H_0 normally provides the parameter or state estimate, but after a change the fast filter gives smaller residuals and the model validator chooses H_1, whose estimate is used for initialization when the slow filter is restarted.

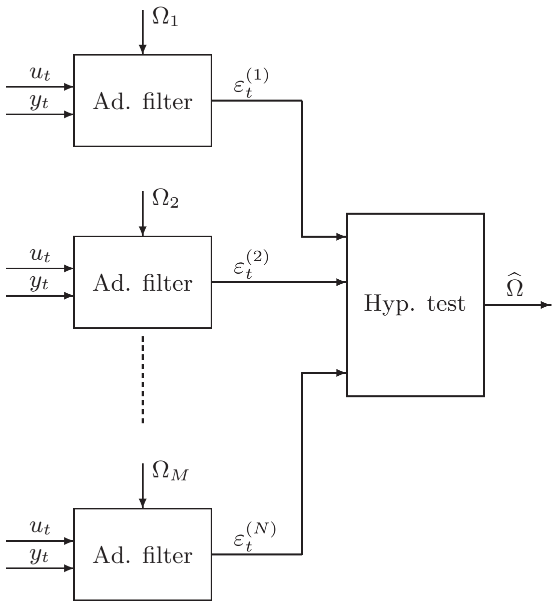


Figure 2.5 A bank of matched filters, each one based on a particular assumption on the set of change times $\Omega = \{k_i\}_{i=1}^n$, that are compared in a hypothesis test.

2.3 Data simulation

Adaptive filtering will be illustrated on a simulated ARX model

$$y(t) = \frac{\left(\theta_3 q^{-1} + \theta_4 q^{-2}\right) u(t) + 0.1e(t)}{1 + \theta_1 q^{-1} + \theta_2 q^{-2}}, \quad Eu^2 = Ee^2 = 1$$

of length 100 with abrupt changes at times 40 and 70. Both input and noise are generated as Gaussian white noises.

```
randn('seed',1)
u=randn(100,1);     % Input
e=randn(100,1);     % Noise
jumptimes=[40 70];
TH=[1.5 .8 2 0.5;...
    1.5 .8 2 1;...
    1.5 .6 2 1]';
nnn=[2 2 1];        % Model structure
y=simchange([0.1*e u],nnn,jumptimes,TH);
z=[y u];
segplot(y,jumptimes)
title('Signal and real change times')
```

From Figure 2.6 it is not easy to estimate the change times by visual inspection.

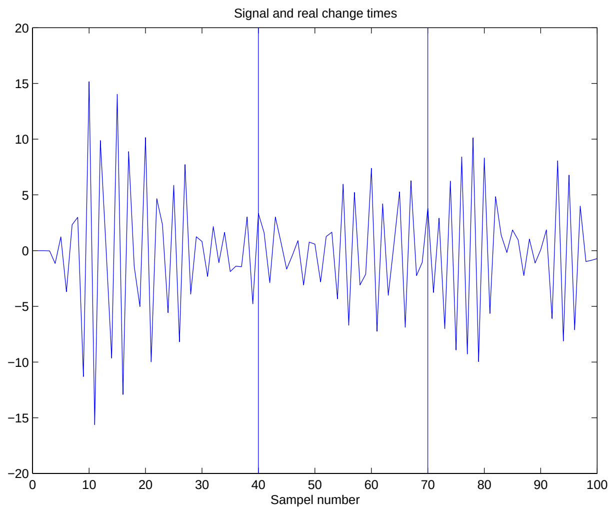


Figure 2.6 Simulated signal and change times.

2.4 Linear filter

First, we examine the signal using RLS with forgetting factor 0.9.

```
thRLS=adfilter(z,nnn,0.9,'RLS');
thplot(thRLS,TH,jumptimes);
```

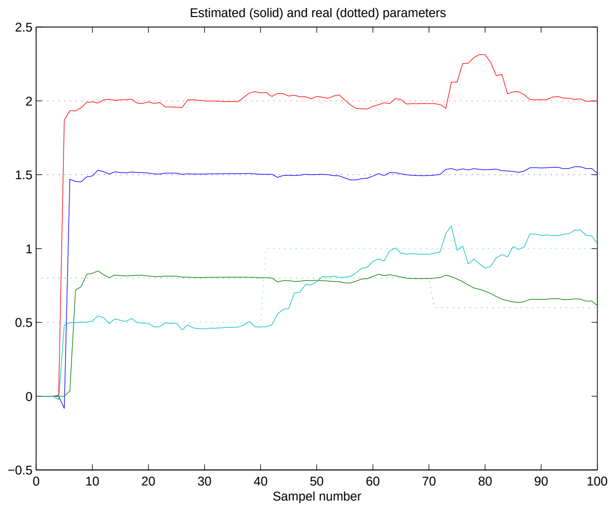


Figure 2.7 Parameter estimates from RLS.

You can guess where the abrupt changes are located by visual inspection. Other forgetting factors than 0.9 give either slower response or a more noisy estimate.

2.5 Whiteness test detection

The one-filter approach tries to estimate the jumps by just looking at the residuals from a standard filter. Two different *distance measures* are possible, but the residual does not make sense here. The squared residual (`dm=1`) is therefore used. Two conceivable *stopping rules* are compared: the *CUSUM* and *GMA* detectors. Only the one-sided version is computed, since the squared residual will not decrease after a change. The stopping rules decide when the squared residual is large enough to alarm. Let us test the CUSUM test with threshold 10 and drift parameter 5 (`sr=[1 10 5]`).

```
[threc,jhat,thseg,gt2,ta]=detect1(z,nnn,2,[1 10 5],0.01);
jhat
jhat =
    42
    72
thplot(thseg,TH,jumptimes);
```

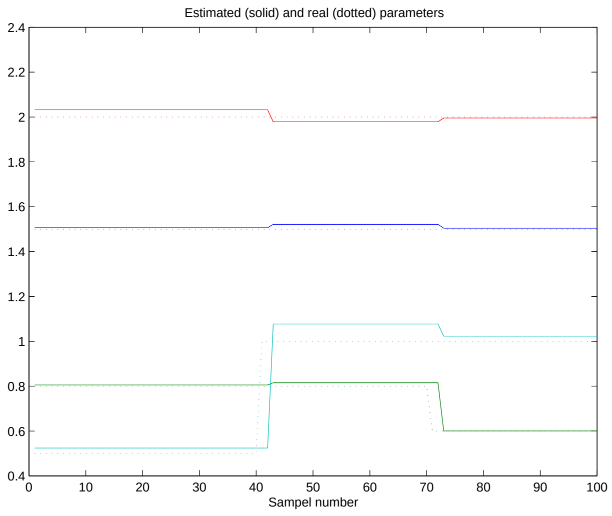


Figure 2.8 Parameter estimates from `detect1`.

Once again the result is very good. The one-filter approach is furthermore extremely cheap to compute and easy to implement.

Let us look at the distance measure in Figure 2.9, where an alarm is obtained when the curve exceeds the threshold value 10. The delay for detection is about two samples.

```
h2=cdfigure(2,1);
```

```
plot([gt2(:,1)])
title('Distance measure for the stopping rule')
```

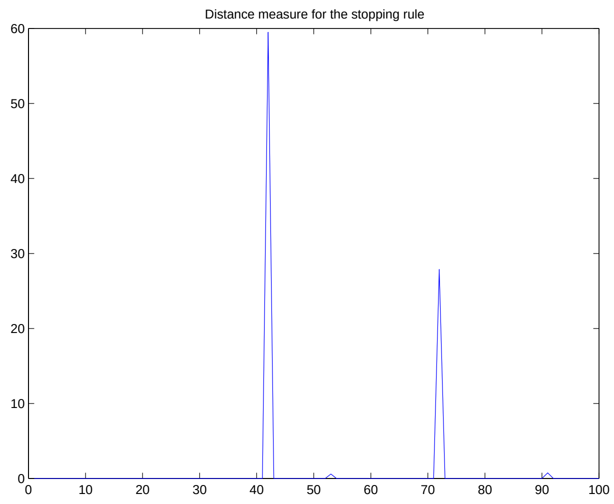


Figure 2.9 Distance measure for `detect1`.

2.6 Parallel filter detection

In the *two-filter approach*, one model is estimated on all past data and compared to one estimated in a sliding window. There are three available distance measures, *Brandt's GLR test* (`dm=1`), the *divergence test* (`dm=2`) and the *asymptotic local approach* (`dm=3`). The same stopping rules as for the one-filter approach are possible, but the design parameters are generally different.

```
[threc,jhat,thseg1,gt3,ta]=detect2(z,nnn,1,[1 5 0.3],10,-1);jhat
jhat =
    41
    71
[threc,jhat,thseg2,gt4,ta]=detect2(z,nnn,2,[1 10 3],10,-1);jhat
jhat =
    41
    72
h1=cdfigure(1);
thplot(thseg1,TH,jumptimes);
h2=cdfigure(2,1);
thplot(thseg2,TH,jumptimes);
```

The jump times are estimated correctly but the parameter estimates are not as good as in the `detectM` function.

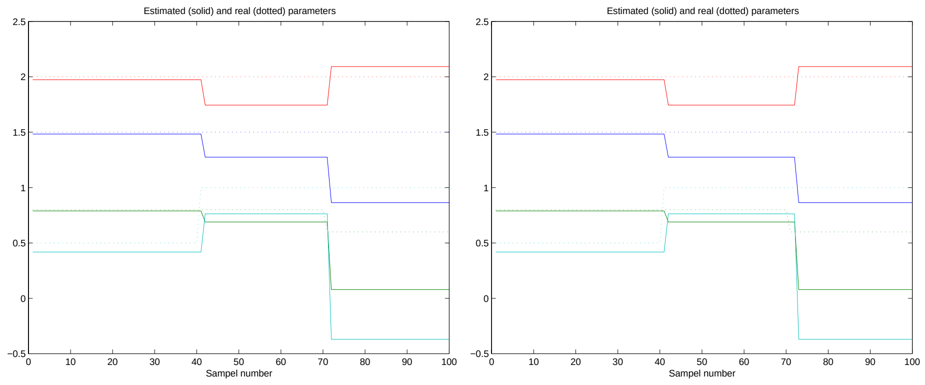


Figure 2.10 Parameter estimates from `detect1` using stopping rules CUSUM and GMA, respectively.

```
h3=cdfigure(3,2);
plot([gt3(:,1) gt4(:,1)])
title('Distance measure for the stopping rule')
```

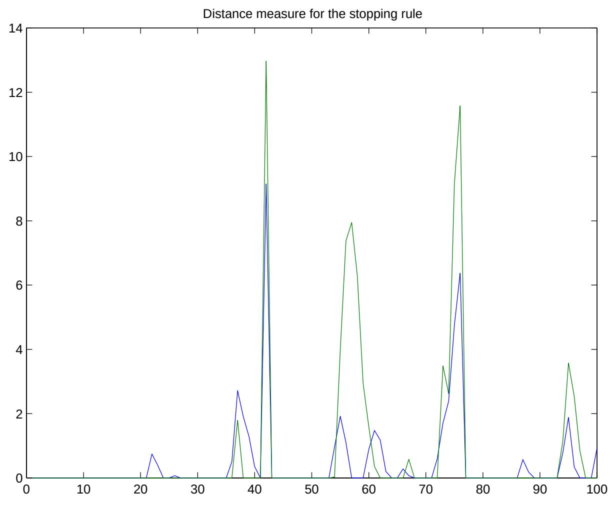


Figure 2.11 Distance measure for `detect2`.

Note from Figures 2.11 and 2.9 that different thresholds are required in the stopping rules! Generally, the design of a stopping rule is a bit difficult. See the command reference for some general guidelines, and Section 2.8 for a dedicated study of the performance of a stopping rule.

2.7 Multiple-model detection

Now, the function `detectM` is applied. Here a number of different filters are run in parallel. A weighting coefficient (the likelihood) determines which is the best one. Note that the default design parameters can be used, as is the case for most signals.

```
[threc,jhat,thseg,lamhat,Alfa]=detectM(z,nnn);jhat
jhat =
    41
    71
h1=cdfigure(1);
segplot(y,jhat)
title('Signal and estimated change times')
h2=cdfigure(2,1);
thplot(thseg,TH,jumptimes);
```

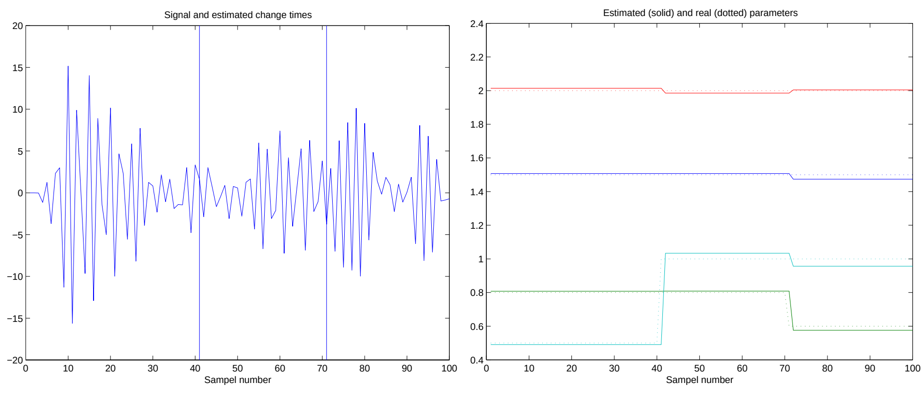


Figure 2.12 Signal segmentation and parameter estimates from `detectM`.

The true change times are mostly estimated correctly in this example.

2.8 Stopping rules

We will here demonstrate how to apply and design a stopping rule with particular attention to the CUSUM detector. Compare with Section 12.2 in the text book. Assume we observe a signal $y(t)$ which is $N(0, \sigma^2)$ before the change and $N(\theta, \sigma)$ after the change. We apply the *CUSUM test* with threshold h

and drift ν:

$$g_t = g_{t-1} + y_t - \nu \tag{2.1a}$$
$$g_t = 0,\ \text{and}\ \hat{k}_{temp} = t \text{ if } g_t < 0 \tag{2.1b}$$
$$g_t = 0,\ t_a = t,\ \hat{k} = \hat{k}_{temp} \text{ and alarm if } g_t > h > 0. \tag{2.1c}$$

A successful design requires $\theta > \nu$.

The alternative is to use the *geometrical moving average* (*GMA*) test:

$$g_t = g_{t-1} + \lambda y_t \tag{2.2a}$$
$$g_t = 0,\ t_a = t \text{ and alarm if } g_t > h > 0. \tag{2.2b}$$

The *average run length* (*ARL*) function for a stopping rule for detecting a change θ in the mean of a signal is defined as

$$E(t_a|\theta) \tag{2.3}$$

where t_a is the stopping time. There are two design parameters in the *CUSUM* test, the *threshold* h and the *drift parameter* ν. If σ is the standard deviation of the noise, then

$$t_a = f\left(\frac{h}{\sigma}, \frac{\theta - \nu}{\sigma}\right) \tag{2.4}$$

That is, it is a function of two arguments. We can assume that $\sigma = 1$ and, for simplicity, $\nu = \theta/2$ (a standard choice of drift if θ is known in advance). The mean time between false alarms is $f(h, -\theta/2)$ and the mean time for detection is $f(h, +\theta/2)$. To check these for a given h, do as follows:

```
h=3;
th=1;
nu=th/2;
mu=th-nu;
[ta0,N0] = cusumarl(h,-mu);
[ta1,N0] = cusumarl(h,+mu);
disp([ta0, ta1])
     127.5000     6.5000
```

That is, the mean time between false alarms is 6.5 samples. `N0` denotes the mean time beween resets in the CUSUM algorithm. A reset is caused by either an alarm or a negative test statistic (the reset operation in (2.1b)).

The exact value of the ARL function is given by a so called *Fredholm integral equation* of the second kind, which must be solved by a numerical algorithm. A direct approximation suggested by Siegmund is obtained by

```
[ta0siegmund] = cusumarl(h,-mu,0.5,-1);
[ta1siegmund] = cusumarl(h,+mu,0.5,-1);
```

and another approximation suggested by Wald is obtained by

```
[ta0wald] = cusumarl(h,-mu,0.5,-2);
[ta1wald] = cusumarl(h,+mu,0.5,-2);
```

These values are displayed in the table below. The mean times do not say anything about the distribution of the run length, which can be quite unsymmetric. Monte Carlo simulations can be used for further analysis of the run length function:

```
NMC=1000;
L0=cusumMC(h,-mu,NMC);
h1=cdfigure(1);
hist(L0,30);
title('Distribution of false alarm times')
L1=cusumMC(h,mu,NMC);
h2=cdfigure(2,1);
hist(L1,30);
title('Distribution of delay for detection')
```

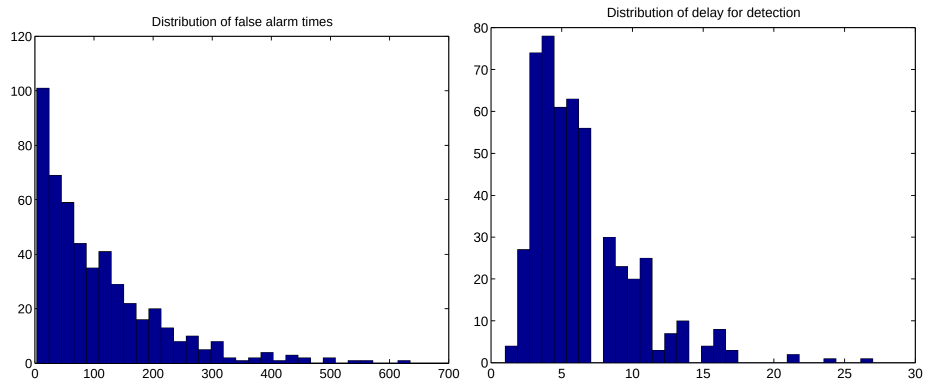


Figure 2.13 Distribution for false alarm times and delay for detections, respectively.

Compare theoretical with achieved mean times;

```
disp('Numerical    Wald      Siegmund       MC')
disp([ta0 ta0wald ta0siegmund mean(L0)])
disp([ta1 ta1wald ta1siegmund mean(L1)])
Numerical    Wald      Siegmund       MC
```

```
  127.5000   32.0000  118.5000  116.8040
    6.5000    4.0000    6.5000    6.2080
```

The distribution for false alarms is basically binominal. A rough estimate of the delay for detection is $L_0 = h/(\theta - \nu)$, which is 6 in this example.

What is really needed in applications is to determine h from a specified mean time for detection L_0. This is computed by a numerical search in

```
hmax=10;
L0=10;
h=cusumdesign(L0,mu,hmax)
h =
     5
```

Evaluate the design by Monte Carlo simulations

```
L1=cusumMC(h,mu,NMC);
mean(L1)
ans =
    10.28
```

Finally, we can generate a table. This is the table on page 443 in the textbook and Table 5.1 in [2].

```
MU=[-2 -1.5 -1 -0.5 0 0.5 1 1.5 2.0];
h=3;
for i=1:length(MU);
  [ta0,N0] = cusumarl(h,MU(i),0.1);
  [tas,N0] = cusumarl(h,MU(i),0.1,-1);
  [taw,N0] = cusumarl(h,MU(i),0.1,-2);
  L1=cusumMC(h,MU(i),NMC);
  tab(i,:)=[MU(i),ta0,taw,tas,mean(L1)];
end
format short e
disp(' ')
disp('       mu        Exact   Wald     Siegmund      MC')
disp(tab(1:3,:));
format short
disp(tab(4:9,:));
        mu       Exact   Wald     Siegmund      MC
  -2.00e+00   2.36e+02   2.03e+04   2.15e+06   9.99e+02
```

```
-1.50e+00   1.98e+02   1.79e+03   5.95e+04   9.92e+02
-1.00e+00   2.11e+02   1.98e+02   2.07e+03   7.65e+02
 -0.50   127.7000   32.2000  118.6000  119.9640
      0   19.4000    9.0000   17.4000   16.7620
 0.5000    6.7000    4.1000    6.4000    6.4620
 1.0000    3.9000    2.5000    3.7000    3.7600
 1.5000    2.7000    1.8000    2.6000    2.6560
 2.0000    2.1000    1.4000    2.0000    2.1740
```

The table is also illustrated in Figure 2.14.

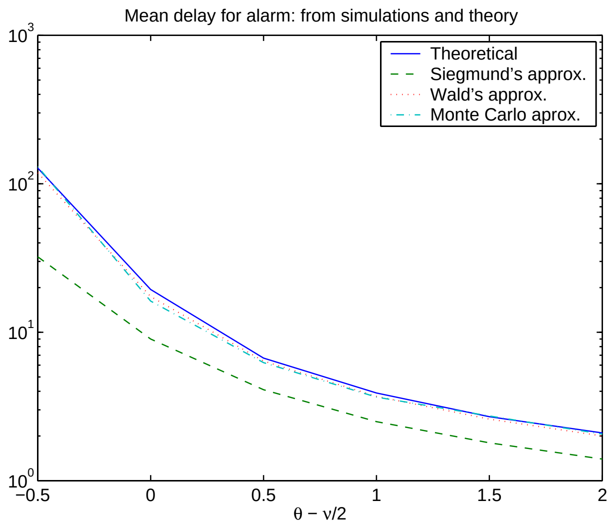


Figure 2.14 ARL function for different approximations as a function of $\mu = \theta - \nu$.

The ARL function can only be computed analytically for very simple cases, but the approach based on Monte Carlo simulations is always applicable. See the GUI chapter how this work can be automated.

3 Kalman filtering and change detection

Purpose	Syntax
Simulation	`[y,x] = simchange (z,nnn,jumptimes,TH)`

Adaptive filtering	`xhat = adfilter (z,nnn,adg,adm)`
	`xhat = adkalman (z,nnn,options)`
	`xhat = detect1 (z,nnn,options)`
	`xhat = detect2 (z,nnn,options)`
	`xhat = detectM (z,nnn,options)`

Plot	`segplot (z,jumptimes)`
	`thplot (xhat,x)`

3.1 Introduction

As for adaptive filtering, the basic idea is that a two-liner as

```
[y,x] = simchange(z,nnn,jumptimes,TH);
xhat = adfilter([y u],nnn);
```

will simulate a state space model and recover the state from the output with a *Kalman filter*, respectively. Compare with Figure 3.1

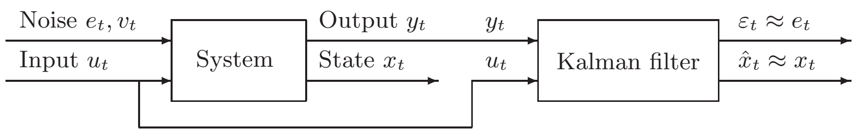


Figure 3.1 Kalman filter structure.

That is, the simulation and filter functions will be each others inverses. The difference of `adfilter` and `adkalman` is that the latter is a dedicated Kalman filter, while the former covers more models than a state space one. Consequently, `adkalman` has more features implemented:

- It can deliver filter, prediction and smoothing forms. (Only filter in `adfilter`.)
- It applies time-varying or stationary Kalman filter. (Only time-varying in `adfilter`.)
- It may use a numerically more well-conditioned, but slightly more complex, implementation, using a *square root algorithm*.

For change detection, the two-liner becomes

```
[y,x] = simchange(u,nnn,jumptimes,TH)
[xhat,jumphat] = detect1([y u],nnn,options);
```

The change detection function for `detect1` has the same limitations on generality as `adfilter`.

3.2 Data simulation

The functions will be illustrated on the discrete time correspondence to the continuous time model

$$\begin{aligned} \ddot{x}(t) &= w(t) + f\delta(t - kT_s) && (3.1)\\ y(t) &= x(t) + e(t). && (3.2) \end{aligned}$$

This double integrator model is common in navigation and tracking applications, where the model simplifies reality and assumes that the object is moving as a random walk. Here f is an unknown force which influences the object at time kT_s, where T_s is the sample interval. The sampled model is defined (see also `c2d` in CSTB), simulated and filtered below:

```
A=[1 1;0 1]; B=[0.5;1]; C=[1 0]; D=0;
Q=0.01*eye(2);
R=0.01;
P0=1*eye(2);
nnn=ss2nnn(A,B,C,D,Q,R,P0);
jumptime=50;
TH=[10; 20];
u=randn(100,1);
[y,X]=simchange([u],nnn,jumptime,TH);
[Xhat,lamhat,epsi]=adfilter([y u],nnn);
h1=cdfigure(1);
plot(epsi)
```

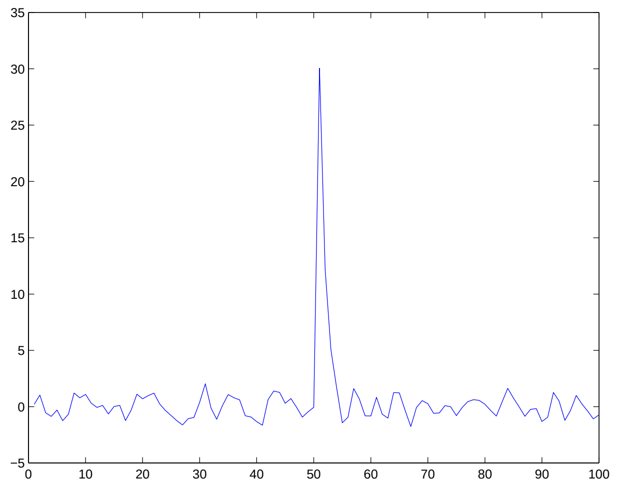


Figure 3.2 Prediction error from Kalman filter.

Figure 3.2 shows the prediction error from the Kalman filter, where the peak comes from the abrupt force change.

`mlr` is suitable for this kind of motion model. It provides a very efficient method for locating the change point.

```
[jhat,lr]=mlr([y u],nnn);jhat
jhat =
    50
h2=cdfigure(2,1);
plot(lr)
```

The `glr` approach is somewhat slower, but gives on the other hand an estimate of the change magnitude. That is, the state vector can be compensated for the estimated change as soon as a decision is made about the change time.

```
[Xhatglr,jumphat,ta,lr,numat,mu]=glr([y u],nnn,10,6e4);
disp([jumphat])
disp([nuhat])
h3=cdfigure(3,2);
plot(lr)
```

```
h4=cdfigure(4,3);
leg1=plot(X(1,:),':'); hold on
leg2=plot(Xhat(1,:),'--');
leg3=plot(Xhatglr(1,:),'-'); hold off
```

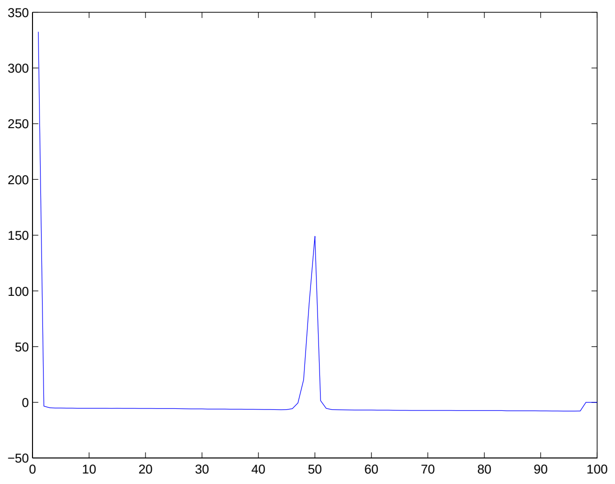


Figure 3.3 Marginalized likelihood ratio function $p(y^{100}|k)$ for `mlr` computed off-line.

```
legend([leg2 leg2 leg3],'True states',...
   'Kalman filtered states','GLR filtered states',2 );

h5=cdfigure(5,4);
leg1=plot(X(2,:),':'); hold on
leg2=plot(Xhat(2,:),'--');
leg3=plot(Xhatglr(2,:),'-'); hold off
legend([leg2 leg2 leg3],'True states',...
   'Kalman filtered states','GLR filtered states',2);
```

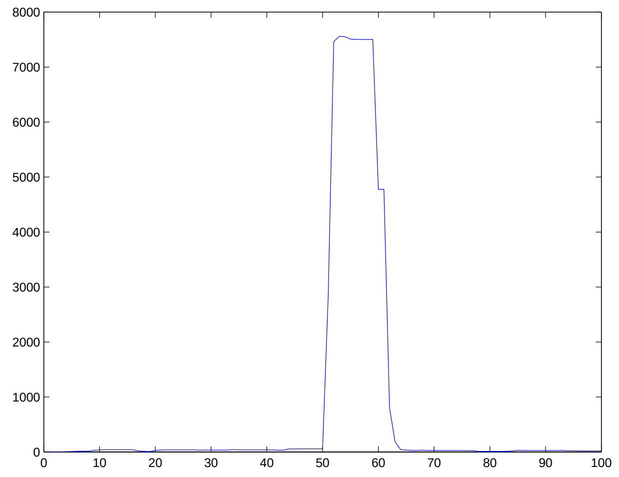


Figure 3.4 Generalized likelihood ratio function $p(y^t|k = t - L)$ for `glr` computed using a sliding window of size $L = 10$.

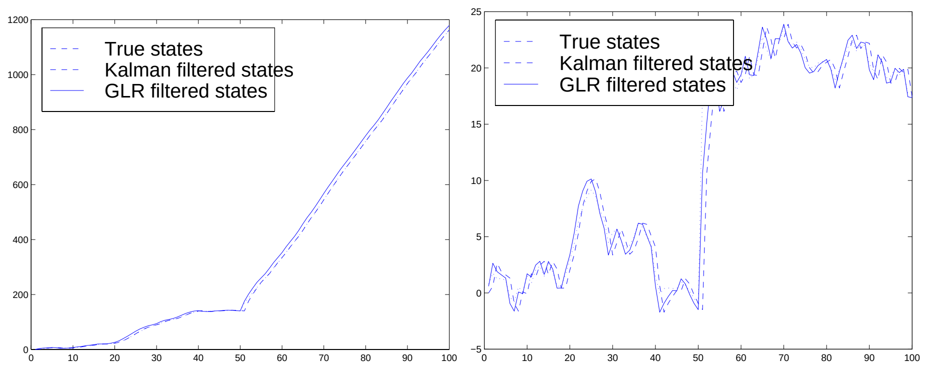


Figure 3.5 True state and estimates from Kalman filter and GLR.

4 Equalization

Purpose	Syntax
Simulation	`[y,u,th] = simchannel (alphabet,nb,N,th)`
Equalization	`[uhat] = viterbi (y,th,alphabet,lam,utrain)`
Blind equalization	`[threc,uhat] = blindeq (y,nb,adg,adm,th0)`
	`[uhat,threc] = blindeqM (y,nb,alphabet)`
Utilities	`ber = u2ber (u,uhat)`
	`m = openeye (bch,beq)`
Demo	`demo_equalization`

4.1 Introduction

The contribution to *digital communication* in this toolbox is the set of functions `simchannel` , `viterbi` , `blindeq` and `blindeqM` . These all operate on FIR filters, as is the standard model in this area. The FIR orders are specified in `nb`. `simchannel` simulates a channel, where the input alphabet and channel parameters are specified. One can also simulate time-varying fading channel, where channel fading parameters are specified. `blindeq` implements classical methods based on LMS like algorithms, and `blindeqM` implements multiple-model algorithms, or *maximum likelihood sequence detector* algorithms. In structure and code, these are very similar to `adfilter` and `detectM` .

The basic idea is that a simulation and estimation are inverses as follows:

```
[y,u,th]=simchannel(alphabet,nn,N);
[threc,uhat]=blindeqM(y,nb,alphabet)
```

Here `uhat` should come close to `u` and `threc` should approximate `th`.

There are two principles for equalization:

1. The channel is estimated from a training sequence, this can be done inside `viterbi`, and the channel estimate is specified to `viterbi` which implements the *Viterbi equalization* algorithm for recovering the unknown input.

2. The channel and the unknown input are estimated simultaneously with one of the two classes of blind equalizers, as is done in `blindeq` and `blindeqM`.

4.2 Equalization using the Viterbi algorithm

In this approach, it is assumed that an estimate of the channel is available. It might have been estimated using a known training sequence. The idea is then to enumerate all possible input sequences, and the one that produces the output most similar to the measured output is taken as the estimate.

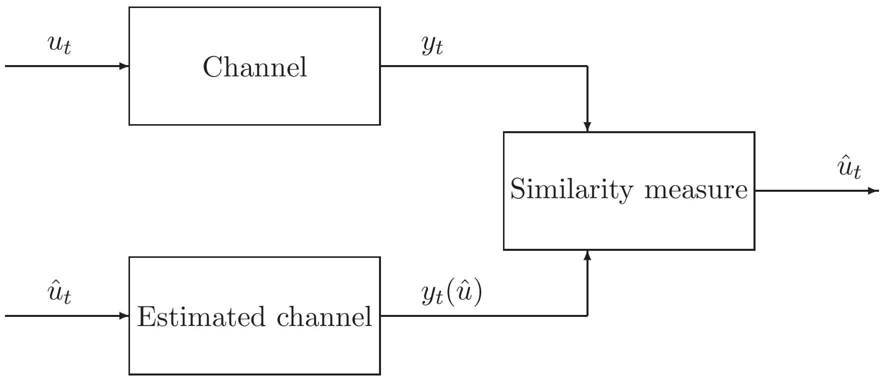


Figure 4.1 Viterbi equalization.

Technically, the similarity measure is the maximum likelihood criterion, assuming Gaussian noise. This is quite similar to taking the sum of squared residuals $\sum_t (y_t - y_t(\hat{u}))^2$ for each possible sequence $\hat{u}$. Luckily, not all sequences have to be considered. It turns out that only $n_s^{n_b}$ sequences have to be examined, where n_s is the number of symbols in the finite alphabet, and n_b is the channel order.

The *bit error rate* (*BER*) is defined as

$$\text{BER} = \frac{\text{number of non-zero } (u_t - \hat{u}_t)}{N} \tag{4.1}$$

where trivial phase shifts (sign change) and time delays of the estimate should be discarded. Use `u2ber` to compute this.

The example below shows different possibilities of guessing or estimating the channel model, and what the resulting bit error rate is.

```
% Simulate a channel
```

```
N=100;
randn('seed',1);
b=[1 1 1];
alphabet=[-1 1];
[y,u]=simchannel(alphabet,3,N,b);

% BER for bad channel estimate
bhat=[1 0 0];
uhat=viterbi(y,bhat,[-1 1],0.01);
ber=u2ber(u,uhat)
    0.2400
% BER for good channel estimate
bhat=[1 0.7 1.3];
uhat=viterbi(y,bhat,[-1 1],0.01);
ber=u2ber(u,uhat)
    0.0400
% BER when using a short training sequence
[uhat,thhat]=viterbi(y,3,[-1 1],0.01,u(1:5));
disp(thhat)
    0.9589
    0.9589
    0.9679
ber=u2ber(u,uhat)
    0
% Using a general input alphabet
alphabet=[-1 1 sqrt(-1) -sqrt(-1)];
[y,u]=simchannel(alphabet,b,N);
uhat=viterbi(y,b,alphabet);
ber=u2ber(u,uhat)
    0
```

4.3 Adaptive blind equalizer

Figure 4.2 shows a block diagram for the adaptive *blind equalizer* approach.

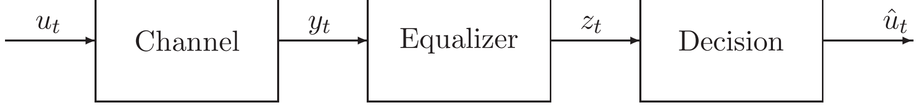


Figure 4.2 Adaptive blind equalizer.

The channel is modeled as a FIR filter

$$B(q) = b_t^1 q^{-1} + b_t^2 q^{-2} ... + b_t^{n_b} q^{-n_b} \tag{4.2}$$

The same model structure is used in the equalizer

$$C(q) = c_t^1 q^{-1} + c_t^2 q^{-2} ... + c_t^{n_c} q^{-n_c} \tag{4.3}$$

Both the channel and equalizer FIR parameters can either be constant (a row vector) or time-varying (a matrix). MIMO models are defined analogously, where the FIR model $B^{ij}(q)$ denotes the channel distortion from input i to output j.

The impulse response of the combined channel and equalizer, assuming FIR models, is

$$h_t = (b * c)_t$$

where $*$ denotes convolution. The best one can hope for is $h_t \approx m\delta_{t-k}$, where k is an unknown time-delay, and m with $|m| = 1$ is an unknown modulus.

The two most popular loss functions defining the adaptive algorithm are given below:

$$\mathrm{E}[(1 - z^2)^2] \qquad \textit{modulus restoral (Godard)} \tag{4.4}$$

$$\mathrm{E}[(\mathrm{sign}(z) - z)^2] \qquad \textit{decision feedback (Sato)} \tag{4.5}$$

These are minimized by an LMS like algorithm in `blindeq`.

The modulus and delay do not matter for the performance and can in applications be ignored. However, for evaluation using the bit error rate, these have to be determined. The algorithm implemented in `u2ber` estimates the covariance function (using FFT for maximum speed) $R(k) = \mathrm{E}(u_t \hat{u}_{t-k})$. The time delay is then estimated as $\hat{k} = \arg\max_k |\hat{R}(k)|$, and the maximum of $\hat{R}(k)$ can be taken as the corresponding estimate of the modulus.

Consider the case of input alphabet $u_t = \pm 1$. For succesful demodulation, assuming no noise on the measurements, it is enough that the largest component of h_t is larger than the sum of the other components. That is, $m_t > 0$, where

$$m_t = 2 - \frac{\sum(|h_t|)}{\max_t |h_t|}.$$

This measure is computed by `openeye`. If the equalizer is a perfect inverse of the channel (which is impossible for FIR channel and equalizer), then $m_t = 1$. The standard definition of an open-eye condition corresponds to $m_t > 0$, when perfect reconstruction is possible, if there is no noise. The larger m_t, the larger noise can be tolerated.

Look at the plot in Figure 4.3 for an example of how a blind equalizer improves the open-eye measure with time.

In the example below, the channel $0.3q^{-1} + 1q^{-1} + 0.3q^{-1}$ is simulated using an input sequence taken from the alphabet $[-1, 1]$. The initial equalizer parameters `b0` are quite critical for the performance. They should at least satisfy the open-eye condition $m_t > 0$ (here $m(0) \approx 0.5$).

```
N=1000;
randn('seed',1);
b=[0.3 1 0.3]
alphabet=[-1 1];
[y,u]=simchannel(alphabet,3,N,b);
y=y; %+0.1*randn(N,1);
nc=5;   % equalizer order for blindeq1

c0=[0 -0.1 1 -0.1 0]';

[theq1,uhat1,epsi1]=blindeq(y,nc,0.01,1,c0);
[theq2,uhat2,epsi2]=blindeq(y,nc,0.01,2,c0);

m1=openeye(b,theq1);
m2=openeye(b,theq2);
cdfigure(1);
h1=plot(m1,'-');  hold on
h2=plot(m2,'--'); hold off
title('Open eye condition')
legend([h1 h2],'Sato','Godard',3)


cdfigure(2,1);
plot([epsi1' epsi2'])

%cdfigure(3,2);
figure
h1=plot(1:3,b,'b-');                   hold on
h2=plot(1:5,theq1(:,N)','r:');
h3=plot(1:7,conv(b,theq1(:,N)'),'g--'); hold off
legend([h1 h2 h3],'Channel','Equalizer',...
       'Combined channel+equalizer',2)

ber1=u2ber(u,uhat1);
    0
ber2=u2ber(u,uhat2);
    0
```

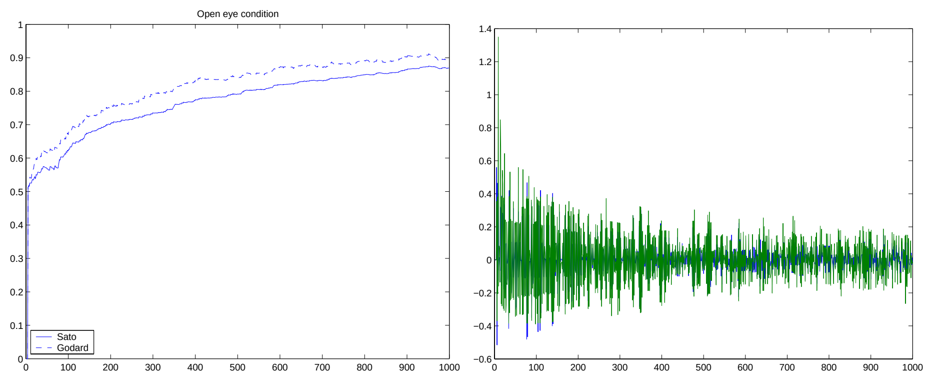


Figure 4.3 Open-eye measure and residuals for Sato's and Godard's algorithms.

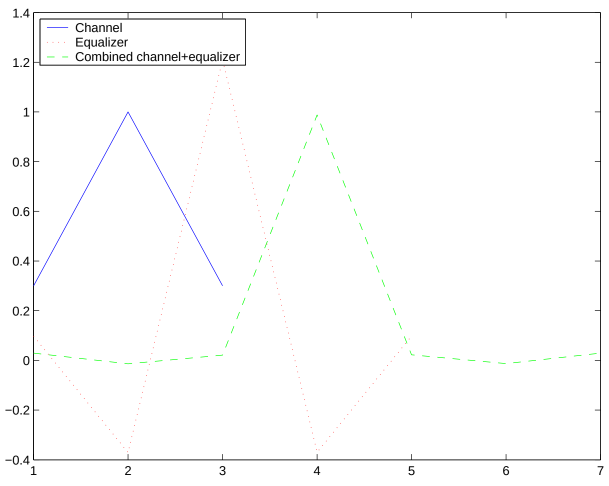


Figure 4.4 Impulse response of channel, equalizer and combined channel–equalizer (almost an impulse function).

4.4 Maximum likelihood sequence detection

This direct approach to channel equalization using *maximum likelihood sequence detection* does not involve an explicit equalizer, but rather a bank of channel estimators, each one matched to a particular input sequence. These input sequences are enumerated in a Viterbi like fashion.

The main difference to the Viterbi algorithm is that the channel is estimated using a standard adaptive filter *conditioned* on the input sequence. Another difference is that here all input sequences must be examined to get the optimal estimate, and a key question is how to approximate the estimate by only considering M sequences at any time. The most important design variable is M, while the noise variance should also be of the right order of magnitude (but this may not be so critical).

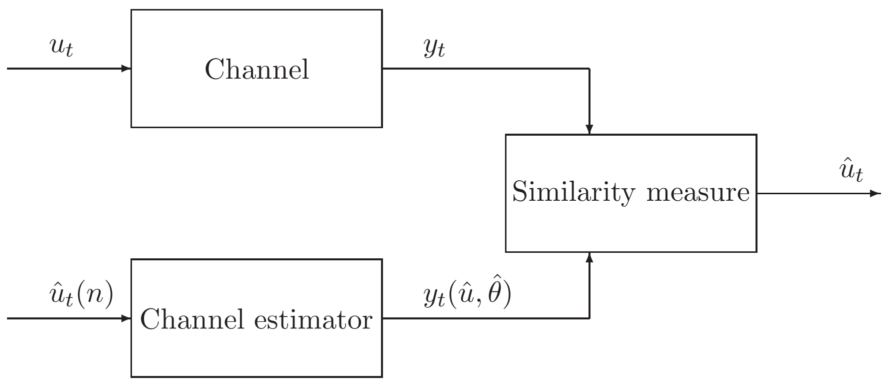


Figure 4.5 Maximum likelihood sequence detection.

Below a number of examples of increasing generality are given.

4.4.1 A time-invariant SISO channel

We start with the simplest case of a time constant SISO channel. Note that the order of the equalizer should be the same as the channel here, in contrast to the adaptive blind equalizer where $n_c > n_b$ generally.

```
N=100;
randn('seed',1);
rand('seed',1);
b=[0.5 1 0.7];
alphabet=[-1 1];
[y,u]=simchannel(alphabet,3,N,b);
y=y+0.1*randn(N,1); % Noise on y
nb=3;     % channel order for blindeqM

[thhat,uhat]=blindeqM(y,nb,[-1,1],0.001,16,.1,10,0);

[ber,uhatout,delay,m]=u2ber(u,uhat);
ber
ber =
    0
cdfigure(1);
plot(m*thhat') % Correct modulus
hold on
plot(ones(N,1)*b,'--')
hold off
```

Note how we use the estimated modulus to correct the sign of the estimated parameters. That is, if $\hat{u}(t) = u(t-\tau)$, then compare $-\hat{\theta}$ with θ_0. See Figure 4.6.

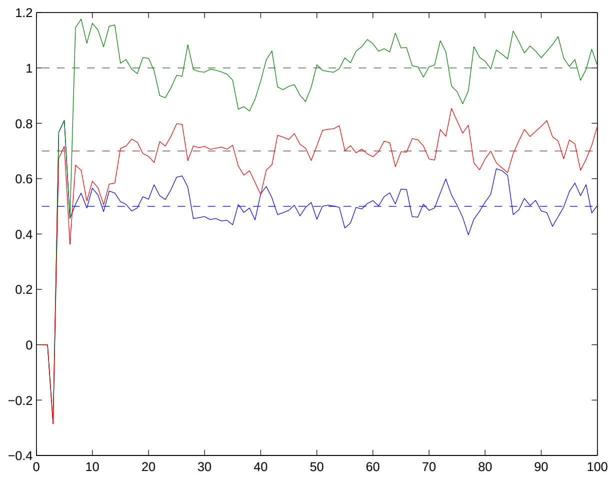


Figure 4.6 Estimated channel parameters by `blindeqM` for a time-invariant SISO channel.

4.4.2 A time-varying SISO channel

Next, we use `simchannel` to simulate a Rayleigh fading channel.

```
N=100;
[y,u,b]=simchannel([-1 1],2,N,[]);
nb=2;

[thhat,uhat]=blindeqM([y u],nb,[-1,1],.001,64,.1,10,0);

[ber,uhatout,delay,m]=u2ber(u,uhat);
ber
ber =
    0
cdfigure(1);
plot(m*thhat') % Correct modulus
hold on
plot(b')
hold off
```

Figure 4.7 illustrates the parameter tracking.

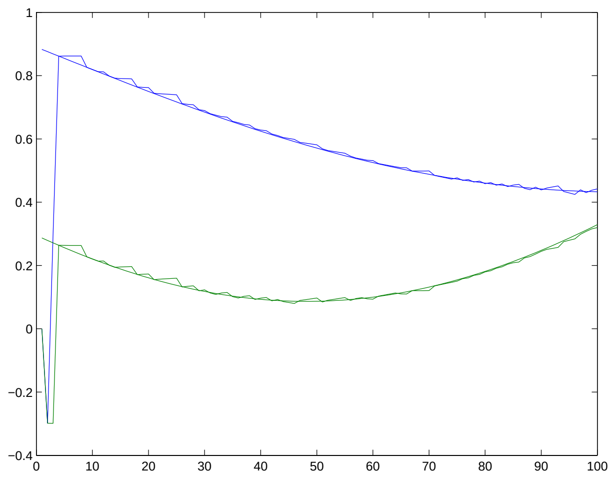


Figure 4.7 Estimated channel parameters by `blindeqM` on a noise-free time-varying channel.

4.4.3 A time-invariant SIMO channel

A SIMO model can be simulated as follows:

```
rand('seed',0);
b=[3 2 1   6 5 4];
a=1;
nb=[3;3];
N=200;
sigma=1;

[y,u]=simchannel([-1 1],nn,N,b(:));
y=y+sigma*randn(N,2);
[thhat,uhat,ber]=blindeqM(y,nb,[-1,1],sigma^2,64);

[ber,uhatout,delay,m]=u2ber(u,uhat);
ber
ber =
    0
cdfigure(2,1);
plot(m*thhat')
hold on
bdum=b';
b=bdum(:);
plot((ones(N,1)*b'))
hold off
axis([0 N 0 7])
title('Estimated (solid) and true (dashed) parameters')
```

Figure 4.8 shows how the six parameters are estimated.

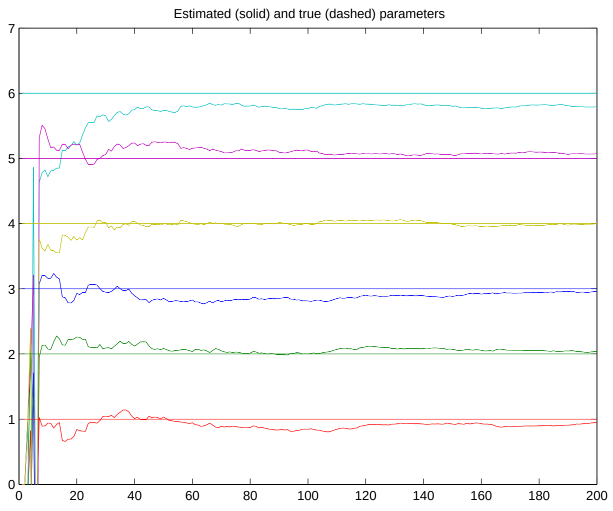


Figure 4.8 Estimated channel parameters by `blindeqM` on a noisy time-invariant SIMO channel.

4.4.4 A time-varying SIMO channel

A time-varying SIMO model is simulated below.

```
randn('seed',0);
N=100;
nb=2; ny=2; nu=1;
sigma=0.01;

randn('seed',1);
[y,u,th]=simchannel([-1 1],nb*ones(ny,nu),N,[]);
y=y+sigma*randn(N,ny);

[thhat,uhat]=blindeqM(y,[2;2],[-1,1],sigma^2,64,.1,10,0);
[ber,uhatout,delay,m]=u2ber(u,uhat);
ber
ber =
    0
cdfigure(3,2);
plot(m*thhat','-')
hold on
plot(th','--')
hold off
```

Tracking of the four time-varying parameters are illustrated in Figure 4.9.

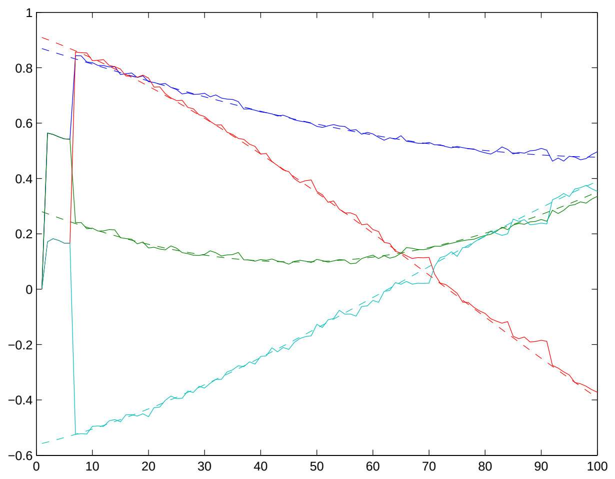


Figure 4.9 Estimated channel parameters by `blindeqM` on a noisy time-varying SIMO channel.

4.4.5 A time-invariant MIMO channel

Finally, the most general MIMO model is simulated.

```
N=64;
sigma=0.1;

th=[1 0.9   0.8 0.7    0.6 0.5   0.4 0.3];
nb=2;
ny=2;
nu=2;

[y,u]=simchannel([-1 1],nb*ones(ny,nu),N,th);
y=y+sigma*randn(N,ny);

[thhat,uhat]=blindeqM(y,nb*ones(ny,nu),[-1,1]);
[ber,uhatout,delay,m]=u2ber(u,uhat);
ber
ber =
     0
disp([th' thhat(:,N)])

cdfigure(4,3);
plot(m*thhat')
hold on
plot(ones(N,1)*th,'--')
hold off
axis([0 N 0.2 1.1])
```

See Figure 4.10 for the resulting plot.

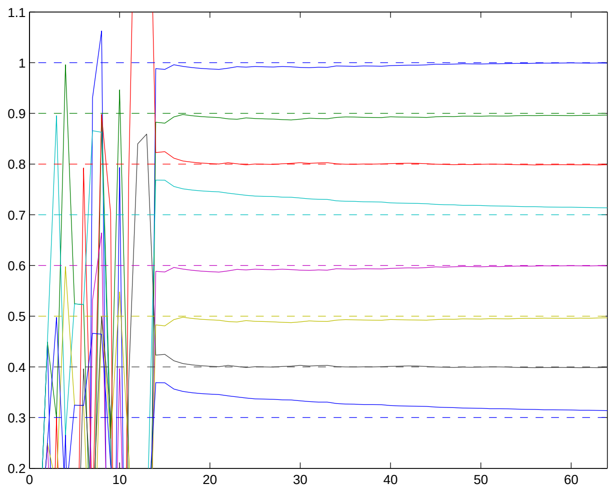


Figure 4.10 Estimated channel parameters by `blindeqM` on a noisy time-invariant MIMO channel.

4.4.6 Remarks

From the examples in this section, one may conclude that everything is simple and works perfectly. All the plots show perfect channel reconstruction and the BER is (almost) zero. However, the main practical problems are as follows:

- For `blindeq` , the problem is to find a good initialization. It is almost impossible to just guess a value.
- For `blindeqM` , the complexity grows in the channel order n_b and the size of the alphabet n_s as $n_s^{n_b}$, which may be prohibitive for certain applications.

Another problem for MIMO channels is to decide which input belongs to which output. This may cause an estimate which is repeatingly switching between the different possibilities, where the probability for one combination is about the same as the probability for the other input–output combination.

5 Target tracking

Purpose	Syntax
Simulation	`[y,x]=simchange (N,nnn,jumptimes,TH)`

Tracking	`[xhat]=adkalman (y,nnn,options)`
	`[xhat,jumptimes]=detect1 (y,nnn,options)`
	`[xhat,jumptimes]=detectM (y,nnn,options)`
	`[xhat,jumptimes]=imm (y,nnn,options)`

Plot	`radarplot (y,xhat,x)`
Model	`nnnplane (t,x,f,flag)`
Demo	`demo_tracking`

5.1 Introduction

For target tracking, the usual Kalman filter applies, but there are particular algorithms suggested, like IMM. There are also many standard motion models. The main contribution of this toolbox to target tracking, is a general motion model `nnnplane` . It implements a variety of models used in the area, controlled by the *global* parameter `NNNmethod`.

The state variables are defined in continuous time as as subset of the following:

- x_1 and x_2 denote the cartesian position in the horizontal plane.
- v_1 and v_2 denote the cartesian velocity in the horizontal plane.
- v denotes the speed.
- h denotes the heading angle, so $v_1 = v\sin(h)$ and $v_2 = v\cos(h)$.
- $\omega = \dot{h}$ denotes the turn rate.

The state space model must be transformed to discrete time before the Kalman filter applies. For non-linear models, there are two alternatives, either to linearize the non-linear model and then apply standard sampling of state space models, or to try to sample the continuous time non-linear model to a discrete time non-linear model, which is then linearized. We refer to these principles as *discretized linearization* and *linearized discretization*, respectively. The global switching parameter `NNNmethod` can now be defined as follows:

1. Four state linear model $x = (x^{(1)}, x^{(2)}, v^{(1)}, v^{(2)})^T$.
2. Four state linear model as above with a time-varying and state dependent covariance matrix $Q(x_t)$, corresponding to velocity noise mainly in lateral direction, as will be explained below.
3. Six state linear model with $x = (x^{(1)}, x^{(2)}, v^{(1)}, v^{(2)}, a^{(1)}, a^{(2)})^T$.
4. Six state linear model as above with a time-varying and state dependent covariance matrix $Q(x_t)$, corresponding to velocity noise mainly in lateral direction, as will be explained below.
5. Five state coordinated turn model, cartesian velocity, linearized discretization, with $x = (x^{(1)}, x^{(2)}, v^{(1)}, v^{(2)}, \omega)^T$ (ω is turn rate).
6. Five state coordinated turn model, polar velocity, linearized discretization, with $x = (x^{(1)}, x^{(2)}, v, h, \omega)^T$ (h is heading angle).
7. Five state coordinated turn model, cartesian velocity, discretized linearization.
8. Five state coordinated turn model, polar velocity, discretized linearization.

The first five ones seem to be the most common ones in applications. A covariance matrix Q corresponding to, for example, 100 times larger maximal acceleration in lateral direction compared to longitudinal direction ($q_v = 0.01 q_w$) is given by

$$\begin{aligned}
\theta &= \arctan(x^{(4)}/x^{(3)}) \\
Q &= \begin{pmatrix} q_v \cos^2(\theta) + q_w \sin^2(\theta) & (q_v - q_w)\sin(\theta)\cos(\theta) \\ (q_v - q_w)\sin(\theta)\cos(\theta) & q_v \sin^2(\theta) + q_w \cos^2(\theta) \end{pmatrix} \\
B_v &= \begin{pmatrix} T^2/2 & 0 \\ 0 & T^2/2 \\ T & 0 \\ 0 & T \end{pmatrix} \quad \text{or} \quad B_v = \begin{pmatrix} T^3/3 & 0 \\ 0 & T^3/3 \\ T^2/2 & 0 \\ 0 & T^2/2 \\ T & 0 \\ 0 & T \end{pmatrix}.
\end{aligned}$$

The reason for introducing an ugly global variable for switching purposes is solely to keep the Simulink-like syntax. See Section 8.9 in the text book for a thorough treatment of model definitions and their extended Kalman filters. The flexibility of `nnnplane` is illustrated by the fact that the same

Kalman filter adkalman applies to all of these non-linear and linear models. All adkalman needs to know is defined inside the model.

Change detection means in this context maneuver detection, and here the alternatives are detect1 and detectM . radarplot gives a nice illustration of the result.

The only taylored function for this problem is imm (*Interactive Multiple Model*), which is one of the most cited method in this area. Besides the extended Kalman filter, it has become a standard in the field. The main difference between detectM and imm , which algorithmically are very similar, is that the former prunes different hypotheses of maneuvers, while the latter merges them.

5.2 Simulation

A flight trajectory with maneuvers is simulated using abrupt changes in the turn rate of a fifth order non-linear state space model.

```
N=70; % Simulation length
lam=1;
nnn='nnnplane';  % Motion model
global NNNmethod % with global switch
NNNmethod=5;     % Five state model for simulation
u=zeros(N,1);    % No input
th0=[-0.025;...  % Define state changes in turn rate
     +0.025;...
     0.04;...
     -0.04;...
     0.039;...
     -0.039]';
jumps=[10 16 35 41 55 61];  % Define true jump times
  % Start a simulation
  [y,x0]=simchange(N,nnn,jumps,th0,lam,'pulse');
```

An illustration of the experiment is obtained as follows:

```
p1=radarplot(y,x0); % Nice illustration
legend(p1,'Measured position','True position')
```

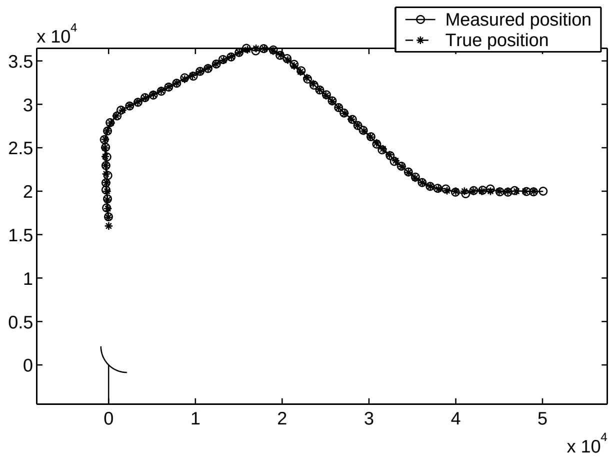


Figure 5.1 Target tracking: simulated trajectory

5.3 Kalman filtering

Default values work fine for the Kalman filter as illustrated below:

```
NNNmethod=1; % Fourth order linear model for estimation
[xhat]=adfilter(y,nnn,1); % Kalman filtering
radarplot(y,xhat);
title('Kalman filter')
```

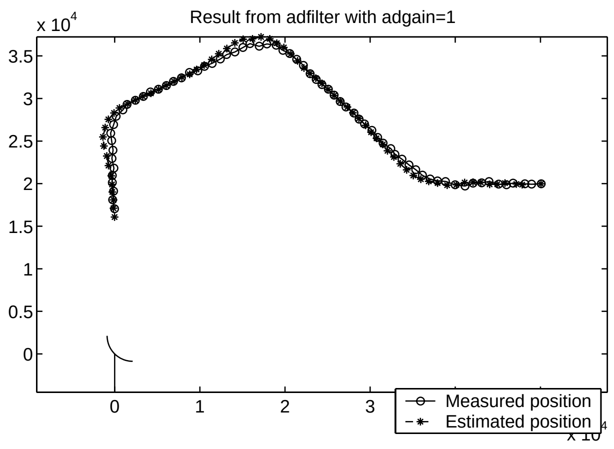


Figure 5.2 Target tracking: position predictions from the Kalman filter.

Note that it is not possible to change the initial conditions in `adfilter`. To get full freedom and alternative implementations, use the Kalman filter in `adkalman` .

```
[xhat1]=adkalman(y,nnn,1,1); % stationary Kalman filter
[xhat2]=adkalman(y,nnn,1,2); % Kalman filter
[xhat3]=adkalman(y,nnn,1,3); % Kalman filter square root
[xhat4]=adkalman(y,nnn,1,4); % Kalman smoother

rmse0=sqrt(sum( (x0(1:2,:)-xhat0(1:2,:)).^2 ));
rmse1=sqrt(sum( (x0(1:2,:)-xhat1(1:2,:)).^2 ));
rmse2=sqrt(sum( (x0(1:2,:)-xhat2(1:2,:)).^2 ));
rmse3=sqrt(sum( (x0(1:2,:)-xhat3(1:2,:)).^2 ));
rmse4=sqrt(sum( (x0(1:2,:)-xhat4(1:2,:)).^2 ));

figure
plot([rmse1' rmse2' rmse3' rmse4'])
legend('adkalman: stationary','adkalman: time-varying',...
       'adkalman: square-root','adkalman: smoothing');
axis([0 N 600 1600])
```

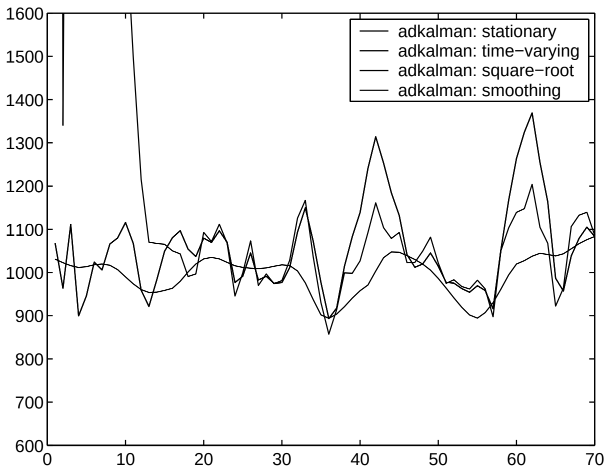


Figure 5.3 Target tracking: position error for different variants of the Kalman filter.

```
figure
plot(x0(1,:),x0(2,:),'-d')
hold on
plot(xhat1(1,:),xhat1(2,:),'g-')
plot(xhat2(1,:),xhat2(2,:),'g--')
plot(xhat3(1,:),xhat3(2,:),'r-.')
plot(xhat4(1,:),xhat4(2,:),'m:')
hold off
axis('equal')
  legend('true','adkalman: time-varying','adkalman: stationary',...
         'adkalman: square-root','adkalman: smoothing');
axis([1 2.5 3 4]*1e4)
```

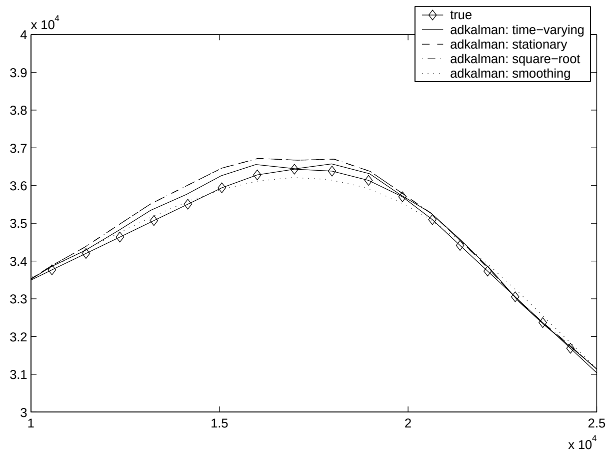


Figure 5.4 Target tracking: tracking a manoeuvre for different variants of the Kalman filter.

Optimization of the adaptation gain is straightforward:

```
adg=[1e-2 1e-1 1e0 1e1 1e2 1e3 1e4];
leg=[];
for i=1:length(adg)
  [xhat]=adkalman(y,nnn,adg(i),[2 1]); % Kalman predictor
  rmse(:,i)=sqrt(sum( (x0(1:2,:)-xhat(1:2,:)).^2 ))';
  leg=str2mat(leg,num2str(adg(i)));
end
leg(1,:)=[];
rmse0=sqrt(sum( (y'-x0(1:2,:)).^2 ))';
figure
p1=plot([rmse]);
hold on, p2=plot(rmse0,'r--'); hold off
leg=str2mat(leg,'Ad-hoc filter');
legend([p1;p2],leg,2)
xlabel('Time [samples]')
ylabel('Predictive RMSE(t)')
```

A plot of RMSE versus adaptation gain is instructive.

```
semilogx(adg,sqrt(sum(rmse.^2)))
hold on,
semilogx(adg,sqrt(sum(rmse0.^2))*ones(size(adg)),'r--');
hold off
xlabel('Adaptation gain')
ylabel('Predictive RMSE')
```

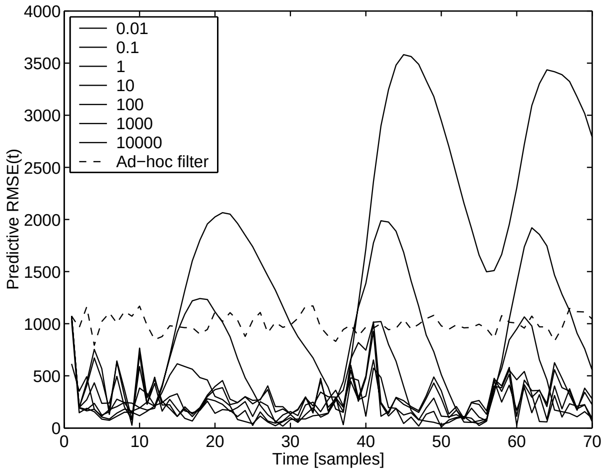


Figure 5.5 Target tracking: position prediction error using different Q scalings. The *ad-hoc* filter uses y_t as the position estimate.

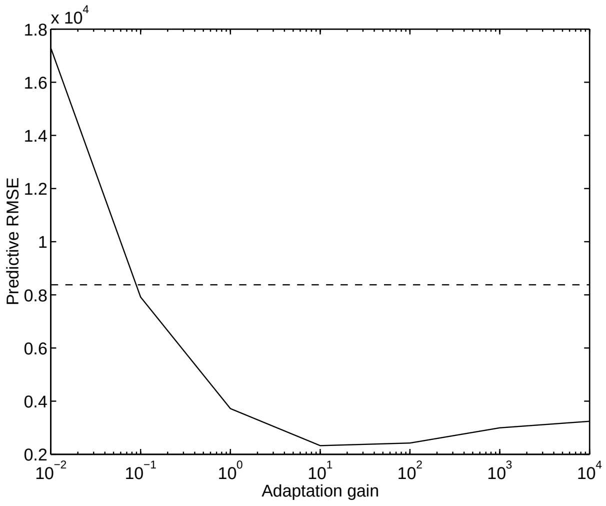


Figure 5.6 Target tracking: time-averaged RMSE as a function of the Q scaling. Dashed line is the RMSE for the *ad-hoc* filter using y_t as the position estimator.

Finally, the different state coordinates implemented in `nnnplance` can be compared in the Kalman filter.

```
leg=[];
for i=1:5
  NNNmethod=i; % State coordinates
  [xhat]=adkalman(y,nnn,1,[2 1]); % Kalman predictor
  rmse(:,i)=sqrt(sum( (x0(1:2,:)-xhat(1:2,:)).^2 ))';
  leg=str2mat(leg,num2str(i));
end
leg(1,:)=[];
plot([rmse])
legend(leg,2)
xlabel('Time [samples]')
ylabel('Predictive RMSE(t)')
```

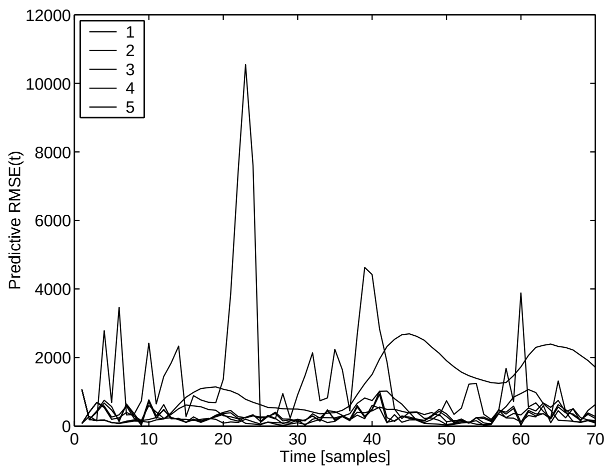


Figure 5.7 Target tracking: position prediction error using the diffferent state coordinates in `nnnplane`, listed on page 5.1. Note that Q is not optimized, so one cannot say which state coordinate system is best from this plot.

5.4 Change detection with whiteness test

The basic whiteness test idea, where a stopping rule monitors a distance measure, and the alarm kicks on the adaptiation gain, is implemented in `detect1`. We can redo the estimation, and see whether the detector improves tracking during the manoeuvres.

```
h=5;
```

```
nu=0.5;
[xhat,jhat,xseg,gt,ta]=detect1(y,nnn,1,[2 1e10 nu],1);
[xhat1,jhat1,xseg1,gt,ta]=detect1(y,nnn,1,[2 h nu],1);
jumps,jhat1=jhat1'
jumps =
  10    16    35    41    55    61
jhat1 =
  15    39    64
figure
plot(x0(1,:),x0(2,:),'-',...
     xhat(1,:),xhat(2,:),'--',...
     xhat1(1,:),xhat1(2,:),'-.')
hold on
plot(xhat1(1,jhat1),xhat1(2,jhat1),'o')
hold off
legend('True trajectory','Kalman filter',...
       'KF with change detector')
axis('equal')
```

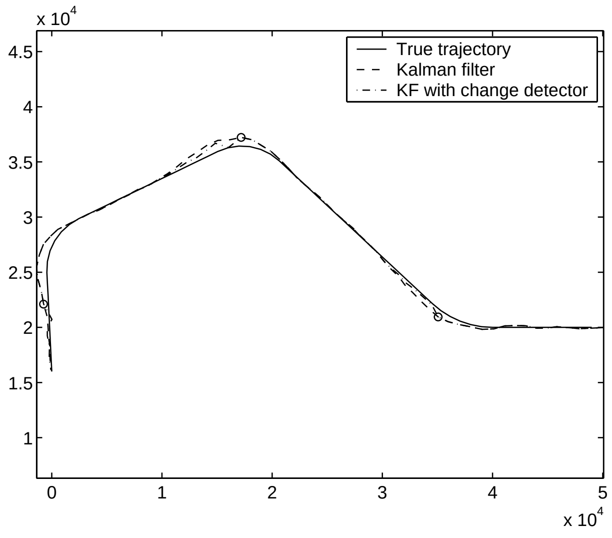


Figure 5.8 Target tracking: `detect1` improves position tracking during manoeuvres due to an increase in adaptation gain when the CUSUM test gives an alarm.

As for `adkalman`, changing the design parameters in the Kalman filter (x_0, P_0, Q, R), requires editing the model. That is, make a local copy of `nnnplane` and edit these definitions.

The estimation error of the Kalman filter with and without change detector is plotted next.

```
subplot(211)
```

```
segplot([x0(1,:)'-xhat(1,:)' x0(1,:)'-xhat1(1,:)'],jhat1);
ylabel('Error in x_1')
xlabel(' '), title(' ')
subplot(212)
segplot([x0(2,:)'-xhat(2,:)' x0(2,:)'-xhat1(2,:)'],jhat1);
ylabel('Error in x_2')
xlabel('Time [samples]')
title(' ')
legend('Kalman filter','KF with change detector',4)
```

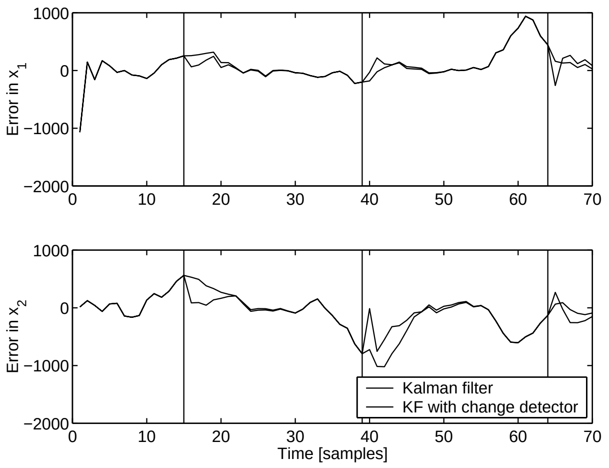


Figure 5.9 Target tracking: position error in x_1 and x_2 using a Kalman filter with and without change detection. Note the faster recovery with detection.

5.5 Change detection with filter banks

Here, filter banks is applied, where each filter is matched to a specific manoeuvre hypothesis. The fourth order linear motion model is used. First, `detectM` tests manoeuvres by increasing Q, allowing momentarily faster tracking.

```
NNNmethod=1;
[xhat,jhat,xseg,gt,ta]=detectM(y,nnn,[1 10],1e-15,1);
[xhatM,jhatM,xsegM,gt,ta]=detectM(y,nnn,[1 10],0.5,[]);
jumps,jhatM=jhatM'
jumps =
    10    16    35    41    55    61
jhatM =
    14    38    58
figure
```

```
plot(x0(1,:),x0(2,:),'-',...
     xhat(1,:),xhat(2,:),'--',...
     xhatM(1,:),xhatM(2,:),'-.')
hold on
plot(xhatM(1,jhatM),xhatM(2,jhatM),'o')
hold off
legend('True trajectory','Kalman filter','Multiple model KF')
axis('equal')
```

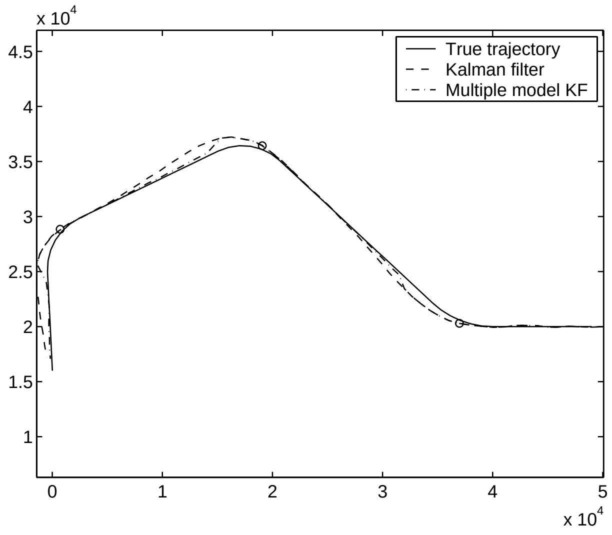


Figure 5.10 Target tracking: tracking with `detectM` gives in this example similar detection performance to `detect1`. Note the improved recovery after the alarms in Figure 5.11 though.

We get one estimated jump in each maneouvre, where theoretically there should be two. The position plot in Figure 5.10 is similar to the one in Figure 5.8, so `detect1` and `detectM` perform similarly regarding detection. An estimation error plot is generated below. Here one can see that recovery after a detected change is much better for `detectM`, due to that it remembers old data before the change, and can adapt accordingly.

```
subplot(211)
plotfix
segplot([x0(1,:)'-xhat(1,:)' x0(1,:)'-xhatM(1,:)'],jhatM);
ylabel('Error in x_1')
xlabel(' '), title(' ')
axis([0 N -2000 2000])
subplot(212)
plotfix
segplot([x0(2,:)'-xhat(2,:)' x0(2,:)'-xhatM(2,:)'],jhatM);
```

```
ylabel('Error in x_2')
xlabel('Time [samples]')
title(' ')
axis([0 N -2000 2000])
legend('Kalman filter','Multiple model KF',3)
```

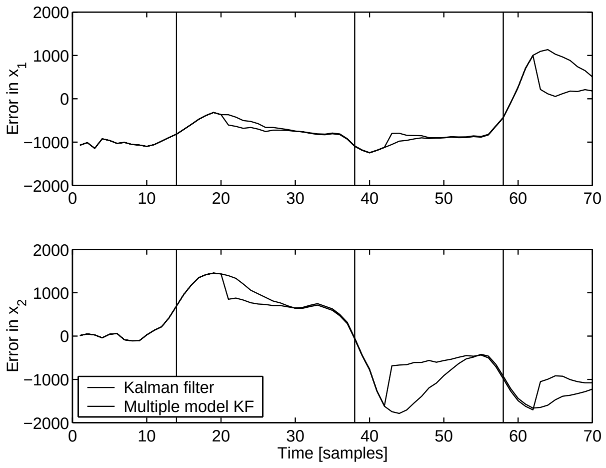


Figure 5.11 Target tracking: tracking with `detectM` gives superior recovery after an alarm compared to `detect1` in Figure 5.9.

IMM is based on a model with heading angle as one state. We can compare `imm` with `detectM` as follows:

```
NNNmethod=5;      % Coordinated turn model
e=randn(N,2);     % Measurement noise
yn=y+500*e;       % Add extra noise
[xhat,jhat,xseg,gt,ta]=detectM(yn,nnn,[1 1e9],1e-15,1);
[xhatM,jhatM,xsegM,gt,ta]=detectM(yn,nnn,[1 1e9],0.5,[]);
[xhatimm]=immnew(yn,nnn);
jumps,jhatM=jhatM'
figure
plotfix
plot(x0(1,:),x0(2,:),'-',...
     xhat(1,:),xhat(2,:),'--',...
     xhatimm(1,:),xhatimm(2,:),':',...
     xhatM(1,:),xhatM(2,:),'-.',...
     yn(:,1),yn(:,2),'o')
hold on
plot(xhatM(1,jhatM),xhatM(2,jhatM),'o')
hold off
legend('True trajectory','Kalman filter',...
  'IMM','Multiple model KF','Measurements')
```

```
axis('equal')
```

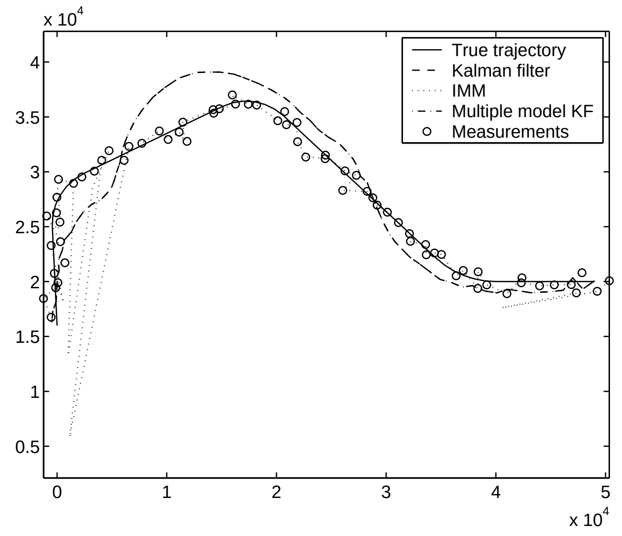


Figure 5.12 Target tracking: comparison of `IMM`, `detectM` and `adkalman`.

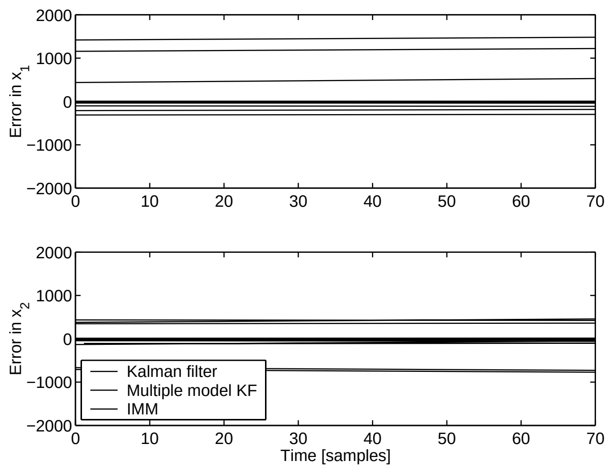


Figure 5.13 Target tracking: comparison of `IMM`, `detectM` and `adkalman`.

6 Applications

6.1 Fuel Monitoring

The following application illustrates the use of change detection for improving signal quality. The data consist of measurements of instantaneous fuel consumption available from the electronic injection system in a Volvo 850 GLT used as a test car.[1] The raw data are quite noisy and need some kind of filtering before being displayed to the driver at the dashboard. There are two requirements on the filter:

- Good attenuation of noise is necessary in order to being able to tune the accelerator during cruising.
- Good tracking ability. Tests show that fuel consumption very often changes abruptly, especially in city traffic.

These requirements are contradictory for standard linear filters. Figure 6.1 shows the raw data together with a filter implemented by Volvo[2]. Volvo uses a quite fast low-pass filter to get good tracking ability and then quantize the result to a multiple of 0.3 to attenuate some of the noise. However, the quantization introduces a difficulty when trying to minimize fuel consumption manually and the response to changes could be faster.

The CUSUM algorithm, Brandt's GLR test and the ML sequence estimator are applied. These algorithms are only capable to follow abrupt changes. For incipient changes, the algorithm will give an alarm only after the total change is large or after a long time. In both algorithms, it is advisable to include data forgetting in the parameter estimation to allow for a slow drift in the mean of the signal.

Table 6.1 shows how some change detection algorithms perform on this signal. As a starter, one can try

```
load fuel
```

[1]Thanks to Volvo AB for lending the test car.

[2]This is not exactly the same filter as Volvo uses, but the functionality is the same.

```
N=length(y);
t=0.2:0.2:0.2*N;
threc=adfilter(y,-1,0.9);
threc1=detect1(y,-1,2,[2 3 2]);
threc2=detect2(y,-1,1,[2 20 3],3);
threcM=detectM(y,-1,[1 3],[],5);
```

which generates information for Table 6.1. The demo `demo_fuel` gives more examples.

Method	Design parameters	$\hat{n}$	MDL	kFlops
RLS	$\lambda = 0.9$, quant $= 0.3$	–	–	19
CUSUM	$h = 3,\ \nu = 2$	14	8.39	20
Brandt's GLR	$h = 20,\ \nu = 3,\ L = 3$	13	8.40	60
Multiple filter ML	$\sigma^2 = 3$	14	8.02	256

Table 6.1 Simulation result for fuel consumption

Figure 6.2 shows the result using the recursive ML detector as a non-linear filter, using 5 parallel filters. Compared to the existing filter, the tracking ability has improved slightly and, more importantly, the accuracy gets better and better in segments with constant fuel consumption.

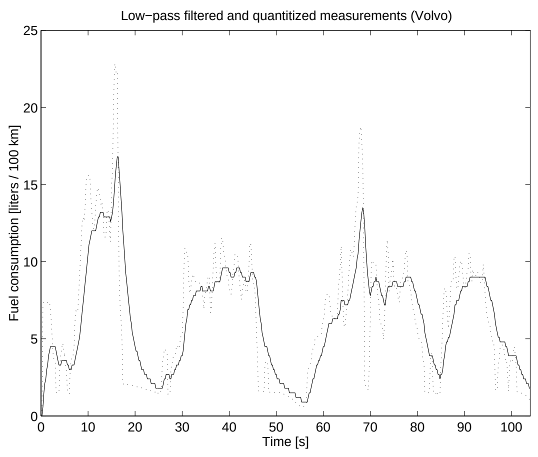


Figure 6.1 Measurements of fuel consumption and Volvo's proposed filter.

6.2 EEG signals

A human and a set of rat EEG signals are investigated. A full demo is provided in `demo_eeg`.

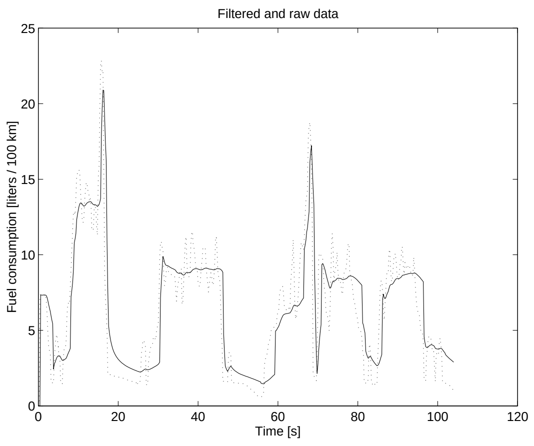


Figure 6.2 Filtering with recursive ML detection using `detectM`.

6.2.1 Human health diagnosis

The data are measured from human ocdipital area.[3] Before a certain time, t_b, the lights are on in test room and the test person is looking at something interesting. The neurons are processing information in visual cortex, and only noise is seen in measurements. When the light is turned off, the visual cortex is at rest. The neuron clusters start 10 Hz periodical "rest rythm". The delay between t_b and the actual time when the rhythm starts varies strongly. It is believed that the delay correlates with e.g. Alzheimer decease, and methods for estimating the delay would be useful in for example medicin tests. An fourth order AR model is used in the multi-filter approach using `detectM`.

```
load eeg_human
[threc,jhat,ths]=detectM(Y(:,1),4);jhat
jhat =
    429
h1=cdfigure(1);
segplot(Y(:,1),jhat)
title('EEG signal on a human when lights turned on at time 387')
```

The plot shows a rather precise location of the change point. That is, medical diagnosis can be automized.

[3]Thanks to Pasi Karjalainen, Dept. of Applied Physics, University of Kuopio, Finland. The original data have been rescaled.

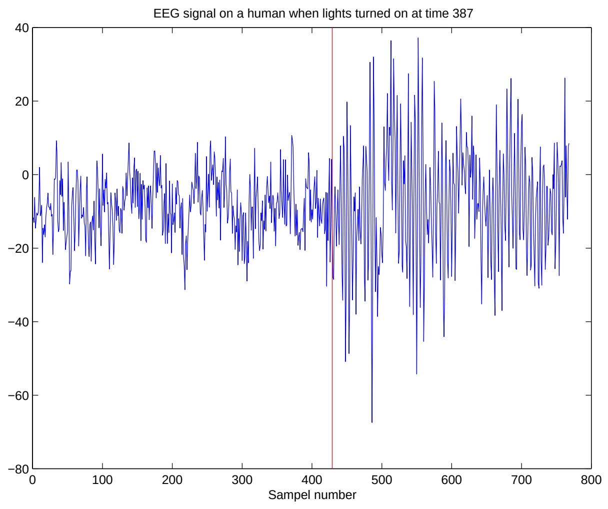


Figure 6.3 Human EEG signal with detection change point.

6.2.2 Rat EEG

The EEG signal here is measured on a rat.[4] The goal is to classify the signal into segments of so called "spindles" or back ground noise. Currently researchers are using narrow band filter and they have some treshold for the output power of that. That method gives

```
[1096 1543 1887 2265 2980 3455 3832 3934].
```

This is a signal where the ML estimator for changes in noise variance can be applied. The `multihyp` function is used, but the `detectM` function is also applicable.

```
load eeg_rat
plot(y)
[th,lam,epsi]=adfilter(w1,0,.97);
[jtime,jtype,ths,lams]=multihyp(w1,0,300,[0 1 0]);
[jtime;jtype]

ans =
    754 1058 1358 1891 2192 2492 2796 3098 3398 3699
      2    2    2    2    2    2    2    2    2    2

h3=cdfigure(3,2);
plot([lam lams'])
```

[4]Thanks to Pasi Karjalainen, Dept. of Applied Physics, University of Kuopio, Finland. The original data have been rescaled.

It can be remarked that the changes are hardly abrupt, but rather incipient, for this signal. Note also that the type of change will always be 2 (meaning variance change) since the masking option `[0 1 0]` is used.

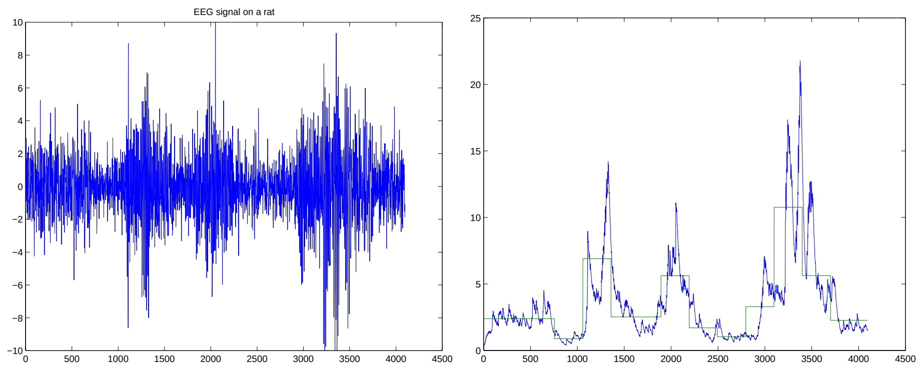


Figure 6.4 Rat EEG

6.3 Paper refinery

The data here are process data from a refiner in the paper industry. [5] The interesting signal is a raw engine power signal in kW which is extremely noisy. The refinery engine grinds tree fibers for paper production, and it is used to compute the reference value in a control system for quality control. The requirements on the power filter are:

- The noise must be considerably attenuated to be useful in the feedback loop.
- It is very important to quickly detect abrupt power decreases to be able to remove the grinding discs immediately and avoid physical disc damages.

That is, both tracking and detection is important, but for two different reasons. RLS provides som useful information, as seen from the left plots in Figure 6.5.

- There are two segments where the power clearly decreases quickly. Furthermore, there is a starting and stopping transient that should be detected as change times.
- The noise level is fairly constant (0.05) during the observed interval.

[5]Thanks to Thore Lindgren at Sunds Defibrator AB, Sundsvall, Sweden. The original data have been rescaled.

The ML recursive sequence estimator has been applied, with the assumption of constant and known noise variance:

```
load defibrator
y=y/1100;  % y=power
[thff,lamff]=adfilter(y,-1,0.98); %implemented filter
figure
subplot(211)
plot(thff)
subplot(212)
plot(lamff)
[jhat,thseg,lamseg,threc]=segm(y,-1,[1 0.02]);
jhat
figure
subplot(311)
segplot([y],jhat)
subplot(312)
plot([threc' thseg'])
title('Recursive and smoothed power estimate')
subplot(313)
plot([filtpower/1100 threc'])
title('Filtpower and recursive power estimate')
```

Comments:

- It is generally advisable to use known noise variance when it is known.
- It is also advisable to scale the signal so the parameter is in the order of 1.
- Default design parameters are used.

The resulting power estimate (both recursive and smoothed) are compared to the filter implemented by Sund in the right plots of Figure 6.5.

Both tracking and noise rejection are improved.

6.4 Poisson process of photon arrivals

Tracking the brightness changes of galactical and extragalactical objects is an important subject in astronomy. The data examined here are obtained from X-ray and γ-ray observatories.[6] The signal contains even integers represent-

[6]Thanks to Dr. Jeffrey D. Scargle at NASA for providing these data. The original data have been rescaled.

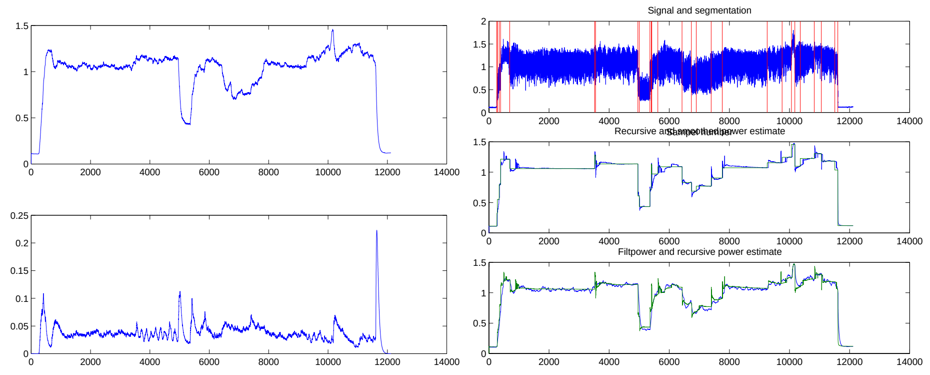


Figure 6.5 Power signal in Sund's defibrator. Left: Low-pass filtered signal using RLS with forgetting factor 0.98 (upper plot). The estimated noise variance (lower plot) indicates that it is fairly constant. Right: Original signal (upper plot) where estimated change times are marked, Sund's filter and smoothed ML estimate (middle plot), and finally, Sund's filter and recursive ML estimate (lower plot).

ing the time of arrival of the photon, in units of microseconds, where the fundamental sampling interval of the instrument is 2 microseconds. This is a typical queue process where a Poisson process is plausible. A Poisson process can easily be converted to a *change in the mean* model by taking the time difference between the arrival times. By definition, these will be independently exponentially distributed (modulo quantization errors). That is, the model is

$$y_t = \theta_t + e_t, \quad p(y_t) = \frac{1}{\theta_t} e^{-\frac{y_t}{\theta_t}}, \quad \mathrm{E}(y_t) = \theta_t.$$

Thus, e_t is white noise. There is no problems in modifying the algorithms with respect to any distribution of the noise. The distribution essentially influences only the line where the likelihood is computed. This is however a perfect illustration of the robustness of the algorithms with respect to incorrectly modeled noise distribution. The following lines compute the RLS estimate and ML segmentation of the changing mean model:

```
load photons
y=diff(y1)/30;
thrls=adfilter(y,-1,0.99);
[jhat,thseg,lamseg,threc]=segm(y/30,-1,[1 1],0.01,[10 3 8]);
jhat
jhat =
    [3001, 5012, 6509, 11674]
cdfigure(1)
thplot(thrls,thseg,threc)
cdfigure(2,1)
segplot(decimate(y,100),jhat/100)
```

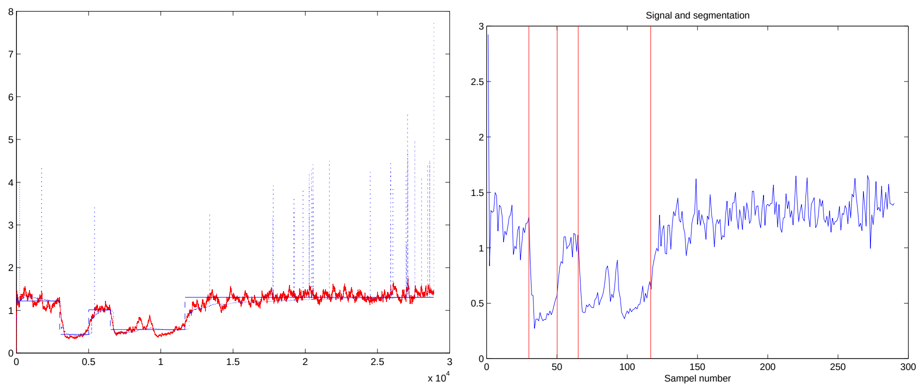


Figure 6.6 Time difference between photon arrival times for astronomical data

Comments:

- A typical feature of the exponential distribution is frequent large outliers. This is the reason why the estimated parameter from RLS contains spikes.
- For the same reason, the design parameters in the MAP segmentation need to be tuned. Here we have increased the minimum segment length to 8. With the default settings, outliers will be put into very short segments.
- Note that it is advisable to scale the signal so θ is in the order of 1.

We can play around with the design parameter q and see what segmentations it will lead to. Table 6.2 shows that more change points are added in the end of the signal as we let the probability for a change increase.

q	Change times
0.001	3001 5012 6509 11674
0.01	3001 5012 6509 11674
0.02	3001 5012 6509 11674
0.05	3001 5012 6509 11674 27071 27099
0.1	3001 5012 6509 11674 27071 27099
0.2	3001 5012 6509 11674 20266 20277 27071 27102
0.5	3001 5012 6509 11674 20266 20277 25884 27071 27102

Table 6.2 Change times as a function of change probability q

6.5 Maneuver detection for a driven path

These data were collected from test drives with a Volvo 850 GLT using sensor signals from the ABS system.[7] From these data, the position of the car can be calculated very roughly using dead-reckoning.

The heading angle within each segment is modeled as a first order polynomial in t,

$$h(t) = c_1 + c_2 t,$$

and the `detectM` function is used with 10 parallel filters and required accuracy of the prediction errors corresponding to a variance of 0.05.

```
load path
N=length(x1);
h1=cdfigure(1);
plot(v)
title('Velocity')
h2=cdfigure(2,1);
plot(x1,x2)
axis('equal')
xlabel('Meter')
ylabel('Meter')
title('Position')
text(x1(1),x2(1)-50,'t=0 s')
text(x1(N)+50,x2(N)+50,'t=225 s')

phi=phase(i*diff(x2)'+diff(x1)')';
N=length(phi);
M=10;
nnn=-2;
t=(1:N)';
t0=clock;
[jhat,thseg]=segm([phi ones(size(t)) t],nnn,[1 0.05],.05,M);
consumedtime=etime(clock,t0)
estpath=thseg'.*[ones(size(t)) t]*[1;1];

figure(h2)
plot(x1,x2,'-',x1(jhat),x2(jhat),'o')
axis('equal')
xlabel('Meter')
ylabel('Meter')
title('Position')
text(x1(1),x2(1)-50,'t=0 s')
```

[7]Thanks to Volvo AB for lending the test car.

```
text(x1(N)+50,x2(N)+50,'t=225 s')

figure(h1)
plot(t,phi,'-',jhat,phi(jhat),'o',t,estpath,'--')
title('Actual and segmented heading angle')
xlabel('Time [s]')
ylabel('Heading angle [rad]')
```

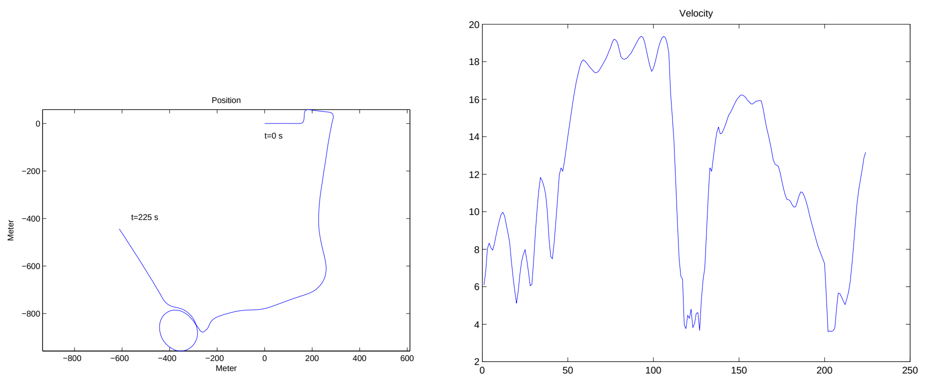


Figure 6.7 Driven path and velocity profile.

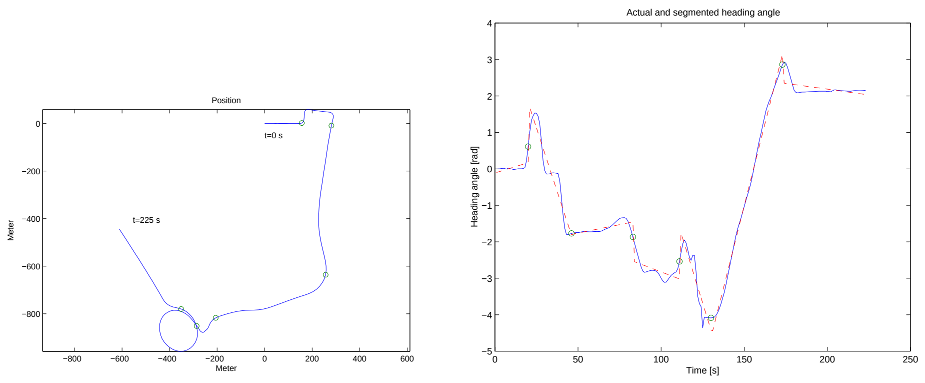


Figure 6.8 Driven path, with detected maneuver times marked, and segmented heading angle.

The estimated change times come quite close to what visual inspection gives. The second corner is not estimated, but the estimated heading angle shows a good mean-square approximation with the measured one. If more changes are to be estimated, the acceptable deviation in terms of the noise variance (set to 0.05 above) should be decreased.

6.6 Altitude sensor quality

A barometric air pressure sensor is used in airborne navigation systems for stabilizing the inertial navigation system in height. The barometric sensor is not very accurate and gives measurements of height with both a bias and large variance error. The sensor is particularly sensitive to the so called transonic passage, that is, when Mach 1 is passed. It is a good idea to detect for which velocities the measurements are useful, and perhaps also to try to find a table for mapping velocity to noise variance. Figure 6.9 shows the errors from a calibrated (no bias) barometric sensor, and low-pass filtered squared errors.[8] Figure 6.10 shows how the ML variance segmentation algorithm estimates the change times and noise variance as a function of sample number.

```
load altdata
y=hdiff(1:4:15608);

h1=cdfigure(1);
subplot(211)
plot(y)
title('Barometric height residuals')
nnn=0;
[thhat,lamhat]=adfilter(y,nnn,0.97);
subplot(212)
plot(lamhat)
title('Low-pass filtered squared residuals')

q=0.01;
[jhat,jtype,thseg,lamseg]=multihyp(y,nnn,100,[0 1 0],q);
h2=cdfigure(2,1);
semilogy([lamhat,lamseg'])
title('Low-pass filtered and segmented squared residuals')
```

6.7 Belching sheep

The input is the lung volume of a sheep and the output the air flow through the throat.[9] The goal is get a model for how the input relates to the output, and how different medicines affect this relation. A problem with a straightforward system identification approach is that the sheep belches regularly. Therefor, belching segments must be detected before modeling. The approach here is

[8]Thanks Saab AB and Dr. Jan Palmqvist for providing the data. The data have been rescaled.

[9]Thanks to Draco for sharing their data. The data have been rescaled.

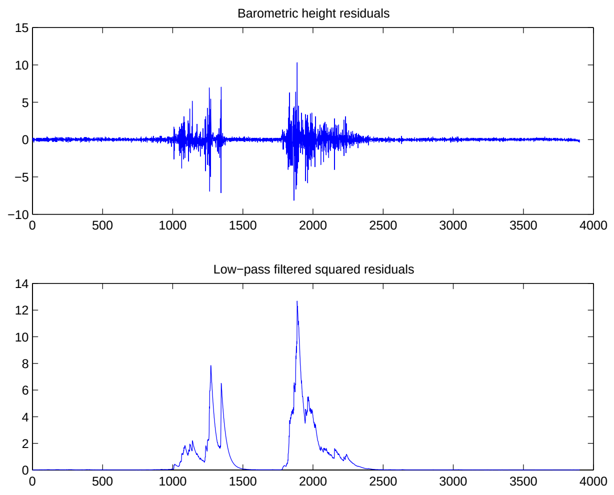


Figure 6.9 Barometric altitude residuals and low-pass filtered squared residuals (RLS with forgetting factor 0.97).

that the residuals from an ARX model is segmented according to the variance level. A good alternative is to use `detectM` with an ARX model directly, instead of explicitely computing the residuals.

```
load sheep
z=[sheep(:,2) sheep(:,1)];
[th,lamhat,e]=adfilter(z,[5 5 0],1);

nnn=0;
[jhat,jtype,thseg1,lamseg1]=multihyp(e,nnn,100,[0 1 0]);
jhat
jhat =
  [837 937 2345 2445 4195 4295 5564 5664 7009 7109]
N=length(e);
g=zeros(N,1);
lambda=0.97;
g=filter([1-lambda 0],[1 -lambda],e.^2);

h1=cdfigure(1);
subplot(211);
plot(1000*(1+sheep-2*[ones(N,1) zeros(N,1)]));
ax=axis;ax(2)=N;axis(ax);
title('Pressure and air volume')
xlabel('Sample number')
subplot(212);
thplot(g,lamseg1);
ax=axis;  ax(2)=N; ax(4)=4e-4; axis(ax);
title('Filtered model residual')
```

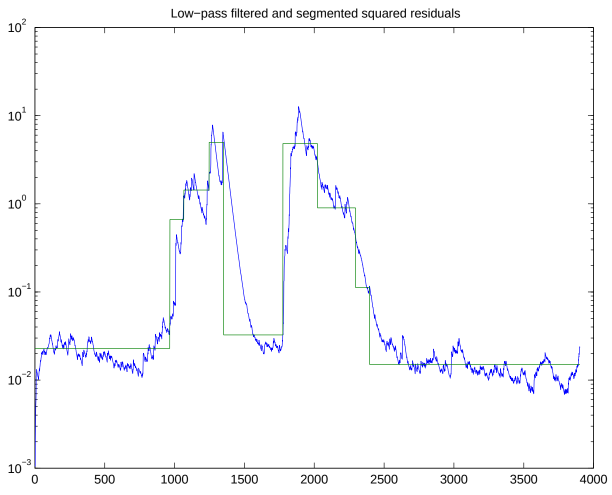


Figure 6.10 Low-pass filtered and segmented squared residuals

```
xlabel('Sample number')
```

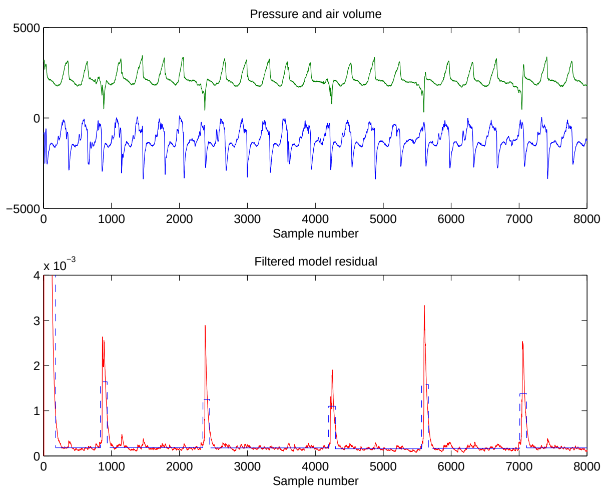


Figure 6.11 Input and output (above), and low-pass filtered and segmented noise variance (below)

7 Blockset and alternative implementations

The core functions have dual implementations. There are three different implementations of most of the filters. The recursive m-file implementation is useful for people who want to include toolbox functions in a larger application specific context, or customize for instance what is done after a detected change. The S-functions written in C are intended for Simulink and Real Time Workshop.

For example, `detect1` appears in the following shapes:

- `detect1.m` Basic loop-based implementation.
- `detect1c.c` C implementation.
- `detect1c.mexsol` C implementation compiled for unix.
- `detect1c.dll` C implementation compiled for PC.
- `rdetect1.m` Recursive implementation, doing one time recursion.
- `rdetect1c.c` Recursive C implementation.
- `rdetect1c.mexsol` C implementation compiled for unix.
- `rdetect1c.dll` C implementation compiled for PC.
- `sdetect1.mdl` Simulink block for compiled C-version.
- `srdetect1.mdl` Simulink block for compiled recursive C-version.
- `detect1x.m` Fast implementation using simulink but matlab syntax as `detect1`.

The recursive version is contained in the main directory, while all others are collected in the subdirectory `blockset`.

7.1 Recursive whiteness test detection

There is a number of recursive implementations of the functions in the toolbox. These have the prefix 'r', e.g. `radfilter`. The syntax of these functions are almost the same as in the original, except that a state has to be included in the input and output arguments. The state is initialized by calling the function itself with no data. In this way, the user can use his own filters, e.g. EKF, and make appropriate actions after a detected change. In this example, a first order ARX model with one jump is simulated.

```
randn('seed',1)
N=100;
u=randn(N,1);
e=randn(N,1);
th0=[0.5 1.5;0.8 1.8]';
d=2;
lam=0.1;
nnn=[1 1 1];
y=simchange([lam*e u],nnn,N/2,th0);
```

Then, we initialize the recursive versions of `adfilter` and `detect1`

```
[Xfilt]=radfilter([NaN NaN],nnn);
dm=2;
h=5;
nu=lam;
sr=[1 h nu];
[Xone]=rdetect1([],[],dm,sr,-1);
```

Let's also initialize some parameters we want to save for plots

```
TH=[];
LAM=[];
GT1=[];
alarmtimes=[];
```

The following loop does what

```
detect1([y u],nnn,dm,sr,-1)
```

would do. Note that the tasks of residual generation and detection are split into separate functions in contrast to `detect1`.

```
for t=1:N;
  [thhat,lamhat,epsi,S,Xfilt]=radfilter([y(t) u(t)],nnn,0.97,...
                                      'RLS',Xfilt);
  [alarm,Xone]=rdetect1(epsi/sqrt(S),Xone);
  GT1=[GT1;Xone(1)];
  TH=[TH thhat];
  LAM=[LAM;lamhat];
  if alarm~=0;
    alarmtimes=[alarmtimes t];
  end;
end;
```

For comparison, we generate the smoothed parameter estimates from `detectM`, the test statistic off-line from `detect1` and the estimates from `adfilter`.

```
[jhat,thseg]=detectM([y u],nnn);
[jdum,thdum,GT2]=detect1([y u],nnn,dm,sr,-1);
[TH2,LAM2]=adfilter([y u],nnn,0.97);
h1=cdfigure(1);
plot([thseg' TH' TH2'])
axis([0 N -1 3])
h2=cdfigure(2,1);
segplot([GT1],alarmtimes)
hold on, plot(GT2), hold off
title(['Test statistic and alarm times, Threshold = ',...
       num2str(h)])
```

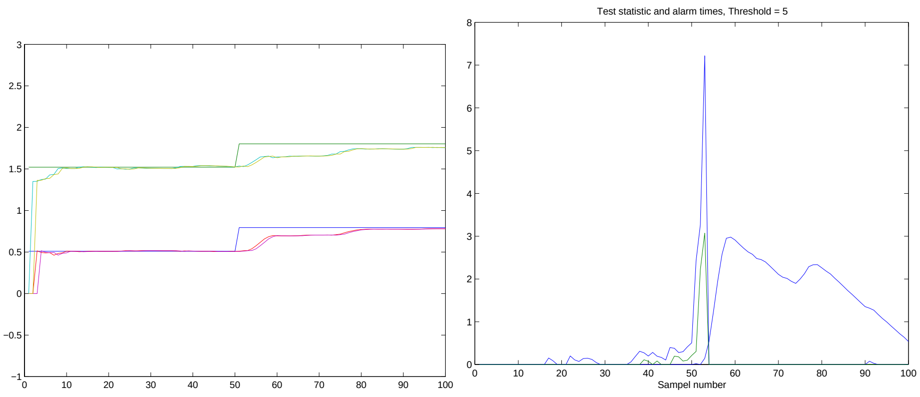


Figure 7.1 Estimated parameters from `adfilter` and `detect1`, and test statistics.

From the left plot of Figure 7.1 we conclude that the same estimates are generated and from the right one that we get several alarms after the change.

The most appropriate thing to do after the alarm is to increase the tracking ability of the filter. This is done below by increasing the covariance matrix of the parameters a factor 10.

```
[Xfilt]=radfilter([NaN NaN],nnn);
[Xone]=rdetect1([],[],dm,sr,-1);
TH=[];
LAM=[];
GT1=[];
alarmtimes=[];
for t=1:N;
  [thhat,lamhat,epsi,S,Xfilt]=radfilter([y(t) u(t)],nnn,0.97,...
                                        'RLS',Xfilt);
  [alarm,Xone]=rdetect1(epsi/sqrt(S),Xone);
  GT1=[GT1;Xone(1)];
  TH=[TH thhat];
  LAM=[LAM;lamhat];
  if alarm~=0;
    alarmtimes=[alarmtimes t];
    Xfilt(:,5:4+d)=10*Xfilt(:,5:4+d);
  end;
end;
h3=cdfigure(3,2);
segplot([GT1],alarmtimes)
title(['Test statistic and alarm times, Threshold = ',...
      num2str(h)])
```

This time there is only one alarm.

7.2 Recursive parallel filter detection

Consider the first order ARX model with one jump from the previous example. Then, we initialize the recursive versions of `adfilter`, one using all data and one using a sliding window and `detect2`.

```
L=10;
[X0]=radfilter([NaN NaN],nnn,1,'RLS');
[X1]=radfilter([NaN NaN],nnn,L,'WLS');
dm=2;
h=10;
nu=lam;
sr=[1 h nu];
[Xdet]=rdetect2([],[],[],[],[],dm,sr);
```

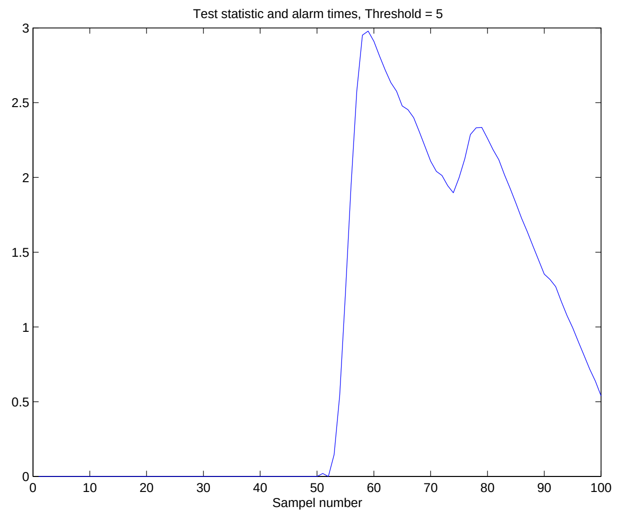


Figure 7.2 Test statistics for recursive `rdetect1`.

Let's also initialize some parameters we want to save for plots

```
TH0=[];
TH1=[];
LAM=[];
GT1=[];
alarmtimes=[];
```

The following loop does what `detect2([y u],nnn,dm,sr,-1)` would do. Note that the task of residual generation and the detector is split into separate functions in contrast to `detect2`.

```
for t=1:N;
  [thhat0,lam0,epsi0,S0,X0]=radfilter([y(t) u(t)],nnn,1,'RLS',X0);
  [thhat1,lam1,epsi1,S1,X1]=radfilter([y(t) u(t)],nnn,L,'WLS',X1);
  [alarm,Xdet]=rdetect2(...
             epsi0/sqrt(S0),epsi1/sqrt(S1),lam0,lam1,Xdet);
  GT1=[GT1;Xdet(1)];
  TH0=[TH0 thhat0];
  TH1=[TH1 thhat1];
  LAM=[LAM;lam0];
  if alarm~=0;
    alarmtimes=[alarmtimes t];
  end;
end;
```

An illustration of the result follows:

```
plot([TH0' TH1'])
axis([0 N -1 3])
segplot([real(GT1)],alarmtimes)
title(['Test statistic and alarm times, Threshold = ',...
       num2str(h)])
```

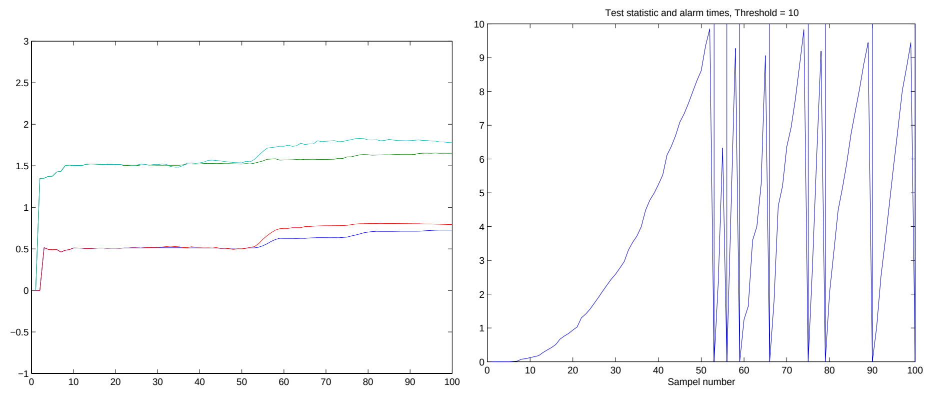


Figure 7.3 Estimated parameters from `radfilter` with different adaptation, and test statistics.

For comparison, we generate the smoothed parameter estimates from `detectM`, the test statistic off-line from `detect2` and the estimates from `adfilter`.

```
[jhat,thseg]=detectM([y u],nnn);
[jdum,thdum,GT2]=detect2([y u],nnn,dm,sr,-1);
[TH2,LAM2]=adfilter([y u],nnn,0.97);
figure(h1);
plot([thseg' TH0' TH1' TH2'])
axis([0 N -1 3])
figure(h2);
segplot([real(GT1)],alarmtimes)
hold on, plot(real(GT2)), hold off
title(['Test statistic and alarm times, Threshold = ',...
       num2str(h)])
```

From the left plot of Figure 7.4, we conclude that the same estimates are generated, and from the right plot that we get several alarms after the change.

The most appropriate thing to do after the alarm is to increase the tracking ability of the filter. This is done below by increasing the covariance matrix of the parameters a factor 10.

```
[X0]=radfilter([NaN NaN],nnn,1,'RLS');
```

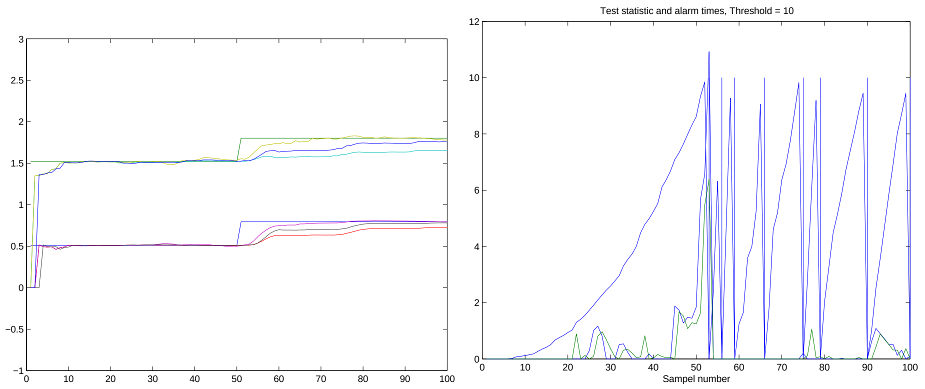


Figure 7.4 Estimated parameters from `adfilter`, `detect2` and `detectM`, and test statistics for `detect1` and `detect2`.

```
[X1]=radfilter([NaN NaN],nnn,L,'WLS');
[Xdet]=rdetect2([],[],[],[],[],dm,sr);
TH=[];
LAM=[];
GT1=[];
alarmtimes=[];
```

Start recursion

```
for t=1:N;
  [thhat0,lam0,epsi0,S0,X0]=radfilter(...
           [y(t) u(t)],nnn,1,'RLS',X0);
  [thhat1,lam1,epsi1,S1,X1]=radfilter(...
           [y(t) u(t)],nnn,L,'WLS',X1);
  [alarm,Xdet]=rdetect2(...
           epsi0/sqrt(S0),epsi1/sqrt(S1),lam0,lam1,Xdet);
  GT1=[GT1;Xdet(1)];
  TH=[TH thhat0];
  LAM=[LAM;lam0];
  if alarm~=0;
    alarmtimes=[alarmtimes t];
    X0(:,6:5+d)=10*X0(:,6:5+d);
  end;
end;
h3=cdfigure(3,2);
segplot([real(GT1)],alarmtimes)
title(['Test statistic and alarm times, Threshold = ',...
       num2str(h)])
```

This time there is only one alarm. There is one main advantage of this ap-

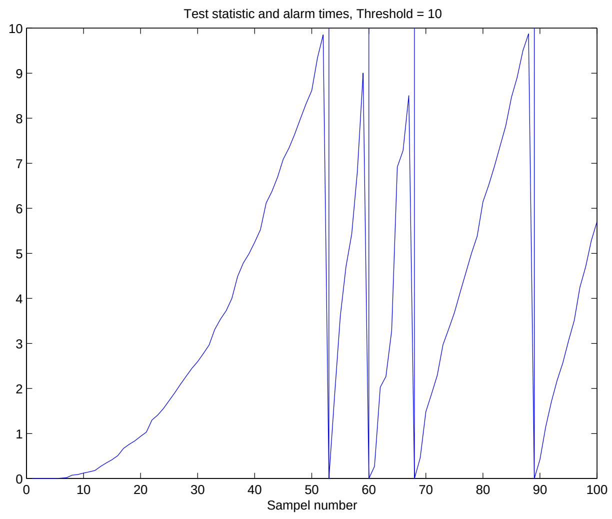


Figure 7.5 Test statistics for recursive `rdetect2`.

proach, compared to using `detect2`: it is possible to use arbitrary adaptive filters in parallel. For instance, we might try two RLS algorithms with different forgetting factors.

7.3 Recursive GLR detection

First, a second order state space model with one jump is simulated.

```
randn('seed',0)
N=100;
d=2;
%w=randn(N,2);
u=randn(N,1);
%e=randn(N,1);
A=[1 1;0 1]; B=[0.5;1]; C=[1 0]; D=0;
Q=0.01*eye(2); R=0.1; P0=1*eye(2);
nnn=ss2nnn(A,B,C,D,Q,R,P0);
th0=[1;...
     2];
y=simchange([u],nnn,N/2,th0);
h1=cdfigure(1);
plot(y)
title('Signal')
```

Let's examine the off-line versions of `glr` and `mlr`

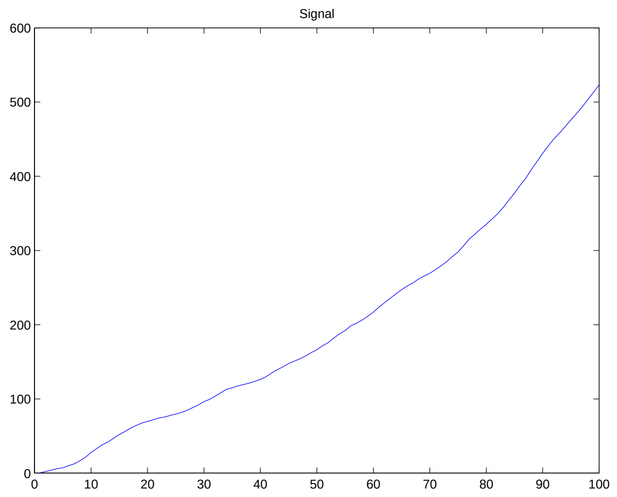


Figure 7.6 Signal.

```
M=-10;  % size of sliding window, batch test here
flops(0)
[jhatglr,taglr,lrglr,Xhat,nuhatglr,mu]=glr([y u],nnn,M);
flopsglr=flops;
[jhatmlr,lrmlr,flopsmlr]=mlr([y u],nnn,1);
disp([jhatglr,jhatmlr])
    49    50
nuhatglr
    0.1120
    2.7933
figure(h1)
plot([y, (C*Xhat)'])
hold on
plot(N,C*(Xhat(:,N)+mu*nuhatglr(:,1)),'o')
hold off
title('Signal and Kalman filtered estimate')
h2=cdfigure(2,1);
plot([lrglr lrmlr])
title('Decision function for GLR and MLR')
```

That is, a threshold of 70 seems appropriate. The number of flops compares as follows

```
disp([flopsglr flopsmlr])
      241986       37983
```

Then, we initialize the recursive versions of `adfilter` and `glr`.

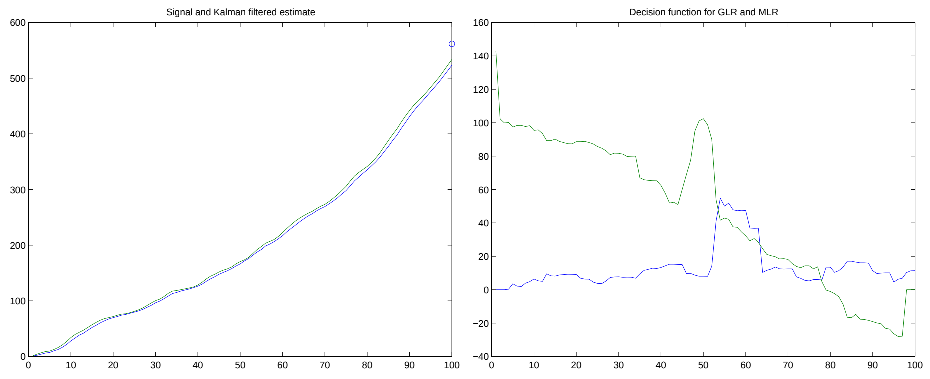


Figure 7.7 Signal and Kalman filter prediction (left), and decision functions of GLR and MLR (right), respectively.

```
[Xfilt]=radfilter([NaN NaN],nnn);
h=72;
M=10;
[Xglr]=rglr([],[],[],nnn,[],M,h);
```

Let's also initialize some parameters we want to save for plots

```
TH=[];
EPSI=[];
LAM=[];
GT1=[];
alarmtimes=[];
```

The following loop uses the recursive implementation of `glr`. Note that the task of residual generation and the detector is split into separate functions.

```
for t=1:N;
  [thhat,lamhat,epsi,S,Xfilt,K]=radfilter(...
                     [y(t) u(t)],nnn,1,'RLS',Xfilt);
  f0=flops;
  [alarm,Xglr,nuhat,jhat,lrmax,P]=rglr(epsi,S,K,nnn,Xglr);
  f1=flops;
  GT1=[GT1;lrmax];
  EPSI=[EPSI;epsi];
  TH=[TH nuhat];
  if alarm~=0;
    nuhat
    P
    alarmtimes=[alarmtimes t];
    % Increase P in the Kalman filter
    Xfilt(:,6:5+d)=100*Xfilt(:,6:5+d);
  end;
end;
```

The number of flops per iteration is

```
disp([f1-f0])
        2548
```

```
h3=cdfigure(3,2);
segplot([GT1],alarmtimes)
title(['Test statistic and alarm times, Threshold = ',num2str(h)])
```

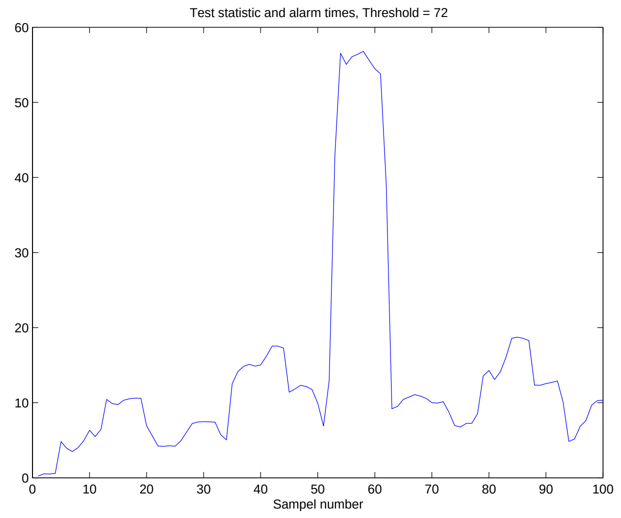


Figure 7.8 Test statistics for recursive `rglr`.

8 Graphical User Interface

Generally, a *graphical user interface* (*GUI*) has the following advantages:

- The unexperienced user gets a structured overview of the functionality and user choices and does not have to bother about syntax.
- The experienced user gets help from the track record of previous experiments. That is, the GUI can be used to organize a large amount of data and design parameters.
- There are certain computations that are so complex that they can hardly be implemented in one file because of the large number of arguments. The alternative to a GUI is to work with big script files, which after a while are hard to maintain.

The first section gives and overview of the functionality of the GUI, and the second one describes the implementation of `guidetect` .

8.1 What do adaptive filtering problems have in common?

Both industrial and acamedic researchers using adaptive filters in control system design, signal processing or statistical time series analysis essentially do the same investigations:

- Tune the filters to get maximal performance.
- Perform Monte Carlo simulations to evaluate the mean performance.
- Examine the RMS errors in states/parameters and other performance measures.
- Compare different methods and present tables with results.

The GUI automates these procedures:

- Manual tuning is facilitated by storing the result in tables. Auto-tuning of the most important parameters is possible. That is, the optimal design parameter is computed automatically to minimize a specified performance measure.

- Monte Carlo simulations are done with a mouse-click, and all statistics are kept.

- RMS plots are available and various performance measures are stored.

- Automatic table generation gives overview of the investigations.

8.2 Getting acquainted with the frontend

The frontend consists of a block diagram illustrating the filter structure. The figure is dynamic, so if a different ***filter structure*** (buttons will in the sequel be boldface and emphasized) is chosen, it will change.

The basic principle is that the experimental setup is done from submenus started in the upper row of pushbuttons, a simulation is started by pushing the respective blocks and design tools are started from the pushbuttons in the right column. The signals shown in the block diagram are push buttons, and a mouse click opens a plot window.

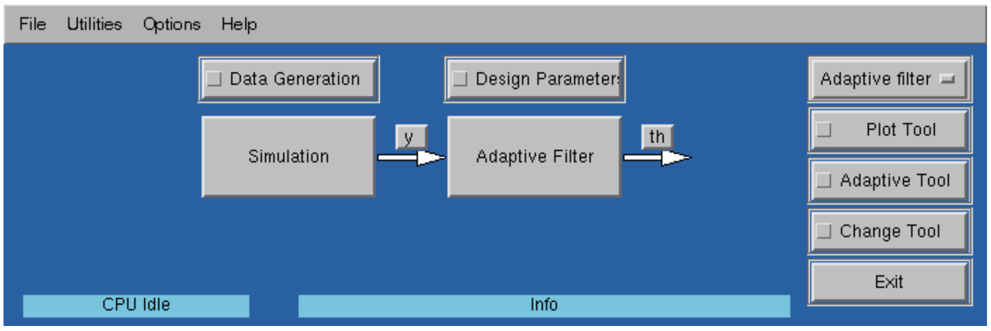


Figure 8.1 Frontend of the GUI. When the ***filter structure*** is changed, the block diagram is adapted.

From the menus, you can

- Get ***help***. On-line help is available from all subwindows.

- Load real data with default values of design parameters (***File/Load Real Data***).

- Start examples ***Utilities/Examples***, where design parameters have good default values.
- ***Options*** include functions for windows management and printouts in the matlab command window.

Once the structure is decided on, a model needs to be specified in the ***Data Generation***. This opens the window in Figure 8.2.

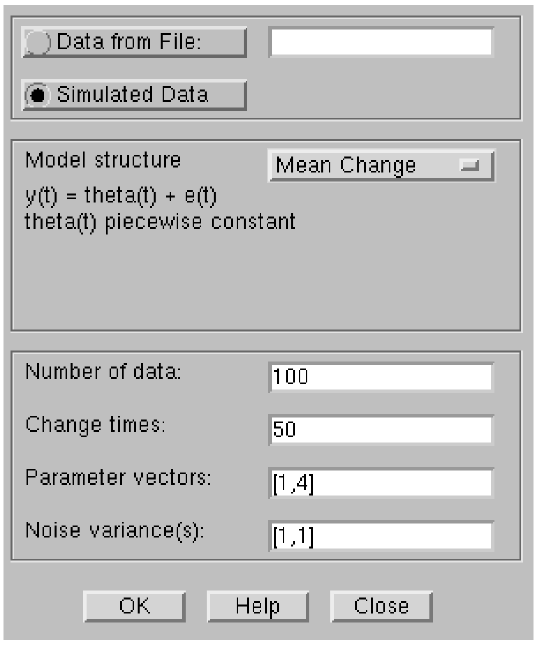

Data from File:
● Simulated Data
Model structure: Mean Change
y(t) = theta(t) + e(t)
theta(t) piecewise constant
Number of data: 100
Change times: 50
Parameter vectors: [1,4]
Noise variance(s): [1,1]

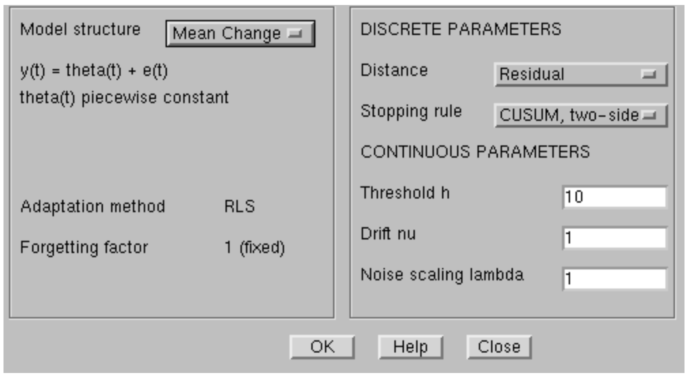
Figure 8.2 Data Generation window. You can switch between logged data on file, or a simulation where you first specify the model structure, then its parameters and change times.

In the ***Design Parameters*** window, you specify a system model and its design parameters.

There are three design tools: ***plot tool***, ***adaptive tool*** and ***change tool***. These windows are shown in Figures 8.4, 8.5 and 8.7, respectively.

From the ***plot tool***, all kind of signal plots are available, *e.g.* parameter and root mean square error plots. Either the recursive or smoothed, or both of them simultaneously, parameter estimates can be illustrated in the plots.

In the ***adaptive tool***, several norms are displayed for the last filter action. Monte Carlo simulations can be initiated, after which mean values of the norms are displayed, and mean values are shown in all plots. The ***Optimize*** button auto-tunes the adaptation gain for adaptive filters and threshold for change detectors. A plot shows the sensitivity of the chosen norm with respect



Model structure: Mean Change

y(t) = theta(t) + e(t)
theta(t) piecewise constant

Adaptation method: RLS

Forgetting factor: 1 (fixed)

DISCRETE PARAMETERS

Distance: Residual

Stopping rule: CUSUM, two-side

CONTINUOUS PARAMETERS

Threshold h: 10

Drift nu: 1

Noise scaling lambda: 1

Figure 8.3 Filter Design Parameters. The window is dynamically depending on the ***filter structure***. To each one, there is a number of options for the adaptation gains, stopping rules and distance measures. You can also try out different model structures for the filter, which does not need to be the same as for the simulation.

to the design parameter, and the plots are updated with the optimal design parameter. The current norms and a summary of the method can be stored in a table.

The ***change tool*** has also a Monte Carlo simulation facility. The performance is here also computed in terms of detections, and some useful norms are displayed in a separate window. These figures and a summary of the method can be stored in a table.

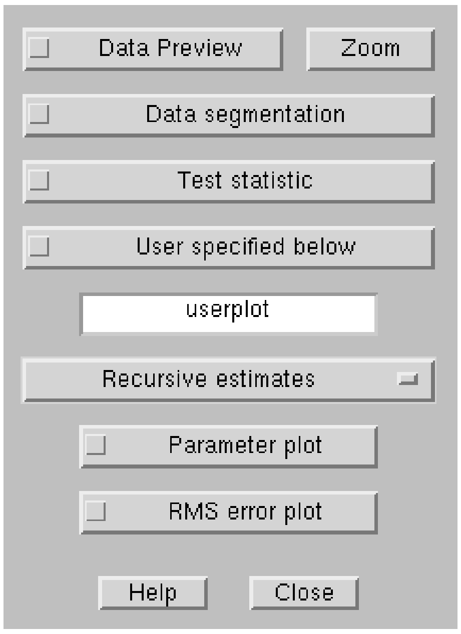
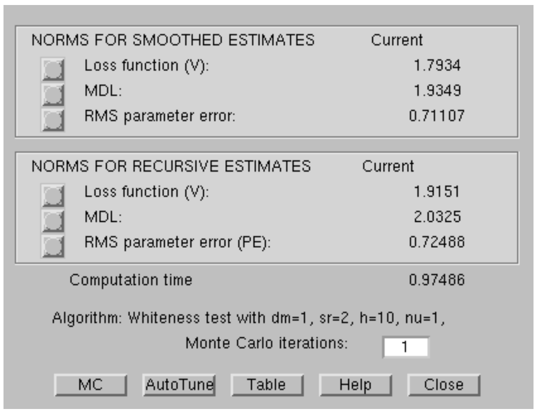


Figure 8.4 Plot tool. Different views of the filtering result. There is a possibility to zoom in (but not out) the signal to be filtered.



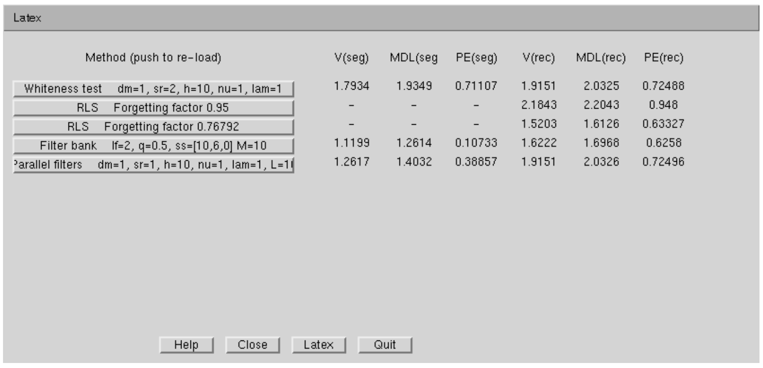
NORMS FOR SMOOTHED ESTIMATES	Current
Loss function (V):	1.7934
MDL:	1.9349
RMS parameter error:	0.71107

NORMS FOR RECURSIVE ESTIMATES	Current
Loss function (V):	1.9151
MDL:	2.0325
RMS parameter error (PE):	0.72488

Computation time 0.97486

Algorithm: Whiteness test with dm=1, sr=2, h=10, nu=1,

Monte Carlo iterations: 1



Latex

Method (push to re-load)	V(seg)	MDL(seg	PE(seg)	V(rec)	MDL(rec)	PE(rec)
Whiteness test dm=1, sr=2, h=10, nu=1, lam=1	1.7934	1.9349	0.71107	1.9151	2.0325	0.72488
RLS Forgetting factor 0.95	-	-	-	2.1843	2.2043	0.948
RLS Forgetting factor 0.76792	-	-	-	1.5203	1.6126	0.63327
Filter bank lf=2, q=0.5, ss=[10,6,0] M=10	1.1199	1.2614	0.10733	1.6222	1.6968	0.6258
ʾarallel filters dm=1, sr=1, h=10, nu=1, lam=1, L=1(	1.2617	1.4032	0.38857	1.9151	2.0326	0.72496

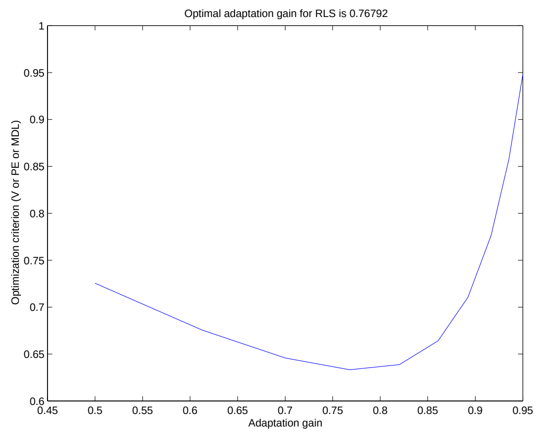
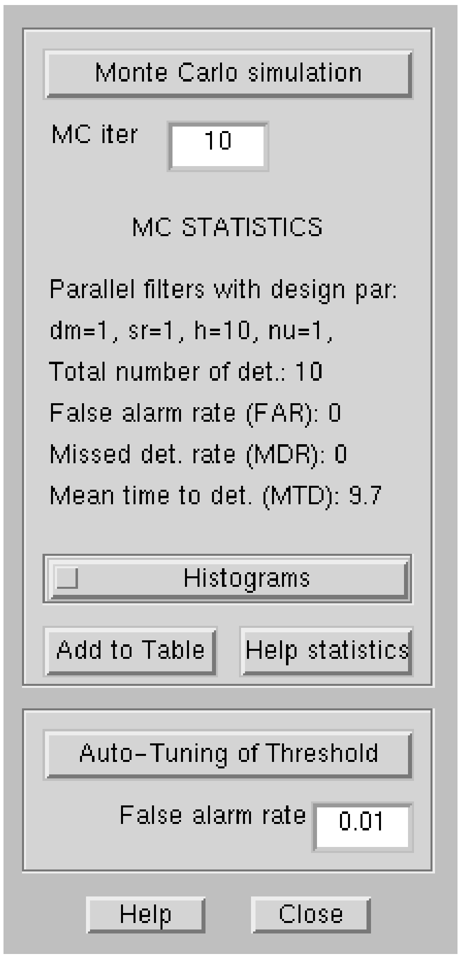


Figure 8.5 Adaptive tool for design of adaptive filters. Here the performance is printed out, a table item can be generated and optimization (auto-tuning) started. For all cases, Monte Carlo simulations can be used. MC by itself, iterates the filter for the current settings. For auto-tuning, first choose a norm (only the lower three ones available for filters. The function then performs a line search over the most critical design parameter, and a plot of the performance versus design parameter is delivered. If the optimum is at the investigated border, push auto-tune another time. Table below is handy for comparison and administration, since you can quickly load all quantities from a specific test from the table.



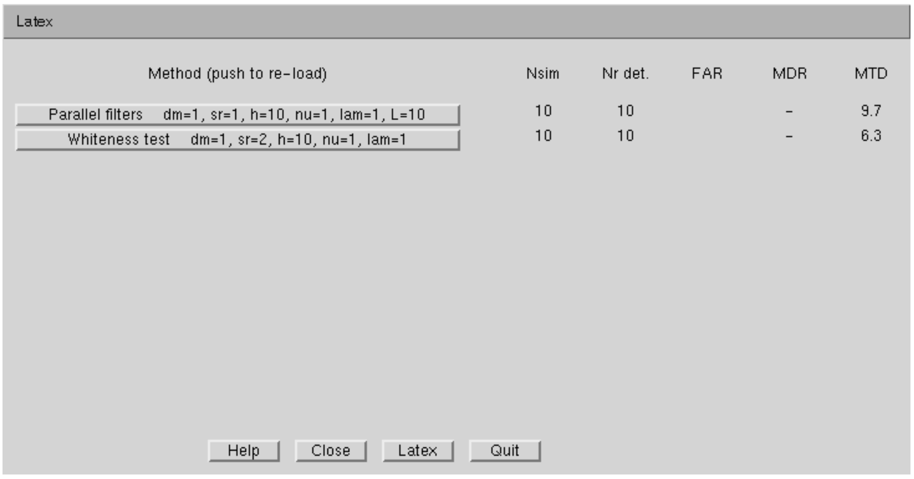
MC iter 10

MC STATISTICS

Parallel filters with design par:
dm=1, sr=1, h=10, nu=1,
Total number of det.: 10
False alarm rate (FAR): 0
Missed det. rate (MDR): 0
Mean time to det. (MTD): 9.7

False alarm rate 0.01

Figure 8.6 Change tool design of change detectors. Here Monte Carlo simulations are started and the relevant statistics are presented. There is a possibility to plot histograms and make tables, as illustrated in Figure **??**. There is also a way to perform auto-tuning of the threshold in the detector to get a specified false alarm rate.



Latex

Method (push to re-load)	Nsim	Nr det.	FAR	MDR	MTD
Parallel filters dm=1, sr=1, h=10, nu=1, lam=1, L=10	10	10		-	9.7
Whiteness test dm=1, sr=2, h=10, nu=1, lam=1	10	10		-	6.3

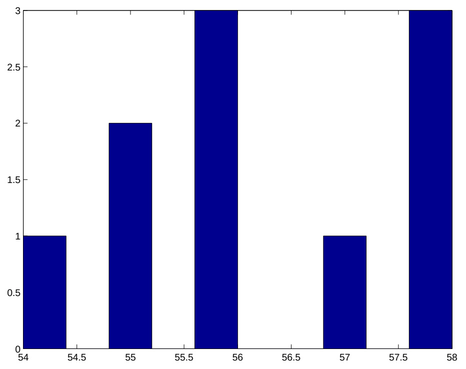


Figure 8.7 The change tool table is suitable for comparisons, overview and administration of experiments. A previous experiment can be downloaded by clicking on the filter settings, and all plots and text fields will be updated.

9 Command Reference

9.1 Commands Grouped by Function

Simulation	
`y=simchange (z,nnn,jumptimes,TH)`	Abrupt parameter/state changes
`y=simadfilter (z,nnn,adm,adg)`	Parameter variations for RLS, LMS
`y=simresid (z,nnn,threc)`	Linear regression residuals
`[y,th]=simchannel (z,nb,options)`	Communication channels

Parameter estimation	
`[threc] = adfilter (z,nnn,adg,adm)`	Linear adaptive filters
`[threc] = detect1 (z,nnn,options)`	Non-linear adaptive filters
`[threc] = detect2 (z,nnn,options)`	Non-linear adaptive filters
`[threc] = detectM (z,nnn,options)`	Non-linear adaptive filters
`[threc] = multihyp (z,nnn,options)`	Non-linear adaptive filters

State estimation	
`[xhat] = adfilter (z,nnn,adg,adm)`	Kalman filter
`[xhat] = adkalman (z,nnn,adg)`	Kalman filter, general
`[xhat] = detect1 (z,nnn,options)`	Kalman filter with whiteness test
`[xhat] = detectM (z,nnn,options)`	Kalman filter bank
`[xhat] = gibbs (z,nnn,options)`	MCMC resampling algorithm
`[xhat] = mlr (z,nnn,options)`	Kalman filter using MLR
`[xhat] = glr (z,nnn,options)`	Kalman filter using GLR
`[r] = faultdetect (z,nnn,options)`	Parity space residual generation

Change Detection	
`[threc,jhat,thseg] = detect1 (z,nnn,options)`	Whiteness tests
`[threc,jhat,thseg] = detect2 (z,nnn,options)`	Two parallel filters
`[threc,jhat,thseg] = detectM (z,nnn,options)`	Multi-filter approach
`[threc,jumpsmc,jhat] = gibbs (z,nnn,options)`	MCMC resampling algorithm
`[threc,jhat,thseg] = multihyp (z,nnn,options)`	Multiple hypotheses
`[jhat,thseg] = cpe (z,nnn,options)`	Change point estimation
`[jhat] = mlr (z,nnn,options)`	Marginalized likelihood ratio test
`[xhat,jhat] = glr (z,nnn,options)`	Generalized likelihood ratio test
`[xhat] = imm (z,nnn,options)`	Interactive multiple model

Equalizers	
`[uhat]=viterbi (y,nn,th,lam)`	Viterbi algorithm
`[threc,uhat]=blindeq (y,nb,adg,adm)`	Adaptive blind equalizer
`[threc,uhat]=blindeqM (y,nb,options)`	ML sequence detection
`ber=u2ber (u,uhat)`	Computation of bit error rate
`m=openeye (bch,beq)`	Computation of open eye measure

Conversions	
`thseg=par2segm (TH,jumptimes,N)`	Compressed to full format
`TH=segm2par (thseg)`	Full to compressed format
`phi=z2phi (z,nnn)`	Construct the regression vector

Plots	
`segplot (z,jumptimes)`	Mark jump times in signal plot
`thplot (th,TH,jumptimes,threc)`	Compare parameter estimates
`hypplot (XdetectM)`	Filter bank hypotheses
`radarplot (y,xhat,x)`	Illustrate target tracking

Utilities	
`Contents`	General overview
`guidetect`	Start the GUI
`demodetect`	Command line demo
`reference`	Examples in this manual
`tutorial`	Tutorials in this manual
`book`	Examples in the accompanying text book
`signal`	Examples in the text book `Signal Processing` (in swedish)

Implementations	
`detect1.m`	Basic loop-based implementation
`detect1c.c`	C implementation
`detect1c.mexsol`	C implementation compiled for unix
`detect1c.dll`	C implementation compiled for PC
`rdetect1.m`	Recursive implementation, doing one time recursion
`rdetect1c.c`	Recursive C implementation
`rdetect1c.mexsol`	C implementation compiled for unix
`rdetect1c.dll`	C implementation compiled for PC
`sdetect1.mdl`	Simulink block for compiled C-version
`srdetect1.mdl`	Simulink block for compiled recursive C-version
`detect1x.m`	Fast matlab call using simulink

`General Utilities`	
`cdfigure`	As `figure`, but creates non-overlapping windows in a 3×3 pattern.
`cdnum2str`	As `num2str`, but the exact inverse operation of `str2num`.
`cdfiltfilt`	Improvement of Matlab's `filtfilt`, by optimizing the initial conditions and thus minimizing transient effects.
`fastfilter`	Improvement of Matlab's `filter`, by performing the filter operation in the frequency domain when it pays off (long input sequence).
`fastfiltfilt`	Improvement of Matlab's `filtfilt`, by performing the filter operation in the frequency domain when it pays off (long input sequence).
`MakeMatrix`	Generates a Latex file which formats the input matrix appropriately.
`MakeTable`	Generates a Latex file which formats a table with entries from the input matrix.

9.2 adfilter

Purpose `adfilter` implements various linear filters for estimating time-varying parameters in filters or linear regression models or the states in a state space model.

Synopsis

```
[thhat,lamhat,Epsi]=adfilter(z,nnn,adg,adm,fflam);
```

Description `adfilter` implements *recursive least squares* (*RLS*), *least mean square* (*LMS*), *normalized least mean square* (*NLMS*) or the *Kalman filter* (*KF*). For parametric models (FIR, ARX, linear regressions *etc.*), `adg` defines the adaptation gain and `adm` the adaptation method. For state space models, the `adg` parameter adjusts the tracking ability of the Kalman filter by scaling $\|Q\|/\|R\|$, and `adm` is not used. A more general implementation of the Kalman filter is found in `adkalman` .

`z=[y u]` Output-input data.

`nnn` Model structure, see `nnn`.

`adg` Adaption gain: forgetting factor (RLS), step size (LMS and NLMS), window size (WLS) or scaling of Q (Kalman filter).

`adm` Adaptation method: RLS (default for parametric models), WLS, LMS, NLMS or KF (always for state space models). The default value depends on `adg`. If `adg`<0.1 then LMS is used, or if $0.1<$`adg`<1 then RLS is used, or if `adg`>1 then WLS is used.

`thhat` Estimated parameter/state vector for each t.

`lamhat` Estimated noise variance for each t

`Epsi` The prediction errors for each t.

`fflam` Forgetting factor for estimating noise variance `lambda`, which should be estimated with higher gain than the parameters.

The syntax is compatible with other recursive algorithms, like `threc=detect1(z,nnn)`.

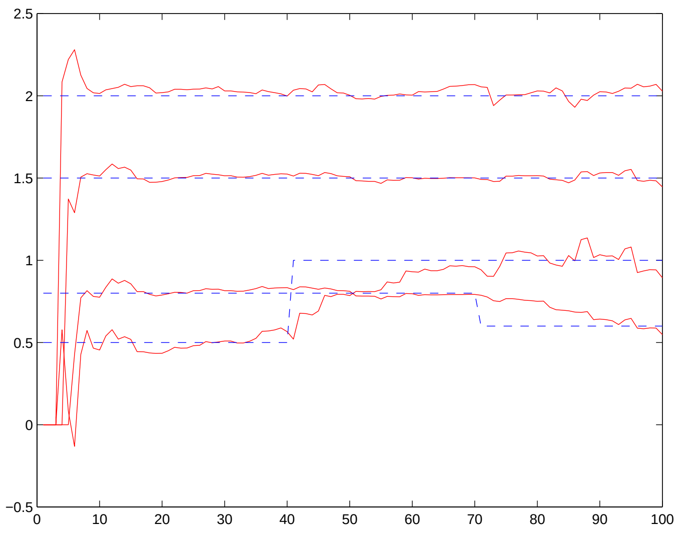
This function is useful prior to segmentation or detection to find out the nature of the parameter changes (abrupt or inicipative). Use high gain in this case.

Tuning Tuning the adaptation gain is a trade-off between variance and tracking errors. If the estimate is not able to follow the true variations (poor tracking), then increase the adaptation gain. If the tracking is acceptable but the variance error is large, then try to decrease the adaptation gain.

Examples

Define and simulate a second order ARX model with two abrupt changes. Then estimate its parameters recursively with a linear filter and compare the result with the true parameters in a plot.

```
u=randn(100,1);
e=randn(100,1);
jumptimes=[40 70];
TH=[1.5 0.8 2 0.5;...
    1.5 0.8 2 1;...
    1.5 0.6 2 1]';
nnn=[2 2 1];
y=simchange([0.1*e u],nnn,jumptimes,TH);
z=[y u];
thRLS=adfilter(z,nnn,0.9);
thplot(thRLS,segm2par(TH,jumptimes,100));
```

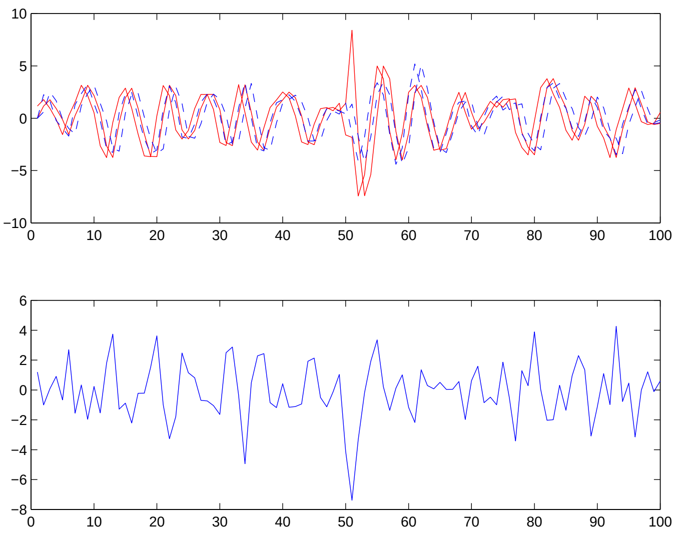


This example is revisited in detect1 , detect2 , detectM .

A state space model is defined and simulated and the Kalman filter is applied below.

```
[a,b,c,d]=tf2ss([0 1 -1],[1 -1 0.8]);
Q=0.0001*eye(2);
R=0.01;
nnn=ss2nnn(a,b,c,d,Q,R);
jumptime=50;
TH=[10 10]';
[y,X]=simchange([u],nnn,jumptime,TH);
[xhat,lamhat,epsi]=adfilter([y u],nnn);
subplot(211)
```

```
thplot(X,xhat)
subplot(212)
plot(epsi)
```



This example is revisited in `glr` , `mlr` .

Algorithm

The algorithms are described in Fredrik Gustafsson. *Adaptive filtering and change detection*. John Wiley & Sons, Ltd, 2000:

LMS : Algorithm 5.2, p 134.

NLMS : Algorithm 5.2, p 134, and equations (5.38)-(5.39), p. 136.

WLS : Lemma 5.1, p 193.

RLS : Algorithm 5.3, p 138.

KF for parameter estimation: Algorithm 5.4, p. 142.

KF for state estimation: equations (8.34)-(8.37), p. 278.

The underlying models are described in Appendix A.

References

A thorough treatment of Kalman filtering is found in B.D.O. Anderson and J.B. Moore. *Optimal filtering*. Prentice Hall, Englewood Cliffs, NJ., 1979 and T. Kailath, A.H. Sayed, and B. Hassibi. *Linear estimation*. Information and System Sciences. Prentice-Hall, Upper Saddle Riber, New Jersey, 2000, and of recursive parameter estimation in L. Ljung and T. Söderström. *Theory and practice of recursive identification*. MIT Press, Cambridge, MA, 1983.

See Also

`adkalman` , `detect1` , `detect2` , `detectM` , `glr` , `mlr` , `radfilter`

9.3 adkalman

Purpose `adkalman` includes various implementations of the Kalman filter

Synopsis

```
[xhat,yhat,P]=adkalman(z,nnn,adgain,kftype);
```

Description `adkalman` generalizes the Kalman filter implemented in `adfilter` a number of alternative formulations and implementations.

`z=[y u]` Output-input data.

`nnn` Model structure, see `nnn` .

`adgain` Adaptation gain which scales P_0 and Q.

`kftype` Type of implementation:

1. stationary KF
2. time-varying KF on prediction form
3. time-varying KF on filter form (default)
4. fixed interval smoother
5. square root filter
6. square root predictor

Algorithm The algorithms are described in Fredrik Gustafsson. *Adaptive filtering and change detection.* John Wiley & Sons, Ltd, 2000:

1 Stationary Kalman filter: Algorithm 8.2.

2 Time-varying Kalman filter on predictor form: Algorithm 8.1.

3 Time-varying Kalman filter on filter form: Algorithm 8.1.

4 Fixed interval smoother: Algorithm 8.3.

5 Square root filter: Algorithm 8.8.

6 Square root predictor: Algorithm 8.7.

The used state space model is defined in (A.19)-(A.20) in Appendix A.

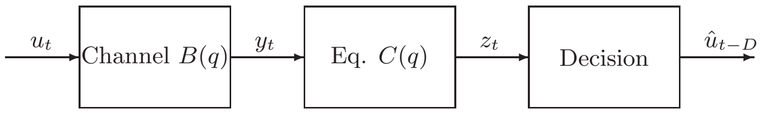
See Also `adfilter`

9.4 blindeq

Purpose `blindeq` implements classical blind equalization algorithms.

Synopsis `[uhat,threc]=blindeq(y,nb,adg,adm);`

Description A linear filter is adaptively estimated in order to inverse filter the channel dynamics. The output from the equalizer is compared to the known input alphabet, and its nearest neighbor is chosen. To be able to converge, `blindeq` assumes that the adaptive equalizer is initially such that its output $\hat{u}$ is approximately equal to the channel input u, and adaptation aims at refining the initial guess to improve bit error rate.



`y` Output data

`uhat` Estimated input

`adg=1` Loss function $E[(1-z^2)^2]$ (modulus restoral, Godard)

`adg=2` Loss function $E[(\text{sign}(z)-z)^2]$ (decision feedback, Sato)

`adm` Adaptation gain in a stochastic gradient (LMS like) method.

`th0` Initial guess of equalizer parameters in $C(q)$. Quite critical for convergence.

Limitations

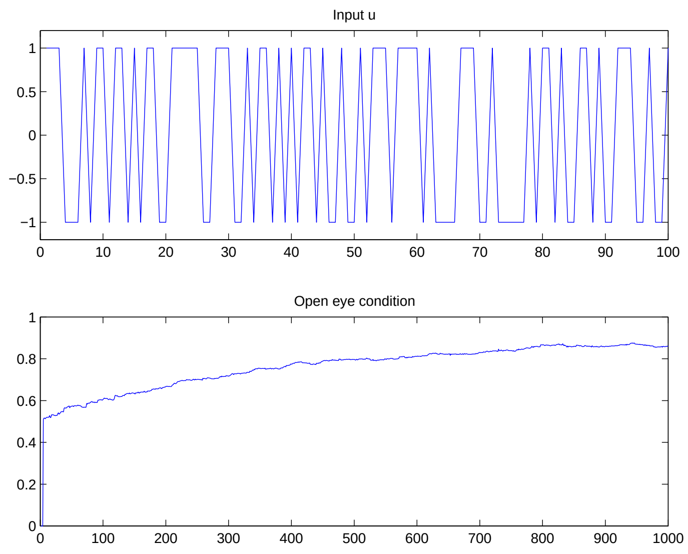
1. Only FIR equalizers
2. Only input alphabets [-1,+1]
3. Only scalar channels

Examples

```
N=1000;
randn('seed',1);
b=[0.3 1 0.3];   % Channel dynamics
```

```
nb=length(b);
alphabet=[-1 1]; % Binary channel
[y,u]=simchannel(alphabet,nb,N,b); % Simulate channel input and output
y=y+0.1*randn(N,1);  % Add noise

nc=5;   % equalizer order for blindeq
c0=[0 -0.1 1 -0.1 0]';  % Initial equalizer
[theq1,uhat1,epsi1]=blindeq(y,nc,0.01,1,c0);
m1=openeye(b,theq1);
% m1=1 is a perfect equalizer, m1>0 gives perfect
% reconstruction in the noise free case
plot(m1)
title('Open eye condition')
conv(b,theq1(:,N))'
ans =
    0.0308   -0.0122    0.0170    1.0004    0.0180
   -0.0111    0.0311
 % Impulse response of combined channel and equalizer
```



Algorithm

The algorithms are described in Algorithm 5.6, p. 166, in Fredrik Gustafsson. *Adaptive filtering and change detection*. John Wiley & Sons, Ltd, 2000.

References

The modulus restoral algorithm is presented in D.N. Godard. Self-recovering equalization and carrier tracking in two-dimensional data communication systems. *IEEE Transactions on Communications*, 28:1867–1875, 1980, and decision feedback in R.W. Lucky. Techniques for adaptive equalization of digital communication systems. *Bell System Technical Journal*, 45:255–286, 1966.

See Also

blindeqM , openeye , u2ber , viterbi

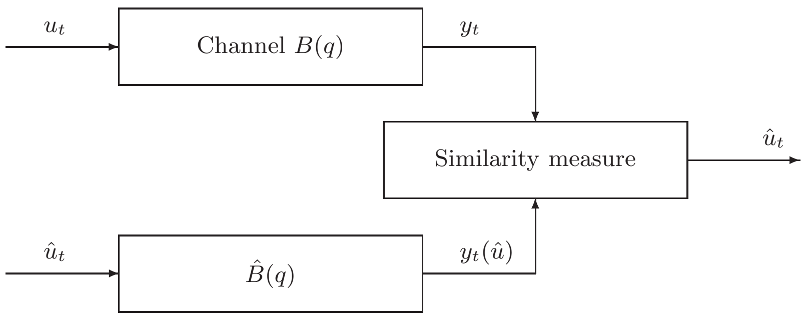
9.5 blindeqM

Purpose `blindeqM` is a multi-filter approach to blind equalization.

Synopsis

```
[uhat,threc]=blindeqM(y,nb,alphabet,r2,M,r1,p0,dsp)
```

Description `blindeqM` estimates the unknown input $u(t)$ taken from a finite alphabet simulataneously with the unknown channel, using a Viterbi-like search algorithm. The principle is that each possible input sequence implies one estimated channel model, and the one that fits the data best in the maximum likelihood sense wins. Since the number of input sequences in finite (though quite large), all of them can be evaluated. A search algorithm eliminates unlikely sequences to limit the number of parallel hypotheses (and filter) to a finte number M.



`y` or `z` Output data.

`nb` `ny` times `nu` matrix, where `ny` is the number of outputs and `nu` the number of input. The elements denote the corresponding FIR orders.

`q` Covariance of parameter drift is `Q=eye(n)*q`. Here `n=sum(sum(nb))` is the total number of parameters. Default `q=0`.

`alphabet` Finite alphabet of input symbols.

`r` Measurement noise variance. Default `R=0.1*eye(ny)`.

`p0` Covariance matrix of initial estimate, `P=eye(N)*p0`. Default `p0=10`.

`M` Number of models. Default `M=64`. This number must be an integer multiple of the size of the input alphabet.

`dsp` Display some information if `dsp~=0`. Default `dsp=0`.

`uhat` Estimated input.

`threc` Estimated parameter vector as a function of time (`n x N` matrix).

The implementation handles MIMO FIR channel models and any input alphabet (in contrast to blindeq). Use u2ber to correct the modulus in the estimated sequence and the channel parameters.

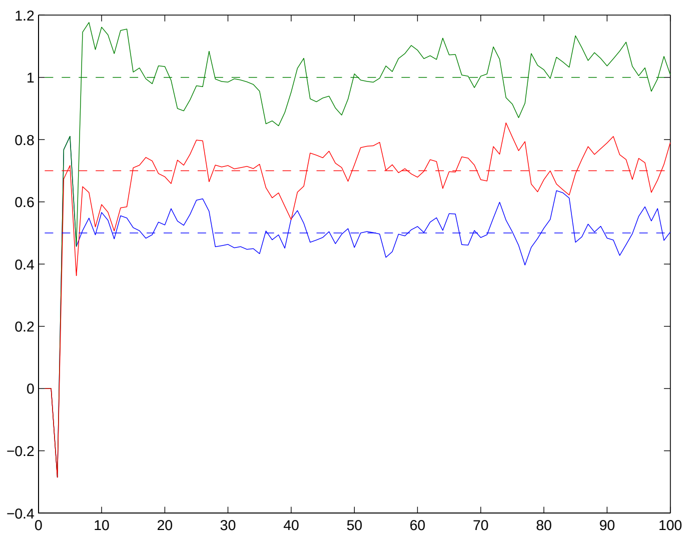
Limitations

1. Only FIR channel models.
2. The MIMO implementation only works for `nb` all having constant elements (`nb*ones(ny,nu)` where `nb` is the common FIR order).

Examples

```
N=100;
randn('seed',1);
rand('seed',1);
b=[0.5 1 0.7];  % Channel dynamics
nb=length(b);
alphabet=[-1 1]; % Binary channel
[y,u]=simchannel(alphabet,nb,N,b); % Simulate channel input and output
y=y+0.1*randn(N,1);  % Add noise

nb=3;     % channel order for blindeqM
[thhat,uhat]=blindeqM(y,nb,.001,16,.1,10,0);
cdfigure(1);
plot(thhat'),           hold on
plot(ones(N,1)*b,'--'), hold off
```



Algorithm

The algorithms are described in Algorithm 10.7, p. 395, in Fredrik Gustafsson. *Adaptive filtering and change detection.* John Wiley & Sons, Ltd, 2000.

References

F. Gustafsson and B. Wahlberg. Blind equalization by direct examination of the input sequences. *IEEE Transactions on Communications*, 43(7):2213–2222, 1995.

See Also

`blindeq` , `viterbi`

9.6 cdfigure

Purpose `cdfigure` creates non-overlapping plot windows.

Synopsis `h=cdfigures(n);`

Description Set up `n<10` figures, starting at the top at the right side of the screen.

Examples

```
h=cdfigure(9);
```

will set up 9 nicely placed figure windows of the same size in a 3 times 3 pattern.

```
cdfigure(1); plot(y1)
cdfigure(2); plot(y2)
```

will place the first plot in the upper right corner, and then the second plot is placed below the first one.

9.7 cpe

Purpose

`cpe` implements Change Point Estimation methods from the statistical literature for a change in the mean model.

Synopsis

```
[jumphat,thseg,gt]=cpe(z,nnn,dm,h,lambda,dsp);
```

Description

`cpe` implements some statistical off-line methods for estimating the change point for mean in white noise.

$$y_t = \begin{cases} \theta_0 + e_t \text{ for } t \leq k \\ \theta_1 + e_t \text{ for } t > k \end{cases}$$

The maximum likelihood and Bayesian change point estimates are based on a Gaussian assumption on the noise, while the non-parametric options only assume symmetric distributions and examine the sign of the signal.

`z=[y u]` output-input data

`nnn` model structure is always set to [0 1 0] in this approach! It is kept for conformity.

`dm` distance measure = `[approach problem]`:

`approach`

1 = Bayesian (Gaussian noise)

2 = ML (Gaussian noise)

3 = Non-parametric I (default)

4 = Non-parametric II

`problem`

1 = Detect increase in mean, θ_0 unknown

2 = Detect increase in mean, $\theta_0 = 0$ known

3 = Detect change in mean, θ_0 unknown (default)

4 = Detect change in mean, $\theta_0 = 0$ known

`h` Threshold

`lambda` Known noise variance

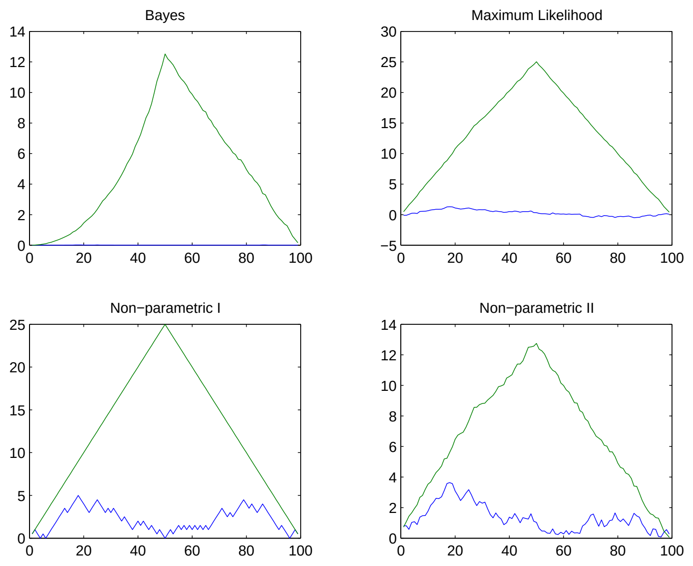
`jumphat` estimated change points

`thseg` estimated parameters

`gt` The distance measure, which is compared to the threshold `h`.

Examples

```
% H0 signal:
y0=[1*ones(50,1);1*ones(50,1)]+.1*randn(100,1);
% H1 signal:
y1=[1*ones(50,1);2*ones(50,1)]+.1*randn(100,1);
% Use an unknown mean model (required in cpe)
nnn=-1;
% Bayesian approach
[jh,ths,GT01,U01]=cpe(y0,nnn,1);
[jh,ths,GT11,U11]=cpe(y1,nnn,1);
% Maximum likelihood approach
[jh,ths,GT02,U12]=cpe(y0,nnn,2);
[jh,ths,GT12,U12]=cpe(y1,nnn,2);
% Non-parametric approach I
[jh,ths,GT03,U13]=cpe(y0,nnn,3);
[jh,ths,GT13,U13]=cpe(y1,nnn,3);
% Non-parametric approach II
[jh,ths,GT04,U14]=cpe(y0,nnn,4);
[jh,ths,GT14,U14]=cpe(y1,nnn,4);
subplot(221), plot([GT01,GT11]), title('Bayes')
subplot(222), plot([GT02,GT12]), title('Maximum Likelihood')
subplot(223), plot([GT03,GT13]), title('Non-parametric I')
subplot(224), plot([GT04,GT14]), title('Non-parametric II')
```



Algorithm See Section 4.5, p. 102–105, in Fredrik Gustafsson. *Adaptive filtering and change detection.* John Wiley & Sons, Ltd, 2000.

References

These methods are surveyed in A. Sen and M.S. Srivastava. On tests for detecting change in the mean. *Annals of Statistics*, 3:98–108, 1975 where further references can be found.

See Also

`detect1` , `detect2` , `detectM` , `glr`

9.8 cusumarl

Purpose `cusumarl` computes the Average Run Length function for the CUSUM test.

Synopsis `L0=cusumarl(h,mu,acc,Nstep,sigma,dsp);`

Description The `cusumarl` function computes the mean time to detection

$$L_0 = E[t_a|h,\theta]$$

where t_a is the stopping time from the CUSUM algorithm:

$$g_t = \max(g_{t-1} + s_t - \nu/2, 0), \quad \text{alarm and} t_a = t \text{ if } g_t > h.$$

The signal is assumed to be white with mean `th` and standard deviation `sigma`, where

$$\text{E}(s_t) = \theta, \quad \text{Var}(s_t) = \sigma^2.$$

It also computes the more general average run length function, where run length is the time the algorithm is running until it is reset (g_t becomes 0). Reset can be caused either by an alarm, when $g_t > h$, or when $g_{t-1} + s_t - \nu/2 < 0$.

Zero initial condition in the CUSUM test is assumed. The solution is given by a *Fredholm integral equation*, which is solved numerically. Siegmund's and Wald's approximations can be obtained as well.

`L0` Mean time between false alarms

`N0` Mean number of samples between resets of CUSUM

`h` Threshold in CUSUM algorithm

`mu` Mean of the signal `th` - the drift parameter `nu`/2

`sigma` Standard deviation of the noise

`acc` Relative error in `L0`.

`Nstep` Number of grid points in the numerical function approximation (default is 200).

`Nstep=-1` gives Siegmund's approximation.

`Nstep=-2` gives Wald's approximation.

Examples

Compute theoretically the mean time between false alarms `ta0` and the mean time to detection `ta1` after a change in the mean of `th=1` for the CUSUM test with threshold `h=5` and drift `nu=th=1`.

```
h=3;
nu=1;
th=1;
mu=th-nu/2;
[ta0,N0] = cusumarl(h,-mu);  % H0
[ta1,N0] = cusumarl(h,+mu);  % H1
[ta0siegmund] = cusumarl(h,-mu,0.5,-1);
[ta1siegmund] = cusumarl(h,+mu,0.5,-1);
[ta0wald] = cusumarl(h,-mu,0.5,-2);
[ta1wald] = cusumarl(h,+mu,0.5,-2);
L0=cusumMC(h,-mu,500);
L1=cusumMC(h,mu,500);
disp([ta0 ta0wald ta0siegmund mean(L0);...
      ta1 ta1wald ta1siegmund mean(L1)])
  127.5000   32.0000  118.5000  115.3580
    6.5000    4.0000    6.5000    6.2680
```

Algorithm

See Section 12.2.2, p. 441–444, in Fredrik Gustafsson. *Adaptive filtering and change detection.* John Wiley & Sons, Ltd, 2000.

References

A thorough treatment is given in C.S. Van Dobben de Bruyn. *Cumulative sum tests: theory and practice.* Hafner, New York, 1968 and M. Basseville and I.V. Nikiforov. *Detection of abrupt changes: theory and application.* Information and system science series. Prentice Hall, Englewood Cliffs, NJ., 1993.

See Also

`cusumdesign` , `cusumMC`

9.9 cusumdesign

Purpose `cusumdesign` converts specified delay for detection to design threshold in the CUSUM algorithm.

Synopsis `h=cusumdesign(L0,mu,h0,acc);`

Description `cusumdesign` finds the threshold `h` in the CUSUM algorithm,

$$g_t = \max(g_{t-1} + s_t - \nu/2, 0), \quad \text{alarm if } g_t > h.$$

which gives the mean delay for detection `L0`. A line search is implemented, where Siegmund's approximation of the ARL function is used to evaluate `L0`.

`L0`	Mean time between false alarms
`h`	Threshold in CUSUM algorithm
`mu`	Mean of the signal `th` - the drift parameter `nu`/2
`h0`	Initial guess
`acc`	Accuracy of final `L0`

Examples Suppose we want to determine the threshold `h` in the CUSUM test from a specified mean time for detection `L0` of a change of magnitude `th`. Set `nu=th`. The threshold is computed by a numerical search and validated as follows:

```
hmax=10;
th=0.4;
L0=10;
h=cusumdesign(L0,th/2,hmax);
[ta0]=cusumarl(h,th/2);
disp([ta0,L0])
   10.5000   10.0000
```

Algorithm A line search in the threshold h is performed using a bisection technique.

See Also `cusumarl` , `cusumMC`

9.10 cusumMC

Purpose

`cusumMC` computes the distribution for the delay for detection in the CUSUM algorithm, by means of Monte Carlo simulations.

Synopsis

```
L=cusumMC(h,mu,Niter,N);
```

Description

The assumptions are that the increments in the CUSUM algorithm are white and Gaussian. The output `L` is a vector of `Niter` alarm times, do `hist(L)`, and the mean delay for detection is obtained by `L0=mean(L)`. This figure can be compared to the theoretical value obtained from `cusumarl`.

`L` Vector of alarm times

`h` Threshold in CUSUM algorithm

`mu` Mean of the signal `th` - the drift parameter `nu`

`Niter` Number of simulations

`N` Number of data in each simulation. Should be chosen larger than all alarm times in `L` to get a reliable mean `L0=mean(L)`.

Examples

Compare the theoretical delay for detection with a Monte Carlo simulation.

```
h=3;
mu=0.5;
ta1 = cusumarl(h,+mu)
ta1 =
    6.5000
L1=cusumMC(h,mu,500);
mean(L1)
ans =
    6.1620
```

See Also

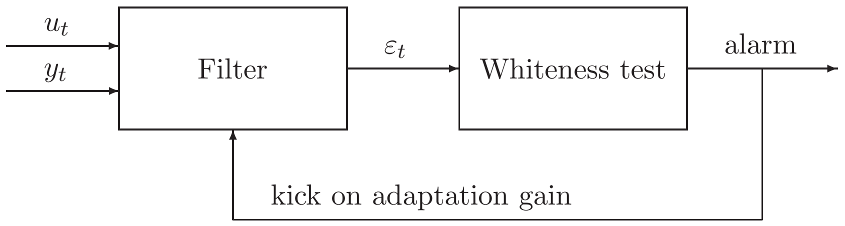
`cusumarl` , `cusumdesign`

9.11 detect1

Purpose `detect1` detects abrupt changes and tracks parameters/states using a whiteness test on the output from an adaptive filter (residual generator).

Synopsis `[threc,jumphat,thseg,gt,ta]=detect1(z,nnn,dm,sr,lambda,dsp)`

Description The adaptive filter in the figure below is matched to the model specified in `nnn` . The output residual is white in the fault free case, and non-whiteness can be tested in various ways.



`z=[y u]` Output-input data

`nnn` Model structure

`dm` Distance measure:

1 = normalized sample mean

2 = normalized squared sample mean

`sr` Stopping rule

`sr=[1 h nu]` CUSUM detector with threshold `h` and drift parameter `nu`

`sr=[2 h nu]` two-sided CUSUM detector with threshold `h` and drift parameter `nu`

`sr=[3 h gamma]` GMA detector with threshold `h` and forgetting `gamma`

`sr=[4 h gamma]` two-sided GMA detector with threshold `h` and forgetting `gamma`

`lambda` Scaling of measurement noise variance. Default: estimated

`jumphat` Estimated change points

`thseg` Smoothed estimated parameters

`threc` Recursively estimated parameters

`gt` The decision function

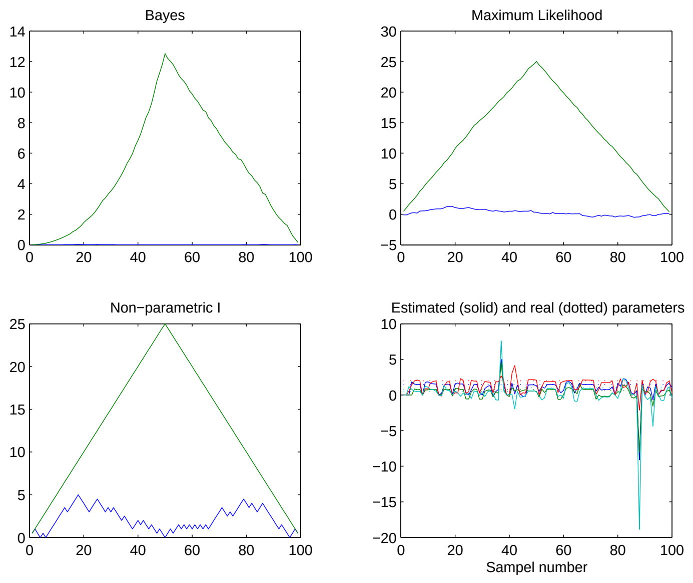
Tuning

Start with a very large threshold h and adjust ν such that $g_t = 0$ more than 50 % of the time. Then set the threshold so the required number of false alarms (this can be done automatically) or delay for detection is obtained. For details, see p. 70 in [10].

Examples

Define and simulate a second order ARX model with two abrupt changes. Then estimate its parameter with the one model filter and compare the recursive estimates with the true parameters in a plot.

```
u=randn(100,1);
e=randn(100,1);
jumptimes=[40 70];
TH=[1.5 0.8 2 0.5;...
    1.5 0.8 2 1;...
    1.5 0.6 2 1]';
nnn=[2 2 1];
y=simchange([0.1*e u],nnn,jumptimes,TH);
z=[y u];
[jhat,thseg,gt,ta,threc]=detect1(z,nnn,2,[1 10 5],0.01);
thplot(threc,TH,jumptimes);
```



Algorithm

Depending on the model structure, `detect1` implements:

- *Change in the mean model*: Algorithm 3.3, p.68.
- *Parametric models*: Section 5.6, p. 148.

- *State space models*: Section 8.10, p. 324.

The *stopping rule* is one of:

- *GMA*: Section 3.2, p. 59.
- *CUSUM*: Algorithm 3.2, p. 66.

Limitations

There is no adaptation between the estimated change points. A quite logical *ad-hoc* algorithm, where for instance RLS is supervised by a change detector, which after an alarm decreases the forgetting factor, must be implemented by `rdetect1` . See Section 7.0.1 for one example.

See Also

`rdetect1` , `detect2` , `detectM` , `adfilter` , `glr` , `mlr`

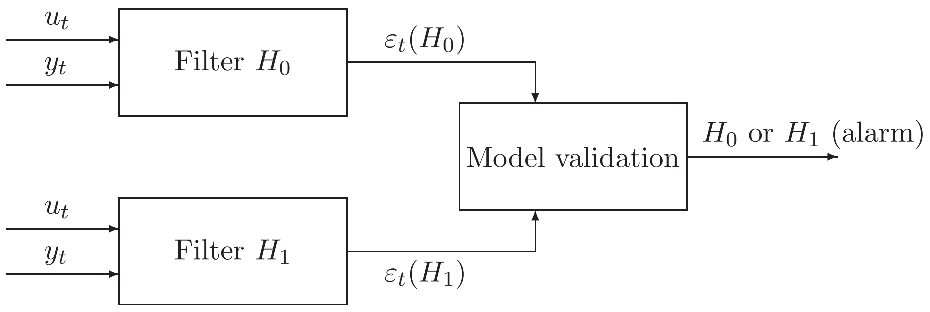
9.12 detect2

Purpose `detect2` detects abrupt changes and tracks parameters with the sliding window approach.

Synopsis

```
[threc,jumphat,thseg,gt,ta]=detect2(z,nnn,dm,sr,L,lambda,dsp);
```

Description The two adaptive filters in the figure below are matched to the hypotheses H_0 and H_1, respectively. These filters are assumed to be windowed least squares estimators with infinite (since last alarm) and size L windows, respectively. A more general approach with different adaptation gain and possibly using different adaptation methods can be achieved with `rdetect2`. Model validation consists in specifying a *distance measure* for the residuals, and a stopping rule.



`z=[y u]` Output-input data.

`nnn` Model structure.

`L` Length of the sliding window.

`dm` Distance measure:

1 = Brandt's GLR algorithm

2 = divergence test

3 = mean

`sr` Stopping rule

`sr=[1 h nu]` CUSUM detector with threshold `h` and drift parameter `nu`

`sr=[2 h nu]` Two-sided CUSUM detector with threshold `h` and drift parameter `nu`

`sr=[3 h gamma]` GMA detector with threshold `h` and forgetting `gamma` `sr=[4 h gamma]` Two-sided GMA detector with threshold `h` and forgetting `gamma`

`sr=[5 h q]` Bayes detector with threshold `h` and jump probability `q`.

`lambda` Noise variance/scaling. Default: estimated

`jumphat` Estimated change points

`thseg` estimated and smoothed parameters

`threc` Recursively estimated parameters

`gt` The decision function

`dsp (0/1)` Display some information (default 0)

Algorithm

See Chapter 6, and in particular Section 6.1 for an overview. The *distance measure* is one of:

- *Brandt's GLR test*: Algorithm 3.6, p.79, and Equation (6.7) on p. 212.
- *The divergence test*: Equation (6.8) on p. 212.

The *stopping rule* is one of:

- *GMA*: Section 3.2, p. 59.
- *CUSUM*: Algorithm 3.2, p. 66.

Limitations

Not prepared for state space models. Use `rdetect2` in that case.

The two filters are not adaptative, except for the restart after each estimated change point. A quite logical *ad-hoc* algorithm, where for instance two RLS filters with different forgetting factors are supervised by model validation, must be implemented by `rdetect2` . See the example in Section 7.0.1, which is easily modified to this case.

Tuning

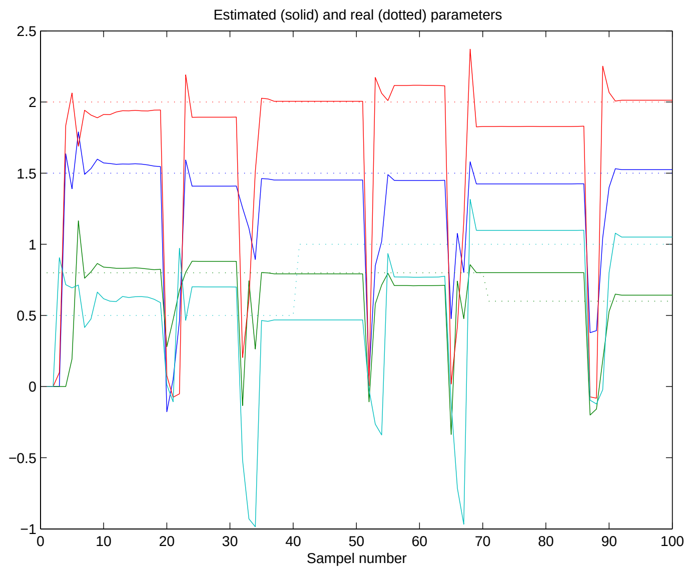
As for `detect1` , the design parameters are a bit tricky to choose. Plot the decision function `gt`. It should be zero most of the time when there is no change. If it isn't, increase the forgetting. If it does not respond fast enough or not at all after a change, decrease the forgetting. Choose then the threshold where you want the alarms.

The window size is coupled to the desired delay for detection. A good starting value is the specified or wanted mean delay for detection. Set the threshold to get the specified false alarm rate. Diagnosis:

- Check visually the variance error in the parameter estimate in the sliding window. If the variance is high, this may lead to many false alarms and the window size should be increased.
- If the estimated change times look like random numbers, too little information is available and the window size should be increased.
- If the change times make sense, the mean delay for detection might be improved by decreasing the window.

Examples Define and simulate a second order ARX model with two abrupt changes. Then estimate its parameter with the two-filter approach and compare the recursive estimates with the true parameters in a plot.

```
u=randn(100,1);
e=randn(100,1);
jumptimes=[40 70];
TH=[1.5 .8 2 0.5;1.5 .8 2 1;1.5 .6 2 1]'
nnn=[2 2 1];
y=simchange([0.1*e u],nnn,jumptimes,TH);
z=[y u];
[jhat,thseg,gt,ta,threc]=detect2(z,nnn,1,[1 0.5 0.3],10,-1);
thplot(threc,TH,jumptimes);
```



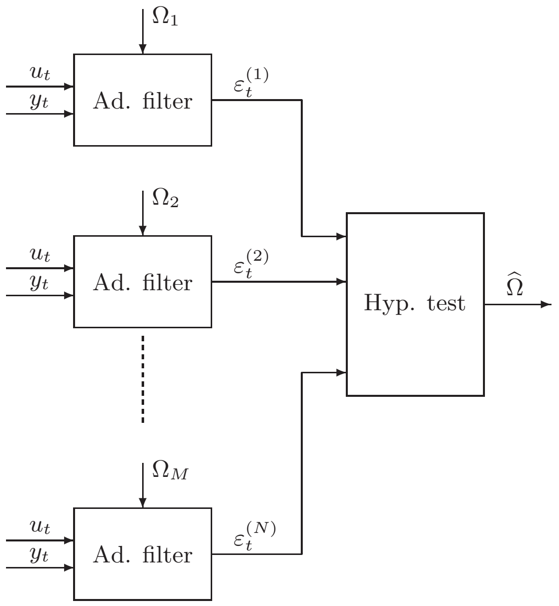
See Also rdetect2 , detect1 , detectM , adfilter , glr , mlr

9.13 detectM

Purpose `detectM` detects changes and tracks parameters/states using an approximation of the MAP/ML estimate of change times.

Synopsis `[threc,jhat,thseg,lamhat,Alfa,XM]=detectM(z,nnn,lf,q,ss,dsp);`

Description The function implements a bank of matched filters that are run in parallel. Each filter is based on a particular assumption on the set of change times $\Omega = \{k_i\}_{i=1}^n$. These hypotheses are compared in a hypothesis test. A clever algorithm is used to reduce the exponentially in time growing number of hypotheses.



Mathematically, the function approximates the *maximum likelihood* (*ML*) and *maximum a posteriori probability* (*MAP*), in a way that the exact likelihood is computed on a subset of all possible combination of change times. The approximation is in many cases identical to the exact solution.

`z` The output-input data vector `z=[y u]` or `z=[y Phi]`.

`nnn` Model structure (see `nnn`)

`lf` Loss function:

`lf=[1 lambda]` MAP estimate with known noise scaling lambda `lf=[2]` MAP estimate with unknown noise scaling (default)

`lf=[3]` BIC/MDL penalty term for complexity

`lf=[4]` AIC penalty term for complexity

`q` The probability that the system jumps at each sample (default 0.5, which gives ML)

`ss` Search scheme = `[M,ll,minseg]`

`M` The number of filters (default 8 plus number of parameters)

`ll` Guaranteed lifelength of a jump sequence (default `M-4`)

`minseg` The minimum allowed segment size (default 0)

`dsp (0/1)` Display some information (default 0)

`jumphat` The estimated jump times

`thseg` Estimated and smoothed piecewise constant parameters.

`threc` Recursive estimate of parameters.

`lamhat` Estimated and smoothed piecewise constant noise variance.

`Alfa` Probability of each filter as a function of time.

`XM` Hypothesis histories for each filter (for `hypplot`).

The ML estimate is achieved if `q=1/2` (default). The exact ML/MAP estimate of change times is obtained if `M=N+1` (and `ll=M-2`) where `N` is the number of data. If `M` is less an approximative local search is performed. `thseg` is the smoothed parameter estimate conditioned on the estimated change times. Note that the syntax is the same as for the parameter estimation methods in SITB, e.g. `arx` and `rarx` and time series analysis is possible as well by letting `z=y` and `nnn=na`.

Tuning

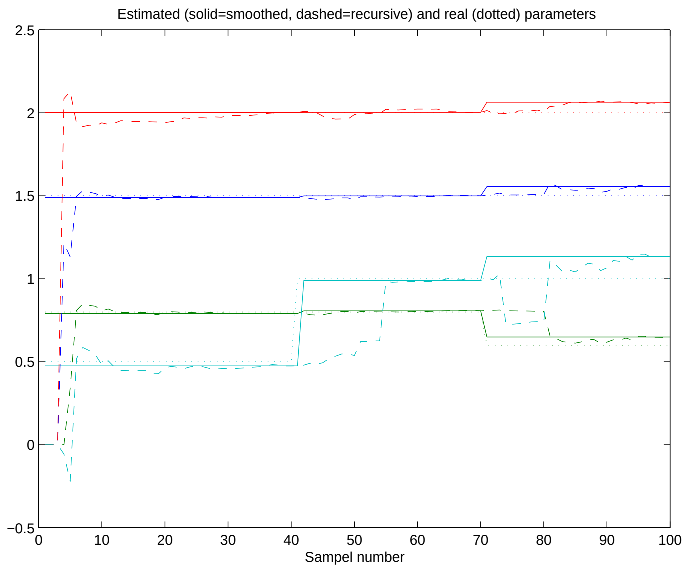
The default design parameter usually works quite well. The number of change times can be tuned by q. The smaller q, the fewer change times. If this is not enough, try to use `lf=[1 lambda]` and tune `lambda` until a sufficient number of changes are detected. Generally, `lf=[1 lambda]` should be used when prior information about the noise variance is available. See Section 7.5.2 in [10] for more information.

Examples

Define and simulate a second order ARX model with two abrupt changes. Then estimate its parameter with the `detectM` filter and compare the recursive and smoothed estimates with the true parameters in a plot.

```
u=randn(100,1);
e=randn(100,1);
```

```
jumptimes=[40 70];
TH=[1.5 .8 2 0.5;1.5 .8 2 1;1.5 .6 2 1]'
nnn=[2 2 1];
y=simchange([0.1*e u],nnn,jumptimes,TH);
z=[y u];
[jhat,thseg,lamhat,threc]=detectM(z,nnn);
thplot(thseg,TH,jumptimes,threc);
```



Algorithm

Depending on the model structure, `detectM` implements:

- *Change in the mean model*: Algorithm 4.1, p. 94.
- *Parametric model*: Algorithm 7.1, p. 244.
- *State space model*: Algorithm 10.2, p. 386.

See Also

`hypplot` , `detect1` , `detect2` , `adfilter` , `glr` , `mlr` , `gibbs`

9.14 faultdetect

Purpose `faultdetect` implements fault detection using the parity space approach.

Synopsis `[r,Ordo,Hu,Hd,Hf,Null,T]=faultdetect(z,nnn,L,R,F,minorder);`

Description

`r` The residuals are zero during non-faulty mode. The optional transformation using `R` and `F` makes the residual parallel to column i of `R` when the fault is parallel to column i of `F`, and the size of the residual is the fault size.

`L` Size of sliding window

`R` Residual structure

`F` Fault pattern structure

`minorder` Compute minimum order residual filters (default 0)

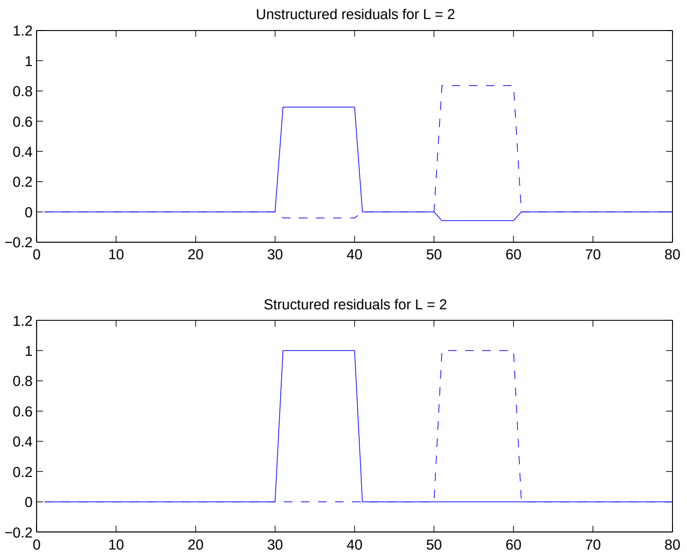
Examples A DC motor is simulated using step-wise (fifth argument of `simchange`) faults in state one and two, respectively. The structured residuals diagnosis which change has happened. The residual vector becomes the first column in `R` after a change parallel to `F` and so on. From the plot we see that one structured residual becomes exactly one after the first fault, which means that the diagnosed fault is exactly the first column of `F`.

See Section 11.5.1, p. 405, for more information on this example.

```
nnn='nnnDCm1';
N=80;  f1=[1;0];  f2=[0;1];
TH=[f1 -f1 f2 -f2];
jumps=[3*N/8 4*N/8 5*N/8 6*N/8];
u=randn(N,1);
d=[zeros(N/8,1); randn(N/8,1); zeros(6*N/8,1)];
[y,x0]=simchange([u d],nnn,jumps,TH,1);

L=2;   z=[y u];
r1=faultdetect(z,nnn,L);
F=eye(2);  R=eye(2);
r2=faultdetect(z,nnn,L,R,F);
subplot(211),  plot(r1)
title(['Unstructured residuals for L = ',num2str(L)])
```

```
subplot(212),  plot(r2)
title(['Structured residuals for L = ',num2str(L)])
```



Algorithm Algorithm 11.1, p. 399.

9.15 gibbs

Purpose

`gibbs` detects changes and tracks parameters/states using an approximation of the MAP/ML estimate of change times.

Synopsis

```
[threc,jhat,thseg,lamhat,Alfa,XM]=detectM(z,nnn,lf,q,ss,dsp);
```

Description

The function implements a Gibbs-Metropolis resampling algorithm for resampling the jump sequence in a MCMC fashion. At each iteration, each jump time is resampled individually, and the new time is kept or rejected with prpobability proportional to its likelihood. Mathematically, the function approximates the *maximum likelihood* (*ML*) and *maximum a posteriori probability* (*MAP*), in a way that the exact likelihood is computed on a subset of all possible combination of change times. That is, `gibbs` and `detectM` approximates the same optimality criteria, but with different algorithms.

`z` The output-input data vector `z=[y u]` or `z=[y Phi]`.

`nnn` Model structure (see `nnn`)

`lf` Loss function:

`lf=[1 lambda]` MAP estimate with known noise scaling lambda

`lf=[2]` MAP estimate with unknown noise scaling (default)

`lf=[3]` BIC/MDL penalty term for complexity

`lf=[4]` AIC penalty term for complexity

`q` The probability that the system jumps at each sample (default 0.5, which gives ML)

`par` Search scheme = `[n M prior]`
`n` is the number of jumps (must be given),
`M` the number of Gibbs iterations and
`prior` denotes the prior distribution on the jump times (`prior`=0 uniform, `prior` > 0 Gaussian with standard deviation = `prior`). The prior controls the resampled jump time.

`opt` `opt=[o1 o2 o3]` creates different plots.
$o1 \neq 0$ gives an off-line jump plot.
$o2 \neq 0$ gives an on-line jump plot.
$o3 \neq 0$ gives a histogram with assumed burnin-time $0.25M$.

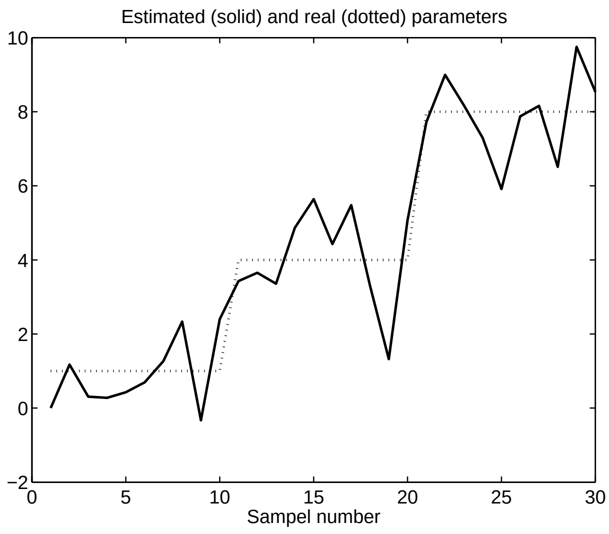
`jumpinit` Initial estimate of jump sequence.

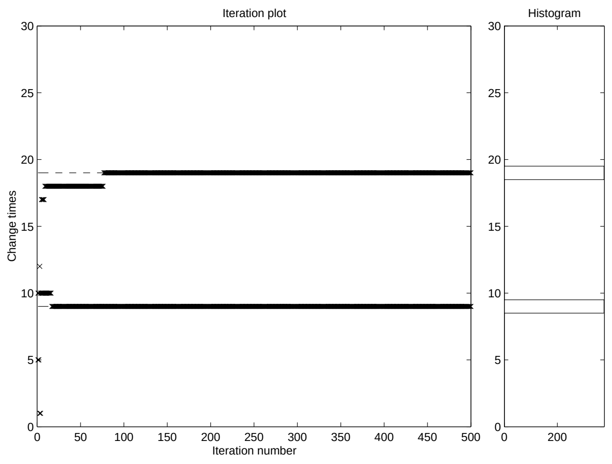
The ML estimate is achieved if `q=1/2` (default).

Tuning

Examples

```
rand('state',1)
randn('state',3)
N=30;
TH=[1 4 8];
jump=[N/3 2*N/3];
y=simchange(randn(N,1),-1,jump,TH);
thplot(y,TH,jump);
[threc,kn]=gibbs(y,-1,[1 1],0.5,[2 500 0],[0 1 1],[5 10]);
```





Algorithm

Depending on the model structure, `gibbs` implements:

- *Change in the mean model*: Algorithm 4.2, p. 102.
- *Parametric model*: Special case of below, not explicitly given.
- *State space model*: Algorithm 10.6, p. 394.

See Also

`detectM`

9.16 glr

Purpose `glr` estimates the jump time and size of an additive change in the state of a linear system.

Synopsis `[Xhat,jumphat,NUhat,ta,LR]=glr(z,nnn,M,h,pnu);`

Description Computes the log likelihood ratio of a jump at time k versus no jump in a state space model

$$\begin{aligned} x_{t+1} &= Ax_t + B_u u_t + B_v v_t + \sigma_{t-k} B_\nu \nu \\ y_t &= Cx_t + Du_t + e_t, \end{aligned}$$

The generalized likelihood ratio replaces the unknown jump magnitude by its most likely value:

$$l(k) = \max_\nu \log \frac{p(Y|\nu,k)}{p(Y)}.$$

Then a change at time $\hat{k}$ is decided if

$$\hat{k} = \arg\max_k l(k) > h.$$

The test can be applied off-line, when the test above is applied for all $1 \leq k \leq N$ first when all data are processed, or on-line, when the test is applied for either all $k < t$ or using a sliding window for $t - M < k < t$.

`jumphat` Estimated jump time, no jump if `jumphat=0`

`lr` Log likelihood ratio

`nuhat` Estimated jump

`z` Output-input data

`nnn` Structure parameters

`M` Size of sliding window of considered change times.

If `M<1` all time instants are considered (default).

If `M=1` a Gaussian prior on the jump is used.

`h` Threshold in the test (default `h=10`)

`pnu` Covariance of the jump when `M=1`. Default is `pnu=100`.

Examples

Simulate a state space model corresponding to the transfer function $(q^{-1}+q^{-2})/(1+q^{-1}+q^{-2})$ with a sudden increase in state with $(10,10)^T$ at time 50. Then `glr` tries to find the change time and magnitude.

```
u=randn(100,1);
[a,b,c,d]=tf2ss([0 1 -1],[1 -1 0.8]);
Q=0.0001*eye(2);
R=0.01;
nnn=ss2nnn(a,b,c,d,Q,R);
jumptime=50;
TH=[10 10]';
[y,x]=simchange(u,nnn,jumptime,TH);
[jhat,nuhat,xhat,lr]=glr([y u],nnn);
thplot(x,xhat)
nuhat,jhat
```

Check the likelihood ratio `lr` so that the threshold is set properly.

Algorithm

Algorithm 9.1, p. 350.

References

The original reference is A.S. Willsky and H.L. Jones. A generalized likelihood ratio approach to the detection and estimation of jumps in linear systems. *IEEE Transactions on Automatic Control*, 21:108–112, 1976. A general treatment is given in M. Basseville and I.V. Nikiforov. *Detection of abrupt changes: theory and application.* Information and system science series. Prentice Hall, Englewood Cliffs, NJ., 1993. Implementation issues and an algorithm are discussed in F. Gustafsson. The marginalized likelihood ratio test for detecting abrupt changes. *IEEE Transactions on Automatic Control*, 41(1):66–78, 1996.

9.17 imm

Purpose `imm` implements the *Interactive Multiple Model* filter.

Synopsis `[xhatp,xhatf,Alfa] = imm(z,nnn,wbar,q)`

Description Common algorithm used for target tracking. Use this in combination with the motion models for the aircraft in `nnnplane` .

`xhatp` Filter predictions $\hat{x}_{t|t-1}$.

`xhatp` Filter estimates $\hat{x}_{t|t}$.

`alfa` Wieghts of the individual filters as a function of time.

`z` Position related measurements y_t for target tracking.

`nnn` Motion model, see `nnnplane` .

`wbar` Possible turn rates. Default `[0 -3 3]` degrees per second.

`q` Transition probabilities. Default is `0.5*eye(3)+0.5/3*ones(3)`.

Algorithm Algorithm 10.4 on page 389.

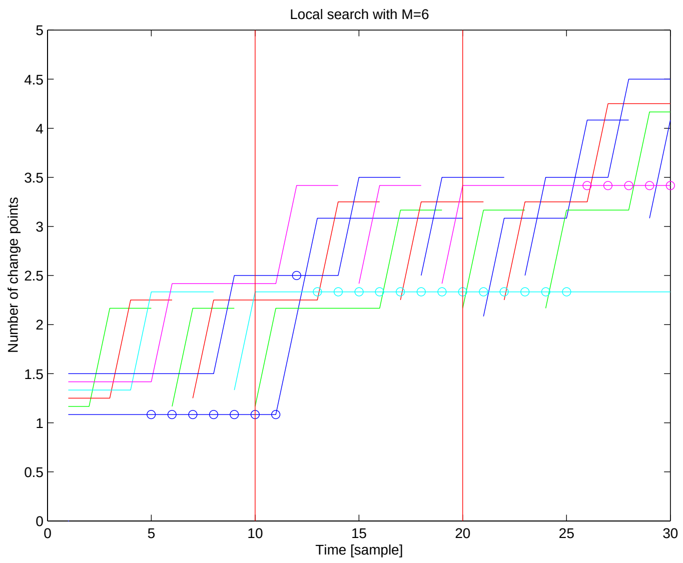
9.18 hypplot

Purpose `hypplot` illustrates the local search of hypotheses in `detectM`.

Synopsis `hypplot(Xn,jump);`

Examples

```
jumps=[10 20];
TH=[0 5 8];
[y,th0]=simchange(randn(30,1),-1,jumps,TH);
M=6;
[jumphat,thseg,lamseg,threc,Alfa,hist]=detectM(y,-1,[1 1],0.5,M,0);
cdfigure(1);
hypplot(hist,jumphat)
title(['Local search with M=',num2str(M)])
```



The plot illustrates a filter bank with 6 parallel matched filters. Each filter is matched to a hypothesis of a sequence of change times. Each line shows one such hypothesis, where each change time is illustrated with a jump. Terminating lines indicate that the hypothesis is rejected, and the filter is initiated to a new one. The circles denote at each time the most likely hypothesis. Thus, the first rather large change is detected by one matched filter after two samples, while the second change require six samples.

See Also `detectM`

9.19 mlr

Purpose `mlr` estimates the jump time for an additive change in the state of a linear system.

Synopsis `[jumphat,lr]=mlr(z,nnn,lf,L)`

Description The function computes the log likelihood ratio of a jump at time k versus no jump in a state space model

$$\begin{aligned} x_{t+1} &= Ax_t + B_u u_t + B_v v_t + \delta_{t-k} B_\nu \nu \\ y_t &= Cx_t + Du_t + e_t, \end{aligned}$$

The marginalized likelihood ratio integrates out the influence of the unknown jump magnitude,

$$l(k) = \int_{-\infty}^{\infty} \log \frac{p(Y|\nu,k)}{p(Y)} d\nu.$$

Then the change time $\hat{k}$ is estimated as

$$\hat{k} = \arg \max_{1 \le k \le N} l(k),$$

where $k = N$ is to be interpreted as no jump. Note the principal difference to the `glr` test, where a threshold is involved. MLR is an estimation approach, and GLR a hypothesis test. The test is aimed to be off-line.

`jumphat` Estimated jump time, no jump if `jumphat=0`.

`lr` Log likelihood ratio.

`z` Output-input data.

`nnn` Model structure.

`lf` Loss function.

`lf = [1 lambda];` known noise level.

`lf = 2;` unknown constant noise level.

`lf = 3;` unknown changing noise level (default).

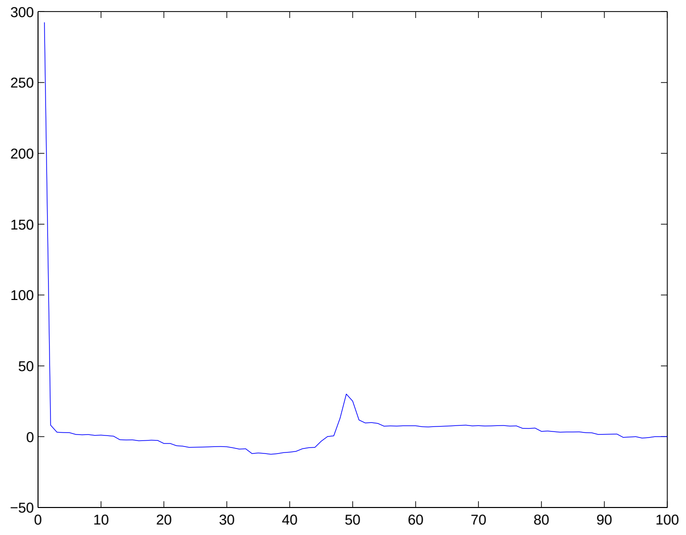
`L` Number of ignorance data at the end. (Default `length(H)`.)

Limitations The algorithm may have numerical problems when the state matrix A has eigenvalues different from one. Examples on A matrices with eigenvalues one are most models used for navigation and tracking and parametric models on state space form.

No estimate of the change magnitude ν is computed, so isolation and diagnosis are unsolved problems.

Examples Simulate a state space model corresponding to the transfer function $(q^{-1}+q^{-2})/(1+q^{-1}+q^{-2})$ with a sudden increase in state with $(10,10)^T$ at time 50. Then `mlr` tries to find the change time and magnitude.

```
[a,b,c,d]=tf2ss([0 1 1],[1 1 1]);
Q=0.0001*eye(2);
R=0.01;
w=randn(100,2);
nnn=ss2nnn(a,b,c,d,Q,R);
jumptime=50;
TH=[0 0;10 10]';
y=simchange([0.1*e u 0.01*w],nnn,jumptime,TH);
[jhat,lr]=mlr([y u],nnn);
jhat
jhat =
    49
plot(lr)
```



Algorithm Algorithm 9.2, p. 358.

See Also `glr`

9.20 modeltype

Purpose `modeltype` determines the type of model implicitely given in `nnn`

Synopsis

```
mtype=modeltype(nnn);
```

Description The `mtype` definition of models are used internally by all filters for switching purposes. It is defined as

`mtype = 0`	No model (only noise)
`mtype = 1`	Unknown mean in noise
`mtype = 2`	Linear regression with regressor in z=[y PHI]
`mtype = 3`	AR model
`mtype = 4`	ARX model
`mtype = 5`	State space model, defined according to Simulink S-functions.

For `mtype<3`, the model formats relate as `mtype=-nnn`. For `mtype=3,4`, the model formats are essentially equivalent to `nn` in the System Identification ToolBox. See `nnn` for more details on how the models are defined.

See Also `nnn`

9.21 multihyp

Purpose `multihyp` detects abrupt changes in parametric models and performs diagnosis on the hypothesis parameter change and noise variance change.

Synopsis `[jhat,jtype,thseg,lamseg]=multihyp(z,nnn,L,hypmask,q,dsp)`

Description The algorithm applies to all parametric models `nnn` (not state space models) and makes the following hypothesis test at each time instant recursively:

`H0` No change in parameters or noise variance.

`H1` Change in parameters and the no change in noise variance.

`H2` Change in noise variance but no change in parameters.

`H3` Change in both parameters and noise variance.

The hypothesis test is performed by comparing the outputs from one non-adaptive filter and one adaptive filter implemented by the sliding window approach. The algorithm is a two-filter one similar to what is done in `detect2` . Variance changes can also be detected in `detectM` , but there no diagnosis is done. That is, here you don't know whether the parameter or variance change is significant.

There is an optional mask, `hypmask`, to exclude one or more of the hypothesis. For instance, default is to exclude `H3`. By excluding both `H2, H3`, the algorithm `detect2` is obtained essentially.

`jumptime` Detected jump times:

`jumptype` Vector of diagnosed changes (0,1,2,3) according to the hypotheses.

`z=[y u]` Output-input data.

`nnn` Model structure.

`L` Length of the sliding window. Default `L=50`.

`hypmask` Possibility to mask, e.g. `[0 1 0]` means only variance changes considered. Default: `[1 1 0]`, that is, either parameter or variance change but not both at the same time.

`q` Prior probability for a jump at time t (default `q=0.5` (ML))

`dsp` Display some information every `dsp` sample. Default `dsp=0`.

Limitations

The algorithm is not intended for state space models.

Examples

Simulate a second order FIR model with noise with 4 abrupt changes: two parameter and two noise variance changes. The `multihyp` filter should be able to find the type of change and approximate position.

```
TH=[1.6 0.64;...
   -1.6 0.64;...
   -1.6 0.64;...
   -1.6 0.64;...
    1.6 0.64;]';
Lam=[1 1 10 1 1];
jumptimes=[50;100;150;200];
N=250;
nnn=[0 2 1];  % FIR model
L=10;         % Size of sliding window
e=randn(N,1);
u=randn(N,1);
y=simchange([e u],nnn,jumptimes,TH,Lam);
[jhat,jtype,thseg,lamseg]=multihyp([y u],nnn,L);
disp([jhat;jtype])
    54   103   157   201
     1     2     2     1
```

Algorithm

Algorithm 6.1, p. 224.

Tuning

See `detect2` .

References

Theory, algorithm and a communication application are found in C. Carlemalm and F. Gustafsson. On detection and discrimination of double talk and echo path changes in a telephone channel. In A. Prochazka, J. Uhlir, P.J.W. Rayner, and N.G. Kingsbury, editors, *Signal analysis and prediction*, Applied and Numerical Harmonic Analysis. Birkhauser Boston, 1998.

See Also

`detect2` , `detectM` , `detect1`

9.22 nnn

Purpose `nnn` displays help documentation for the model structure object `nnn`.

Synopsis `nnn`

Description `nnn` denotes the model structure for data `z`. There are seven different model types, enumerated by `mtype` = 0–6.

0 `nnn = 0` and `z = y`; Assumes noise with piecewise constant variance $\lambda(t)$

$$y(t) = e(t), \quad \mathrm{E}e(t)^2 = \lambda(t).$$

1 `nnn = -1` and z = y; Assumes a piecewise constant mean model

$$y(t) = \theta(t) + e(t), \quad \mathrm{E}e(t)^2 = \lambda(t).$$

2 `nnn = -2` and `z = [y Phi]` Assumes a piecewise constant linear regression model

$$y(t) = \varphi(t)^T\theta(t) + e(t), \quad \mathrm{E}e(t)^2 = \lambda(t).$$

3 `nnn = na` and `z = y` Assumes a piecewise constant AR(na) model:

$$\begin{aligned} y(t) =& -\theta_1(t)y(t-1) - .. - \theta_{n_a}(t)y(t-n_a) + e(t) \\ =& \varphi(t)^T\theta(t) + e(t), \quad \mathrm{E}e(t)^2 = \lambda(t). \end{aligned}$$

4 `nnn = [na nb nk]` and `z = [y u]`
Assumes a piecewise constant ARX(na,nb,nk) model:

$$\begin{aligned} y(t) =& -\theta_1(t)y(t-1) - .. - \theta_{n_a}(t)y(t-n_a) + \\ & + \theta_{n_a+1}u(t-n_k) + .. + \theta_{na+nb}u(t-n_k-n_b) + e(t) \\ =& \varphi(t)^T\theta(t) + e(t), \quad \mathrm{E}e(t)^2 = \lambda(t). \end{aligned}$$

5 `nnn='filename'` and `z=[y u]` is a general user specified state space model using an S-function syntax:

```
[n,x0,P0]=filename(0,0,0,0);
 n = [ nx nu nv nd nf ny];
xp=filename(t,x,th,2);
y =filename(t,x,th,3);
th=[th;lambda];
[A,Bu,Bv,Bd,Bth,Q]=filename(t,x,th,5);
[C,Du,Dth,R]=filename(t,x,th,6);
```

Note that the last two optional calls are not standard simulink, but used to increase speed in filtering when provided. The state space model is defined by

$$\begin{aligned}
x(t+1) &= Ax(t) + Bu(t) + B_v v(t) + B_d d(t) + B_\theta \theta(t) \\
y(t) &= Cx(t) + D_u u(t) + e(t) + D_\theta \theta(t) \\
\mathrm{E} w(t)w(t)^T &= Q \\
\mathrm{E} e(t)e(t)^T &= R\lambda(t) \\
\mathrm{Cov} x(0)x(0)^T &= P_0 \\
\mathrm{E} x(0) &= x_0
\end{aligned}$$

This rather complicated way of expressing a state space model is needed for implementing time-varying or extended Kalman filters. The model format covers all conceivable dynamic systems. There is a m-file template, `nnnmodel`, which is recommended to use when defining your own model. An advantage of the increased generality is that it can also be used for pseudo-linear regressions and general parametric model structures (ARMAX, OE etc.). Then `Xfun` is a state that can be used to store old data that is needed to compute the gradient $C(t) = \varphi(t)^T$.

Note that the state space model generally is MIMO (multi-input multi-output), while the other parametric models are SISO.

Let n denote the number of jump times. For state space model (5), the $\dim\theta = d \times n$ matrix TH (Θ) and $1 \times n$ vector `jumptimes` define the amplitude and time for the additive changes. For the linear regression models (1-4), the $1 \times n$ vector `jumptimes` defines the segments where the parameter vector is piecewise constant and the $\dim\theta = d \times (n+1)$ matrix `TH` contains the corresponding parameter vectors.

This structure is used for simulation, adaptive filtering and change detection.

Examples

Here follows a definition of a DC motor contained in the template file `nnnDCm1` .

```
function [out1,out2,out3,out4,out5,out6]=nnnDCm1(t,x,uinp,flag);

% Sampled state space model of Dc motor with x=[phi w]'
%
%               1
%      G(s) = ------
%             s(s+1)
%
% Inputs are uinp=[u;d;f;lam];
% Noise inputs are simulated within the model

T=0.4;
nx=2; nu=1; nv=0; nd=1; nth=2; ny=2;

out1=[]; out2=[]; out3=[]; out4=[]; out5=[]; out6=[];

if flag==0;    % Initialization
    P0=zeros(nx);
    x0=zeros(nx,1);
    n=[nx nu nv nd nth ny];
    out1=n; out2=x0; out3=P0;
elseif flag==1;
elseif flag==2;
    [A,Bu,Bv,Bd,Bf,Q]=feval('nnnDCm1',0,x,0,5);
    u=uinp(1:nu);
    d=uinp(nu+1:nu+nd);
    f=uinp(nu+nd+1:nu+nd+nth);
    xp=A*x+Bu*u+Bd*d+Bf*f;
    out1=xp;
elseif flag==3;
    [C,Du,Df,R]=feval('nnnDCm1',0,x,0,6);
    u=uinp(1:nu);
    d=uinp(nu+1:nu+nd);
    f=uinp(nu+nd+1:nu+nd+nth);
    y=C*x+Du*u+Df*f;
    out1=y;
elseif flag==4;
  t=t+T;
  out1=t;
elseif flag==5;
  a=exp(-T);
  A=[1 1-a;0 a];
  Bu=[T-1+a; 1-a];
  Bd=[0;0];
  Bf=eye(2);
  Q=[];
  out1=A; out2=Bu; out4=Bd; out5=Bf; out6=Q;
elseif flag==6;  % Output equation matrices
  C=[1 0; 0 0];
  C=eye(2);
  Du=[0;0];
  Df=zeros(2);
  R=[];
  out1=C;
  out2=Du;
  out3=Df;
  out4=R;
end
```

See Also

modeltype

9.23 nnnplane

Purpose

`nnnplane` implements a variety of motion models used in the area of target tracking.

Description

The chosen motion model is controlled by the *global* parameter `NNNmethod`.

The state variables are defined in continuous time as as subset of the following:

- x_1 and x_2 denote the cartesian position in the horizontal plane.
- v_1 and v_2 denote the cartesian velocity in the horizontal plane.
- v denotes the speed.
- h denotes the heading angle, so $v_1 = v\sin(h)$ and $v_2 = v\cos(h)$.
- $\omega = \dot{h}$ denotes the turn rate.

The state space model must be transformed to discrete time before the Kalman filter applies. For non-linear models, there are two alternatives, either to linearize the non-linear model and then apply standard sampling of state space models, or to try to sample the continuous time non-linear model to a discrete time non-linear model, which is then linearized. We refer to these principles as *discretized linearization* and *linearized discretization*, respectively. The global switching parameter `NNNmethod` can now be defined as follows:

1. Four state linear model $x = (x^{(1)}, x^{(2)}, v^{(1)}, v^{(2)})^T$.
2. Four state linear model as above with a time-varying and state dependent covariance matrix $Q(x_t)$, corresponding to velocity noise mainly in lateral direction, as will be explained below.
3. Six state linear model with $x = (x^{(1)}, x^{(2)}, v^{(1)}, v^{(2)}, a^{(1)}, a^{(2)})^T$.
4. Six state linear model as above with a time-varying and state dependent covariance matrix $Q(x_t)$, corresponding to velocity noise mainly in lateral direction, as will be explained below.
5. Five state coordinated turn model, cartesian velocity, linearized discretization, with $x = (x^{(1)}, x^{(2)}, v^{(1)}, v^{(2)}, \omega)^T$ (ω is turn rate).
6. Five state coordinated turn model, polar velocity, linearized discretization, with $x = (x^{(1)}, x^{(2)}, v, h, \omega)^T$ (h is heading angle).

7. Five state coordinated turn model, cartesian velocity, discretized linearization.
8. Five state coordinated turn model, polar velocity, discretized linearization.

The first five ones seem to be the most common ones in applications. A covariance matrix Q corresponding to, for example, 100 times larger maximal aceleration in lateral direction compared to longitudinal direction ($q_v = 0.01 q_w$) is given by

$$\theta = \arctan(x^{(4)}/x^{(3)})$$

$$Q = \begin{pmatrix} q_v \cos^2(\theta) + q_w \sin^2(\theta) & (q_v - q_w)\sin(\theta)\cos(\theta) \\ (q_v - q_w)\sin(\theta)\cos(\theta) & q_v \sin^2(\theta) + q_w \cos^2(\theta) \end{pmatrix}$$

$$B_v = \begin{pmatrix} T^2/2 & 0 \\ 0 & T^2/2 \\ T & 0 \\ 0 & T \end{pmatrix} \quad \text{or} \quad B_v = \begin{pmatrix} T^3/3 & 0 \\ 0 & T^3/3 \\ T^2/2 & 0 \\ 0 & T^2/2 \\ T & 0 \\ 0 & T \end{pmatrix}.$$

The reason for introducing an ugly global variable for switching purposes is solely to keep the Simulink-like syntax. See Section 8.9 in the text book for a thorough treatment of model definitions and their extended Kalman filters. The flexibility of `nnnplane` is illustrated by the fact that all toolbox filters and change detectors apply to all of these non-linear and linear models. For instance, `adkalman` applies the Kalman filter, or extended Kalman filter, depending on the model.

Examples

See Section 5.

See Also

`nnn`

9.24 openeye

Purpose `openeye` computes a so-called open-eye measure, which is a measure of how well an equalizer works for a given channel.

Synopsis

```
m=openeye(b,c);
```

Description The impulse response of the combined channel and equalizer, assuming FIR models, is

$$h_t = (b * c)_t$$

where $*$ denotes convolution. The best one can hope for is $h_t \approx m\delta_{t-\tau}$, where τ is an unknown time-delay, and the modulus m with $|m| = 1$ is unknown. The modulus and delay do not matter for the performance when differential modulation is used and can in such applications be ignored.

Consider the case of input alphabet $u_t = \pm 1$, in which case the modulus is $m = \pm 1$. For succesful demodulation and assuming no noise on the measurements, it is enough that the largest component of h_t is larger than the sum of the other components. That is, $m_t > 0$, where

$$m_t = 2 - \frac{\sum(|h_t|)}{\max_t |h_t|}.$$

If the equalizer is a perfect inverse of the channel (which is impossible for FIR channel and equalizer), then $m_t = 1$. The open-eye condition corresponds to $m_t > 0$, when perfect reconstruction is possible, assuming there is no noise. The larger m_t, the larger noise can be tolerated. Look at the plot in the `blindeq` example to see how a blind equalizer improves the open-eye measure.

Both the channel and equalizer FIR parameters can either be constant (a row vector) or time-varying (a matrix).

`b` FIR parameters for channel.

`c` FIR parameters for equalizer.

`m` Open-eye measure

Examples Compute the open-eye measure for two initial equalizers.

```
b=[0.3 1 0.3];             % channel
```

```
c1=[0 -0.1 1 -0.1 0]';  % equalizer
m1=openeye(b,c1)
m1 =
    0.5106
c2=[0 1 1 1 0]';  % another equalizer
m2=openeye(b,c2)
m2 =
   -1.0000
```

This means that `c1` will work well, but `c2` is useless. See also the example in `blindeq` .

See Also

`blindeq`

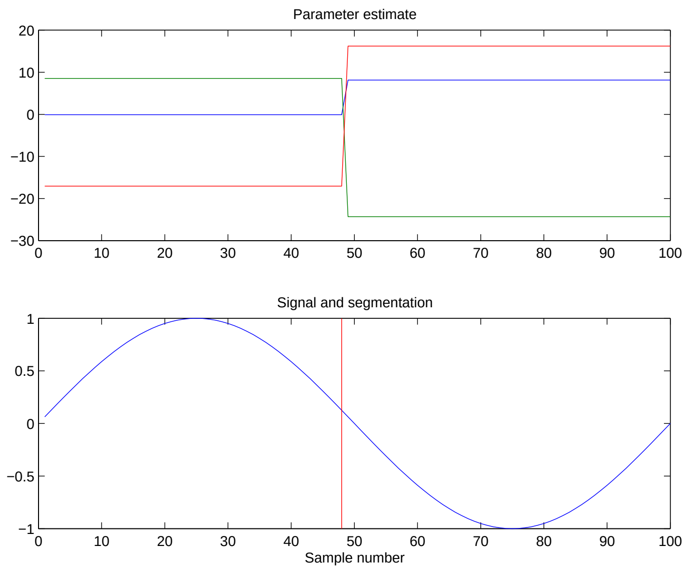
9.25 par2segm

Purpose `par2segm` converts a $d \times N$ matrix of piecewise constant parameter vectors to a $d \times n$ matrix of parameter vectors in each segment.

Synopsis `[TH,jumptimes,N]=par2segm(th);`

Examples Approximate a sinusoid with a piecewise constant second order polynomial, and present the coefficients in each segment.

```
t=(1:100)'/100;
y=sin(t*2*pi);
[jhat,thseg]=detectM([y ones(size(t)) t t.^2],-2,[1 0.001]);
plot(thseg')
par2segm(thseg)
ans =
    -0.0838    8.1438
    8.5421  -24.3353
  -17.0447   16.2328
```



See Also segm2par

9.26 power2

Purpose

`power2` computes the statistical power for testing a parameter change using `detect2` in a FIR model.

Synopsis

```
beta=power2(th0,th1,R,lambda,N,alpha);
```

Description

Asymptotic expression for the power function in the two-filter approach for a constant known noise variance. `th0` is assumed known and `th1` estimated in a sliding window. The result holds for the distance measures GLR, divergence test and normalized parameter norm. For unnormalized parameter norm, replace `N` by `sqrt(N)`. The mean time to detection is approximately `1/beta` while the mean time between false alarms is approximately `1/(1-alpha)`.

Note that the actual value of `beta` is not reliable (see the demo). Qualitatively it is correct in the sense that when comparing different model structures it gives the highest value to the one that performs best. Thus, this function can be used to compare different model structures in the case where the parameter vector before and after the change is known. With a known change time, these can both be estimated from data, and the most powerful model structure can be determined.

Note also that in many cases under-modelling is good for change detection performance.

`th0`	FIR parameter vector before change
`th1`	FIR parameter vector after change
`R`	$N * P(N)$ where P is the covariance matrix
`lambda`	Noise variance
`N`	Size of sliding window
`alpha`	Confidence level in the hypothesis test

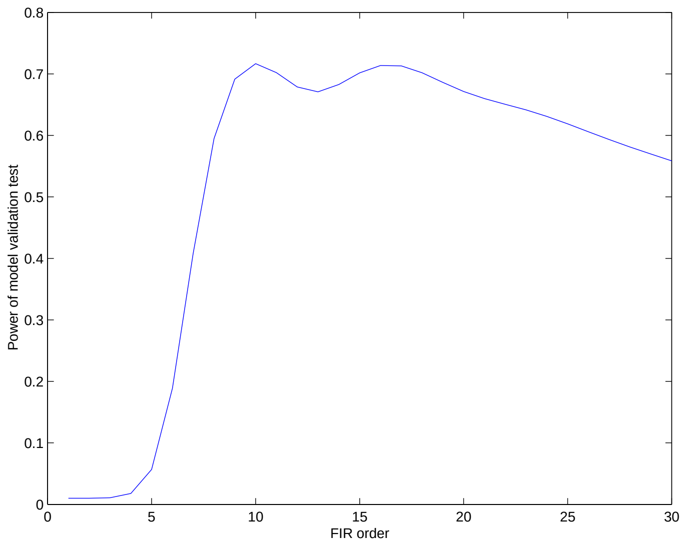
Examples

```
nb=30;
lambda=0.1;
N=50;
alpha0=0.99;                   % confidence level 99%
```

```
Calpha=erfinv(2*(alpha0)-1);    % Assuming AsN of test statistics

% The "Astrom system", pole in 0.8376*exp(i*0.4592)
th0=dimpulse(.2/1.5*[0 1 .5],[1 -1.5 0.7],nb);
r=0.8376*exp(i*0.4);
% Move the phase angle from 0.46 to 0.4
th1=dimpulse(.2/1.5*[0 1 .5],[1 -real(r+r') real(r*r')],nb);

for m=1:nb;
  beta0(m)=power2(th0(1:m),th1(1:m),eye(m),lambda,N,alpha0);
end;
plot(beta0)
xlabel('FIR order'), ylabel('Power of model validation test')
```



In this example we can conclude that using a FIR model of order 10 gives much higher power (smaller mean time to detection) for a given confidence level (mean time between false alarms) than using higher order FIR models.

See also Section 6.4, p. 221, in [10], or, for a full description, F. Gustafsson and B.M. Ninness. Asymptotic power and the benefit of under-modeling in change detection. In *Proceedings on the 1995 European Control Conference, Rome*, pages 1237–1242, 1995.

Limitations

Only for the parallel filter approach in `detect2` and only FIR models.

See Also

`detect2`

9.27 radarplot

Purpose `radarplot` plots a two-dimenhsional path plot with a marker for a radar at the origin.

Synopsis `radarplot(ym,yhat,jhat);`

9.28 radfilter

Purpose `radfilter` is a recursive implementation of `adfilter`.

Synopsis For initialization of state:
`[Xfilt]=radfilter([],nnn);`
For recursion:
`[thhat,lamhat,epsi,S,X,K]=radfilter(z,nnn,adg,adm,X);`

Description The only syntax change from `adfilter` is the inclusion of a state vector `X`. The state is of the form

```
 X=[d,lamhat,mtype,n,th,P,Z],
```

where `Z` is a memory of measurements used only for the WLS option.

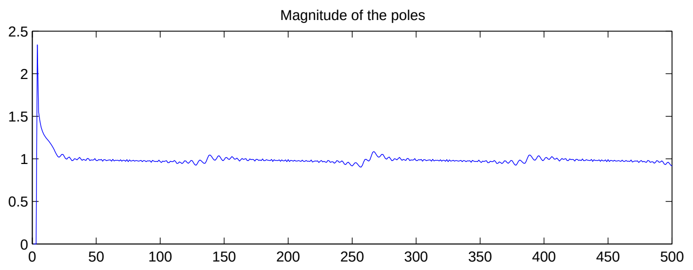
The exact definition is

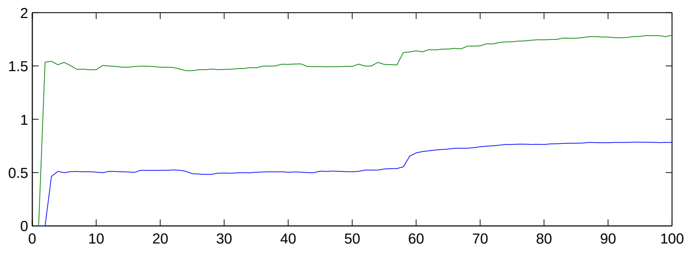
```
X=[[d lamhat mtype n; zeros(L*d+d-1,4)]...
   [xhat P;zeros(L*d,d+1)] Z];
```

Examples Simulate a first order ARX model and identify it recursively.

```
N=100;
u=randn(N,1);
e=randn(N,1);
th0=[0.5 1.5;0.8 1.8]';
d=2;
lam=0.1;
nnn=[1 1 1];
y=simchange([lam*e u],nnn,N/2,th0);
TH=[];

[Xfilt]=radfilter([],nnn);
for t=1:N;
  [thhat,lamhat,epsi,S,Xfilt]=radfilter(...
           [y(t) u(t)],nnn,0.97,'RLS',Xfilt);
  TH=[TH thhat];
end
plot(TH')
```





See Also `rdetect1` , `rdetect2` , `rglr` , `adfilter`

9.29 rdetect1

Purpose

`rdetect1` is a recursive implementation of `detect1` .

Synopsis

For initialization of state:
`[Xone]=rdetect1([],[],dm,sr,-1);`
For recursion:
`[alarm,Xone]=rdetect1(epsi(t),Xone);`

Description

The difference to `detect1` is mainly that the filter and stopping rule are separated, so `rdetect1` only implements the stopping rule. For that reason, the data input is the normalized filter residual `epsi`, such that $\mathrm{Var}(\varepsilon(t)) = \lambda$ if there is no change. This can be computed from the outputs of `radfilter` as `epsi/sqrt(S)`, where `S` is the variance of the residual. See the example in `rdetect2` . `lambda` (λ) is an unknown noise scaling, which is by default estimated.

The *distance measure* `dm` and *stopping rule* `sr` are the same as for `detect1`

The state is defined as

```
X=[gt,gtm,n,lambda,dm,nu,gamma,q,h,hm,noiseest]
```

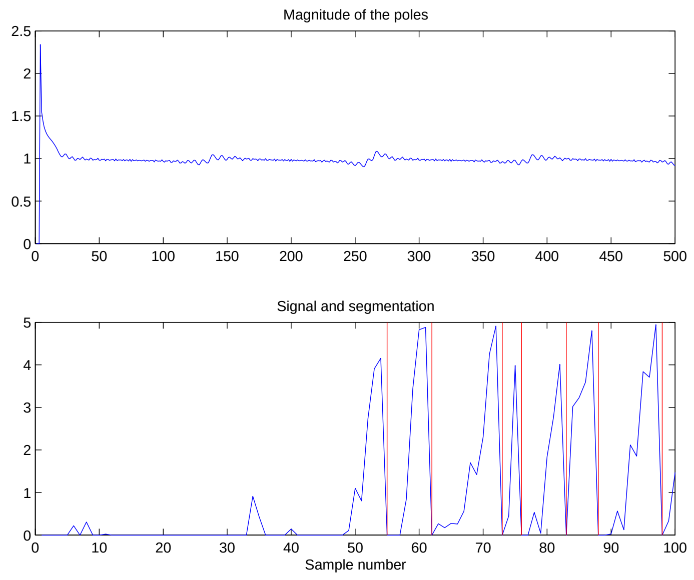
Examples

Simulate a change in the mean, and apply the CUSUM test as a whiteness detector.

```
lambda=0.25;
y=[zeros(50,1);ones(50,1)] + sqrt(lambda)*randn(100,1);
dm=2;
h=5;
nu=.5;
sr=[1 h nu];
GT1=[];
alarmtimes=[];

[Xone]=rdetect1([],[],dm,sr,-1);
for t=1:100;
  [alarm,Xone]=rdetect1(y(t),Xone);
  GT1=[GT1;Xone(1)];
  if alarm~=0;
```

```
    alarmtimes=[alarmtimes t];
  end;
end
segplot(GT1,alarmtimes)
```



After the change, we get several alarms.

See Also

detect1 , radfilter , rdetect2 , rglr

9.30 rdetect2

Purpose `rdetect2` is a recursive implementation of `detect2` .

Synopsis For initialization of state:

```
[X]=rdetect2([],[],[],[],[],dm,sr);
```

For recursion:

```
[alarm,X]=rdetect2(epsi0/sqrt(S0),epsi1/sqrt(S1),lam0,lam1,X);
```

Description The difference to `detect2` is mainly that the filter and stopping rule are separated, so `rdetect2` only implements the stopping rule. For that reason, the data input is the normalized filter residuals `epsi` from two adaptive filters, such that $\mathrm{Var}(\varepsilon(t)) = \lambda$ if there is no change. Each residual can be computed from the outputs of `radfilter` as `epsi/sqrt(S)`, where `S` is the variance of the residual. `lambda` (λ) is an unknown noise scaling, which is by default estimated.

The *distance measure* `dm` and *stopping rule* `sr` are the same as for `detect2` .

The state is defined as

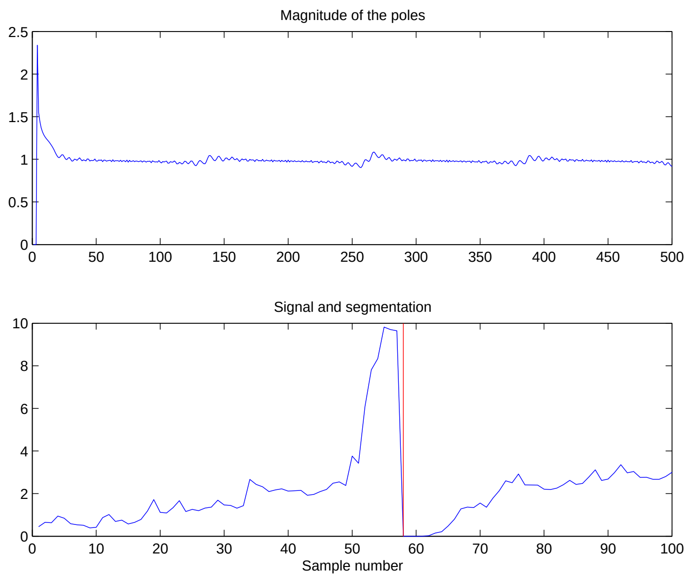
```
X=[gt,gtm,dm,nu,gamma,q,h,hm]
```

Examples Simulate a change in the mean, run two parallel RLS filters with different forgetting factors and apply the two-sided CUSUM test to the divergence measure.

```
lambda=0.25;
y=[zeros(50,1);ones(50,1)] + sqrt(lambda)*randn(100,1);
nnn=-1;
GT2=[];
alarmtimes=[];

[X0]=radfilter([],nnn,1,'RLS');
[X1]=radfilter([],nnn,0.8,'RLS');
dm=2;
h=10;
nu=1;
sr=[1 h nu];
[Xdet]=rdetect2([],[],[],[],[],dm,sr);
```

```
for t=1:100;
  [thhat0,lam0,epsi0,S0,X0]=radfilter([y(t)],nnn,1,'RLS',X0);
  [thhat1,lam1,epsi1,S1,X1]=radfilter([y(t)],nnn,0.8,'RLS',X1);
  [alarm,Xdet]=rdetect2(epsi0/sqrt(S0),epsi1/sqrt(S1),...
                        lam0,lam1,Xdet);
  GT2=[GT2;Xdet(1)];
  if alarm~=0;
  % Let the slow filter overtake the fast filter state
    X0=X1;
    alarmtimes=[alarmtimes t];
  end;
end
segplot(GT2,alarmtimes)
```



See Also

detect2 , rdetect1 , rglr , radfilter

9.31 rglr

Purpose

`rglr` is a recursive implementation of `glr` .

Synopsis

For initialization of state:
`[Xglr]=rglr([],[],[],nnn,[],M,h);`
For recursion:
`[alarm,Xglr,nuhat,jhat,lrmax,P]=rglr(epsi,S,K,nnn,Xglr);`

Description

In `rglr`, the Kalman filter and GLR test are implemented separately.

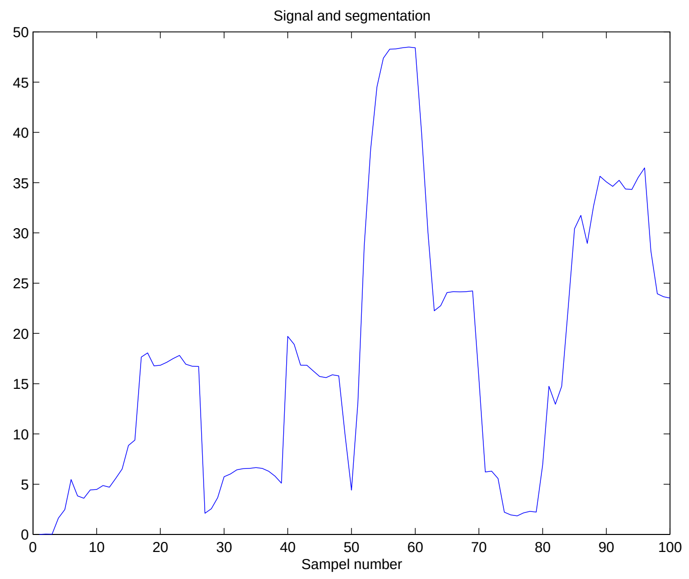
Examples

Simulate a state space model, and run the Kalman filter and GLR test recursively in series.

```
N=100;
d=2;
u=randn(N,1);
A=[1 1;0 1]; B=[0.5;1]; C=[1 0]; D=0;
Q=0.01*eye(2); R=0.1; P0=1*eye(2);
nnn=ss2nnn(A,B,C,D,Q,R,P0);
th0=[1;...
     2];
y=simchange([u],nnn,N/2,th0);

[Xfilt]=radfilter([NaN NaN],nnn);
h=72;
M=10;
[Xglr]=rglr([],[],[],nnn,[],M,h);
GT1=[];
alarmtimes=[];

for t=1:N;
  [thhat,lamhat,epsi,S,Xfilt,K]=radfilter(...
                             [y(t) u(t)],nnn,1,'RLS',Xfilt);
  [alarm,Xglr,nuhat,jhat,lrmax,P]=rglr(epsi,S,K,nnn,Xglr);
  GT1=[GT1;lrmax];
  if alarm~=0;
    alarmtimes=[alarmtimes t];
    % Increase P in the Kalman filter
    Xfilt(:,6:5+d)=100*Xfilt(:,6:5+d);
  end;
end;
segplot([GT1],alarmtimes)
```



See Also radfilter , glr , mlr , rdetect1 , rdetect2

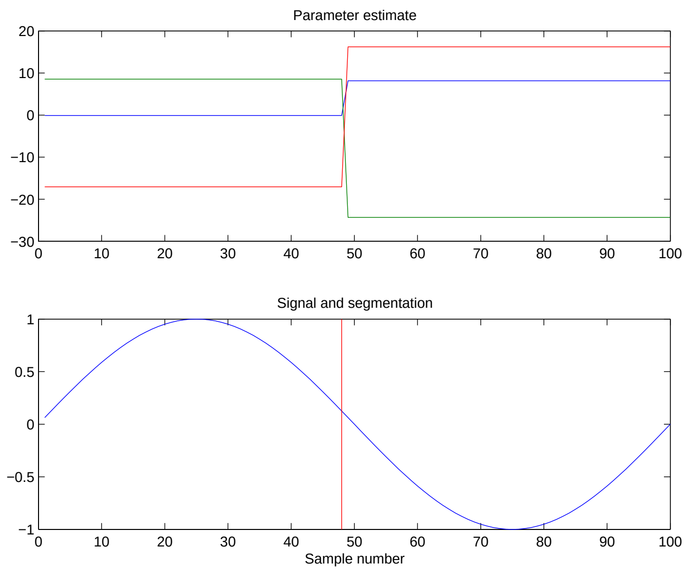
9.32 segm2par

Purpose `segm2par` converts a vector of change times and $d \times n$ matrix of parameters vectors in each segment to a full $d \times N$ parameter matrix.

Synopsis `thseg=segm2par(TH,jumptimes,N);`

Examples

```
jumptimes=[40 70];
TH=[1.5 0.8 2 0.5;...
    1.5 0.8 2 1;...
    1.5 0.6 2 1]';
nnn=[2 2 1];
th=segm2par(TH,jumptimes,100);
plot(th')
axis([0 100 0 2.5])
```



See Also par2segm

9.33 segplot

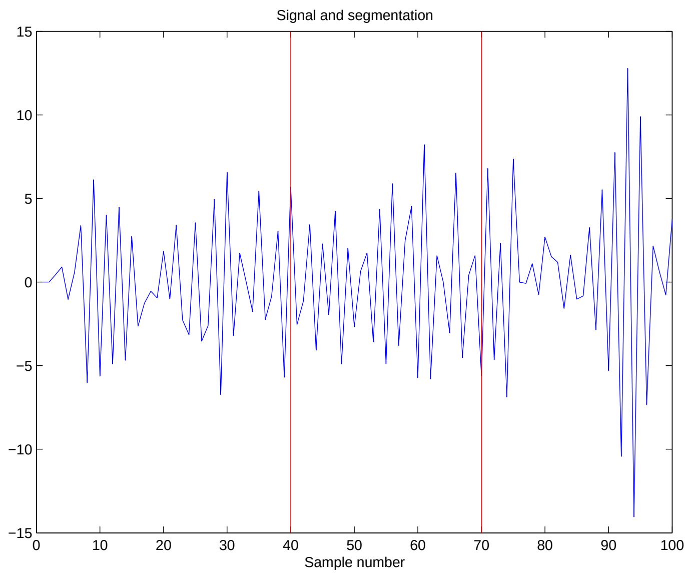
Purpose `segplot` plots the signal and marks the segments.

Synopsis

```
segplot(z,jumphat);
```

Examples

```
u=randn(100,1);
e=randn(100,1);
jumptimes=[40 70];
TH=[1.5 0.8 2 0.5;...
    1.5 0.8 2 1;...
    1.5 0.6 2 1]';
nnn=[2 2 1];
y=simchange([0.1*e u],nnn,jumptimes,TH);
segplot(y,jumptimes)
```



See Also `thplot`

9.34 simadfilter

Purpose `simadfilter` simulates a time-varying parametric model with random walk parameters.

Synopsis

```
[y,th]=simadfilter(z,nnn,adg,adm);
```

Description The parameter time-variations are modelled as random walk:

$$\begin{aligned}\theta(t+1) &= \theta(t)+v(t)\\ \mathrm{Cov}(v(t)) &= Q(t)\end{aligned}$$

The function simulates a trajectory according to this random walk model, and computes the output corresponding to the model specified in `nnn`.

The Kalman filter is the optimal estimator for random walk parameters. Note that some interesting special cases exist, which correspond to models where RLS and NLMS are the optimal filters.

The parameters are:

`z` data matrix equal to `[e]`, `[e u]` or `[e Phi]`

`nnn` Model structure.

`adg` Adaptation gain, see `adfilter` .

`adm` Adaptation method, see `adfilter` .

See Also `simchange` , `simresid`

9.35 simchange

Purpose `simchange` simulates a linear system with abrupt changes.

Synopsis

```
[y,th0]=simchange(z,nnn,jumptimes,TH,Lambda,ipm,ippar);
```

Description The function offers a flexible way to simulate additive changes in all supported model structures. The default change is abrupt, but smooth changes (incipative), can be obtained with the two optional parameters `ipm, ippar`.

`z` Data matrix equal to `[e]`, `[e u]`, `[e Phi]` or `[e u w]`

`nnn` Model structure.

`jumptimes` Times for abrupt changes (denoted n below).

`TH` Additive changes. For regression models, `TH` contains the parameter vectors in each segment, so the number of columns in `TH` equals the number of elements in `jumptimes` plus one ($n + 1$). For state space models `TH` has as many columns as there are jumps (n).

`Lambda` Scaling of measurement noise variance R.

`th0` True parameters, which might be interpolated values of `TH`.

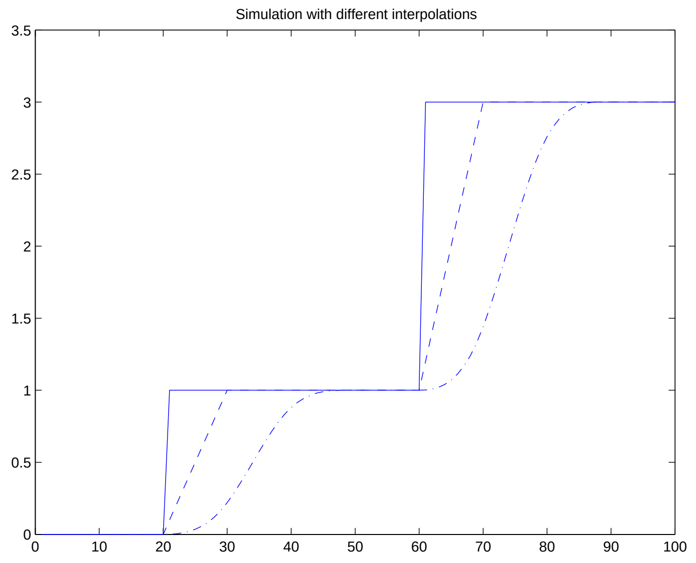
`ipm` Interpolation method for the impulsive changes in TH.

- `ipm='no'` No interpolation (default for parametric models).
- `ipm='lc'` Interpolated change from `jumptime` to `jumptime+m*L`.
- `ipm='ic'` Integrated change (default for state space models).

`ippar=[L m]` Interpolation window size `L` and degree `m` used for interpolated changes (see above). `m=1` gives linear interpolation.

Examples

```
e=randn(100,1);
jumptimes=[20 60];
TH=[0 1 3];
nnn=[-1];
[y,th0]=simchange(e,nnn,jumptimes,TH);
[y,th1]=simchange(e,nnn,jumptimes,TH,1,'lc',[10 1]);
[y,th2]=simchange(e,nnn,jumptimes,TH,1,'lc',[10 3]);
plot(th0),      hold on
plot(th1,'--')
plot(th2,'-.'), hold off
title('Simulation with different interpolations')
```



Interpolation is implemented by convolution with a pulse of width L, so higher order interpolation implies longer response times (mL).

See Also

`simresid` , `simadfilter`

9.36 simchannel

Purpose `simchannel` simulates a digital communication channel.

Synopsis `[y,u,th]=simchannel(alphabet,nn,N,th);`

Description The function generates a random input signal with elements in a finite alphabet and simulates a MIMO (`ny` outputs, `nu` inputs) FIR (order `nb(ny,nu)`) channel.

`alphabet` Finite input symbol alphabet

`nn` Orders of the MIMO FIR channel. `ny x nu` matrix with FIR order `nn(i,j)` for input `i` to output `j`.

`N` Number of simulated data points.

`th` Channel parameters

`th` Matrix of size (`N x sum(sum(nn))`) of channel FIR parameters for a time-varying or a `sum(sum(nn))` vector for a time invariant channel. The parameter vector starts with the FIR from input 1 to output 1, then proceeds with the other inputs to output 1, and then goes on with the second output and so on. If `th=[];` then a fading channel is simulated, which is complex if the alphabet is complex, otherwise real-valued.

`y` Channel output

`u` Channel input

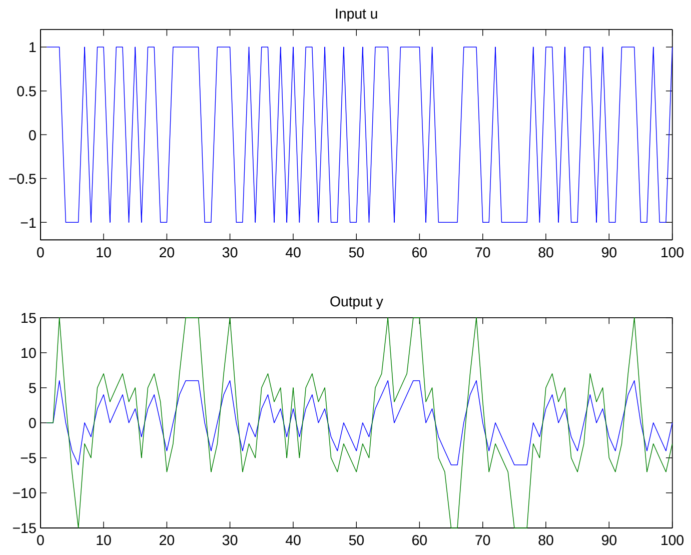
`th` Channel parameters, which is a `N x sum(sum(nn))` matrix

The time-varying parameters are simulated according to a Rayleigh fading model. The alphabet can be taken as $[-11]$ for a binary channel, $[1-1i-i]|$ for *PM4* modulation. *QAM16* may be generated by the script

```
d1=[-3 -1 1 3];
d2=ones(1,4);
a=d1'*d2+d2'*d1*i;
a(:);
```

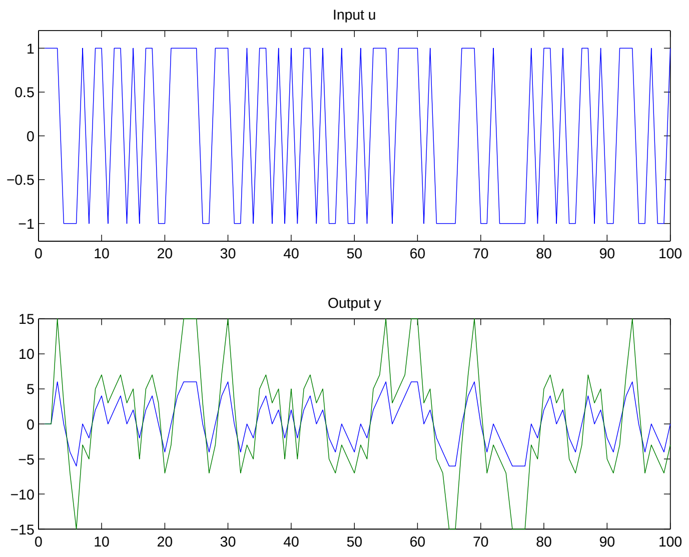
Examples Simulate a second order SISO time-varying channel, with input alphabet $[-1,1]$:

```
N=1000;
nb=2;
alphabet=[-1 1];
[y,u,b]=simchannel(alphabet,nb,N,[]);
```



Change to a time constant SIMO system with FIR channels $3 + 2q^{-1} + q^{-2}$ and $6 + 5q^{-1} + 4q^{-2}$, respectively.

```
b=[3 2 1   6 5 4];
nb=[3;3];
[y,u,b]=simchannel(alphabet,nb,N,b);
subplot(211)
plot(u)
subplot(212)
plot(y)
```



Simulate a time-varying MIMO channel:

```
nb=2; ny=2; nu=2;
[y,u,th]=simchannel(alphabet,nb*ones(ny,nu),N,[]);
```

The MIMO parameters are defined below for the channel $G_{11} = 1 + 0.9q^{-1}$, $G_{12} = 0.8 + 0.7q^{-1}$, $G_{21} = 0.6 + 0.5q^{-1}$ and $G_{22} = 0.4 + 0.3q^{-1}$.

```
th=[1 0.9   0.8 0.7    0.6 0.5   0.4 0.3];
nb=2; ny=2; nu=2;
[y,u]=simchannel(alphabet,nb*ones(ny,nu),N,th);
```

See Also

9.37 simresid

Purpose `simresid` computes the residuals from a given parameter matrix.

Synopsis

```
[epsi,epsif]=simresid(z,nnn,th);
```

Description The function can be used for validation purposes; the smaller residuals the better model and detection performance generally. Formally, `simresid` is the inverse operation of `simadfilter` and `simchange`.

`z` Output-input data.

`nnn` Model structure.

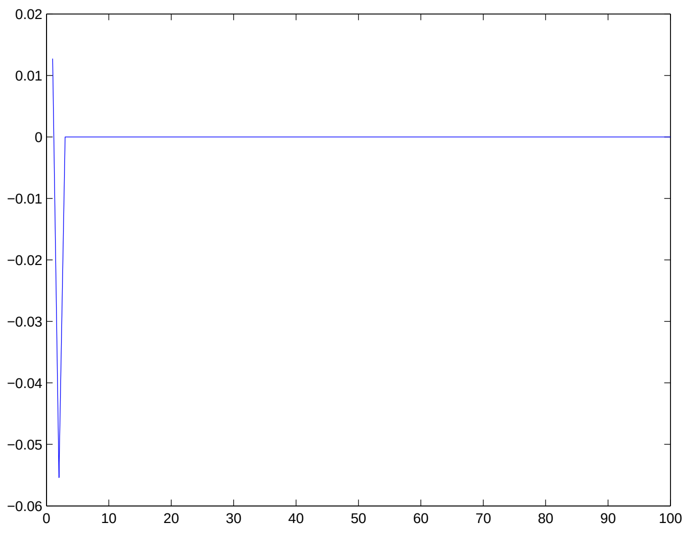
`th` Parameters or states, obtained from an adaptive filter or change detector. Both recursive and smoothed (piecewise constant parameters) can be used of course.

`epsi` Predictor residuals $y_t = \varphi_t^T \theta_{t-1}$ or $y_t = Cx_{t-1}$.

`epsi` Filter residuals $y_t = \varphi_t^T \theta_t$ or $y_t = Cx_t$.

Examples Simulate an ARX model and recover the noise sequence by inverse filtering.

```
randn('seed',0);
u=randn(100,1);
e=0.1*randn(100,1);
nnn=[2 2 1];
jumptimes=[40 70];
TH=[1.5 .8 2 0.5;...
    1.5 .8 2 1;...
    1.5 .6 2 1]';
y=simchange([e u],nnn,jumptimes,TH);
th=segm2par(TH,jumptimes,100);
[epsi,epsif]=simresid([y u],nnn,th);
plot([epsif-e])
```



Except for a short transient depending on unknown initial conditions, the residual is identical to the noise sequence.

See Also `simchange` , `simadfilter`

9.38 th2poles

Purpose `th2poles` plots the absolute value and argument of the poles of the recursively estimated AR parameters in `th`.

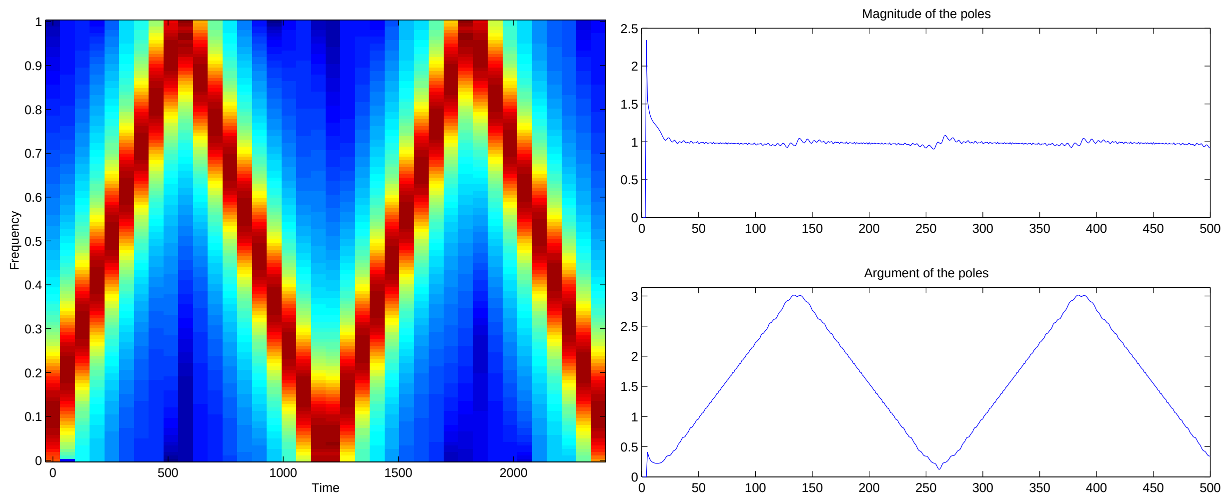
Synopsis `th2poles(th);`

Description This function is useful for time-varying spectral analysis using AR-models. The time-varying resonance frequencies and their relative strengths are directly readable from the plot.

Limitations Works only for AR-models.

Examples Generate a chirp-signal with linearly increasing frequency. Its frequency content can be illustrated non-parametrically by FFT techniques using `specgram` in SPTB, or by an adaptively estimated AR model illustrated by `th2tfd`.

```
cdfigure(1);
N=5000;
y=sin(2*pi*(1:N)'.^2./N);
specgram(y)
cdfigure(2);
N=500;
y=sin(2*pi*(1:N)'.^2./N);
[th,lamhat]=adfilter(y,2,20);
th2poles(th);
```



Look how well the phase of the poles reflects the frequency content.

See Also `th2tfd`

9.39 th2tfd

Purpose `th2tfd` plots the time-frequency description of an adaptively estimated AR model.

Synopsis `[z,w,T]=th2tfd(threc,lamhat,y,wbin,tbin,plottype);`

Description The function computes the transfer function of the recursively estimated AR model in `threc` and plots the result.

`lamhat`, if provided, is the local energy in the AR model.

`y`, if provided, is the original signal used only for plotting.

`wbin` is the number of frequency bins. Default is 32.

`tbin` is the number of time bins. Default is 32.

Both `wbin` and `tbin` could be vectors of actual bin values. `w` and `T` are the frequency and time bins. With no output arguments, one (or more if `plottype` is a vector) of the following is obtained:

`plottype=1 Imagesc` plot (default).

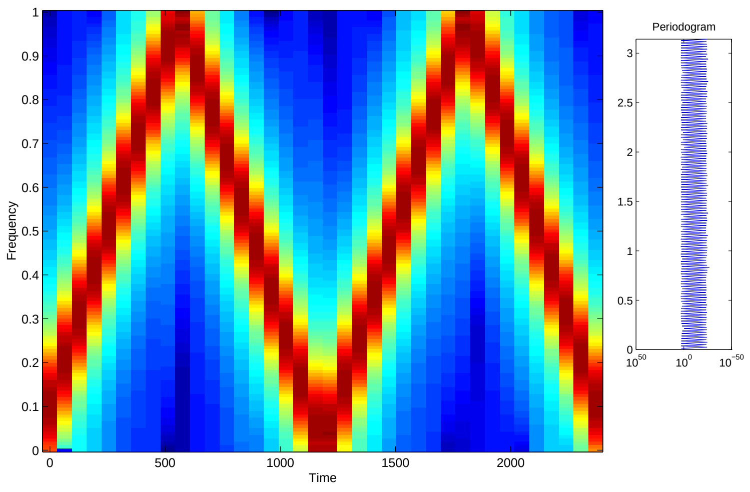
`plottype=2 Waterfall` plot.

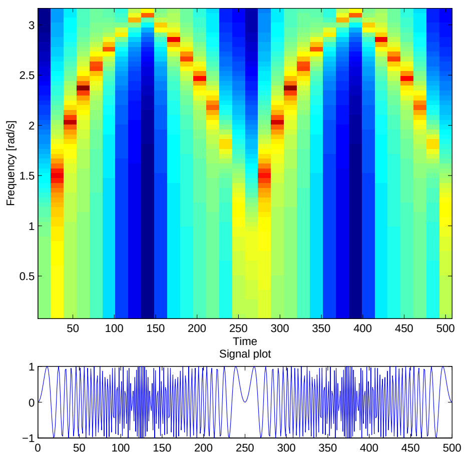
`plottype=3 Contour` plot.

Note: sampling interval 1 assumed.

Examples Generate a chirp-signal with linearly increasing frequency. Its frequency content can be illustrated non-parametrically by FFT techniques using `specgram` in SPTB, or by an adaptively estimated AR model illustrated by `th2tfd`.

```
cdfigure(1);
N=5000;
y=sin(2*pi*(1:N)'.^2./N);
specgram(y)
cdfigure(2);
N=500;
y=sin(2*pi*(1:N)'.^2./N);
[th,lamhat]=adfilter(y,2,20);
th2tfd(th,lamhat,y,64,[],1);
```





See Also `th2poles`

9.40 thplot

Purpose `thplot` plots the estimated parameters and true parameters together for easy comparison.

Synopsis

```
thplot(th,TH,jumptimes,threc);
```

Description

`th` Parameters plotted with solid line (typically smoothed estimate)

`TH` Parameters plotted with dotted line (typically simulated true values)

`threc` Parameters plotted with dashed line (typically recursive values)

`jumptimes` Change times illustrated with vertical bars.

Note: `th` and `threc` are given for each time instant, but `TH` for each segment only.

Examples See `detectM`

See Also `segplot`

9.41 u2ber

Purpose `u2ber` computes bit error rate for the output of an equalizer.

Synopsis

```
[ber,uhatout,delay,m]=u2ber(u,uhat);
```

Description The impulse response of the combined channel and equalizer, assuming FIR models, is

$$h_t = (b * c)_t$$

where $*$ denotes convolution. The best equalizer one can get is $h_t \approx= m\delta_{t-k}$, where k is an unknown time-delay, and m, with $|m| = 1$, an unknown modulus. The modulus and delay do not matter for the performance and can in applications be ignored. However, for evaluation using the bit error rate, these have to be determined.

This algorithm is intelligent and looks for an appropriate time delay and modulus of the combined channel and equalizer, by a correlation technique.

`u` True input (from finite alphabet).

`uhat` Estimated input.

`uhatout` The corrected input estimate with respect to time delay and modulus of the combined channel and equalizer.

`ber` Bit error rate.

`delay` Estimated time delay.

`m` Estimated modulus.

Examples See `blindeq` .

Algorithm The algorithm implemented in `u2ber` estimates the covariance function (using FFT for maximum speed)

$$R(k) = \mathrm{E}(u_t \hat{u}_{t-k}).$$

The time delay is then estimated as

$$\hat{k} = \arg\max_k |\hat{R}(k)|,$$

and the maximum of $\hat{R}(k)$ can be taken as the corresponding estimate of the modulus.

See Also `blindeq` , `openeye`

9.42 viterbi

Purpose `viterbi` equalizes distortion from a channel using an estimate of it.

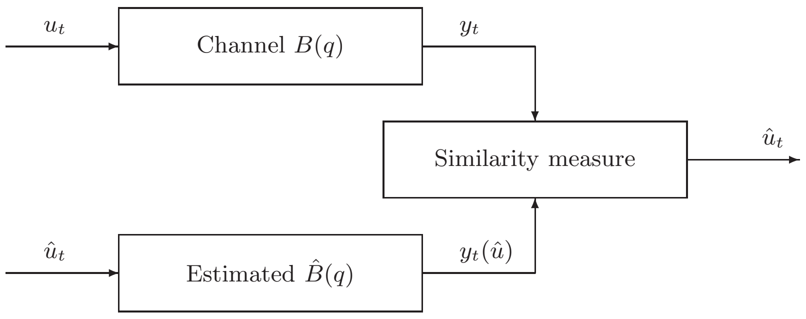
Synopsis

```
[uhat]=viterbi(y,th,alphabet,R)
```

or

```
[uhat,thhat]=viterbi(y,nb,alphabet,R,utrain)
```

Description The Viterbi algorithm enumerates all possible combinations of input signals over the time horizon of the FIR channel estimate `th`. The input sequence that gives the smallest likelihood is taken as the estimate.



The channel estimate may here come from a training sequence. This is also included in the code, so it is possible to specify the part of the input sequence that is known. In this case, also the channel estimate is provided in the out arguments.

The parameters are:

`y` Output from channel.

`utrain` If provided, utrain is the training sequence, `th=nb` is the FIR order and `thhat` is the FIR estimate.

`uhat` Estimated input.

`th` Estimated FIR parameters in the channel.

`alphabet` Finite input alphabet. Can be any set of complex/real numbers.

`R` Noise variance.

Limitations

1. Only FIR equalizers.
2. Only scalar channels.

Examples

```
N=100;
b=[1 1 1];         % True channel
u=sign(randn(N,1));
y=filter(b,1,u)+sqrt(0.1)*randn(N,1);
bhat=[1 0.7 1.3]; %  Channel ''estimate''
uhat=viterbi(y,bhat,[-1 1],0.1);
ber=u2ber(u,uhat)
ber =
    0.0400
% BER when using a short training sequence
[uhat,thhat]=viterbi(y,3,[-1 1],0.1,u(1:5));
thhat
ans =
    0.9589    0.9589    0.9679
ber=u2ber(u,uhat)
ber =
     0
```

References

G.D. Forney. The Viterbi algorithm. *Proceedings of the IEEE*, 61:268–278, 1973

See Also

`blindeq` , `blindeqM` , `u2ber`

9.43 z2phi

Purpose `z2phi` extracts the regression vector and data `z`.

Synopsis

```
[phi,y,u,w]=z2phi(z,t,nnn);
```
or
```
[tstart,d]=z2phi(z,0,nnn);
```

Description This function is called by all filters when a linear regression model is specified.

If $t = 0$, the first possible starting time is delivered together with the number of parameters, otherwise the corresponding regression vector for the linear regression structure in `nnn` is computed.

10 Signal models and notation

General

The signals and notation below are common for the book and toolbox.

Name	Variable	Dimension
N	Number of data (off-line)	Scalar
L	Number of data in sliding window	Scalar
t	Time index (in units of sample intervals)	Scalar
n_*	Dimension of the vector $*$	Scalar
y_t	Measurement	n_y, p
u_t	Input (known)	n_u
e_t	Measurement noise	n_y
θ_t	Parameter vector	n_θ, d
x_t	State	n_x
k_i	Change time	Scalar
n	Number of change times	Scalar
k^n	Vector of change times	n
s_t	Distance measure in stopping rule	Scalar
g_t	Test statistic in stopping rule	Scalar

Signal estimation models

The *change in the mean model* is

$$y_t = \theta_t + e_t,\ \mathrm{Var}(e_t) = \sigma^2, \tag{10.1}$$

and the *change in the variance model* is

$$y_t = e_t,\ \mathrm{Var}(e_t) = \sigma_t^2. \tag{10.2}$$

In change detection, one or both of θ_t and σ_t^2 are assumed piecewise constant, with change times $k_1, k_2, \ldots, k_n = k^n$. Sometimes it is convenient to introduce the start and end of a data set as $k_0 = 0$ and $k_n = N$, leaving $n - 1$ degrees of freedom for the change times.

Parametric models

Name	Variable	Dimension	Covariance matrix
e_t	Measurement noise	n_e	R_t or σ^2
$\hat{\theta}_t$	Parameter estimate	n_θ, d	$P_t = P_{t\|t}$
ε_t	Residual $y_t - \varphi_t^T \hat{\theta}_{t-1}$	n_y	S_t

The generic signal model has the form of a *linear regression*:

$$y_t = \varphi_t^T \theta_t + e_t. \tag{10.3}$$

In adaptive filtering, θ_t is varying for all time indexes, while in change detection it is assumed piecewise constant, or at least slowly varying in between the change times.

Important special cases are the *FIR* model

$$\begin{aligned} y_t &= b_t^1 u_{t-n_k} + b_t^2 u_{t-n_k-1} + \dots + b_t^{n_b} u_{t-n_k-n_b+1} + e_t && (10.4a)\\ &= \varphi_t^T \theta_t + e_t && (10.4b)\\ \varphi_t^T &= (u_{t-n_k},\ u_{t-n_k-1}, \dots, u_{t-n_k-n_b+1}) && (10.4c)\\ \theta_t^T &= (b_t^1,\ b_t^2, \dots, b_t^{n_b}), && (10.4d) \end{aligned}$$

AR model

$$\begin{aligned} y_t &= -a_t^1 y_{t-1} - a_t^2 y_{t-2} - \dots - a_t^{n_a} y_{t-n} + e_t && (10.5a)\\ &= \varphi_t^T \theta_t + e_t && (10.5b)\\ \varphi_t^T &= (-y_{t-1},\ -y_{t-2}, \dots, -y_{t-n}) && (10.5c)\\ \theta_t^T &= (a_t^1,\ a_t^2, \dots, a_t^{n_a}), && (10.5d) \end{aligned}$$

and *ARX* model

$$\begin{aligned} y_t &= -a_t^1 y_{t-1} - a_t^2 y_{t-2} - \dots - a_t^{n_a} y_{t-n_a} && (10.6a)\\ &\quad + b_t^1 u_{t-n_k} + b_t^2 u_{t-n_k-1} + \dots + b_t^{n_b} u_{t-n_k-n_b+1} + e_t && (10.6b)\\ &= \varphi_t^T \theta_t + e_t && (10.6c)\\ \varphi_t^T &= (-y_{t-1},\ -y_{t-2}, \dots, -y_{t-n_a},\ u_{t-n_k},\ u_{t-n_k-1}, \dots, u_{t-n_k-n_b+1}) && (10.6d)\\ \theta_t^T &= (a_t^1,\ a_t^2, \dots, a_t^{n_a},\ b_t^1,\ b_t^2, \dots, b_t^{n_b}). && (10.6e) \end{aligned}$$

A general parametric, dynamic and deterministic model for curve fitting can be written as

$$\begin{aligned} \dot{x}_t &= g_t(x_t; \theta) && (10.7a)\\ y_t &= h_t(x_t; \theta). && (10.7b) \end{aligned}$$

State space models

Name	Variable	Dim.	Cov.
x_t	State	n_x	–
v_t	State noise (unknown stoch. input)	n_v	Q_t
d_t	State disturbance (unknown det. input)	n_d	–
f_t	State fault (unknown input)	n_f	–
e_t	Measurement noise	n_y	R_t, σ^2
A, B, C, D	State space matrices	$--$	–
$\hat{x}_{t\|t}$	Filtered state estimate	n_x	$P_{t\|t}$
$\hat{x}_{t\|t-1}$	Predicted state estimate	n_x	$P_{t\|t-1}$
$\hat{y}_{t\|t}$	Filtered output estimate	n_y	–
$\hat{y}_{t\|t-1}$	Predicted output estimate	n_y	–
ε_t	Innovation or residual $y_t - \hat{y}_{t\|t-1}$	n_y	S_t

The standard form of the *state space model* for linear estimation is

$$x_{t+1} = A_t x_t + B_{u,t} u_t + B_{v,t} v_t, \quad \text{Cov}(v_t) = Q_t \tag{10.8a}$$
$$y_t = C_t x_t + e_t, \quad \text{Cov}(e_t) = R_t. \tag{10.8b}$$

Additive state and sensor changes are caught in the model

$$x_{t+1} = A_t x_t + B_{u,t} u_t + B_{v,t} v_t + \sigma_{t-k} B_\theta \nu \tag{10.9a}$$
$$y_t = C_t x_t + e_t + D_{u,t} u_t + \sigma_{t-k} D_{\theta,t} \nu, \tag{10.9b}$$

where ν is the additive fault magnitude. A completely deterministic state space model is

$$x_{t+1} = A_t x_t + B_{u,t} u_t + B_{d,t} d_t + B_{f,t} f_t \tag{10.10a}$$
$$y_t = C_t x_t + D_{u,t} u_t + D_{d,t} d_t + D_{f,t} f_t, \tag{10.10b}$$

where f_t is the additive time-varying fault *profile*. All in all, the most general linear model is

$$\begin{aligned}
x_{t+1} &= A_t x_t + B_{u,t} u_t + B_{d,t} d_t + B_{f,t} f_t + \sigma_{t-k} B_\theta \nu + B_{v,t} v_t \\
y_t &= C_t x_t + D_{u,t} u_t + D_{d,t} d_t + D_{f,t} f_t + \sigma_{t-k} D_{\theta,t} \nu + e_t \\
\text{Cov}(v_t) &= Q_t \\
\text{Cov}(e_t) &= R_t \\
\text{Cov}(x_0) &= P_0.
\end{aligned}$$

Multi-model approaches can in their most general form be written as

$$x_{t+1} = A_t(\delta) x_t + B_{u,t}(\delta) u_t + B_{v,t}(\delta) v_t \tag{10.12a}$$
$$y_t = C_t(\delta) x_t + D_{u,t}(\delta) u_t + e_t \tag{10.12b}$$
$$v_t \in \text{N}(m_{v,t}(\delta), Q_t(\delta)) \tag{10.12c}$$
$$e_t \in \text{N}(m_{e,t}(\delta), R_t(\delta)), \tag{10.12d}$$

where δ is a discrete and finite *mode* parameter.

Bibliography

[1] B.D.O. Anderson and J.B. Moore. *Optimal filtering.* Prentice Hall, Englewood Cliffs, NJ., 1979.

[2] M. Basseville and I.V. Nikiforov. *Detection of abrupt changes: theory and application.* Information and system science series. Prentice Hall, Englewood Cliffs, NJ., 1993.

[3] C. Carlemalm and F. Gustafsson. On detection and discrimination of double talk and echo path changes in a telephone channel. In A. Prochazka, J. Uhlir, P.J.W. Rayner, and N.G. Kingsbury, editors, *Signal analysis and prediction*, Applied and Numerical Harmonic Analysis. Birkhauser Boston, 1998.

[4] C.S. Van Dobben de Bruyn. *Cumulative sum tests: theory and practice.* Hafner, New York, 1968.

[5] G.D. Forney. The Viterbi algorithm. *Proceedings of the IEEE*, 61:268–278, 1973.

[6] D.N. Godard. Self-recovering equalization and carrier tracking in two-dimensional data communication systems. *IEEE Transactions on Communications*, 28:1867–1875, 1980.

[7] F. Gustafsson. The marginalized likelihood ratio test for detecting abrupt changes. *IEEE Transactions on Automatic Control*, 41(1):66–78, 1996.

[8] F. Gustafsson and B.M. Ninness. Asymptotic power and the benefit of under-modeling in change detection. In *Proceedings on the 1995 European Control Conference, Rome*, pages 1237–1242, 1995.

[9] F. Gustafsson and B. Wahlberg. Blind equalization by direct examination of the input sequences. *IEEE Transactions on Communications*, 43(7):2213–2222, 1995.

[10] Fredrik Gustafsson. *Adaptive filtering and change detection.* John Wiley & Sons, Ltd, 2000.

[11] Fredrik Gustafsson. *Adaptive filtering and change detection.* John Wiley & Sons, Ltd, 2000.

[12] T. Kailath, A.H. Sayed, and B. Hassibi. *Linear estimation.* Information and System Sciences. Prentice-Hall, Upper Saddle Riber, New Jersey, 2000.

[13] L. Ljung and T. Söderström. *Theory and practice of recursive identification.* MIT Press, Cambridge, MA, 1983.

[14] R.W. Lucky. Techniques for adaptive equalization of digital communication systems. *Bell System Technical Journal*, 45:255–286, 1966.

[15] A. Sen and M.S. Srivastava. On tests for detecting change in the mean. *Annals of Statistics*, 3:98–108, 1975.

[16] A.S. Willsky and H.L. Jones. A generalized likelihood ratio approach to the detection and estimation of jumps in linear systems. *IEEE Transactions on Automatic Control*, 21:108–112, 1976.

Index

Purpose	Syntax

Simulation	y=simadfilter (z,nnn,adm,adg)
	y=simchange (z,nnn,jumptimes,TH)
	y=simresid (z,nnn,threc)
	[y,u,th]=simchannel (alphabet,nn,N,th)

Adaptive filtering	[thhat,epsi] =adfilter (z,nnn,adg,adm) [xhat] =adkalman (z,nnn,adg,options)
Change detection	[threc,jhat,thseg]=detect1 (z,nnn,options) [threc,jhat,thseg]=detect2 (z,nnn,options) [threc,jhat,thseg]=detectM (z,nnn,options) [threc,jumpsmc,jhat]=gibbs (z,nnn,options) [jhat,jtype,thseg]=multihyp (z,nnn,options) [jhat,thseg]=cpe (z,nnn,options) [jhat]=mlr (z,nnn,options) [xhat,jhat]=glr (z,nnn,options) [r]=faultdetect (z,nnn,options)
Equalization	[uhat]=viterbi (y,th,alphabet,lam,utrain)

Blind equalization	[threc,uhat]=blindeq (y,nnn,adg,adm,th0)
	[threc,uhat]=blindeqM (y,nnn,alphabet)
	ber=u2ber (u,uhat)
	m=openeye (bch,beq)

Model conversion	thseg=par2segm (TH,jumptimes,N)
	TH=segm2par (thseg)
	phi=z2phi (z,nnn)

Plot	segplot (z,jumptimes) thplot (th1,TH,jumptimes,th2) hypplot (XdetectM) radarplot (y,xhat,x)
Help	helpdetect
Demonstration	demodetect , reference , tutorial
Book examples	book , signal
GUI	guidetect
Real data sets	airbag, altdata, ash, bach, carspeed ceram, defibrator, eeg_human, eeg_rat ekg, equake, fricest, fuel, highway labmotor, nmt450, nmt900, nose, carpath photons, planepath, salesment, sfquake sheep, speech, tpi
Models	nnnplane , nnnDCm1