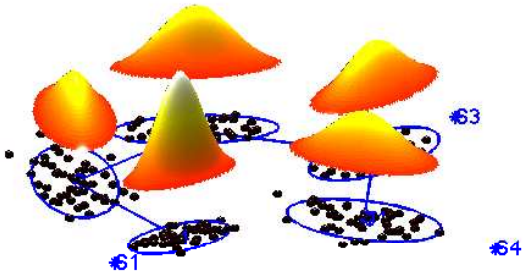
Statistical Sensor Fusion
Matlab Toolbox

Fredrik Gustafsson



Contents

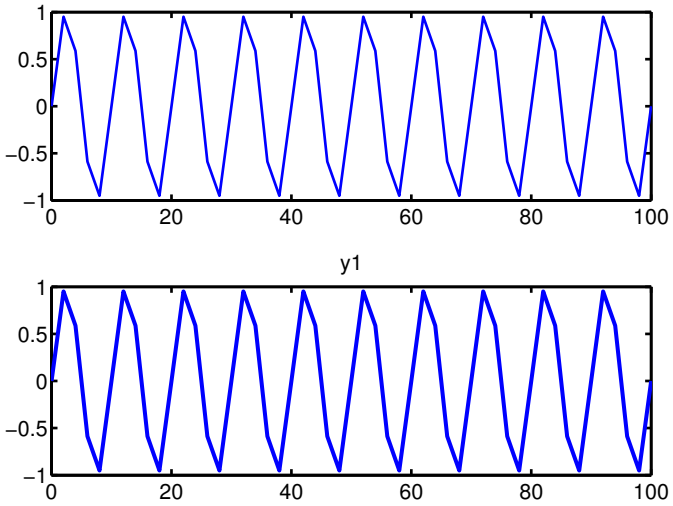
1

Introduction

Functions operating on matrices is the classical way to work with MATLAB^{TM}. *Methods operating on objects* is how *Signals and Systems Lab* is designed. The difference and advantages are best illustrated with an example.

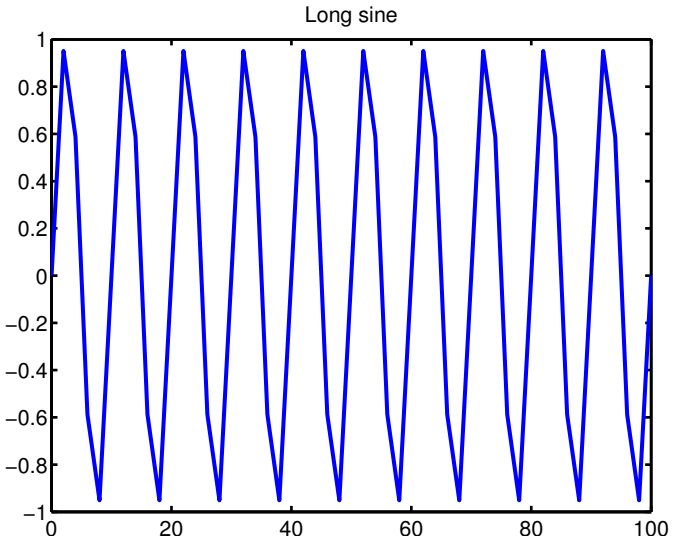
A sinusoid is generated and plotted, first in the classical way as a vector, and then after it is embedded into an object in a class called SIG.

```
f=0.1;
t1=(0:2:100)';
yvec1=sin(2*pi*f*t1);
subplot(2,1,1), plot(t1,yvec1)
subplot(2,1,2), plot(sig(yvec1,1/2))
```



The result is essentially the same. The default values for linewidth and fontsize are different in this case, and there is a default name of the signal in the title. The latter can be changed, and more properties of the signal can be defined as shown below. The advantage is that this meta information is kept in all subsequent operations, as first illustrated in the `plot` method.

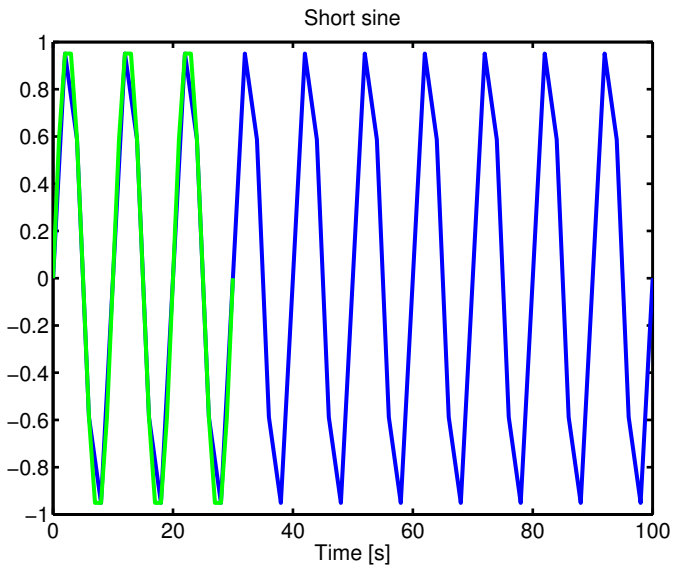
```
y1=sig(yvec1,1/2);
y1.tlabel='Time [s]'; y1.ylabel='Amplitude'; y1.name='Long sine';
plot(y1)
```



Note that `plot` is a method for the SIG class. To access its help, one must type `help sig.plot` to distinguish this plot functions from all other ones (there is one per class).

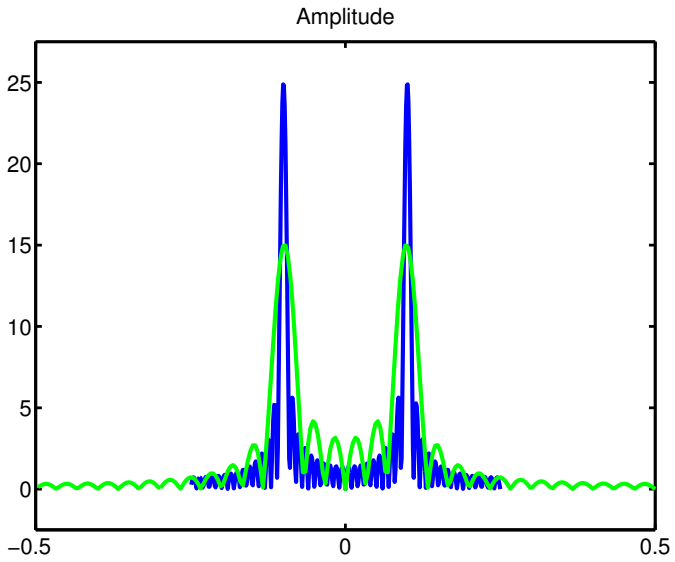
The plot function can sort out different signals automatically. This example shows two signals from the same sinusoid with different sampling intervals and different number of samples.

```
t2=(0:30)';
yvec2=sin(2*pi*f*t2);
y2=sig(yvec2,1);
y2.tlabel='Time [s]'; y2.ylabel='Amplitude'; y2.name='Short sine';
plot(y1,y2)
```



Once the signal object is defined, the user does not need to worry about its size or units anymore. Plotting the Fourier transform of these two signals illustrate one obvious strength with this approach.

```
plot(ft(y1),ft(y2))
```



Using the `fft` function directly requires some skills in setting the frequency axis and zero padding appropriately, but this is taken care of here automatically.

A signal can also be simulated from a model. The following code generates the same sine as above, but from a signal model. First, the sine model is defined.

```
s=sigmod('sin(2*pi*th(1)*t)',[1 1]);
s.pe=0.1;  s.th=0.1;
s.thlabel='Frequency'; s.name='Sine';
s
SIGMOD object: Sine
   y = sin(2*pi*th(1)*t) + N(0,0.1)
   th'  = 0.1
  Outputs: y1
  Param.: Frequency
y=simulate(s,1:40);
```

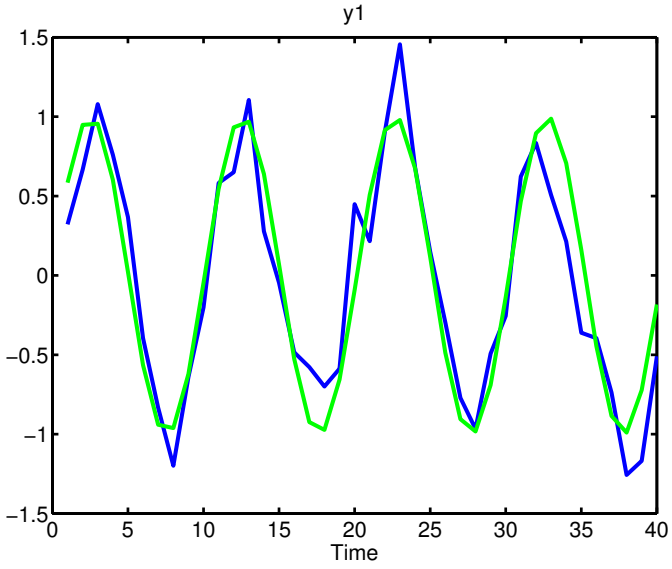
Here also a Gaussian error is added to each sample. The frequency is defined as a parameter called `th` (standardized name in *Signals and Systems Lab*). The `display` method defines what is printed out in the MATLAB™ command window, and it is used to summarize the essential information in models as intuitively as possible with only textual unformatted output.

Using the realization of the signal, the frequency can be estimated.

```
shat=estimate(s,y)
SIGMOD object: Sine (calibrated from data)
   y = sin(2*pi*th(1)*t) + N(0,0.1)
   th'  = 0.099
   std = [0.00047]
  Outputs: y1
  Param.: Frequency
```

Note that the standard deviation of the estimate is also computed (and displayed). By removing the noise, the estimated sine can be simulated and compared to the previous realization that was used for estimation.

```
shat.pe=0;
yhat=simulate(shat,1:40);
plot(y,yhat)
```



The methods `plot`, `simulate`, `display` and `estimate` are defined for essentially all classes in *Signals and Systems Lab*.

2

The SIG object

The representation of signals is fundamental for the *Signals and Systems Lab*, and this section describes how to represent and generate signals. More specifically,

- Section 2.1 provides an overview of the SIG object.
- Section 2.2 describes how to create a SIG object from a vector.
- Section 2.2.4 shows how to preprocess signals and overviews the types of real and artificial signals.
- Section 2.4 introduces the signals in the database of real signals that is included in the *Signals and Systems Lab*.
- Section 2.3 presents some of the benchmark examples in the *Signals and Systems Lab*.

2.1 Fields in the SIG Object

The constructor of the SIG object basically converts a vector signal to an object, where you can provide additional information. The main advantages of using a signal object rather than just a vector are:

- Defining stochastic signals from PDFCLASS objects is highly simplified using calls as `yn=y+ndist(0,1);`. Monte Carlo simulations are here generated as a background process.

- You can apply standard operations such as `+, -, .*`, and `./` to a SIG object just as you can do to a vector signal, where these operations are also applied to the Monte Carlo simulations.
- All plot functions accept multiple signals, which do not need to have the same time vector. The plot functions can visualize the Monte Carlo data as confidence bounds or scatter plots.
- The plot functions use the further information that you input to the SIG object to get correct time axis in plots and frequency axis in Fourier-transform plots. Further, you can obtain appropriate plot titles and legends automatically.

The basic use of the constructor is `y=sig(yvec,fs)` for discrete-time signals and `y=sig(yvec,tvec)` for continuous-time signals. The obtained SIG object can be seen as a structure with the following field names:

- `y` is the signal itself.
- `fs` is the sampling frequency for discrete-time signals.
- `t` contains the sampling times. Continuous-time signals are represented with `y(tk)` (uniformly or nonuniformly sampled), in which case the sampling frequency is set to `fs = NaN`.
- `u` is the input signal, if applicable.
- `x` is the state vector for simulated data.
- `name` is a one-line identifier (string) used for plot titles and various display information.
- `desc` can contain a more detailed description of the signal.
- `marker` contains optional user-specified markers indicating points of interest in the signal. For instance, the markers can be change points in the signal dynamics or known faults in systems.
- `yMC` and `xMC` contains Monte Carlo simulations arranged as matrices where the first dimension is the Monte Carlo index.
- `ylabel`, `xlabel`, `ulabel`, `tlabel`, and `markerlabel` contain labels for plot axes and legends.

The data fields `y, t, u, x, yMC`, and `xMC` are protected and cannot be changed arbitrarily. Checks are done in order to preserve the SIG object's dimensions. The operators `+` (`plus`), `-` (`minus`), `*` (`times`), and `/` (`rdivide`) are overloaded, which means that you can change the signal values linearly and add an offset to the time scale. All other fields are open for both reading and writing.

Table 2.1: SIG *constructor*

`sig(y,fs)`	Uniformly sampled time series $y[k] = y(k/f_s)$.
`sig(y,t)`	Nonuniformly sampled time series $y(t)$, used to represent continuous time signals.
`sig(y,t,u)`	IO system with input
`sig(y,t,u,x)`	State vector from state-space system
`sig(y,fs,u,x,yMC,xMC)`	MC data arranged in an array

2.2 Creating SIG Objects

The SIG constructor accepts inputs as summarized in Table 2.1.

The fields `y, t, fs, u, x, yMC`, and `xMC` are protected. You can change any of these directly, and the software does a basic format check. Using methods detailed in the next table, you can extract subsignals using a matrix-like syntax. For instance, `z(1:10)` picks out the first 10 samples, and `z(:,1,2)` gets the first output response to the second input. You can append signals taken over the same time span to a MIMO signal object using append and concatenate two signals in time or spatially as summarized in Table 2.2. The overloaded operators are summarized in Table 2.3

2.2.1 Defining Discrete-Time Signals

Scalar discrete-time signals are defined by a vector and the sampling frequency:

```
N=100;
fs=10;
t=(0:N-1)'/fs;
yvec=sin(pi*t);
y=sig(yvec,fs);
y.name='Sin signal';
stem(y)
```

Table 2.2: SIG *methods*

`arrayread`	`z=z(t,i,j)`	Pick out subsignals from SIG systems where `t` is the time, and `i` and `j` are output and input indices. `z(t1:t2)` picks out a time interval and is equivalent to `z(t1:t2,:,:)`.
`horzcat`	`z=horzcat(z1,z2) or z=[z1 z2]`	Concatenate two SIG objects to larger output dimension. The time vectors must be equal.
`vertcat`	`z=vertcat(z1,z2) or z=[z1;z2]`	Concatenate two SIG objects in time. The number of inputs and outputs must be the same.
`append`	`z=append(z1,z2)`	Concatenate two SIG objects to MIMO signals

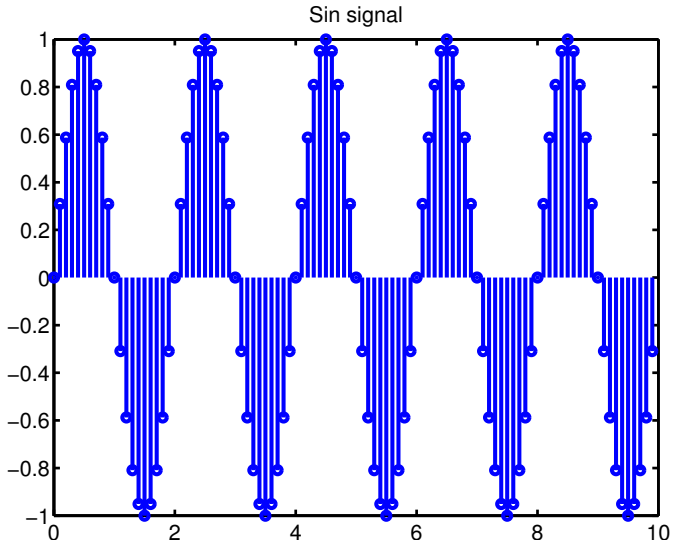
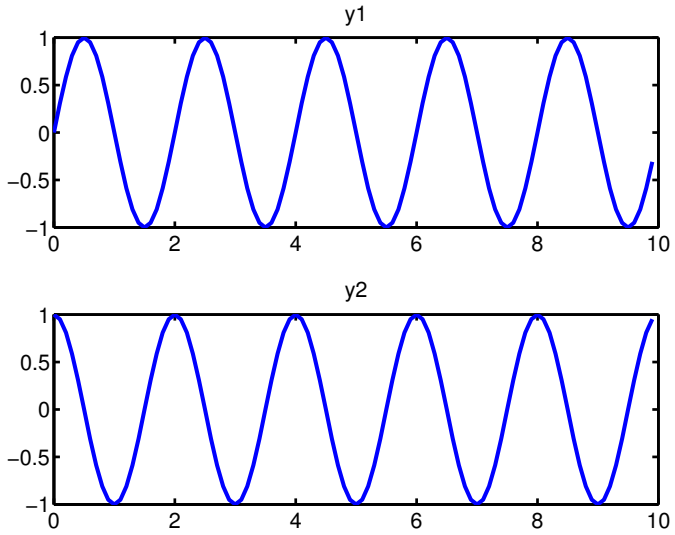


Table 2.3: SIG *overloaded operators*

OPERATOR	EXAMPLE	DESCRIPTION
plus	y=sin(t)+1+expdist(1)	Adds a constant, a vector, another signal, or noise from a PDFCLASS object
minus	y=sin(t)-1-expdist(1)	Subtracts a constant, a vector, another signal, or noise from a PDFCLASS object
times	y=udist(0.9,1.1)*sin(t)	Multiplies a constant, a vector, another signal, or noise from a PDFCLASS object
rdivide	y=sin(t)/2	Divides a signal with a constant, a vector, another signal, or noise from a PDFCLASS object. `divide` and `mrdivide` are also mapped to `rdivide` for convenience.
mean,E	y=E(Y)	Returns the mean of the Monte Carlo data
std	sigma=std(Y)	Returns the standard deviation of the Monte Carlo data
var	sigma2=var(Y)	Returns the variance of the Monte Carlo data
rand	y=rand(Y,10)	Returns one random SIG object or a cell array of random SIG objects
fix	y=fix(Y)	Removes the Monte Carlo simulations from the object

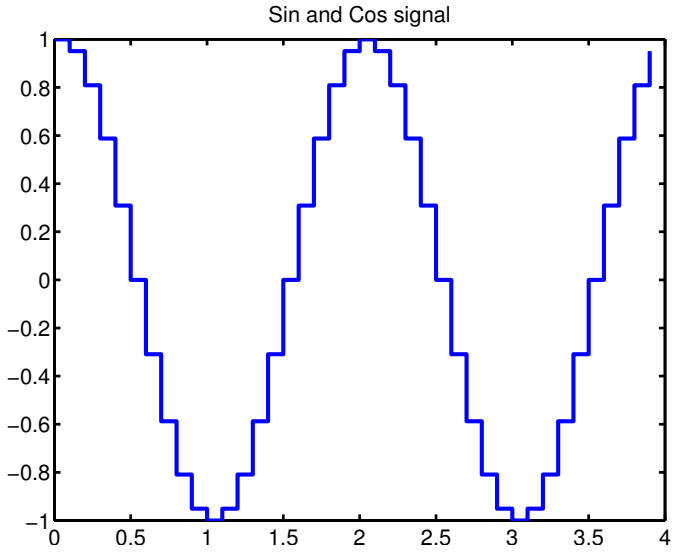
You define a multivariate signal in a similar way using a matrix where time is the first dimension:

```
yvec=[sin(pi*t) cos(pi*t)];
y=sig(yvec,fs);
y.name='Sin and Cos signal';
plot(y)
```



The following zooming-in call of the second signal component illustrates the use of indexing:

```
staircase(y(1:40,2))
```



2.2.2 Defining Continuous-Time Signals

Continuous-time signals are represented by nonuniform time points and the corresponding signal values with the following two conventions:

- Steps and other discontinuities are represented by two identical time stamps with different signal values. For instance,

```
t=[0 1 1 2]';
y=[0 0 1 1]',
z=sig(y,t);}
```

  defines a unit step, where the output `y` changes from 0 to 1 at `t = 1`.

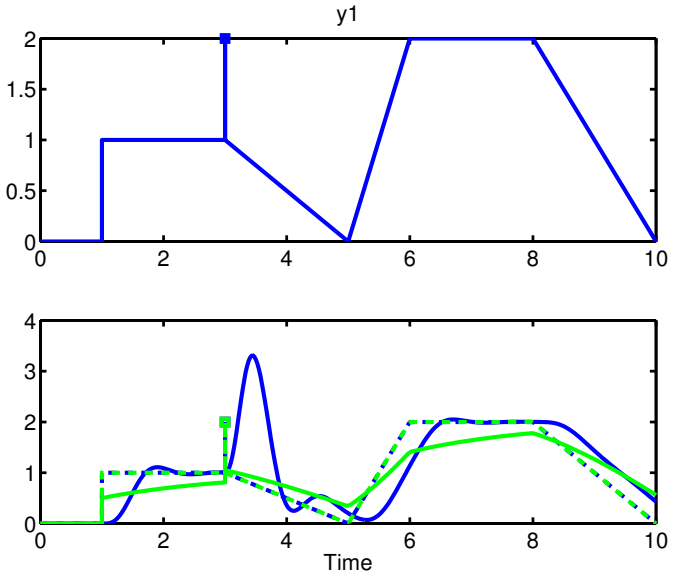
- Impulses are represented by three identical time stamps where the middle signal value represents the area of the impulse. For instance,

```
t=[0 1 1 1 2]';
y=[0 0 1 0]',
z=sig(y,t);
```

  defines a unit impulse at `t = 1`.

These conventions influence how the plots visualize continuous-time signals and also how a simulation is done. The following example illustrates some of the possibilities:

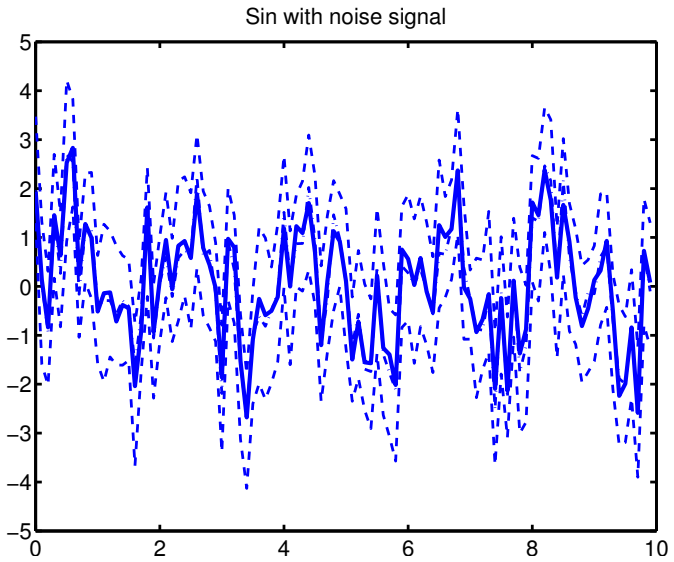
```
t=   [0 1 1 3 3 3 5 6 8 10]';
uvec=[0 0 1 1 2 1 0 2 2  0]';
u=sig(uvec,t);
G1=getfilter(4,1,'fs',NaN);
G2=ltf([0.5 0.5],[1 0.5]);
y1=simulate(lss(G1),u)
SIG object with continuous time input-output state space data
  Name:        Simulation of butter filter of type lp
  Sizes:       N = 421,  ny = 1, nu = 1, nx = 4
y2=simulate(lss(G2),u)
SIG object with continuous time input-output state space data
  Sizes:       N = 201,  ny = 1, nu = 1, nx = 1
subplot(2,1,1), plot(u)
subplot(2,1,2), plot(y1,y2)
```



2.2.3 Defining Stochastic Signals

Stochastic signals are represented by an ensemble of realizations, referred to as the Monte Carlo data. The field y is the nominal signal, and the field `yMC` contains `MC` other realizations of the same signal, where the deterministic part is the same. The most straightforward way to define a stochastic signal in the *Signals and Systems Lab* is to use a PDFCLASS object for the stochastic part:

```
N=100;
fs=10;
t=(0:N-1)'/fs;
yvec=sin(pi*t);
y=sig(yvec,fs);
V=ndist(0,1);
yn=y+V;  % One nominal realization + 20 Monte Carlo realizations
yn.name='Sin with noise signal';
y.MC=0;
yn1=y+V;  % One realization
plot(yn,'conf',90)
```



The last lines show how you can change the number of Monte Carlo simulations beforehand; in this case the Monte Carlo simulation is turned off. The second example is a bit more involved using more overloaded operations and different stochastic processes.

```
yvec=sin(pi*t);
y=sig(yvec,fs);
A=udist(0.5,1.5);
V=ndist(0,1);
yn=A.*y+V;
yn=yn.*yn;
plot(yn,'conf',90)
```

2.2.4 Data Preprocessing

Data preprocessing refers to operations on a signal you usually want to do prior to signal analysis with Fourier transform or model-based approaches. Table 2.4 summarizes the SIG object's methods discussed in this section.

To illustrate the different kind of data operations available, assume that a nonuniformly sampled signal `y(tk)` is in the SIG object `y`, and the task is to reveal the signal's low-frequency content. You can then apply one or more of the following operations:

- Interpolation of a continuous-time (nonuniformly sampled) signal `y(tk)` to an arbitrary time grid is performed by `y2=interp(y1,t);`
- As a special case of the above, sampling interpolates a continuous-time signal to a uniform grid `y[k] = y(kT)`. The call `y2=sample(y1,fs);` is the same as `y2=interp(y1,(0:N-1)/fs);` where `N=ceil(t(end)*fs);`

Table 2.4: SIG *data pre-processing functions*

`interp`	Interpolate from `y(t1)` to `y(t2)`
`sample`	Special case of interp, where `t2` is uniform time instants specified by a sampling frequency `fs`
`detrend`	Remove trends in nonstationary time series
`window`	Compute and apply a data window to SIG objects
`resample`	Resample uniformly sampled signal using a band-limitation assumption
`decimate`	Special case of resample for down-sampling

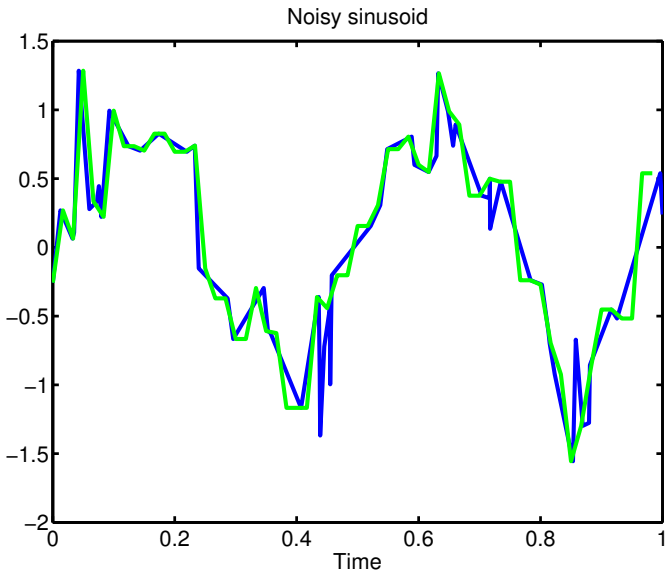
- Resampling a discrete-time signal to a more sparse or dense (using an antialias filter) time grid. You can do this by `y2=resample(y1,n,m);` This operation resamples `y[k] = y(kT)`, `k = 1, 2,..., N` to `y[l] = y(lnT/m)`, `l = 1, 2,..., ceil(mN/n)`.
- As a special case of the previous operation, decimation decreases the sampling frequency by a factor n. The calls `y2=decimate(y1,n);` is the same as `y2=resample(y1,n,1);`
- Prewindowing, using for instance a standard window: `y3=window(y2,'hamming');` The low-level window function `getwindow` is used internally, and it returns the applied window as a vector. There are many window options such as `box`, `Bartlett` (triangular), `Hamming`, `Hanning`, `Kaiser`, `Blackman`, or `spline` windows. The latter convolves a uniform window with itself an optional number of times.
- Filtering, for instance using a standard filter: `G=getfilter(n,fc,type,method);` `y4=filter(G,y3);` The low-level filter function is called inside the LTF (linear transfer function) method `filter`. The main difference is that the LTF method filters each signal in a multivariate signal object individually.

All these operations apply to multivariate signals and stochastic signals (represented by Monte Carlo realizations). The following example illustrates the entire chain. First, generate a windowed sinusoid with random sampling times:

```
N=60;
t=[0;sort(rand(N-2,1));1];
yt=sin(4*pi*t)+sqrt(0.1)*randn(N,1);
y=sig(yt,t);
y.name='Noisy sinusoid';
```

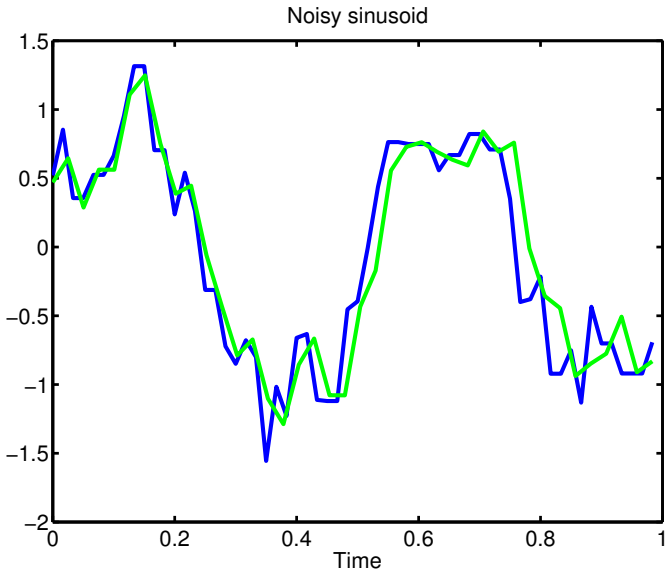
The signal is linearly interpolated at 30 equidistant time instants,

```
fs=N/(t(end)-t(1));
y1=sample(y,fs);
plot(y,y1)
```



and resampled to 20 time instants over the same interval (the *Signals and Systems Lab* automatically uses an antialias filter).

```
y2=resample(y1,3,2);
plot(y1,y2)
```



The signal is now prewindowed by a Hamming window,

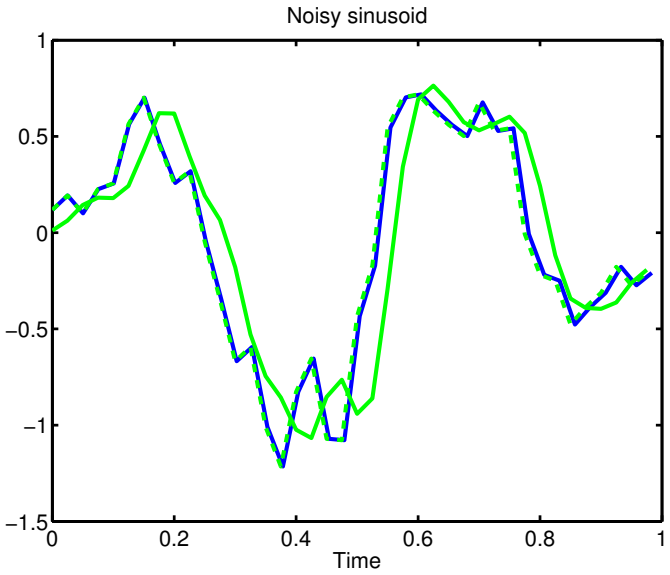
```
y3=window(y2,'kaiser');
```

and low-pass filtered by a Butterworth filter:

```
G=getfilter(4,0.5,'type','LP','alg','butter');
y4=filter(G,y3);
```

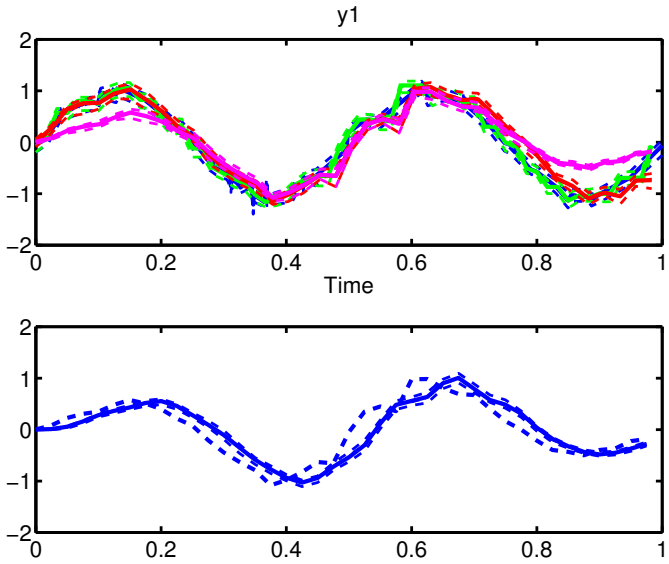
Finally, compare all the signals:

```
plot(y3,y4)
```



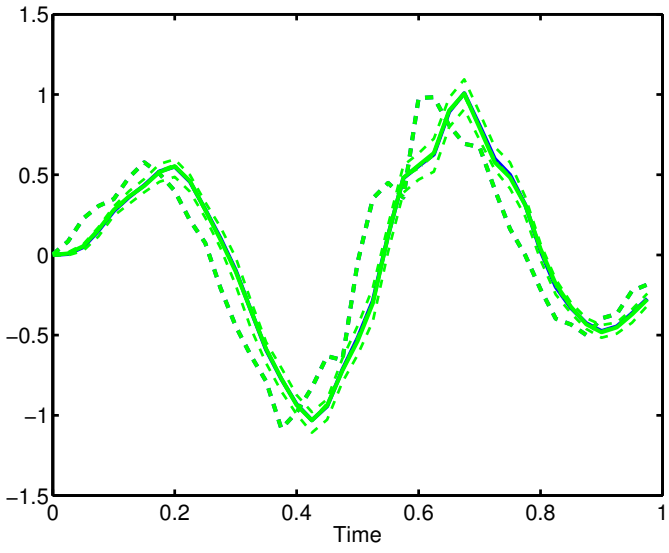
To illustrate the use of Monte Carlo simulations, the next code segment defines a stochastic signal. It does so by adding a PDFCLASS object to the deterministic signal rather than just one realization. All other calls in the following code segment are the same as in the previous example. The main difference is that Monte Carlo realizations of the signal are propagated in each step, which makes it possible to add a confidence bound in the plots.

```
yt=sin(4*pi*t);
y=sig(yt,t);
y=y+0.1*ndist(0,1);
fs=N/(t(end)-t(1));
y1=sample(y,fs);
y2=resample(y1,3,2);
y3=window(y2,'kaiser');
y4=filter(G,y3);
subplot(2,1,1), plot(y,y1,y2,y3,'conf',90)
subplot(2,1,2), plot(y4,'conf',90)
```



All the operations are basically linear, so the expected signal after all four operations is very close to the nominal one as illustrated next.

```
plot(E(y4),y4,'conf',90)
```



The SIG object includes algorithms for converting signals to the following objects:

Table 2.5: SIG *conversions*

`sig2ft`	Compute the Fourier transform (approximation) of a signal
`sig2covfun`	Estimate the (cross-)covariance function of (the columns in) a signal
`sig2spec`	Perform spectral analysis of a signal

- Fourier transform (FT)
- Covariance function (COVFUN)
- Spectrum (SPEC)

The easiest way to invoke these algorithms is to use the call from the constructors. That is, simply use `c = covf(z)` and so on rather than `c = sig2covf(z)`, although the result is the same. Further, estimation algorithms for PDF distributions, time-frequency descriptions, as well as LTF, LSS, and ARX models are contained in the corresponding objects.

2.3 Examples of Real Signals

The *Signals and Systems Lab* contains a database, `dbsignal`, with real-world application data as summarized in Table 2.6.

Each example contains a data file and one M-file, which in turn contains a brief explanation of the data in the help text and provides some initial illustrations of the data. By just calling the function, the *Signals and Systems Lab* shows both the info and a plot.

Now , consider the dataset GENERA,

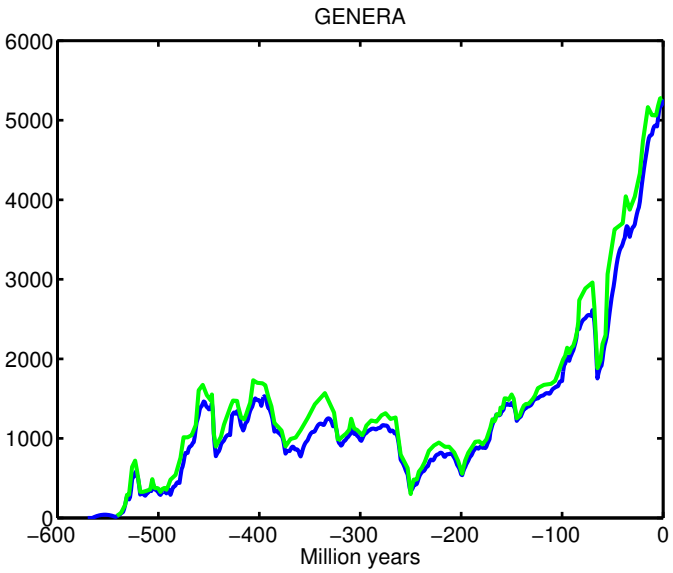
```
load genera
info(y1)
Name:    GENERA
Description:
 S3LAB Signal Database:  GENERA

 The data show how the number of genera evolves over time (million years)
 This number is estimated by counting number of fossils in terms of
 geologic periods, epochs, states and so on.
 These layers of the stratigraphic column have to be assigned dates and
 durations in calender years, which is here done by resampling techniques.

 y1       uniformly resampled data using resampling techniques
 y2       original non-uniformly sampled data

 See Brian Hayes, "Life cycles", American Scientist, July-August, 2005, ➥
     p299-303.


Signals:
Time:  Million years
```

Table 2.6: SIG *real signals examples*

NAME	DESCRIPTION
bach	A piece of music performed by a cellular phone.
carpath	Car position obtained by dead-reckoning of wheel velocities.
current	Current in an overloaded transformer.
eeg_human	The EEG signal y shows the brain activity of a human test subject.
eeg_rat	The EEG signal y shows the brain activity of a rat.
ekg	An EKG signal showing human heartbeats.
equake	Earthquake data where each of the 14 columns shows one time series.
ess	Human speech signal of 's' sound.
fricest	Data z for a linear regression model used for friction estimation
fuel	Data $y = z$ from measurements of instantaneous fuel consumption.
genera	The number of genera on earth during 560 million years.
highway	Measurements of car positions from a helicopter hovering over a highway.
pcg	An PCG signal showing human heartbeats.
photons	Number of detected photons in X-ray and gamma-ray observatories.
planepath	Measurements $y = p$ of aircraft position.

There are two signals in the genera file: `y1` and `y2`. The first is the original data, which is nonuniformly sampled. The second signal is resampled uniformly by stochastic resampling techniques. The following plot illustrates these signals

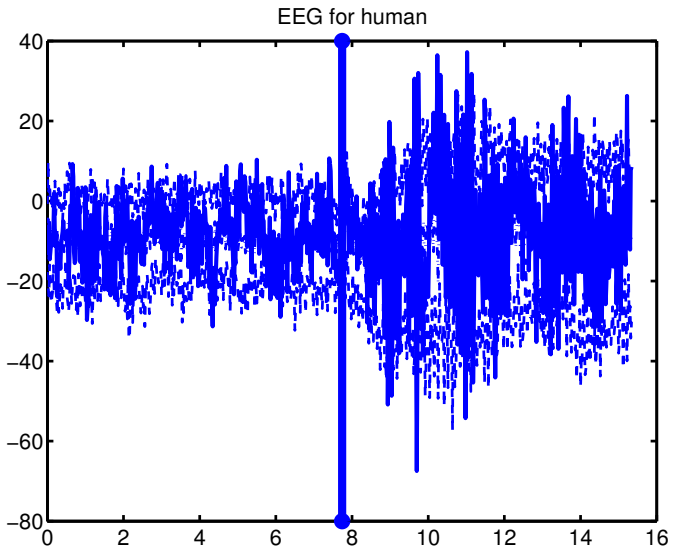
```
plot(y1,y2)
```



Note that the `plot` method can illustrate multiple signals of different kinds (here with uniform and nonuniform samples) at the same time. The time axis is correct, and this time information is kept during further processing. For instance, the frequency axis in frequency analysis is scaled accordingly.

The next example contains a marker field and multiple signal realizations:

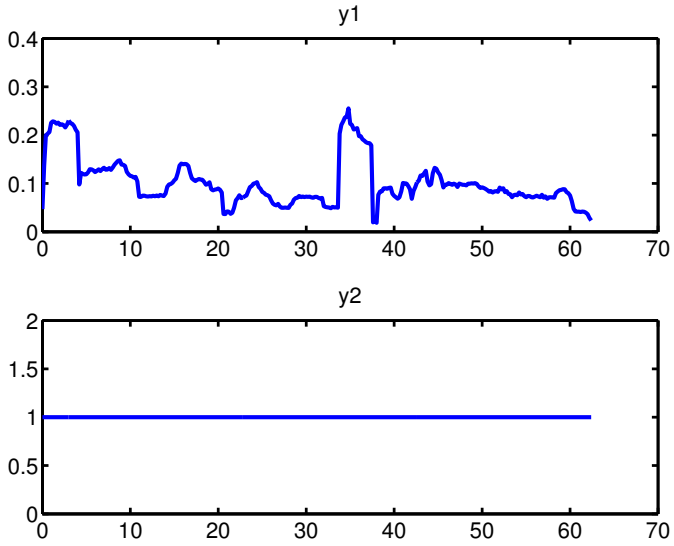
```
load eeg_human
info(y)
Name:    EEG for human
Signals:
Time:  s
plot(y,'conf',90)
```



The visualization of markers in `plot` uses vertical lines of the same color as the plot line for that signal. The different realizations of the signal, corresponding to different individual responses, are used to generate a confidence bound around the nominal realization. This is perhaps not terribly interesting for stochastic signals. However, Fourier analysis applied to these realizations gives confidence in the conclusions, namely that there is a 10 Hz rest rythm starting directly after the light is turned off.

The following is an example with both input and output data. Because the input and output are of different orders of magnitudes, the code creates two separate plots of the input and output.

```
load fricest
info(y)
Name:    Friction estimatin data
Description:
y is the wheel slip, u is the regression vector [mu 1], model is y=u*th+e
Signals:
subplot(2,1,1), plot(y(:,:,[]))  % Only output
subplot(2,1,2), uplot(y)         % Only input
```



2.4 Standard Signal Examples

Besides the database `dbsignal`, a number of standard examples are contained in getsignal.

2.4.1 Discrete-Time Signals

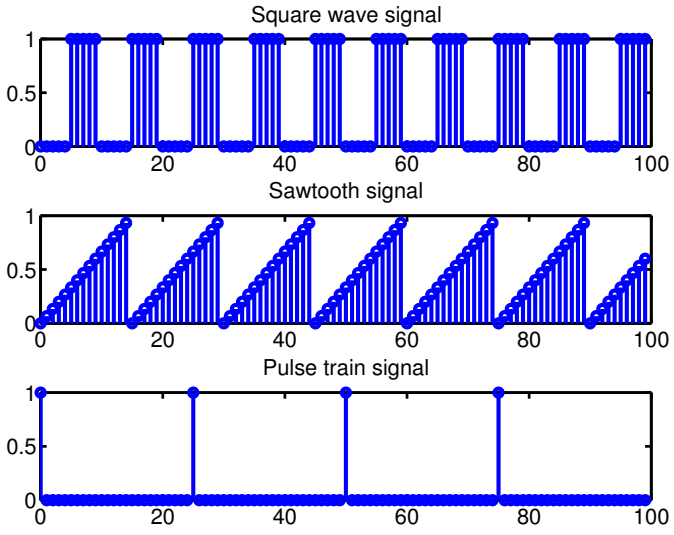
Table 2.4.1 summarizes the options in `dbsignal` for discrete-time signals: There are also a few other signals that come with the *Signals and Systems Lab* for demonstration purposes.

One category of signals contains periodic functions:

```
s1=getsignal('square',100,10);
s2=getsignal('sawtooth',100,15);
s3=getsignal('pulsetrain',100,25);
subplot(3,1,1), stem(s1)
subplot(3,1,2), stem(s2)
subplot(3,1,3), stem(s3)
```

Table 2.7: SIG *examples of discrete-time signals*

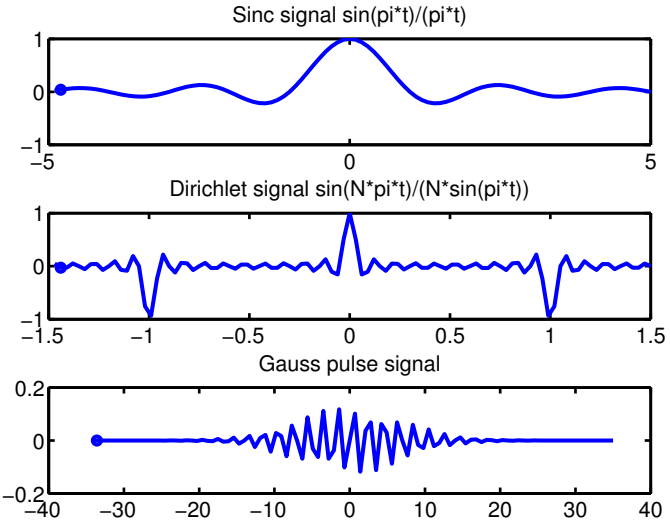
EXAMPLE	DESCRIPTION
`ones`	A unit signal `[1 ... 1]` with `nu=opt1` dimensions
`zeros`	A zero signal `[0 ... 0]` with `nu=opt1` dimensions
`pulse`	A single unit pulse `[0 1 0...0]`
`step`	A unit step `[0 1...1]`
`ramp`	A unit ramp with an initial zero and $N/10$ trailing ones
`square`	Square wave of length `opt1`
`sawtooth`	Sawtooth wave of length `opt1`
`pulsetrain`	Pulse train of length `opt1`
`sinc`	`sin(pi*t)/(pi*t)` with `t=k*T` where `T=opt1`
`diric`	The periodic sinc function `sin(N*pi*t)/(N*sin(pi*t))` with `t=k*T` where `T=opt1` and `N=opt2`
`prbs`	Pseudo-random binary sequence (PRBS) with basic period length `opt1` (default `N/100`) and transition probability `opt2` (default 0.5)
`gausspulse`	`sin(pi*t)*p(t,sigma)` with `t=k*T` where `p` is the Gaussian pdf, `T=opt1` and `sigma=opt2`
`chirp1`	`sin(pi*(t+a*t*t)` with `t=k*T` where `T=opt1` and `a=opt2`
`sin1`	One sinusoid in noise
`sin2`	Two sinusoids in noise
`sin2n`	Two sinusoids in low-pass filtered noise



The second kind of signals contains the following window-shaped oscillat-

ing functions:

```
s4=getsignal('sinc',100,.1);
s5=getsignal('diric',100,0.03);
s6=getsignal('gausspulse',100,0.7);
subplot(3,1,1), stem(s4)
subplot(3,1,2), stem(s5)
subplot(3,1,3), stem(s6)
```



2.4.2 Continuous-Time Signals

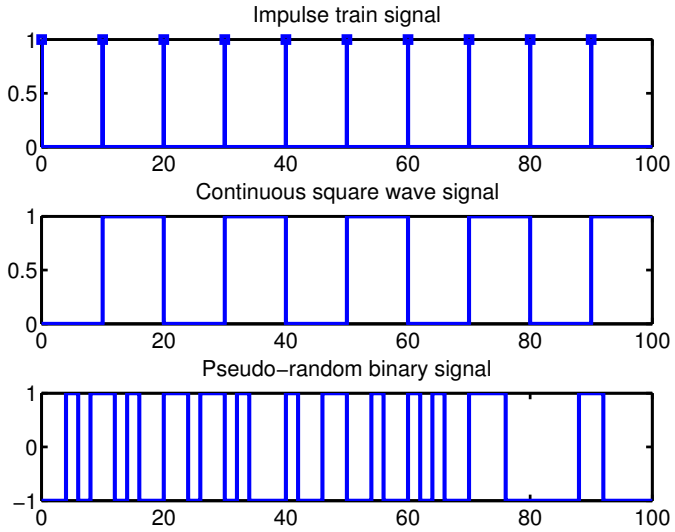
Table 2.4.2 summarizes the options in `dbsignal` for continuous-time signals:

The following example illustrates some of these signals.

```
s7=getsignal('impulsetrain',100);
s8=getsignal('csquare',100);
s9=getsignal('cprbs',100);
subplot(3,1,1), plot(s7)
subplot(3,1,2), plot(s8)
subplot(3,1,3), plot(s9)
```

Table 2.8: SIG *examples of continuous-time signals*

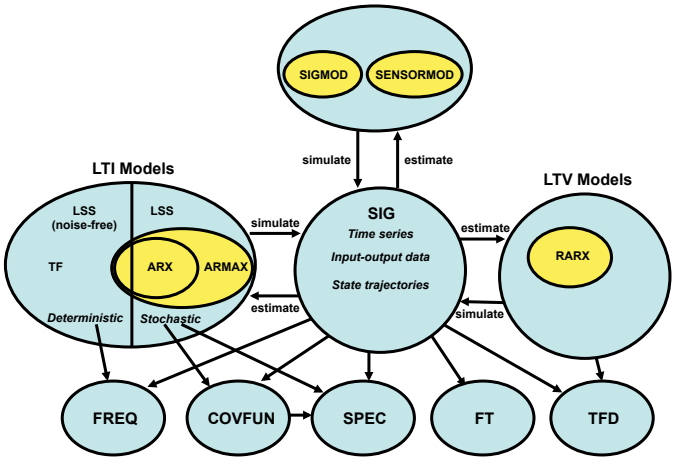
EXAMPLE	DESCRIPTION
`sin1`	One sinusoid in noise
`sin2`	Two sinusoids in noise
`sin2n`	Two sinusoids in low-pass filtered noise
`cones`	A unit signal `[1 1]` with `t=[0 N]` and `ny=opt1`
`czeros`	A zero signal `[0 0]` with `t=[0 N]` and `ny=opt1`
`impulse`	A single unit impulse `[0 1 0 0]` with `t=[0 0 0 N]`
`cstep`	A unit step `[0 1 1]` with `t=[0 0 N]`
`csquare`	Square wave of length `N` and period length `opt1`
`impulsetrain`	Pulse train of length `N` and period length `opt1`
`cprbs`	Pseudo-random binary sequence with basic period length `opt1` (default `N/100`) and transition probability `opt2` (default 0.5)



3

Overview of Signal and Model Objects

One concise way to view the *Signals and Systems Lab* is as a tool that converts a signal among different representations. The purpose might be to analyze the properties of a signal, to remove noise from observations of a signal, or to predict a signal's future values. In the *Signals and Systems Lab*, conversion of the signal into different domains is the key tool. The different domains are covered by different objects as the following picture summarizes:



The following list provides an overview of the main signal representations (objects):

1. SIG object: The signal domain represents an observed signal $y(t)$. Discrete-time signals are represented by their sample values $y[k] =$

$y(kT)$ and their sampling frequency $f_s = 1/T$. Continuous-time signals are approximated by nonuniformly sampled signals $y(t_k)$, $k = 1, 2, ..., N$, where you specify the time vector t_k explicitly. For input-output systems, you can provide an input signal $u[k]$, and for state-space models it is possible to store the state vector $x[k]$ in the SIG object. PLOT illustrates signals as piecewise constant or linear curves, and STEM creates stem plots. For detailed information, see Section 2.

2. FT object: The frequency-domain representation implemented in the Fourier transform (FT) object extends the discrete Fourier transform (DFT) to the discrete-time Fourier transform (DTFT) by using zero-padding. Internally, the fast Fourier transform (FFT) is used. The DFT is defined by

$$Y_N(f) = T_s \sum_{k=1}^{N} y[k] e^{-i2\pi k T_s f}, \tag{3.1}$$

and the DTFT is obtained by padding the signal with trailing zeros. This method approximates the continuous Fourier transform (FT) for the windowed signal. The FFT algorithm is instrumental for all computations involving the frequency domain. For nonuniform sampling, it is possible to approximate the Fourier transform with

$$Y_N(f) = T_s \sum_{k=1}^{N} y(t_k)(t_k - t_{k-1}) e^{-i2\pi t_k f}. \tag{3.2}$$

This can be seen as a Riemann approximation of the FT integral. Note that the *Signals and Systems Lab* uses the Hz frequency convention rather than rad/s.

3. LTI object: Linear time-invariant (LTI) signal models are covered in the following LTI objects:

   (a) LSS (state-space objects)

   (b) LTF (transfer functions)

   (c) ARX models (AutoRegressive eXogenous input) and the special cases FIR and AR models.

   The TF and SS objects cover the important subclass of deterministic filters.

   All LTI objects can be uncertain in that you can specify one or more parameters as stochastic variables, and this uncertainty is propagated in further operations and model conversions, simulations, and so on using the Monte Carlo simulation principle. Table 3.1 summarizes the objects with their structural parameters and model definitions.

Table 3.1: *Definition of model objects*

OBJECT	NN	DEFINITION
`tf`	`[na, nb, nk]`	$A(q)y(t) = B(q)u(t-nk)$
`arx`	`[na, nb, nk]`	$A(q)y(t) = B(q)u(t-nk) + v(t)$
`ss`	`[nx, nu, nv, ny]`	$y[k] = C(qI-A)^{-1}\big(Bu[k] + v[k]\big) + e[k]$

4. FREQ object: The frequency function of an LTI object. For continuous-time transfer functions, $H(s)$, the frequency function is $H(i2\pi f)$, and for discrete-time models, $H(q)$, the frequency function is $H(\exp(i2\pi f/fs))$. Note again that the *Signals and Systems Lab* uses the Hz frequency convention rather than rad/s. For LTI objects with both deterministic and stochastic inputs, the FREQ object represents the frequency function of the deterministic input-output dynamics. For the stochastic signal model part, see the COV and SPEC objects as discussed next.

5. COV object: The covariance function is defined as

$$R(\tau) = \mathrm{E}\big(y(t)y(t-\tau)\big). \tag{3.3}$$

   for zero-mean stationary stochastic signals. It measures how the dependence of samples decays with distance. For white noise, $R(\tau) = 0$ for all τ not equal to 0, meaning that any two signal values are independent.

6. SPEC object: The spectral domain is one of the most important representation of stochastic signals. The spectrum is defined as the Fourier transform of the covariance function:

$$\Phi(f) = \int R(\tau)e^{-i2\pi f\tau}d\tau. \tag{3.4}$$

   It is related to the Fourier transform as

$$\Phi(f) = \lim_{N\to\infty} \frac{T}{N} \left|Y_N(f)\right|^2. \tag{3.5}$$

   Comparing it with Parseval's formula

$$\int_{-\infty}^{\infty} y^2(t)dt = \frac{1}{2\pi}\int_{-\infty}^{\infty} \left|Y_N(f)\right|^2 df, \tag{3.6}$$

   which relates distribution of signal energy over time and frequency, the interpretation is that $\Phi(f)$ measures the energy in a signal $y(t)$ that shows up at frequency f.

7. LTV object: A linear time-varying (LTV) signal model is similar to an LTI object except that the transfer functions depend on time. An adaptive filter produces an estimate of an LTV model.

8. TFD object: While the spectrum and covariance function are suitable representations for stationary signals (spectrum and covariance do not depend on time), the time-frequency description (TFD) denotes the generalization of the Fourier transform and spectrum to nonstationary signals. The time-varying transform can be defined as

$$Y(f,t) = \int w(t-s)y(s)e^{-i2\pi fs}ds, \tag{3.7}$$

   where $w(t-s)$ is a kernel function (window) that provides an analysis window around the time t. You can compute TFDs directly from the signal or from an LTV model.

9. PDFCLASS object: The amplitude distribution of a signal vector is described by its probability density function (PDF). Specific distributions such as the Gaussian, Gaussian mixture, exponential, beta, gamma, student's t, F, and χ^2 are children of the PDFCLASS with names as NDIST, GMDIST, EXPDIST, BETADIST, GAMMADIST, TDIST, FDIST, CHIDIST, and so on. Common tasks in statistics involve:

   (a) Random number generation, where the RAND function is an overloaded method on each distribution.

   (b) Symbolic computations of density functions such as $Z = \tan(Y/X)$. Here the empirical distribution EMPDIST is central, where Monte Carlo samples approximate the true but nonparametric distribution.

   (c) Evaluation of the PDF, the cumulative distribution function (CDF), the error function (ERF), or certain moments (mean, variance, skewness, and kurtosis) of given distributions.

   (d) Fitting parametric distributions to empirical data.

   (e) Visualization of data and PDFs.

The applications touched upon here are illustrated by examples in the following chapters.

4

The SIGMOD Objects

4.1 Definition of the SIGMOD Object

The signal model is defined as

$$y(t) = h(t, \theta) + e(t), \tag{4.1}$$
$$e(t) \sim p_e(e). \tag{4.2}$$

Here, `h` is an inline object or a string with arguments `t,th`, and `nn=[ny nth]` gives the dimensions of `y` and the parameters `th`. The convention of dimensions is as follows:

- `y` is an `(ny,1)` vector or an `(ny,N)` matrix.
- `th` is a vector of length `nth`.

The distribution `pe` is specified as a `pdfclass` object, or as a covariance matrix of suitable dimension, in which case an `ndist` object is created. Other fields are `name, thlabel, ylabel`.

SIGMOD is a child of NL. Most of the methods displayed for SIGMOD are no relevant for this class. The most important useful methods are `simulate` for generating a signal `y`, and `estimate` for estimating the parameters `th`.

5

The SENSORMOD Object

5.1 Definition of the SENSORMOD Object

The signal model is defined as

$$y(t) = h(t, x(t), u(t); \theta) + e(t), \tag{5.1}$$
$$e(t) \sim p_e(e). \tag{5.2}$$

The constructor for this model has the basic syntax `m=sensormod(h,nn)`. Here, `h` is an inline object or a string with arguments `t,th`, and `nn=[nx nu ny nth]` gives the dimensions of `x`, `u`, `y` and the parameters `th`. The convention of dimensions is as follows:

- `y` is an `(ny,1)` vector or an `(ny,N)` matrix.
- `u` is an `(nu,1)` vector or an `(nu,N)` matrix.
- `x` is an `(nx,1)` vector or an `(nx,N)` matrix.
- `th` is a vector of length `nth`.

The distribution `pe` is specified as a `pdfclass` object, or as a covariance matrix of suitable dimension, in which case an `ndist` object is created. Other fields are `name, thlabel, ylabel, ulabel, xlabel`.

SENSORMOD is as SIGMOD a child of NL, but where many more of the methods are relevant. There are also methods for estimation and analysis of a sensor network, that do not apply to the NL class.

5.2 Constructor

The SENSORMOD constructor has the syntax

```
m=sensormod(h,nn)
```

The signal model is defined as

$$y(t) = h(t, x(t), u(t); th) + e(t), \tag{5.3}$$
$$e(t) \sim p_e, \tag{5.4}$$
$$x(0) \sim p_{x_0}, \tag{5.5}$$
$$\mathrm{E}[x(0)] = x_0. \tag{5.6}$$

The constructor `m=sensormod(h,nn)` has two mandatory arguments:

- The argument `h` defines the sensor model and is entered in one of the following ways:
  - A string, with syntax `s=sensormod(h,nn);`. Example:
    ```
    h='-th*x^2';
    ```
  - An inline function, with the same syntax `s=sensormod(h,nn);`. Example:
    ```
    h=inline('-x^2','t','x','u','th');
    ```
  - An M-file. Example:
    ```
    function h=fun(t,x,u,th)
    h=-th*x^2;
    ```
    This m-file can be used in the constructor either as a string with the name, or as a function handle,
    ```
    m=sensormod('fun',h,nn);
    m=sensormod(@fun,h,nn);
    ```
    Here, `feval` is used internally, so the function handle is to prefer for speed reasons.

  It is important to use the standard model parameter names `t, x, u, th`. For inline functions and M-files, the number of arguments must be all these four even if some of them are not used, and the order of the arguments must follow this convention.

- `nn=[nx,nu,ny,nth]` denotes the orders of the input parameters. These must be consistent with the entered `f` and `h`. This apparantly trivial information must be provided by the user, since it is hard to unambigiously interpret all combinations of input dimensions that are possible otherwise. All other tests are done by the constructor, which calls the function `h` with zero inputs of appropriate dimensions according to `nn`, and validates the dimensions of the returned outputs.

All other parameters are set separately:

- `pe`, and `px0` are distributions for the measurement noise and state x, respectively. All of these are entered as objects in the `pdfclass`, or as covariance matrices when a Gaussian distribution is assumed.
- `th` and `P` are the fields for the parameter vector and optional covariance matrix. Only the second order property of model uncertainty is currently supported.
- `fs` denotes, similarly to the LTI objects, the sampling frequency, where the convention is that `fs=NaN` means continuous time systems (which is set by default). All NL objects are set to continuous time models in the constructor, and the user has to specify a numeric value of `fs` after construction if a discrete model is wanted. For sensor models, the sampling time does not influence any functions, but the data simulated by the model inherits this sampling time.
- `xlabel`, `thlabel`, `ulabel`, `ylabel`, and name are used to name the variables and the model, respectively. These names are inherited after simulation in the SIG object, for instance.

5.3 Tips for indexing

It is important to understand the indexing rules when working with model objects. First, the sizes of the signals are summarized below.

`x`	$(n_x, 1)$ vector or (n_x, N) matrix
`u`	$(n_u, 1)$ vector or (n_u, N) matrix
`y`	$(n_y, 1)$ vector or (n_y, N) matrix
`th`	n_{th} vector

Suppose the sensor model is

$$h(x) = x_1^2 + x_2^2. \tag{5.7}$$

This can be defined as

```
  s=sensormod('x(1)^2+x(2)^2',[2 0 1 0])
NL constructor warning: try to vectorize h for increased speed
SENSORMOD object
   y = x(1)^2+x(2)^2
   x0'  = [0,0]
  States:  x1     x2
  Outputs: y1
```

The first line gives a hint of that vectorization can be important for speed, when massive parallel calls are invoked by some of the methods. Using the convention above, and remembering the point power operation in MATLAB™, a vectorized call that works for parallelization looks as follows:

```
s=sensormod('x(1,:).^2+x(2,:).^2',[2 0 1 0])
```

This time, the warning will disappear. An alternative way to compute the squared norm is given below:

```
s=sensormod('sum(x(:,:).^2,1)',[2 0 1 0]);
s=sensormod('sum(x(1:2,:).^2,1)',[2 0 1 0]);
```

The second form is to prefer in general. The reason is that for some further operations, the state is extended or augmented, and then it is important to sum over only the first two states.

5.4 Generating Standard Sensor Networks

There are many pre-defined classes of sensor models easily accessible in the function `exsensor`, with syntax

```
s=exsensor(ex,M,N,nx)
```

The arguments are:

- M is the number of sensors (default 1)
- N is the number of targets (default 1)
- n_x denotes the state dimension of each target (default 2). A value larger than 2 can be used to reserve states for dynamics that are added later.

The rows in `h` are symbolically `h((m-1)*N+n,:)=sensor(pt(n),ps(m))`, or mathematically

$$h_{(m-1)N+n,:} = \|p_t(n), p_s(m)\|, \quad m = 1, 2, \dots M, \quad n = 1, 2, \dots N, \tag{5.8}$$

for the chosen norm between the target position p_t and the sensor position p_s. The position for target n is assumed to be

$$p_t(n) = x\big((n-1)n_x + 1 : (n-1)n_x + 2\big), \tag{5.9}$$

and the position for sensor m is assumed to be

$$p_s(m) = \theta\big(2(m-1) + 1 : 2(m-1) + 2\big). \tag{5.10}$$

If `slam` is appended to the string in `ex`, then a SLAM model is obtained For SLAM objects, also the sensor positions are stored in the state, and

$$p_s(m) = x\big(Nn_x + 2(m-1) + 1 : Nn_x + 2(m-1) + 2\big). \tag{5.11}$$

The default values of target locations `s.x0` and sensor locations `s.th` are randomized uniformly in [0,1]x[0,1]. Change these if desired. The options for `ex` are summarized in Table 5.1.

`l=exsensor('list')` gives a cell array with all current options.

Table 5.1: *Standard sensor networks in* `exsensor`

EX	DESCRIPTION
`'toa'`	2D TOA as range measurement $\|p_t(n) - p_s(m)\|$
`'tdoa1'`	2D TDOA with bias state $\|p_t(n) - p_s(m)\| + x(Nn_x + 1)$
`'tdoa2'`	2D TDOA as range differences $\|p_t(n) - p_s(m)\| - \|p_t(k) - p_s(m)\|$
`'doa'`	2D DOA as bearing measurement $\arctan2(p_t(n), p_s(m))$
`'rss1'`	RSS with parameters $\theta(n) + \theta(N+1) \cdot 10 \cdot \log_{10}(\|p_t(n, 1:2) - p_s(m)\|)$
`'rss2'`	RSS with states $x(n, Nn_x + 1) + x(n, Nn_x + 2) \cdot 10 \cdot \log_{10}(\|p_t(n) - p_s(m)\|)$
`'radar'`	2D combination of TOA and DOA above
`'gps2d'`	2D position
`'gps3d'`	3D position
`'mag2d'`	2D magnetic field disturbance
`'mag3d'`	3D magnetic field disturbance
`'quat'`	Quaternion constraint $q^T q = 1$
`'*slam'`	`*` is one of above, θ is augmented to the states

5.5 Utility Methods

The SENSORMOD utility methods are summarized in Table 5.2, without any further comments. They are inherited from the NL class, where more information can be found.

5.6 Object Modification Methods

The radar sensor can be built up as a TOA and a DOA sensor using `addsensor`.

```
s=exsensor('radar')
SENSORMOD object: RADAR
       / sqrt((x(1,:)-th(1)).^2+(x(2,:)-th(2)).^2) \
   y = \ atan2(x(2,:)-th(2),x(1,:)-th(1))          / + e
   x0'  = [0.52,0.23]
   th'  = [0.18,0.22]
  States:  x1     x2
  Outputs: Range     Bearing
s1=exsensor('toa')
SENSORMOD object: TOA
   y = [sqrt((x(1,:)-th(1)).^2+(x(2,:)-th(2)).^2)] + N(0,0.0001)
   x0'  = [0.97,0.82]
   th'  = [0.37,0.03]
  States:  x1     x2
  Outputs: y1
s2=exsensor('doa')
SENSORMOD object: DOA
   y = [atan2(x(2,:)-th(2),x(1,:)-th(1))] + N(0,0.01)
   x0'  = [0.25,0.57]
```

Table 5.2: SENSORMOD *utility methods*

METHOD	DESCRIPTION
`arrayread`	`mji=arrayread(m,j,i)`, or simpler `mji=m(j:i)` is used to pick out sub-systems by indexing
`plot`	`plot(s1,s2,...,'Property','Value')` illustrates the sensor network
`simulate`	`y=simulate(s)` simulates a sensor model 1. `y=simulate(s,x)` gives $z = s(t, x)$ at times `t` and state `s.x0`. 2. `y=simulate(s,x)` gives $z = s(t, x)$ at times `x.t` and state `x.x`.
`display`	`display(s1,s2,...)` returns an ascii formatted version of the NL model
`nl2lss`	`[mout,zout]=nl2ss(m,z)` returns a linearized model $\mathbf{y} = \mathbf{H}x + \mathbf{e}$ using $H = dh(x)/dx$ evaluated at x as given in `s.x`.

Table 5.3: SENSORMOD *object modification methods*

METHOD	DESCRIPTION
`addsensor`	`ms=addsensor(m,s,Property1,Value1,...)` adds (another) sensor to the object
`removesensor`	`ms=removesensor(m,ind)` removes the sensors numbered `ind` from the object

```
   th'  = [0.43,0.61]
  States:  x1     x2
  Outputs: y1
s12=addsensor(s1,s2)
NL object: Motion model: TOA Sensor model: DOA
       / sqrt((x(1,:)-th(1)).^2+(x(2,:)-th(2)).^2) \
   y = \ atan2(x(2,:)-th(2),x(1,:)-th(1))          / + e
   x0'  = [0.97,0.82]
   th'  = [0.43,0.61]
  States:  x1     x2
  Outputs: y1     y1
  Param.: th1     th2
```

Conversely, a TOA and DOA model can be recovered from the RADAR model using `removesensor`.

```
s1=removesensor(s,2)
NL object
   y = [sqrt((x(1,:)-th(1)).^2+(x(2,:)-th(2)).^2)] + N(0,0.01)
   x0'  = [0.52,0.23]
   th'  = [0.18,0.22]
  States:  x1     x2
  Outputs: Range
  Param.: th1     th2
s2=removesensor(s,1)
```

Table 5.4: SENSORMOD *detection methods*

METHOD	DESCRIPTION
`detect`	`[b,level,h,T]=detect(s,y,pfa)` evaluates a hypothesis test $H_0 : y \sim e$ vs. $H_1 : y \sim s + e$
`pd`	`[p,t,lambda]=pd(m,z,pfa,Property1,Value1,...)` computes the probability of detection P_d using GLRT (that is, x^{ML} is estimated)
`pdplot1`	`pd=pdplot1(s,x1,pfa,ind)` plots detection probability $P_d(x)$ as a function on the grid $x(i(1))$ for given P_{fa} using the LRT (that is, x^o is assumed known)
`pdplot2`	`pd=pdplot2(s,x1,x2,pfa,ind)` plots detection probability $P_d(x)$ as a function on the grid $x(i(1), i(2))$ for given P_{fa}
`roc`	`[pd,pfa]=roc(s,x0,h)` plots the receiver operating characteristics (ROC) curve $P_d(P_{fa})$ as a function of false alarm rate P_{fa}

```
NL object
   y = [atan2(x(2,:)-th(2),x(1,:)-th(1))] + N(0,0.01)
   x0'  = [0.52,0.23]
   th'  = [0.18,0.22]
  States:  x1     x2
  Outputs: Bearing
  Param.: th1     th2
```

However, these functions are not primarily intended for the SENSORMOD class, but rather for the NL class.

5.7 Detection Methods

Detection is based on the hypothesis test

$$H_0 : \mathbf{y} = \mathbf{e},$$
$$H_1 : \mathbf{y} = \mathbf{h}(x^0) + \mathbf{e},$$

where $\mathbf{e} \sim \mathcal{N}(0, R)$ and $R = \mathrm{Cov}(\mathbf{e})$ The likelihood ratio test statistic is defined as

$$T(\mathbf{y}) = \mathbf{y}^T R^{-1} \mathbf{y},$$

and distributed as

$$H_0 : T(\mathbf{y}) \sim \chi^2_{n_x},$$
$$H_1 : T(\mathbf{y}) \sim \chi^2_{n_x}(\lambda),$$
$$\lambda = \mathbf{h}^T(x^0) R^{-1} \mathbf{h}(x^0),$$

Table 5.5: *Arguments for* `detect`

`y`	data sample as SIG model or vector
`pfa`	false alarm rate (default 0.01)
`b`	binary decision
`level`	level of the test
`h`	threshold corresponding to P_{fa}
`T`	Test statistic $T(y) = y^T R^{-1} y$

where $\chi^2_{n_x}(\lambda)$ defines the (non-central) chi-square distribution with n_x degrees of freedom and non-centrality parameter λ, see the classes `chi2dist` and `ncchi2dist`.

The probability of detection P_d can be computed as a function of the false alarm probability P_{fa} as

$$P_d = 1 - \Phi(\Phi^{-1}(1 - P_{fa}) - \lambda)$$

where Φ is the cumulative distribution function for the χ^2 distribution.

5.7.1 Method DETECT

Usage:

```
[b,level,h,T]=detect(s,y,pfa)
```

with arguments defined in Table 5.5. Example:

```
s=exsensor('toa',5,1);
s.pe=eye(5);
y=simulate(s)
SIG object with discrete time (fs = 1) stochastic state space data (no ➠
    input)
  Sizes:       N = 1,  ny = 5, nx = 2
  MC is set to: 30
  #MC samples:  0
[b,l,h,T]=detect(s,y,0.01)
b =
     1
l =
    0.9911
h =
    8.9708
T =
    9.1689
```

5.7.2 Method PD

Usage:

```
[p,T,lambda]=pd(s,y,pfa,Property1,Value1,...)
```

with arguments defined in Table 5.6. Example for the case $H_1 : y = x + e$:

Table 5.6: *Arguments for* `pd`

`y`	data sample as SIG model or vector
`pfa`	false alarm rate (default 0.01)
`p`	probability of detection
`level`	level of the test
`lambda`	non-centrality parameter $\lambda = x^T I(x)x$ evaluated at the ML estimate
`T`	Test statistic $T(y) = (y - h(x))^T R^{-1}(y - h(x))$ evaluated at the ML estimate

Table 5.7: *Arguments for* `pdplot1` *and* `pdplot2`

`x1`	Grid vector for first index
`x2`	Grid vector for second index in `pdplot2`
`pfa`	False alarm rate (default 0.01)
`ind`	Index of `s.x0` to vary, that is, `s.x0(ind)=x1(i)`
`pd`	Vector with $P_D(x)$

```
nx=7;
m1=nl('x','x',[nx 0 nx 0],1);
m1.pe=eye(nx);
[p,T,lambda]=pd(m1,1*ones(nx,1),0.01)
p =
    0.2234
T =
   18.4085
lambda =
     7
```

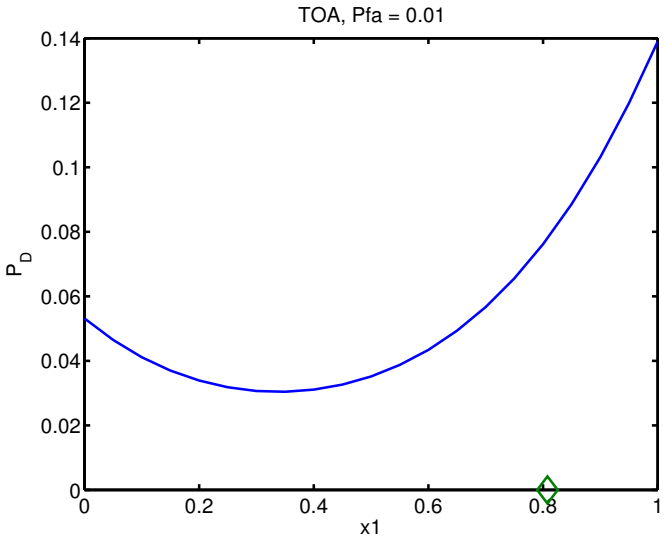
5.7.3 Methods PDPLOT1 and PDPLOT2

Usage:

```
pd=pdplot1(s,x1,pfa,ind)
pd=pdplot2(s,x1,x2,pfa,ind)
```

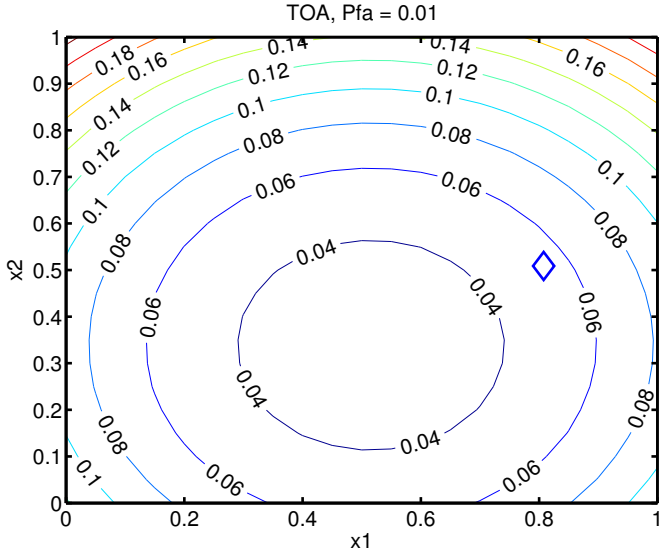
with arguments defined in Table 5.7. Without output argument, a plot is generated. Example with a TOA network, where the first coordinate is varied:

```
s=exsensor('toa',5,1);        % Default network
s.pe=1*eye(5);                % Increase noise level
pdplot1(s,0:0.05:1,0.01,1);   % Plot
```



Same example where both x_1 and x_2 are varied:

```
pdplot2(s,0:0.05:1,0:0.05:1,0.01,[1 2]);
```



5.7.4 Method ROC

Usage:

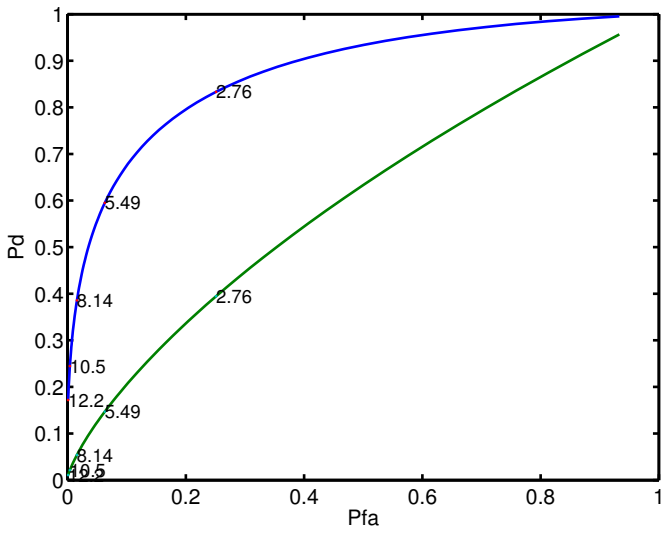
Table 5.8: *Arguments for* `roc`

`x0`	`nx` times `nroc` matrix, where `nroc` is the number of ROC curves that are plotted. Default (if `x0` omitted or empty) `x0=s.x0`
`h`	vector of thresholds. Default, it is gridded as $h(P_{FA})$
`pfa`	false alarm rate
`pd`	probability of detection

```
[pd,pfa]=roc(s,x0,h)
```

with arguments defined in Table 5.8. Example:

```
s=exsensor('toa',5,1); % Default network
s.pe=1*eye(5);         % Increase noise level
roc(s,[1 1;0.5 0.5]'); % Two roc curves
```



5.8 Analysis Methods

The analysis methods are based on the likelihood function

$$l(x) = p_{\mathbf{e}}(\mathbf{y} - \mathbf{h}(x)) = \prod_k p_{e_k}(y_k - h_k(x))$$

From this, one can define the Fisher Information Matrix (FIM)

$$\mathcal{I}(x) = \left(\frac{d\mathbf{h}}{dx}\right)^T \mathbf{R}^{-1} \frac{d\mathbf{h}}{dx},$$

Table 5.9: SENSORMOD *analysis methods*

METHOD	DESCRIPTION
`fim`	`I=fim(m,z,Property1,Value1,...)` computes the Fisher Information Matrix (FIM) $I(x)$
`crlb`	`x=crlb(s,y)` computes the Cramer-Rao lower bound $I^{-1}(x^o)$ for the state x^0 in at `y.x`
`crlb2`	`[cx,X1,X2]=crlb2(s,y,x1,x2,ind,type);` computes a scalar CRLB measure (i.e. trace tr $\left(\mathcal{I}^{-1}(x^o(i(1),i(2)))\right)$) over a 2D state space grid
`lh1`	`[lh,px,px0,x]=lh1(s,y,x,ind);` computes the one-dimensional likelihood function $p_e\big(y-h(x(i))\big)$ over a state space grid
`lh2`	`[lh,x1,x2,px,px0,X1,X2]=lh2(s,y,x1,x2,ind);` computes the two-dimensional likelihood function $p_e\big(y-h(x(i(1),i(2)))\big)$ over a state space grid

and the Cramer-Rao Lower Bound (CRLB)

$$\mathrm{Cov}(\hat{x}) \geq \mathcal{I}^{-1}(x^o),$$

which applies to any unbiased estimator $\hat{x}$. The methods described in this section, see Table 5.9 aim at computing and illustrating these functions.

5.8.1 Method FIM

Usage:

```
I=fim(m,z,Property1,Value1,...)
```

The FIM for

$$y = h(x) + e$$

is defined as

$$\mathcal{I}(x^o) = \left(\frac{dh}{dx}\right)^T R^{-1} \frac{dh}{dx}\bigg|_{x=x^o}.$$

The gradient is evaluated at the true state x^o. The syntax and options are identical to `ekf`.

Example:

```
%      m=nl('x',['sqrt(x(1)^2+x(2)^2);atan2(x(2),x(1))]',[2 0 2 0],1);
m=exsensor('radar');
m.th=[0 0]; % Sensor in origin
m.pe=diag([0.1 0.01]);
for x1=1:3:10;
```

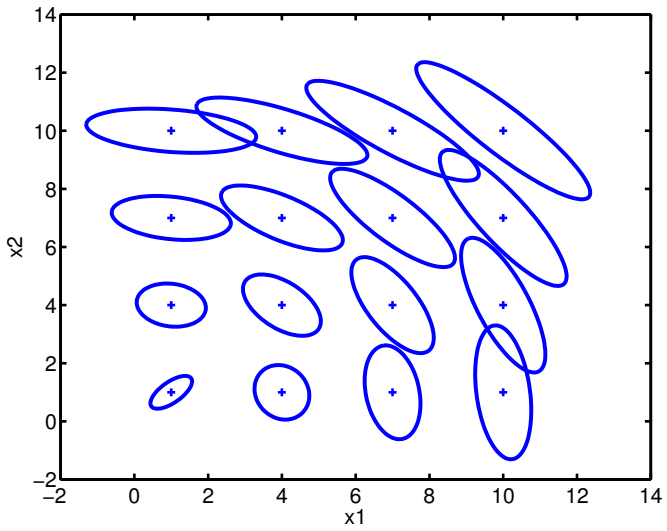
Table 5.10: Arguments for `fim`

`m`	NL object with model
`z`	State vector, or SIG object with true state in `z.x`
`I`	The `(nx,nx)` FIM matrix

```
    for x2=1:3:10;
        I=fim(m,[x1;x2]);
        mm=ndist([x1;x2],inv(I));
        plot2(mm,'legend',''), hold on
        I=fim(m,[x1;x2]);
        mm=ndist([x1;x2],inv(I));
        plot2(mm,'legend',''), hold on
        I=fim(m,[x1;x2]);
        mm=ndist([x1;x2],inv(I));
        plot2(mm,'legend',''), hold on
        I=fim(m,[x1;x2]);
        mm=ndist([x1;x2],inv(I));
        plot2(mm,'legend',''), hold on
    for x2=1:3:10;
        I=fim(m,[x1;x2]);
        mm=ndist([x1;x2],inv(I));
        plot2(mm,'legend',''), hold on
        I=fim(m,[x1;x2]);
        mm=ndist([x1;x2],inv(I));
        plot2(mm,'legend',''), hold on
        I=fim(m,[x1;x2]);
        mm=ndist([x1;x2],inv(I));
        plot2(mm,'legend',''), hold on
        I=fim(m,[x1;x2]);
        mm=ndist([x1;x2],inv(I));
        plot2(mm,'legend',''), hold on
    for x2=1:3:10;
        I=fim(m,[x1;x2]);
        mm=ndist([x1;x2],inv(I));
        plot2(mm,'legend',''), hold on
        I=fim(m,[x1;x2]);
        mm=ndist([x1;x2],inv(I));
        plot2(mm,'legend',''), hold on
        I=fim(m,[x1;x2]);
        mm=ndist([x1;x2],inv(I));
        plot2(mm,'legend',''), hold on
        I=fim(m,[x1;x2]);
        mm=ndist([x1;x2],inv(I));
        plot2(mm,'legend',''), hold on
    for x2=1:3:10;
        I=fim(m,[x1;x2]);
        mm=ndist([x1;x2],inv(I));
        plot2(mm,'legend',''), hold on
        I=fim(m,[x1;x2]);
        mm=ndist([x1;x2],inv(I));
        plot2(mm,'legend',''), hold on
        I=fim(m,[x1;x2]);
        mm=ndist([x1;x2],inv(I));
        plot2(mm,'legend',''), hold on
        I=fim(m,[x1;x2]);
        mm=ndist([x1;x2],inv(I));
        plot2(mm,'legend',''), hold on
    end
end
```

Table 5.11: *Arguments for* `crlb` *and* `crlb2`.

`s`	SENSORMOD object
`y`	SIG object of length 1, where `y.x` (and `y.u` if applicable) is used, if `y` is omitted, `s.x0` is used as `x`
`x`	SIG object where `x.x=y.x` and `x.Px` contains the CRLB
`x1,x2`	grid for the states `x(ind)` (values in `x0` are omitted for these two dimensions)
`ind`	vector with two integers indicating which states `x1` and `x2` refer to in `x`. Default `[1 2]`
`type`	Scalar measure of the FIM. The options are `'trace'`, `'rmse'` (default) defined as `rmse=sqrt(trace(P))`, `'det'`, `'max'` operator for transforming `P(x)` to scalar `c(x)`



5.8.2 Methods CRLB and CRLB2

Usage:

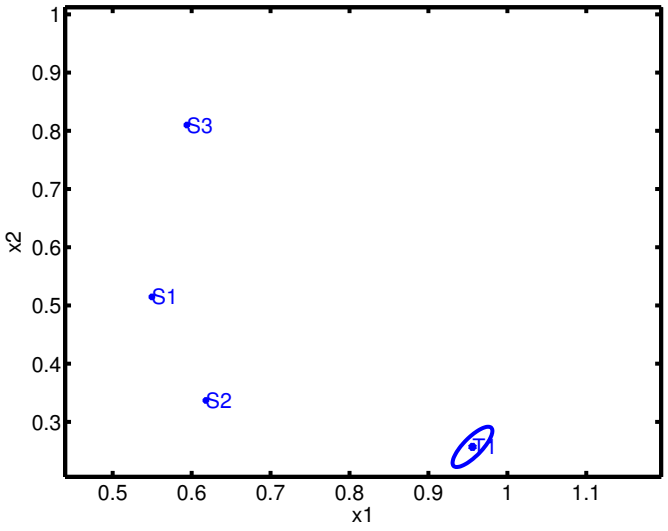
```
x=crlb(s,y)
```

Without output arguments, a confidence ellipse is plotted.

Example: The CRLB at a specific point in a simple TOA network is first illustrated.

```
s=exsensor('toa',3,1);
plot(s)
hold on
crlb(s)
```

```
SIG object with discrete time (fs = 1) stochastic state space data (no ➥
    input)
  Name:         TOA
  Sizes:        N = 1,  ny = 3, nx = 2
  MC is set to: 30
  #MC samples:  0
hold off
```

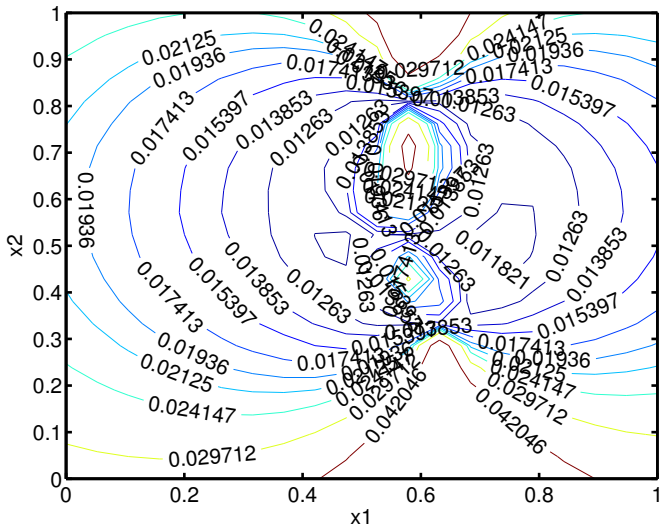


The trace of FIM corresponds to a lower bound on position error in length units, and this RMSE bound can be illustrated with a contour plot as a function of true position.

```
crlb2(s);
```

Table 5.12: *Arguments for* `lh1` *and* `lh2`.

`s`	SENSORMOD object
`y`	SIG object of length 1, where `y.x` (and `y.u` if applicable) is used, if `y` is omitted, `s.x0` is used as `x`
`x`	SIG object where `x.x=y.x` and `x.Px` contains the CRLB
`x1,x2`	grid for the states `x(ind)` (values in `x0` are omitted for these two dimensions)
`ind`	vector with two integers indicating which states `x1` and `x2` refer to in `x`. Default `[1 2]`
`type`	Scalar measure of the FIM. The options are `'trace'`, `'rmse'` (default) defined as `rmse=sqrt(trace(P))`, `'det'`, `'max'` operator for transforming `P(x)` to scalar `c(x)`



5.8.3 Methods LH1 and LH2

Usage:

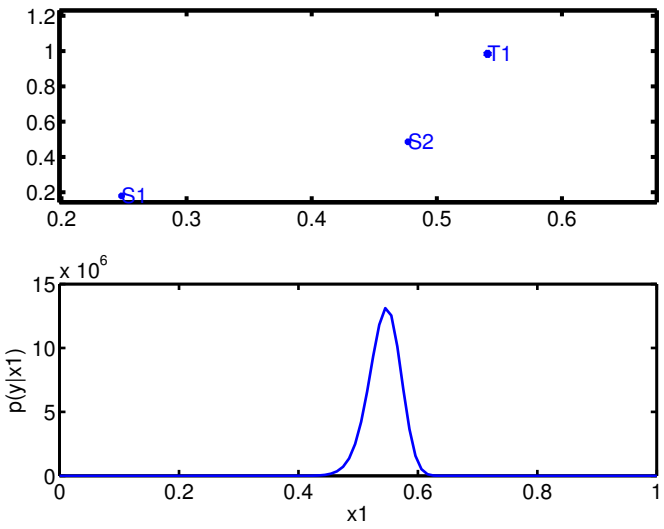
```
[lh,x1,px,px0,x]=lh1(s,y,x1,ind);
[lh,x1,x2,px,px0,X1,X2]=lh2(s,y,x1,x2,ind);
```

Without output arguments, a contour plot is generated. The arguments are described in Table 5.12.

Example: First, a TOA network with two sensors is generated. Then, the likelihood function $p(y|x_1)$ is evaluated and plotted.

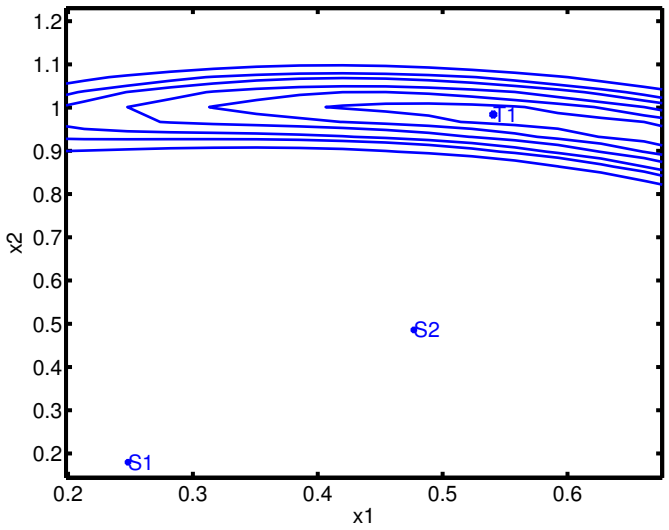
```
s=exsensor('toa',2,1);
y=simulate(s,1);
```

```
subplot(2,1,1), plot(s);
subplot(2,1,2), lh1(s,y);
```



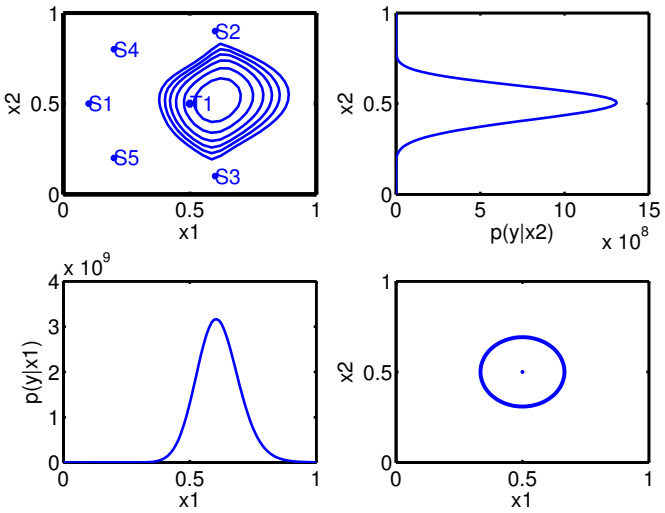
A two-dimensional likelihood $p(y|x_1, x_2)$ is generated next.

```
plot(s);
hold on, lh2(s,y); hold off
```



A more sophisticated plot is obtained below, using a DOA network with five sensors. Both the marginal likelihoods and the two-dimensional ones are computed, and the fourth quadrant is used to compare with the CRLB.

```
M=5; N=1;
s=exsensor('doa',M,1);
s.x0=[0.5;0.5];  s.pe=0.1*eye(M);
s.th=[0.1 0.5 0.6 0.9 0.6 0.1 0.2 0.8 0.2 0.2];
y=simulate(s);
subplot(2,2,1),  plot(s),
hold on, lh2(s,y);   hold off,   axis([0 1 0 1])
subplot(2,2,3),   lh1(s,y,[],1); set(gca,'Xlim',[0 1])
subplot(2,2,2),  [p,x,dum,dum]=lh1(s,y,[],2);
plot(p,x), set(gca,'Ylim',[0 1])
xlabel('p(y|x2)'), ylabel('x2')
subplot(2,2,4), crlb(s); axis([0 1 0 1])
```



5.9 Estimation Methods

The estimation methods in the SENSORMOD class are summarized in Table 5.13.

5.9.1 Methods LS and WLS

Usage:

```
[xhat,shat]=ls(s,y);
[xhat,shat]=wls(s,y);
```

where the arguments are explained in the table below.

`s`	SENSORMOD object
`y`	SENSORMOD object generated from the `s` model
`xhat`	SENSORMOD object with $\hat{x}_k$, P_k (and $\hat{y}_k$)
`shat`	SENSORMOD object, same as `s` except for that x is estimated

Table 5.13: SENSORMOD *estimation methods*

METHOD	DESCRIPTION
`ls`	`[xhat,shat]=ls(s,y);` computes the least squares estimate $\hat{x}^{LS}$ $\arg\min(\mathbf{y}-\mathbf{H}x)^T(\mathbf{y}-\mathbf{H}x))$ using $H = dh(x)/dx$ evaluated at x as given in `s.x`.
`wls`	`[xhat,shat]=wls(s,y);` computes the weighted least squares estimate $\hat{x}^{WLS}$ $\arg\min(\mathbf{y}-\mathbf{H}x)^T R^{-1}(\mathbf{y}-\mathbf{H}x))$ using $H = dh(x)/dx$ evaluated at x as given in `s.x`.
`ml`	`[xhat,shat,res]=ml(s,y);` computes the ML/NLS parameter estimate in the sensormod object `s` from `y` $\hat{x}^{ML}$ $\arg\min -\log p_e(\mathbf{y}-h(x))$ using NLS in `nls.m`
`calibrate`	`shat=calibrate(s,y,Property1,Value1,...);` computes the NLS parameter estimate $\hat{\theta}$ in `s` from measurements in `y`. A signal object of the target position can be obtained by `xhat=sig(shat)`.
`estimate`	`[shat,res]=estimate(s,y,property1,value1,...)` computes the joint parameter estimate $\hat{x}, \hat{\theta}$ in `s` from measurements in `y`. A subset of x, θ can be masked out for estimation. A signal object of the target position can be obtained by `xhat=sig(shat)`.

The function computes

$$\hat{x}_k = \arg\min_x V(x) = \arg\min(y_k - H_k x)^T \big(\mathrm{Cov}(e_k)\big)^{-1}(y_k - H_k x).$$

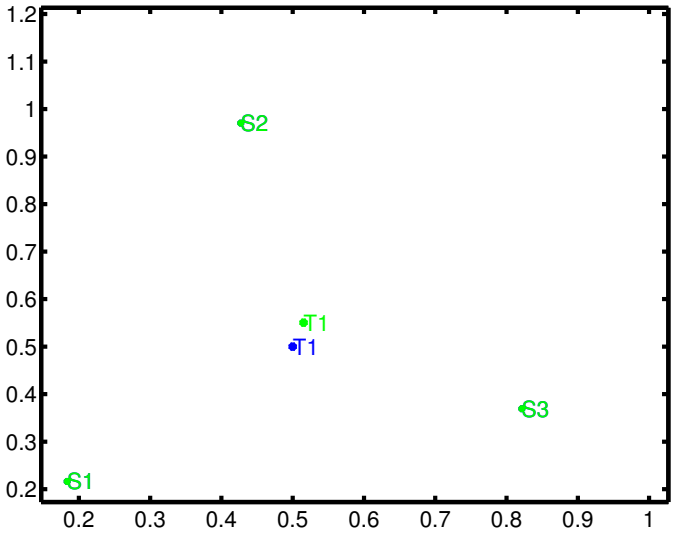
For nonlinear functions `s.h`, `numgrad` is used to linearize $H = dh(x)/dx$ around `s.x0`, which then becomes a crucial parameter. Why two different output objects?

- `xhat` is a `sig` object, corresponding to $\hat{x}_k$. This is useful for comparing the time independent estimates from LS and WLS to the state estimate from filtering methods.
- `shat` is computed only if $N = 1$, in case $\hat{x}_1$ is used as the estimate of target position in the SENSORMOD object, which is otherwise identical to the input object. This is useful for illustrating the estimation result in the sensor network.

Generate a TOA network with three sensors, simulate one measurement, and estimate the position of the target with WLS.

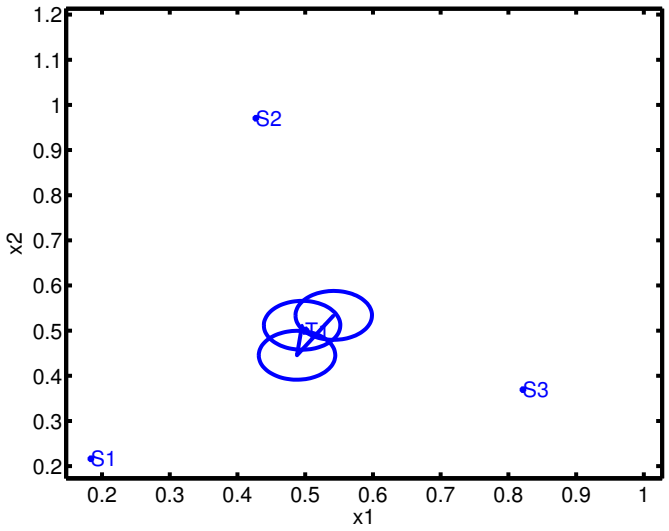
```
s=exsensor('toa',3,1);
s.pe=0.001*eye(3);
s.x0=[0.5 0.5];
y1=simulate(s,1)
```

```
SIG object with discrete time (fs = 1) stochastic state space data (no
    input)
  Sizes:       N = 1,  ny = 3, nx = 2
  MC is set to: 30
  #MC samples:  0
[xhat,shat]=wls(s,y1); shat % Note x0 distribution
SENSORMOD object: TOA
       / sqrt((x(1,:)-th(1)).^2+(x(2,:)-th(2)).^2) \
   y = | sqrt((x(1,:)-th(3)).^2+(x(2,:)-th(4)).^2) | + e
       \ sqrt((x(1,:)-th(5)).^2+(x(2,:)-th(6)).^2) /
   x0'  = [0.52,0.55]   + N(0,[0.0007,1.3e-06;1.3e-06,0.00064])
   th'  = [0.18,0.22,0.43,0.97,0.82,0.37]
  States:  x1     x2
  Outputs: y1     y2     y3
plot(s,shat)                % overlaid sensor network
```



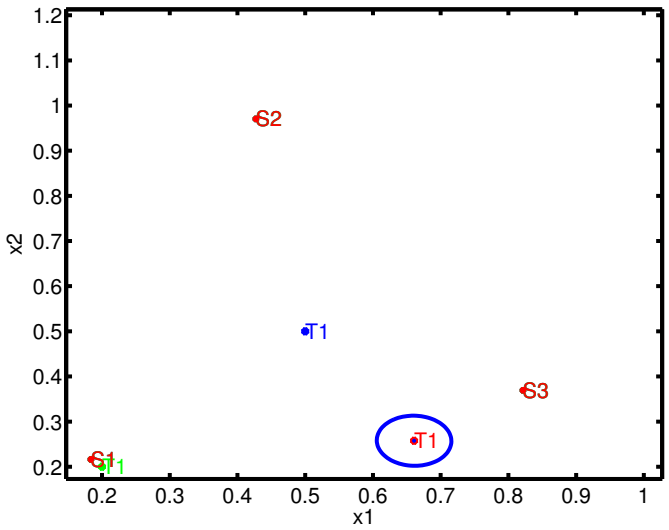
Simulate three measurements, and estimate the target for each one of them, and compare the estimates on the sensor network.

```
y3=simulate(s,1:3);          % Three measurements
[xhat,shat]=wls(s,y3);       % Three estimates
plot(s)
hold on
xplot2(xhat,'conf',90)
```



Now, we kind of cheated, since the WLS used an H obtained by linearizing around the true x! It is more fair to use an perturbed initial position to WLS.

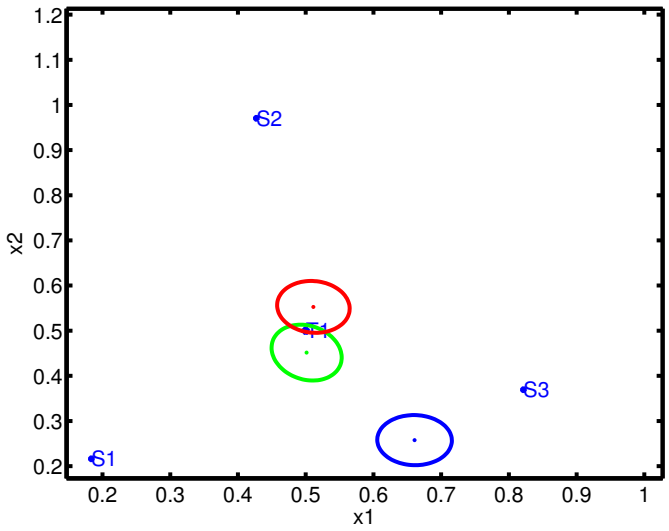
```
sinit=s;
sinit.x0=[0.2 0.2];          % Perturbed inital value for H
[xhat1,shat1]=wls(sinit,y1);
plot(s,sinit,shat1)
hold on
xplot2(xhat1,'conf',90)
```



Now, it does not work so well.

However, the WLS method can be called iteratively, and hopefully it will get closer to the best solution for each iteration.

```
[xhat2,shat2]=wls(shat1,y1);
[xhat3,shat3]=wls(shat2,y1);
plot(s)
hold on
xplot2(xhat1,xhat2,xhat3,'conf',90)
```



As seen, the estimate converges in a few iterations. This is basically how the ML method is solved.

5.9.2 Method ML

Usage:

```
[xhat,shat,res]=ml(s,y);
```

where the arguments are explained in the table below.

`s`	SENSORMOD object
`y`	SENSORMOD object generated from the `s` model
`xhat`	SENSORMOD object with $\hat{x}_k$, P_k (and $\hat{y}_k$)
`shat`	SENSORMOD object, same as `s` except for that x is estimated
`res`	details from the NLS algorithm in NLS

The function can be seen as an iterative WLS algorithm, initialized with $x^{(0)}$ given by `s.x0`:

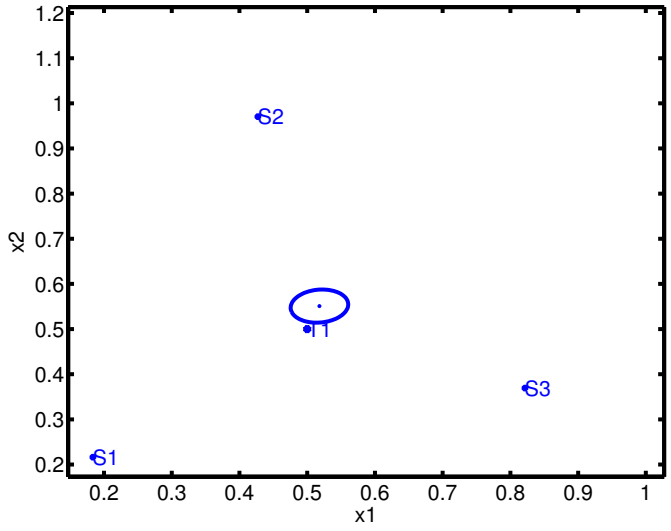
$$\hat{x}_k^{(i+1)} = \arg\min_x V(x) = \arg\min (y_k - H_k^{(i)} x)^T \big(\mathrm{Cov}(e_k)\big)^{-1} (y_k - H_k^{(i)} x),$$

$$H_k^{(i)} = \left. \frac{dh(x)}{dx} \right|_{x=\hat{x}_k^{(i+1)}}.$$

However, this is a rather simplified view of what the NLS function really performs. It is based on a Gauss-Newton algorithm with all kind of tricks to speed up convergence and improve robustness.

As an illustration, the same example with incorrect initialization as above, is here revisited, using the ML method instead of WLS.

```
[xhatml,shatml]=ml(sinit,y1);
plot(s)
hold on
xplot2(xhatml,'conf',90)
```



5.9.3 Methods CALIBRATE and ESTIMATE

Usage:

```
shat=calibrate(s,y);
[shat,res]=estimate(s,y);
```

where the arguments are explained in the table below.

`s`	SENSORMOD object
`y`	SENSORMOD object generated from the `s` model
`shat`	SENSORMOD object, same as `s` except for that x is estimated
`res`	details from the NLS algorithm in NLS

Note that no SIG object is outputted here, in contrast to WLS and ML. The reasons are the following:

- WLS and ML are methods of the SENSORMOD class. One estimate $\hat{x}_k$ is generated for each measurement y_k, and also it makes sense to predict $\hat{y}_k$ using the prior on the target position. That is, the SIG object is non-trivial.

- CALIBRATE and ESTIMATE are methods of the more general NL class. The intention here is to estimate the parameters θ and *initial conditions* x_0. That is, it does not make sense in general to define a SIG object. However, for the SENSORMOD class, it might be handy to get a SIG object in the same way as for the WLS and ML methods. The remedy is to use `xhat=sig(shat);`, that simply makes a SIG object of the state x (assuming that $y = x$!).

Similar to the WLS method, these functions computes a NLS solution using the NLS m-file. The difference is what is considered as the parameter in the model

$$y = h(x; \theta) + e.$$

- In WLS, x is the parameter:

$$\hat{x} = \arg\min_x V(x) = \arg\min_x (y - h(x;\theta))^T R^{-1} (y - h(x;\theta)).$$

- In CALIBRATE, θ is the parameter.

$$\hat{\theta} = \arg\min_\theta V(\theta) = \arg\min_\theta (y - h(x;\theta))^T R^{-1} (y - h(x;\theta)).$$

- In ESTIMATE, any combination of elements in x and θ is considered to be the parameter.

$$\begin{aligned}\widehat{(\theta(i_\theta), x(j_x))} &= \arg\min_{\theta(i_\theta), x(j_x)} V(\theta(i_\theta), x(j_x)) \\ &= \arg\min_{\theta(i_\theta), x(j_x)} (y - h(x(j_x); \theta(i_\theta))^T R^{-1} (y - h(x(j_x); \theta(i_\theta)).\end{aligned}$$

  The index pair i_θ and j_x are defined as binary mask vectors. Both ML and CALIBRATE can be seen as special cases of ESTIMATE where i_θ and j_x are all zeros, respectively.

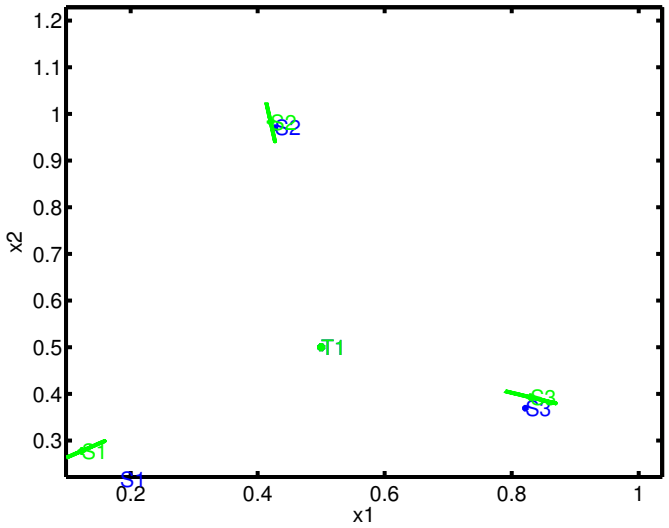
The typical call to ESTIMATE is as follows.

```
shat=estimate(sinit,y3,'thmask',thmask,'x0mask',x0mask)
```

where `x0mask` is a binary vector of the same dimension as x, and analogously for `thmask`. All other property-value pairs are passed on the the NLS function.

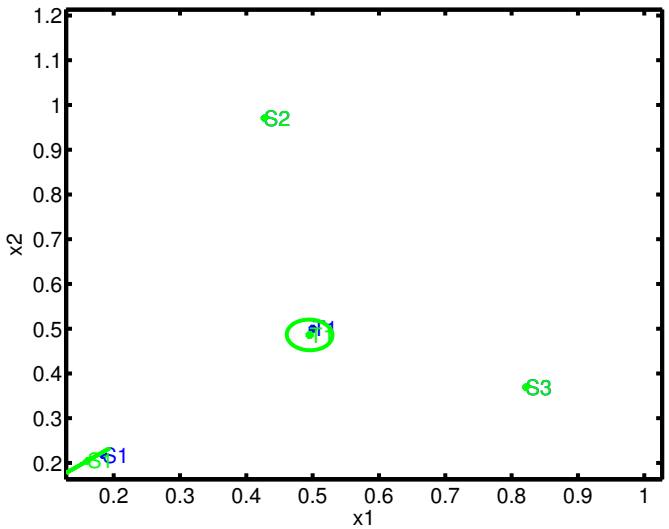
To illustrate CALIBRATE, we perturb the sensor positions randomly, and try to estimate them from the data simulated using their actual positions. The PLOT allows to plot confidence ellipsoids.

```
sinit=s;
sinit.th=sinit.th+0.04*randn(6,1);
shatcal=calibrate(sinit,y3);
plot(s,shatcal,'conf',90)   % Sensor positions estimated
```



In the next example, we assume that the position of the second and third sensors are known (anchor nodes), and we estimate the first sensor position and target position jointly. To make it more challenging, we first perturb their positions.

```
sinit=s;
sinit.th(1:2)=sinit.th(1:2)+0.04*randn(2,1);  % Perturb first sensor
sinit.x0=[0.4 0.4];                           % Perturb target
shat=estimate(sinit,y3,'thmask',[1 1 0 0 0 0],'x0mask',[1 1]);
plot(s,shat,'conf',90)   % Target and first sensor position estimated
```

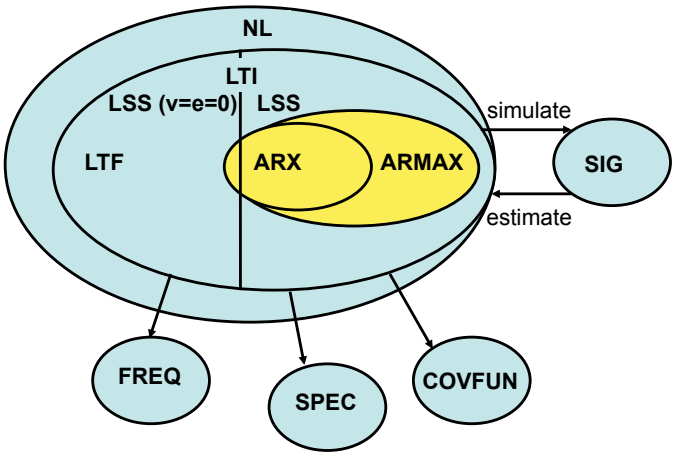


6

The NL Object

6.1 Definition of the NL Object

The nonlinear (NL) model object extends the models of the LTI class to both time-varying and nonlinear models.



Any LTI object can thus be converted to a NL object, but the reverse operation is not possible generally. However, a local linearized LTI model can be constructed once an operating point is selected.

The general definition of the NL model is in continuous time

$$\begin{aligned} \dot{x}(t) &= f(t, x(t), u(t); \theta) + v(t), \\ y(t_k) &= h(t_k, x(t_k), u(t_k); \theta) + e(t_k), \\ x(0) &= x_0. \end{aligned}$$

and in discrete time

$$\begin{aligned} x_{k+1} &= f(k, x_k, u_k; \theta) + v_k, \\ y_k &= h(k, x_k, u_k, e_k; \theta), \\ x(0) &= x_0. \end{aligned}$$

The involved signals and functions are:

- x denotes the state vector. t is time, and t_k denotes the sampling times that are monotonously increasing. For discrete time models, k refers to time kT, where T is the sampling interval.
- u is a known (control) input signal.
- v is an unknown stochastic input signal specified with its probability density function $p_v(v)$.
- e is a stochastic measurement noise specified with its probability density function $p_e(e)$.
- x_0 is the known or unknown initial state. In the latter case, it may be considered as a stochastic variable specified with its probability density function $p_0(x_0)$.
- θ contains the unknown parameters in the model. There might be prior information available, characterized with its mean and covariance.

For deterministic systems, when v and e are not present above, these model definitions are quite general. The only restriction from a general stochastic nonlinear model is that both process noise v and measurement noise e have to be additive.

The constructor `m=nl(f,h,nn)` has three mandatory arguments:

- The argument `f` defines the dynamics and is entered in one of the following ways:
  - A string, with syntax `m=nl(f,h,nn);`. Example:
    ```
    f='-th*x^2';
    ```
  - An inline function, with the same syntax `m=nl(f,h,nn);`. Example:

```
f=inline('-x^2','t','x','u','th');
```

– An M-file. Example:

```
function f=fun(t,x,u,th)
f=-th*x^2;
```

This m-file can be used in the constructor either as a string with the name, or as a function handle.

```
m=nl('fun',h,nn);
m=nl(@fun,h,nn);
Here, \sfc{feval} is used internally, so the function handle ➥
     is to
prefer for speed reasons.
```

It is important to use the standard model parameter names `t, x, u, th`. For inline functions and M-files, the number of arguments must be all these four even if some of them are not used, and the order of the arguments must follow this convention.

- `h` is defined analogously to `f` above.
- `nn=[nx,nu,ny,nth]` denotes the orders of the input parameters. These must be consistent with the entered `f` and `h`. This apparantly trivial information must be provided by the user, since it is hard to unambigiously interpret all combinations of input dimensions that are possible otherwise. All other tests are done by the constructor, which calls both functions `f` and `h` with zero inputs of appropriate dimensions according to `nn`, and validates the dimensions of the returned outputs.

All other parameters are set separately:

- `pv`, `pe`, and `px0` are distributions for the process noise, measurement noise and initial state, respectively. All of these are entered as objects in the `pdfclass`, or as covariance matrices when a Gaussian distribution is assumed.
- `th` and `P` are the fields for the parameter vector and optional covariance matrix. The latter option is used to represent uncertain systems. Only the second order property of model uncertainty is currently supported for NL objects, in contrast to the LTI objects SS and TF.
- `fs` is similarly to the LTI objects the sampling frequency, where the convention is that `fs=NaN` means continuous time systems (which is set by default). All NL objects are set to continuous time models in the constructor, and the user has to specify a numeric value of `fs` after construction if a discrete model is wanted.

Table 6.1: `NL methods`

METHOD	DESCRIPTION
`arrayread`	Used to pick out sub-systems by indexing. Ex: `m(2,:)` picks out the dynamics from all inpus to output number 2. Only the output dimension can be decreased for NL objects, as opposed to LTI objects.
`display`	Returns an ascii formatted version of the NL model
`estimate`	Estimates/calibrates the parameters in an NL system using data
`simulate`	Simulates the NL system using `ode45` in continuous time, and a straightforward for loop in discrete time.
`nl2ss`	Returns a linearized state space model
`ekf`	Implements the extended Kalman filter for state estimation
`nltf`	Implements a class of Ricatti-free filters, where the unscented Kalman filter and extended Kalman filter are special cases
`pf`	implements the particle filter for state estimation
`crlb`	Computes the Cramer-Rao Lower Bound for state estimation

- `xlabel`, `thlabel`, `ulabel`, `ylabel`, and name are used to name the variables and the model, respectively. These names are inherited after simulation in the SIG object, for instance.

The methods of the NL object are listed in Table 6.1. The filtering methods are described in detail in Chapter 7, The most fundamental usage of the NL objects is illustrated with a couple of examples:

- The van der Pol system illustrates definition of a scond order continuous time nonlinear system with known initial state and parameters.

- Bouncing ball dynamics is used to illustrate an uncertain second-order continuous-time nonlinear system with an unknown parameter and with stochastic process noise and measurement noise. The NL object is fully annotated with signal names.

- NL objects as provided after conversion from LTI objects.

- A first-order discrete-time nonlinear model used in many papers on particle filtering.

Table 6.2: *Standard nonlinear models in* `exnl`

EX	DESCRIPTION
`ctcv2d`	Coordinated turn model, cartesian velocity, 2D, `Ts=opt1`
`ct, ctpv2d`	Coordinated turn model, polar velocity, 2D, `Ts=opt1`
`cv2d`	Cartesian velocity linear model in 2D, `Ts=opt1`
`pfex`	Classic particle filter 1D example used in many papers
`vdp`	Van der Pol system
`ball`	Model of a bouncing ball
`pendulum`	One-dimensional continuous time pendulum model
`pendulum2`	Two-dimensional continuous time pendulum model

6.2 Generating Standard Nonlinear Models

There are some examples of nonlinear models to play around with. These are accessed by `exnl`, with syntax

```
s=exnl(ex,opt)
```

Table 6.2 summarizes some of the options. `exnl('list')` gives a cell array with all current options.

6.3 Generating Standard Motion Models

There are many pre-defined classes of motion models easily accessible in the function `exmotion`, with syntax

```
s=exmotion(ex,opt)
```

Table 6.3 summarizes some of the options.
`exmotion('list')` gives a cell array with all current options.

6.4 Utility Methods

The SENSORMOD utility methods are summarized in Table 6.4.

6.5 Object Modification Methods

See Table 6.5.

A coordinated turn motion model is below created and merged with a radar sensor using the `addsensor` method.

```
m=exmotion('ctcv2d');  % Motion model without sensor
s=exsensor('radar',1)  % Radar sensor
SENSORMOD object: RADAR
       / sqrt((x(1,:)-th(1)).^2+(x(2,:)-th(2)).^2) \
```

Table 6.3: *Standard motion models in* `exmotion`

EX	DESCRIPTION
`ctcv2d`	Coordinated turn model, cartesian velocity, 2D, `Ts=opt`
`ct, ctpv2d`	Coordinated turn model, polar velocity, 2D, `Ts=opt`
`ctpva2d`	Coordinated turn model, polar velocity, acc state, 2D, `s=opt`
`cv2d`	Cartesian velocity linear model in 2D, `Ts=opt`
`imu2d`	Dead-reckoning of acceleration and yaw rate, `Ts=opt`
`imukin2d`	Two-dimensional inertial model with a_X, a_Y and ω_X as inputs
`imukin2dbias`	As `imukin2d` but with 3 bias states for the inertial measurements
`imukin3d`	Three-dimensional inertial model with a and ω as the 6D input
`imukin3dbias`	As `imukin3d` but with 6 bias states for the inertial measurements.

Table 6.4: NL *utility methods*

METHOD	DESCRIPTION
`arrayread`	`mji=arrayread(m,j,i)`, or simpler `mji=m(j:i)` is used to pick out sub-systems by indexing
`simulate`	`y=simulate(s)` simulates a sensor model 1. y=simulate(s,x) gives $z = s(t, x)$ at times t and state s.x0. 2. y=simulate(s,x), gives $z = s(t, x)$ at times x.t and state x.x.
`display`	`display(s1,s2,...)` returns an ascii formatted version of the NL model
`nl2lss`	`[mout,zout]=nl2ss(m,z)` returns a linearized model $x+ = Fx + Gv$ and $\mathbf{y} = \mathbf{H}x + \mathbf{e}$ using $F = df(x)/dx$, $G = df(x)/du$ and $H = dh(x)/dx$ evaluated at x as given in `s.x`.

Table 6.5: NL *object modification methods*

METHOD	DESCRIPTION
`addsensor`	`ms=addsensor(m,s,Property1,Value1,...)` adds (another) sensor to the object
`removesensor`	`ms=removesensor(m,ind)` removes the sensors numbered `ind` from the object

```
   y = \ atan2(x(2,:)-th(2),x(1,:)-th(1))          / + e
   x0'  = [0.56,0.59]
   th'  = [0.35,0.15]
  States:  x1     x2
  Outputs: Range     Bearing
ms=addsensor(m,s);     % Motion model with one radar
mss=addsensor(ms,s);   % Motion model with two radars
```

Sensors can be added and removed using `addsensor` and `removesensor` methods. Below, the DOA sensor is first removed from the model created above, and then added again.

```
m=removesensor(s,2)
NL object
   y = [sqrt((x(1,:)-th(1)).^2+(x(2,:)-th(2)).^2)] + N(0,0.01)
   x0'  = [0.56,0.59]
   th'  = [0.35,0.15]
  States:  x1     x2
  Outputs: Range
  Param.: th1     th2
s=exsensor('doa');
m2=addsensor(m,s)
NL object: Motion model:  Sensor model: DOA
       / sqrt((x(1,:)-th(1)).^2+(x(2,:)-th(2)).^2) \
   y = \ atan2(x(2,:)-th(2),x(1,:)-th(1))          / + e
   x0'  = [0.56,0.59]
   th'  = [0.028,0.99]
  States:  x1     x2
  Outputs: Range     y1
  Param.: th1     th2
```

6.6 Examples

A note 2013: the continuous time simulation does not work. For that reason, all nonlinear models are discretized below.

6.6.1 The van der Pol System

First, a nonlinear dynamic system known as the van der Pol equations is defined as an NL object with two states, where the output is the same as the state. This is example is available as a demo `m=exnl('vdp')`.

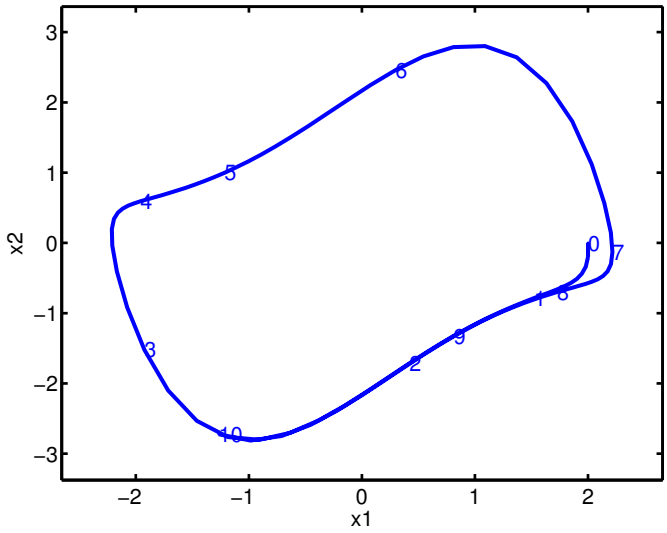
```
f='[x(2);(1-x(1)^2)*x(2)-x(1)]';
h='x';
m=nl(f,h,[2 0 2 0]);
NL warning: Try to vectorize f for increased speed
Hint: NL constructor: size of f not consistent with nn
m.name='Van der Pol system';
m.x0=[2;0];
m
NL object: Van der Pol system
         / x(2)                \
 dx/dt = \ (1-x(1)^2)*x(2)-x(1) /
   y = x
   x0'  = [2,0]
  States:  x1     x2
  Outputs: y1     y2
y=simulate(c2d(m,10),10);
```

```
plot(y)
```

The constructor checks if the sizes in `nn` are consistent with the functions specified in `f` and `h`. If not, an error is given. The constructor also checks if `f` and `h` allow vectorized computations. Since this is not the case here, a warning is given. A try/catch approach is used whenever `f` and `h` are to be evalutated, where first a vectorized call is tried, followed by a for loop if this fails.

The strange behavior of the van der Pol equations results in a periodic state trajectory, which can be visualized as follows.

```
xplot2(y)
```



Now, consider again the definition of the model. The constructor complained about the input format for `f`. The definition below allows for vectorized evaluations, which should be more efficient

```
f='[x(2,:);(1-x(1,:).^2).*x(2,:)-x(1,:)]';
h='x';
m2=nl(f,h,[2 0 2 0]);
m2.x0=[2;0];
tic, simulate(c2d(m,10),10); toc
Elapsed time is 0.109939 seconds.
tic, simulate(c2d(m2,10),10); toc
Elapsed time is 0.099563 seconds.
```

Unfortunately, the vectorized model gives longer simulation time in this case low-dimensional case, but, generally, it should be more efficient.

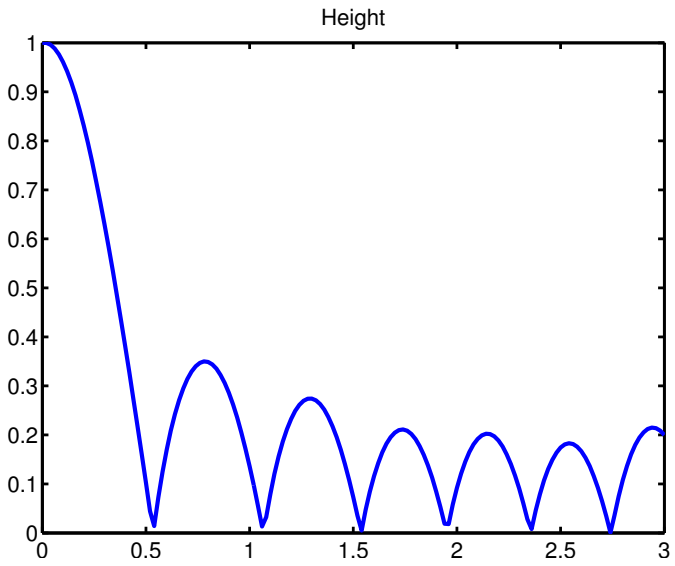
Another reason to use the notation is for the genfilt method, where an implicit state augmentation forces the user to specify state indeces explicitly. For this purpose, it is also important to avoid the single colon operator and

to replace any `end` with `nx`, that is use `x(1:nx,:)` rather than `x(1:end,:)` or `x(:,:)`.

6.6.2 Bouncing Ball

The following example simulates the height of a bouncing ball with completely elastic bounce and an air drag. A trick to avoid the discontinuity in speed at the bounce is to admit a fictive negative height in the state vector, and letting the output relation take care of the sign. The following model can be used (available as the demo `m=exnl('ball')`).

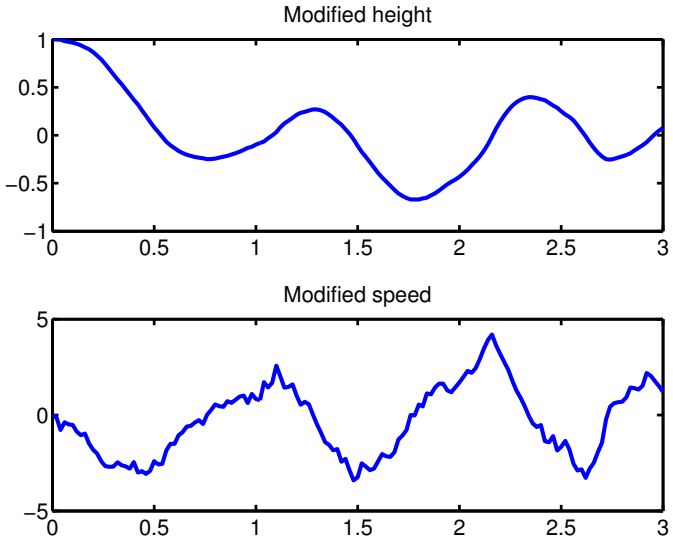
```
f='[x(2,:);-th(1)*x(2,:).^2.*sign(x(2,:))-9.8.*sign(x(1,:))]';
h='abs(x(1,:))';
m=nl(f,h,[2 0 1 1]);
m.x0=[1;0];
m.th=1;
m.name='Bouncing ball';
m.xlabel={'Modified height','Modified speed'};
m.ylabel='Height';
m.thlabel='Air drag';
m
NL object: Bouncing ball
          / x(2,:)                                                \
 dx/dt = \ -th(1)*x(2,:).^2.*sign(x(2,:))-9.8.*sign(x(1,:)) /
   y = abs(x(1,:))
   x0'  = [1,0]
   th'  = 1
  States:  Modified height     Modified speed
  Outputs: Height
  Param.: Air drag
m=c2d(m,50);
y=simulate(m,0:0.02:3);
plot(y)
```



The advantage of representing air drag with a parameter rather than just typing in its value will be illustrated later on in the context of uncertain systems.

The model is modified below to a stochastic model corresponding to wind disturbances on the speed and measurement error.

```
m.pv=diag([0 0.1]);
m.pe=0.1;
m
NL object: Bouncing ball
         / x(1,:)+0.02*(x(2,:)) ➟
                                                                  \
x[k+1] = \ ➟
     x(2,:)+0.02*(-th(1)*x(2,:).^2.*sign(x(2,:))-9.8.*sign(x(1,:))) / + v
   y = abs(x(1,:)) + N(0,0.1)
   x0'  = [1,0]
   th'  = 1
  States:  Modified height     Modified speed
  Outputs: Height
  Param.: th1
y=simulate(m,0:0.02:3);
xplot(y)
```

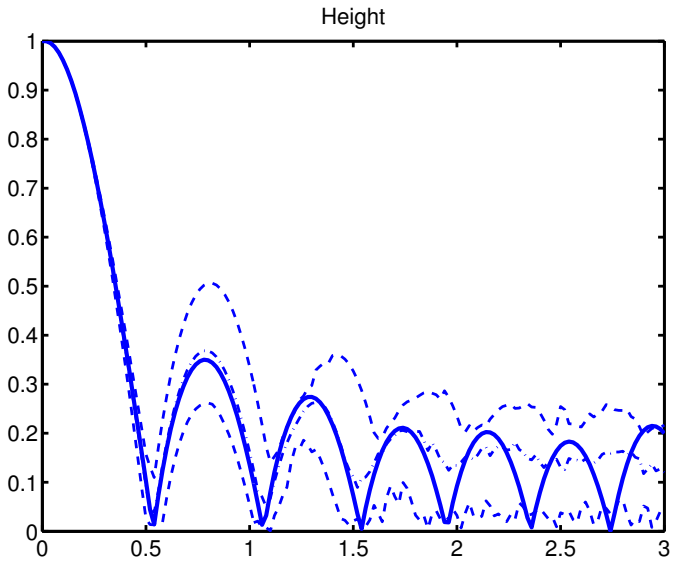


Note that the process and measurement noise are printed out symbolically by the display function.

The model will next be defined to be uncertain due to unknown air drag coefficient. This is very simple to do because this coefficient is defined symbolically rather than numerically in the model. First, the process and measurement noises are removed.

```
m.pv=[];
m.pe=0;
m.P=0.1;
```

```
m
NL object: Bouncing ball
         / x(1,:)+0.02*(x(2,:)) ➠
                                                                  \
x[k+1] = \ ➠
     x(2,:)+0.02*(-th(1)*x(2,:).^2.*sign(x(2,:))-9.8.*sign(x(1,:))) /
   y = abs(x(1,:)) + N(0,0)
   x0'  = [1,0]
   th'  = 1
   std = [0.32]
  States:  Modified height     Modified speed
  Outputs: Height
  Param.: th1
y=simulate(m,0:0.02:3);
plot(y,'conf',90)
```



The confidence bound is found by simulating a large number of systems with different air drag coefficient according to the specified distribution (which has to be Gaussian for NL objects). Note that the display function shows the standard deviation of each parameter (the full covariance matrix is used internally).

6.6.3 Conversions from LTI Objects

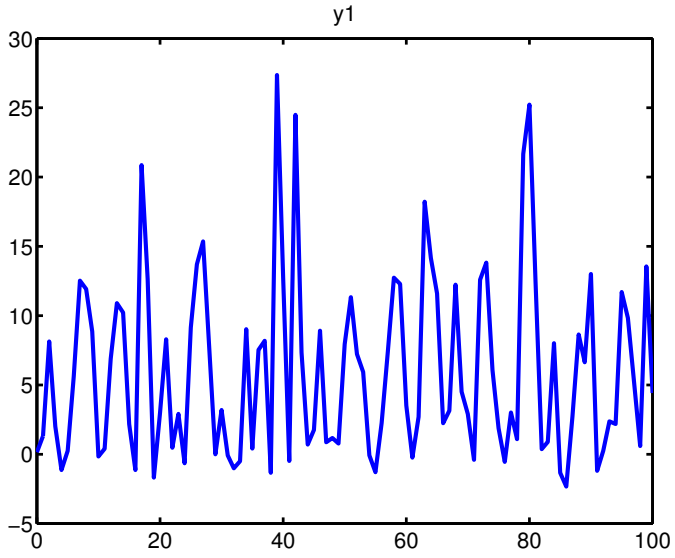
Since the class of LTI models is a special case of NL models, any LTI object (SS or TF) can be converted to an NL object:

```
nl(rand(lss([2 1 0 1])))
NL object
 dx/dt = [-1.00168940004469 -0.574608056403271;1 ➠
      0]*x(1:2,:)+[1;0]*u(1:1,:) + v
   y = [0.89043079103883 2.12068248707517]*x(1:2,:)+1*u(1:1,:) + N(0,0)
   x0'  = [0,0]
```

6.6.4 A Benchmark Example for (Particle) Filtering

The following model has been used extensively for illustrating the particle filter, and compare it to extended and unscented Kalman filters. It is here used to exemplify a stochastic discrete time system (also avaiable as `m=exnl('pfex')`).

```
f='x/2+25*x./(1+x.^2)+8*cos(t)';
h='x.^2/20';
m=nl(f,h,[1 0 1 0]);
m.fs=1;
m.px0=5;  % P0=cov(x0)=5
m.pv=10;  % Q=cov(v)=10
m.pe=1;   % R=cov(e)=1
m
NL object
x[k+1] = x/2+25*x./(1+x.^2)+8*cos(t) + N(0,10)
   y = x.^2/20 + N(0,1)
   x0'  = 0   + N(0,5)
  States:  x1
  Outputs: y1
y=simulate(m,100);
plot(y)
```



The important thing to remember is that all NL objects are assumed to be continuous time models when defined. Once `m` is set, the meaning of `f` changes from continuous to discrete time.

7

Filtering

Filtering, or state estimation, is one of the key tools in model-based control and signal processing applications. In control theory, it is required for state feedback control laws. Target tracking and navigation are technical drivers in the signal processing community. It all started in 1960 by the seminal paper by Kalman, where the Kalman filter for linear state space models was first presented. Smith later in the sixties derived an approximation for nonlinear state space models now known as the extended Kalman filter. Van Merwe with coauthors presented an improvement in the late nineties named the unscented Kalman filter, which improves over the EKF in many cases where the nonlinearity is more severe. The most flexible filter used today is the particle filter (PF), presented by Gordon with coauthors in 1993. The PF can handle any degree of nonlinearity, state constraints and non-Gaussian noise in an optimal way, at the cost of large computational complexity.

7.1 Kalman Filtering for LSS Models

7.1.1 Algorithm

For a linear state space model, the Kalman filter (KF) is defined by

$$
\begin{aligned}
\hat{x}_{k+1|k} &= F_k \hat{x}_{k|k} + G_{u,k} u_k \\
P_{k+1|k} &= F_k P_{k|k} F_k^T + G_{v,k} Q_k G_{v,k}^T \\
\hat{x}_{k|k} &= \hat{x}_{k|k-1} + P_{k|k-1} H_k^T (H_k P_{k|k-1} H_k^T + R_k)^{-1} (y_k - H_k \hat{x}_{k|k-1} - D_{u,k} u_k) \\
P_{k|k} &= P_{k|k-1} - P_{k|k-1} H_k^T (H_k P_{k|k-1} H_k^T + R_k)^{-1} H_k P_{k|k-1}.
\end{aligned}
$$

The inputs to the KF are the LSS object and a SIG object containing the observations y_k and possible an input u_k, and the outputs are the state estimate $\hat{x}_{k|k}$ and its covariance matrix $P_{k|k}$. There is also a possibility to predict future states $\hat{x}_{k+m|k}, P_{k+m|k}$ with the m-step ahead predictor, or to compute the smoothed estimate $\hat{x}_{k|N}, P_{k|N}$ using the complete data sequence $y_{1:N}$.

7.1.2 Usage

Call the KF with

```
[x,V]=kalman(m,z,Property1,Value1,...)
```

The arguments are as follows:

- `m` is a LSS object defining the model matrices A, B, C, D, Q, R.
- `z` is a SIG object with measurements y and inputs u if applicable. The state field is not used by the KF.
- `x` is a SIG object with state estimates. `xhat=x.x` and signal estimate `yhat=x.y`.
- `V` is the normalized sum of squared innovations, which should be a sequence of `chi2dist(nx)` variables when the model is correct.

The optional parameters are summarized in the table below.

7.1.3 Examples

Target Tracking

A constant velocity model is one of the most used linear models in target tracking, and it is one demo model. In this initial example, the model is re-defined using first principles to illustrate how to build an LSS object for filtering.

First, generate model.

```
% Motion model
m=exlti('CV2D');  % Pre-defined model, or...
T=1;
A=[1 0 T 0; 0 1 0 T; 0 0 1 0; 0 0 0 1];
B=[T^2/2 0; 0 T^2/2; T 0; 0 T];
C=[1 0 0 0; 0 1 0 0];
R=0.01*eye(2);
m=lss(A,[],C,[],B*B',R,1/T);
m.xlabel={'X','Y','vX','vY'};
m.ylabel={'X','Y'};
m.name='Constant velocity motion model';
m
          / 1  0  1  0 \
          | 0  1  0  1 |
x[k+1] = | 0  0  1  0 | x[k] + v[k]
          \ 0  0  0  1 /
         /1  0  0  0\
```

Table 7.1: `lss.kalman`

PROPERTY	VALUE	DESCRIPTION
`alg`	`1,2,3,4`	Type of implementation: 1 stationary KF. 2, time-varying KF. 3, square root filter. 4, fixed interval KF smoother Rauch-Tung-Striebel. 5, sliding window KF, delivering xhat(t|y(t-k+1:t)), where k is the length of the sliding window.
`k`	`k>0 0`	Prediction horizon: 0 for filter (default), 1 for one-step ahead predictor, k>0 gives $\hat{x}_{t+k\|t}$ and $\hat{y}_{t+k\|t}$ for `alg=1,2`. In case `alg=5`, `k=L` is the size of the sliding window.
`P0`	`[]`	Initial covariance matrix. Scalar value scales identity matrix. Empty matrix gives a large identity matrix.
`x0`	`[]`	Initial state matrix. Empty matrix gives a zero vector.
`Q`	`[]`	Process noise covariance (overrides the value in `m.Q`). Scalar value scales `m.Q`.
`R`	`[]`	Measurement noise covariance (overrides the value in `m`). Scalar value scales `m.R`.

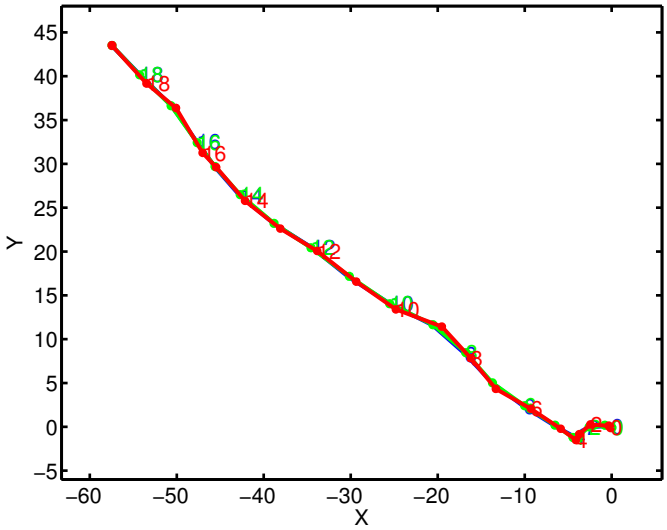
```
y[k] =  \0  1  0  0/  x[k] + e[k]
             /0.25          0         0.5          0\
            |    0       0.25          0         0.5 |
Q = Cov(v) = |  0.5          0          1           0 |
             \   0        0.5          0           1/
            / 0.01          0\
R = Cov(e) = \    0       0.01/
```

Then, simulate data.

```
z=simulate(m,20);
```

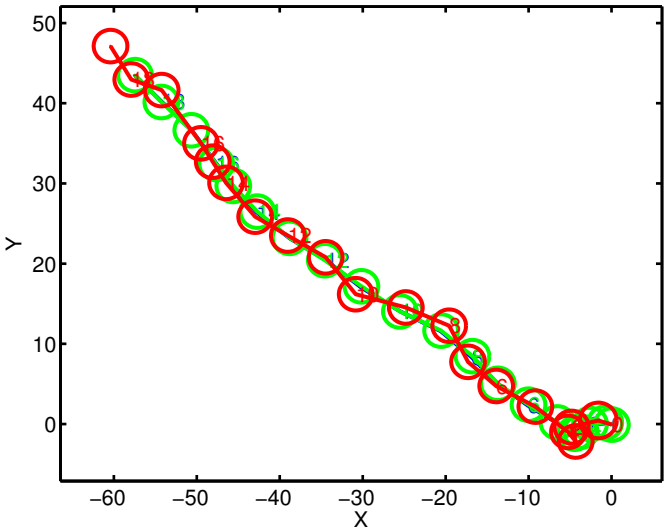
Now, various implementations of the Kalman filter for filtering are compared.

```
xhat10=kalman(m,z,'alg',1,'k',0);
xhat20=kalman(m,z,'alg',2,'k',0);
xhat40=kalman(m,z,'alg',4,'k',0);
xplot2(z,xhat20,xhat40,'conf',99,[1 2])
```



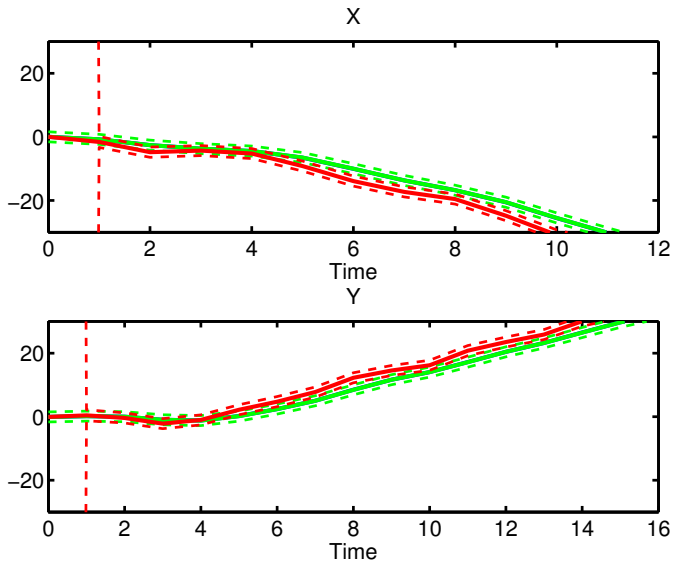
The SNR is good, so all estimated trajectories are very close to the simulated one, and the covariance ellipses are virtually invisible on this scale. The time-varying and stationary Kalman filter can be used for one-step ahead prediction.

```
xhat12=kalman(m,z,'alg',1,'k',1);
xhat22=kalman(m,z,'alg',2,'k',1);
xplot2(z,xhat12,xhat22,'conf',99,[1 2])
```



In this case, the uncertainty is visible. Also, the output can be predicted with confidence intervals.

```
plot(z,xhat12,xhat22,'conf',99,'Ylim',[-30 30])
```



In this plot, the initial transient phase is noticeable. First after two samples, the position can be accurately estimated.

The main difference to the KF is that EKF does not predict further in the future than one sample, and that smoothing is not implemented. Further, there is no square root filter implemented, and there is no such thing as a stationary EKF.

7.2 Extended Kalman Filtering for NL Objects

7.2.1 Algorithm

An NL object of a nonlinear time-varying model can be converted to an SS object by linearization around the current state estimate using `mss=nl2ss(mnl,xhat)`. This is the key idea in the extended Kalman filter, where the A and C matrices are replaced by the linearized model in the Riccati equation. For the state and measurement prediction, the nonlinear functions are used. In total, the EKF implements the following recursion (where some of the arguments

to f and h are dropped for simplicity):

$$
\begin{aligned}
\hat{x}_{k+1|k} &= f(\hat{x}_{k|k}) \\
P_{k+1|k} &= f'_x(\hat{x}_{k|k}) P_{k|k} (f'_x(\hat{x}_{k|k}))^T + f'_v(\hat{x}_{k|k}) Q_k (f'_x(\hat{x}_{k|k}))^T \\
S_k &= h'_x(\hat{x}_{k|k-1}) P_{k|k-1} (h'_x(\hat{x}_{k|k-1}))^T + h'_e(\hat{x}_{k|k-1}) R_k (h'_e(\hat{x}_{k|k-1}))^T \\
K_k &= P_{k|k-1} (h'_x(\hat{x}_{k|k-1}))^T S_k^{-1} \\
\varepsilon_k &= y_k - h(\hat{x}_{k|k-1}) \\
\hat{x}_{k|k} &= \hat{x}_{k|k-1} + K_k \varepsilon_k \\
P_{k|k} &= P_{k|k-1} - P_{k|k-1} (h'_x(\hat{x}_{k|k-1}))^T S_k^{-1} h'_x(\hat{x}_{k|k-1}) P_{k|k-1}.
\end{aligned}
$$

The EKF can be expected to perform well when the linearization error is small. Here small relates both to the state estimation error and the degree of nonlinearity in the model. As a rule of thumb, EKF works well the following cases:

- The model is almost linear.
- The SNR is high and the filter does converge. In such cases, the estimation error will be small, and the neglected rest term in a linearization becomes small.
- If either process or measurement noise are multimodel (many peaks), then EKF may work fine, but nevertheless perform worse than nonlinear filter approximations as the particle filter.

Design guidelines include the following useful tricks to mitigate linearization errors:

- Increase the state noise covariance Q to compensate for higher order nonlinearities in the state dynamic equation.
- Increase the measurement noise covariance R to compensate for higher order nonlinearities in the measurement equation.

7.2.2 Usage

The EKF is used very similarly to the KF. The EKF is called with

```
x=ekf(m,z,Property1,Value1,...)
```

The arguments are as follows:

- `m` is a NL object defining the model.
- `z` is a SIG object with measurements `y`, and inputs `u` if applicable. The state field is not used by the EKF, but is handy to have for evaluation purposes in subsequent plots.

Table 7.2: `nl.ekf`

PROPERTY	VALUE	DESCRIPTION
k	k>0 0	Prediction horizon: 0 for filter (default), 1 for one-step ahead predictor.
P0	[]	Initial covariance matrix. Scalar value scales identity matrix. Empty matrix gives a large identity matrix.
x0	[]	Initial state matrix. Empty matrix gives a zero vector.
Q	[]	Process noise covariance (overrides the value in m.Q). Scalar value scales m.Q.
R	[]	Measurement noise covariance (overrides the value in m). Scalar value scales m.R.

- x is a SIG object with state estimates. xhat=x.x and signal estimate yhat=x.y.

The optional parameters are summarized in Table 7.2.

The main difference to the KF is that EKF does not predict further in the future than one sample, and that smoothing is not implemented. Further, there is no square root filter implemented, and there is no such thing as a stationary EKF.

7.2.3 Examples

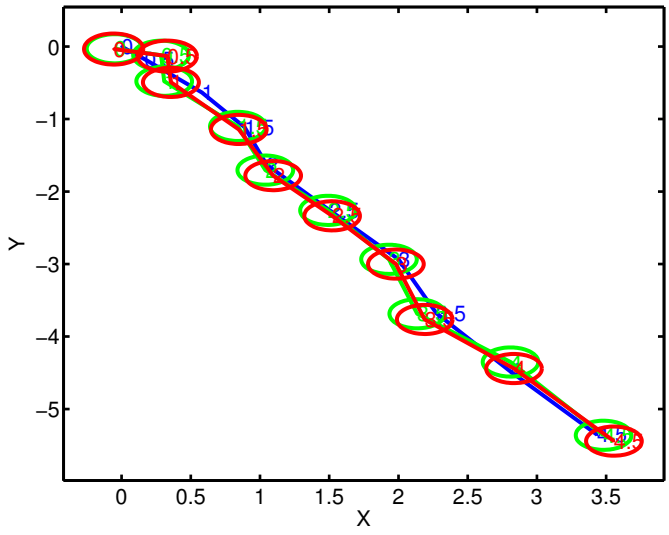
Target Tracking

Since LSS is a special case of NL objects, the Kalman filter is a kind of special case of the EKF method. To illustrate this, let us return to the previous tracking example. All LSS objects can be converted to an NL object.

```
mlss=exlti('cv2d')
          / 1   0   0.5     0 \
          | 0   1     0   0.5 |
x[k+1] = | 0   0     1     0 | x[k] + v[k]
          \ 0   0     0     1 /
         /1   0   0   0\
y[k] =   \0   1   0   0/  x[k] + e[k]
               /0.016           0       0.062           0\
              |      0       0.016           0       0.062 |
Q = Cov(v) = | 0.062           0        0.25           0 |
              \      0       0.062           0        0.25/
              / 0.01            0\
R = Cov(e) = \     0        0.01/
mnl=nl(mlss)
NL object: Constant velocity motion model
x[k+1] = [1 0 0.5 0;0 1 0 0.5;0 0 1 0;0 0 0 1]*x(1:4,:) + v
   y = [1 0 0 0;0 1 0 0]*x(1:4,:) + e
   x0'  = [0,0,0,0]
  States:  X      Y      vX      vY
  Outputs: X      Y
```

The model is simulated using the LSS method (the NL method should give the same result), and the KF is compared to the EKF.

```
z=simulate(mlss,10);
zhat1=kalman(mlss,z);
zhat2=ekf(mnl,z);
NL.EKF warning: px0 not defined, using a default value instead
xplot2(z,zhat1,zhat2,'conf',90)
```



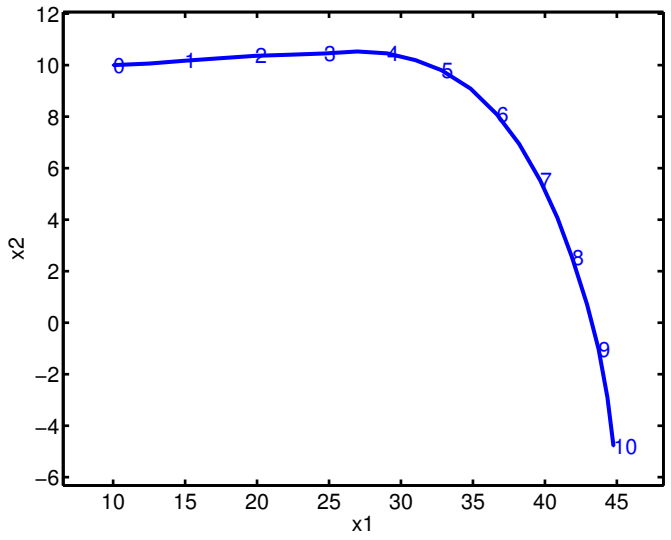
Except for some numerical differences, the results are comparable. A coordinated turn model is common in target tracking applications, where the target is known to turn smoothly along circular segments. There are various models available as demos. Here, a five-state coordinated turn (CT) model with polar velocity (PV) in two dimensions (2D) is used. Predefined alternatives include permutations with Cartesian velocity (CV) and three dimensions (3D).

```
m=exnl('ctpv2d')
NL object: Coordinated turn model with polar velocity
          / ➥
              x(1,:)+2*x(3,:)./(eps+x(5,:)).*sin((eps+x(5,:))*0.5/2).*cos(x(4,:)+x(5,:)*0.5/2) ➥
                    \
          | ➥
              x(2,:)+2*x(3,:)./(eps+x(5,:)).*sin((eps+x(5,:))*0.5/2).*sin(x(4,:)+(eps+x(5,:))*0.5/2) ➥
              |
x[k+1] = | x(3,:) ➥
                                                                                    ➥

     |
          | x(4,:)+x(5,:)*0.5 ➥
                                                                                    ➥
              |
          \ x(5,:) ➥
                                                                                    ➥
              /
```

```
+ v

      / sqrt(x(1,:).^2+x(2,:).^2) \
  y = \ atan2(x(2,:),x(1,:))      / + e
  x0'  = [10,10,5,0,0.1]    + ➠
       N(0,[10,0,0,0,0;0,10,0,0,0;0,0,1e+02,0,0;0,0,0,10,0;0,0,0,0,1])
States:  x1     x2     v     h     w
Outputs: R     phi
```

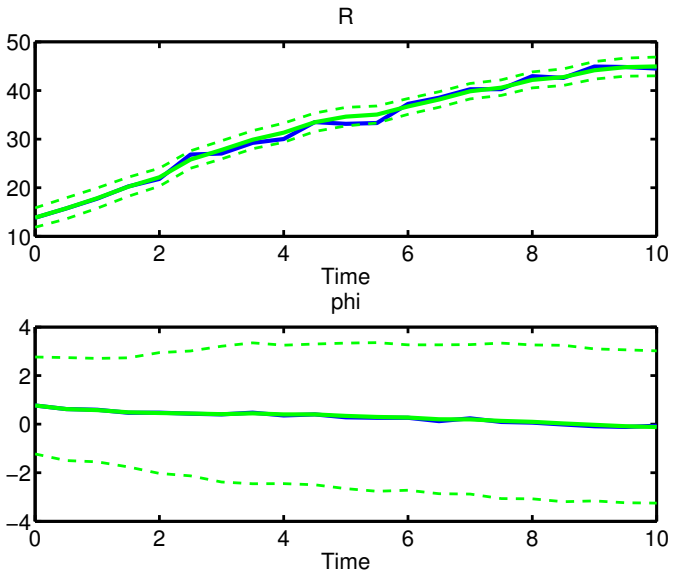
The measurement model assumes range and bearing sensors as in for instance a radar. These are nonlinear functions of the state vector. Also, the state dynamics is nonlinear. The model is simulated, then the first two states (`ind=[1 2]` is default) are plotted.

```
y=simulate(m,10);
xplot2(y,'conf',90)
```



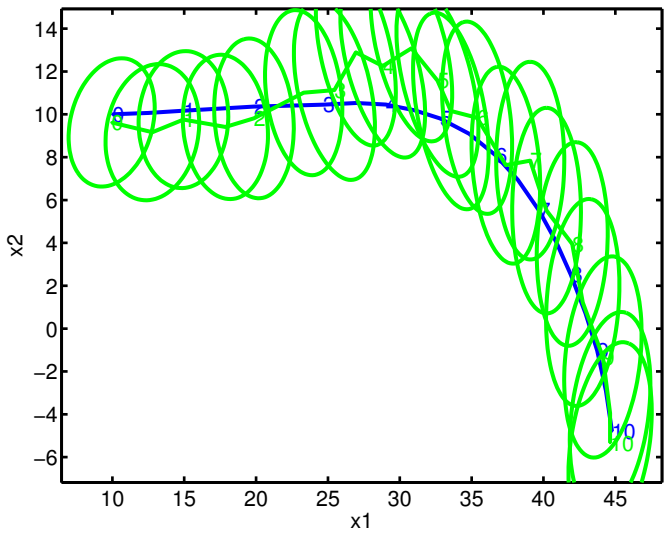
The EKF is applied, and the estimated output is compared to the measured output with a confidence interval.

```
xhat=ekf(m,y);
plot(y,xhat,'conf',99)
```



The position trajectory reveals that the signal to noise ratio is quite poor, and there are significant linearization errors (the confidence ellipsoids do not cover the simulated state).

```
xplot2(y,xhat,'conf',99.9,[1 2])
```



7.3 Particle Filtering for NL Objects

7.3.1 Algorithm

The general Bayesian solution to estimating the state in the nonlinear model

$$x_{k+1} = f(x_k) + v_k, \qquad v_k \sim p_{v_k}, \quad x_0 \sim p_{x_0}, \tag{7.1a}$$

$$y_k = h(x_k) + e_k, \qquad e_k \sim p_{e_k}. \tag{7.1b}$$

The particle filter (PF) approximates the infinite dimensional integrals by stochastic sampling techniques, which leads to a (perhaps surprisingly simple) numerical solution.

Choose a proposal distribution $q(x_{k+1}|x_{1:k}, y_{k+1})$, resampling strategy and the number of particles N.
Initialization: Generate $x_1^i \sim p_{x_0}, i = 1, \ldots, N$ and let $w_{1|0}^i = 1/N$. Each sample of the state vector is referred to as a *particle*. The general PF algorithm consists of the following recursion for $k = 1, 2, \ldots$.

1. *Measurement update:* For $i = 1, 2, \ldots, N$,

$$w_{k|k}^i = \frac{1}{c_k} w_{k|k-1}^i p(y_k|x_k^i), \tag{7.2a}$$

   where the normalization weight is given by

$$c_k = \sum_{i=1}^{N} w_{k|k-1}^i p(y_k|x_k^i) \tag{7.2b}$$

2. *Estimation:* The filtering density is approximated by $\hat{p}(x_{1:k}|y_{1:k}) = \sum_{i=1}^{N} w_{k|k}^i \delta(x_{1:k} - x_{1:k}^i)$ and the mean is approximated by $\hat{x}_{1:k} \approx \sum_{i=1}^{N} w_{k|k}^i x_{1:k}^i$

3. *Resampling:* Optionally at each time, take N samples with replacement from the set $\{x_{1:k}^i\}_{i=1}^N$ where the probability to take sample i is $w_{k|k}^i$ and let $w_{k|k}^i = 1/N$.

4. *Time update:* Generate predictions according to the proposal distribution

$$x_{k+1}^i \sim q(x_{k+1}|x_k^i, y_{k+1}) \tag{7.2c}$$

   and compensate for the importance weight

$$w_{k+1|k}^i = w_{k|k}^i \frac{p(x_{k+1}^i|x_k^i)}{q(x_{k+1}^i|x_k^i, y_{k+1})}, \tag{7.2d}$$

The pf method of the NL object implements the standard SIR filter. The principal code is given below:

```
y=z.y.';
u=z.u.';
xp=ones(Np,1)*m.x0.' + rand(m.px0,Np); % Initialization
for k=1:N;
% Time update
v=rand(m.pv,Np); % Random process noise
xp=m.f(k,xp,u(:,k),m.th).'+v; % State prediction
% Measurement update
yp=m.h(k,xp,u(k,:).',m.th).'; % Measurement prediction
w=pdf(m.pe,repmat(y(:,k).',Np,1)-yp); % Likelihood
xhat(k,:)=mean(repmat(w(:),1,Np).*xp); % Estimation
[xp,w]=resample(xp,w); % Resampling
xMC(:,k,:)=xp; % MC uncertainty repr.
end
zhat=sig(yp.',z.t,u.',xhat.',[],xMC);
```

The particle filter suffers from some divergence problems caused by sample impoverishment. In short, this implies that one or a few particles are contributing to the estimate, while all other ones have almost zero weight. Some mitigation tricks are useful to know:

- Medium SNR. The SIR PF usually works alright for medium signal-to-noise ratios (SNR). That is, the state noise and measurement noise are comparable in some diffuse measure.
- Low SNR. When the state noise is very small, the total state space is not explored satisfactorily by the particles, and some extra excitation needs to be injected to spread out the particles. Dithering (or jittering or roughening) is one way to robustify the PF in this case, and the trick is to increase the state noise in the PF model.
- High SNR. Using the dynamic state model as proposal density is a good idea generally, but it should be remembered that it is theoretically unsound when the signal to noise ratio is very high. What happens when the measurement noise is very small is that most or even all particles obtained after the time prediction step get zero weight from the likelihood function. In such cases, try to increase the measurement noise in the PF model.

As another user guideline, try out the PF on a subset of the complete measurement record. Start with a small number of particles (100 is the default value). Increase an order of magnitude and compare the results. One of the examples below illustrates how the result eventually will be consistent. Then, run the PF on the whole data set, after having extrapolated the computation time from the smaller subset. Generally, the PF is linear in both the number of particles and number of samples, which facilitates estimation of computation time.

7.3.2 Usage

The PF is called with

Table 7.3: `nl.pf`

PROPERTY	VALUE	DESCRIPTION
Np	Np>0 0	Number of particles
k	k=0,1	Prediction horizon: 0 for filter (default) 1 for one-step ahead predictor,
sampling		'simple' Standard algorithm 'systematic' 'residual' 'stratified'
animate	[],ind	Animate the states x(ind)

```
x=pf(m,z,Property1,Value1,...)
```

where the arguments are as follows:

- `m` is a NL object defining the model.
- `z` is a SIG object with measurements `y`, and inputs `u` if applicable. The state field is not used by the EKF.
- `x` is a SIG object with state estimates. `xhat=x.x` and signal estimate `yhat=x.y`.

The optional parameters are summarized in Table 7.3.

7.3.3 Examples

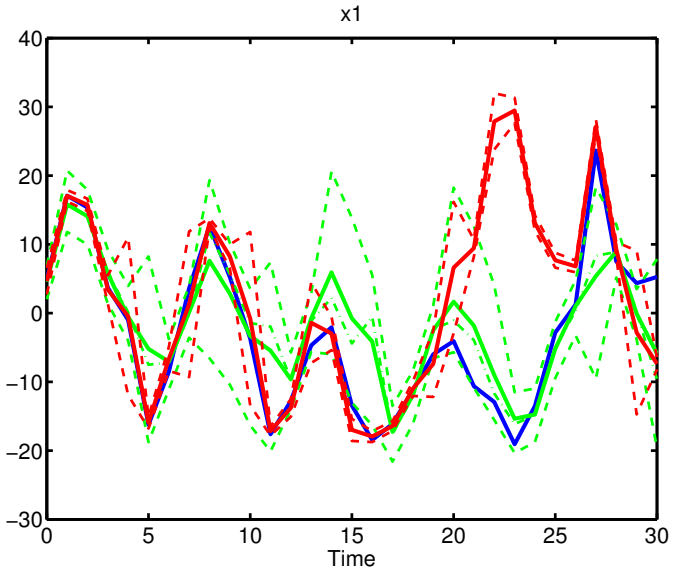
The Benchmark Example

The following dynamic system has been used in many publications to illustrate the particle filter.

```
m=exnl('pfex')
NL object
x[k+1] = x(1,:)/2+25*x(1,:)./(1+x(1,:).^2)+8*cos(t) + N(0,10)
   y = x(1,:).^2/20 + N(0,1)
   x0'  = 5   + N(0,5)
  States:  x1
  Outputs: y1
```

It was for instance used in the seminal paper by Neil Gordon in 1993. The PF is indeed much better than EKF as illustrated below, where the state estimate is plotted with confidence bounds.

```
z=simulate(m,30);
mpf=m;
mpf.pe=10*cov(m.pe);  % Some dithering required
zekf=ekf(m,z);
zpf=pf(mpf,z,'k',1);
xplot(z,zpf,zekf,'conf',90,'view','cont')
```



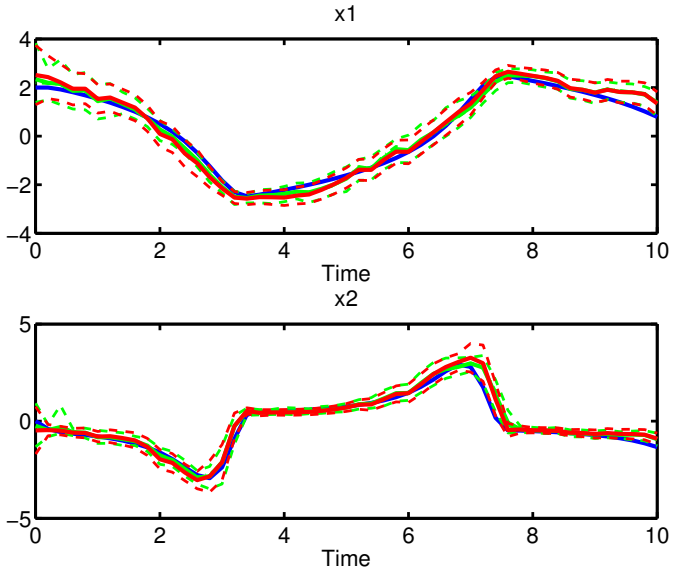
The van der Pol System

The van der Pol system is interesting from a dynamic system point of view. The van der Pol equation is discretized using Euler sampling with a sampling frequency of 5 Hz in the pre-defined example.

```
m=exnl('vdpdisc')
NL object: Discretized van der Pol system (Euler Ts=0.2)
          / x(1,:)+0.2*x(2,:)                              \
x[k+1] = \ x(2,:)+0.2*((1-x(1,:).^2).*x(2,:)-x(1,:)) / + v
   y = x + e
   x0'  = [2,0]    + N(0,[10,0;0,10])
  States:  x1      x2
  Outputs: y1      y2
```

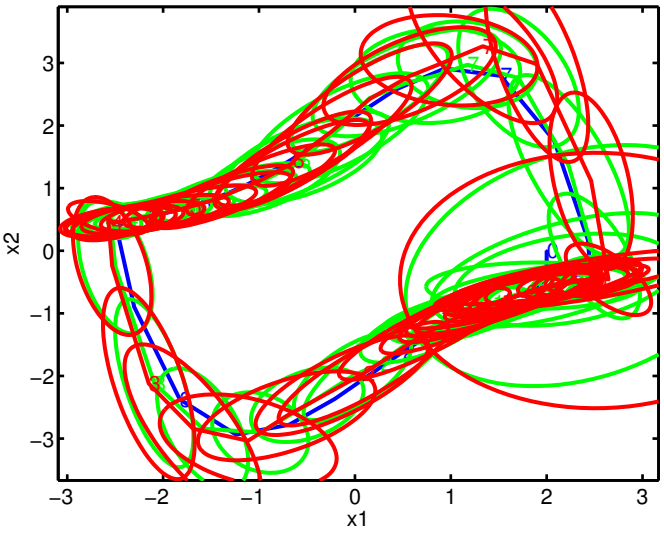
The process noise is zero in this system, which gives rise to periodic state trajectories, see below. The PF is compared to the EKF and ground truth with confidence bounds.

```
z=simulate(m,10);
mpf=m;
mpf.pv=0.01*eye(2);  % Some dithering required
zekf=ekf(mpf,z);
zpf=pf(mpf,z,'k',0);
xplot(z,zpf,zekf,'conf',90)
```



Both EKF and PF perform well with a useful confidence band. The state trajectory can be illustrated with a phase plot of the the two states.

```
xplot2(z,zpf,zekf,'conf',90)
```



The particle filter fits an ellipsoid to the set of particles, and in this example it gives much smaller estimation uncertainty.

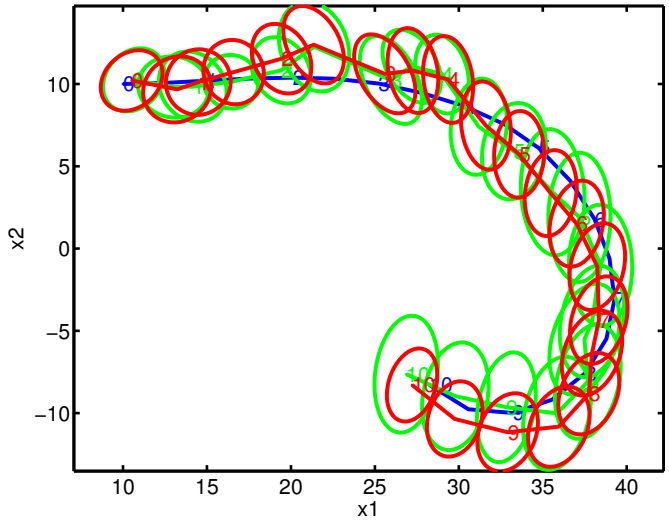
Target Tracking with Coordinated Turns

Target tracking is in this section studied using a coordinated turn model.

```
m=exnl('ctpv2d')
NL object: Coordinated turn model with polar velocity
          / ➡
               x(1,:)+2*x(3,:)./(eps+x(5,:)).*sin((eps+x(5,:))*0.5/2).*cos(x(4,:)+x(5,:)*0.5/2) ➡
                    \
          | ➡
               x(2,:)+2*x(3,:)./(eps+x(5,:)).*sin((eps+x(5,:))*0.5/2).*sin(x(4,:)+(eps+x(5,:))*0.5/2) ➡
               |
x[k+1] = | x(3,:) ➡
                                                                                                  ➡
     |
          | x(4,:)+x(5,:)*0.5 ➡
                                                                                                ➡
               |
          \ x(5,:) ➡
                                                                                                     ➡
               /


+ v


        / sqrt(x(1,:).^2+x(2,:).^2) \
   y = \ atan2(x(2,:),x(1,:))       / + e
   x0'  = [10,10,5,0,0.1]    + ➡
        N(0,[10,0,0,0,0;0,10,0,0,0;0,0,1e+02,0,0;0,0,0,10,0;0,0,0,0,1])
  States:  x1     x2     v     h     w
  Outputs: R     phi
```

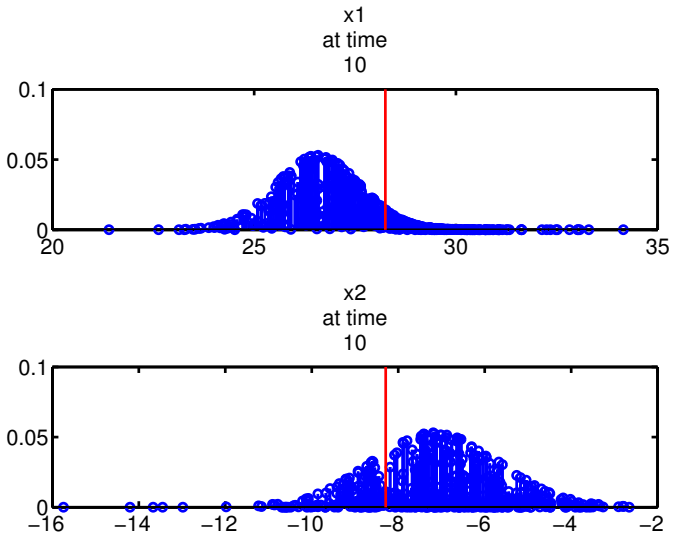
A trajectory is simulated, and the EKF and PF are applied to the noisy observations. The position estimates are then compared.

```
z=simulate(m,10);
zekf=ekf(m,z);
mpf=m;
mpf.pv=20*cov(m.pv);  % Dithering required for the PF
zpf=pf(mpf,z,'Np',1000);
xplot2(z,zpf,zekf,'conf',90)
```



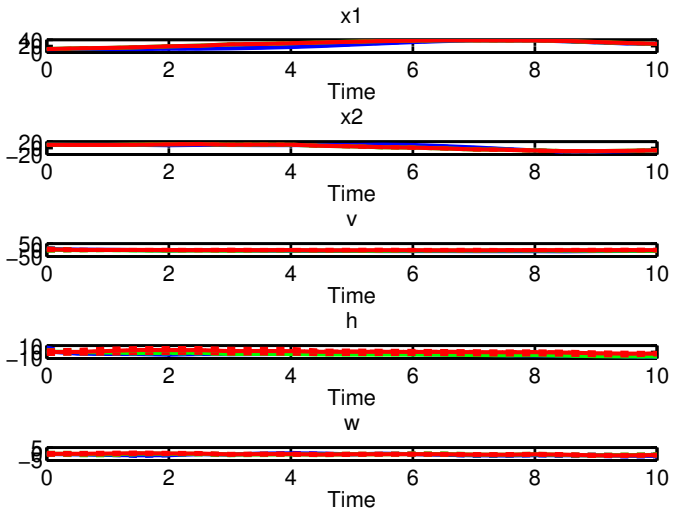
The estimated trajectories follow each other quite closely. There is an animation option, that illustrates the particle cloud for an arbitrary choice of states. Here, the position states are monitored.

```
zpf=pf(mpf,z,'animate','on','Np',1000,'ind',[1 2]);
```



The figure shows the final plot in the animation. The time consumption is roughly proportional to the number of particles, accept for a small overhead constant, as the following regression shows.

```
tic, zpf100=pf(mpf,z,'Np',100); t100=toc;
tic, zpf1000=pf(mpf,z,'Np',1000); t1000=toc;
tic, zpf10000=pf(mpf,z,'Np',10000); t10000=toc;
[t100 t1000 t10000]
ans =
    0.0754    0.0880    0.3242
xplot(zpf100,zpf1000,zpf10000,'conf',90,'ind',[1])
```



The plot shows that the estimate and its confidence bound are consistent for the two largest number of particles. The practically important conclusion is that 100 particles is not sufficient for this application.

7.4 Unscented Kalman Filters

The KF and EKF are characterized by a state and covariance update, where the covariance is updated according to a Ricatti equation. The class of filters in this section completely avoids the Ricatti equation. Instead, they propagate the estimate and covariance by approximating nonlinear transformations of Gaussian variables. Function evaluations and gradient approximations replace the Ricatti equation. The most well-known member of this class of filters is the unscented Kalman filter (UKF). A certain variant of the standard EKF is another special case.

7.4.1 Algorithms

The `ndist` object has a number of nonlinear transformation approximations of the kind

$$x \in \mathcal{N}(m_x, P_x) \Rightarrow z = g(x) \approx \mathcal{N}(m_z, P_z). \tag{7.3}$$

The following options exist:

TT1 in `ndist.tt1eval`: First order Taylor approximation, which gives Gauss' approximation formula

$$\text{TT1: } x \sim \mathcal{N}\big(\mu_x, P\big) \to z \sim \mathcal{N}\big(g(\mu_x), g'(\mu_x) P \big(g'(\mu_x)\big)^T\big), \tag{7.4}$$

TT2 in `ndist.tt2eval`: Second-order Taylor approximation

$$\begin{aligned}\text{TT2: } x \sim \mathcal{N}\big(\mu_x, P\big) \to z \sim \mathcal{N}\Big(& g(\mu_x) + \tfrac{1}{2}[\operatorname{tr}(g_i''(\mu_x)P)]_i, \\ & g'(\mu_x) P \big(g'(\mu_x)\big)^T + \tfrac{1}{2}\Big[\operatorname{tr}(P g_i''(\mu_x) P g_j''(\mu_x))\Big]_{ij}\Big).\end{aligned} \tag{7.5}$$

MCT in `ndist.mcteval`: the Monte Carlo transformation, which takes samples $x^{(i)}$ which are propagated through the nonlinear transformation, and the mean and covariance are fitted to these points.

$$x^{(i)} \sim \mathcal{N}\big(\mu_x, P\big), \quad i = 1, \ldots, N, \tag{7.6a}$$

$$z^{(i)} = g(x^{(i)}), \tag{7.6b}$$

$$\mu_z = \frac{1}{N} \sum_{i=1}^{N} z^{(i)}, \tag{7.6c}$$

$$P_z = \frac{1}{N-1} \sum_{i=1}^{N} \big(z^{(i)} - \mu_z\big)\big(z^{(i)} - \mu_z\big)^T. \tag{7.6d}$$

UT in `ndist.uteval`: the unscented transformation, which is characterized by its so called sigma point distributed along the semi-axis of the covariance matrix. These are propagated through the nonlinear transfor-

mation, and the mean and covariance are fitted to these points.

$$
\begin{aligned}
P &= U\Sigma U^T = \sum_{i=1}^{n_x} \sigma_i^2 u_i u_i^T, \\
x^{(0)} &= \mu_x, \\
x^{(\pm i)} &= \mu_x \pm \sqrt{n_x + \lambda}\sigma_i u_i, \\
\omega^{(0)} &= \frac{\lambda}{n_x + \lambda}, \\
\omega^{(\pm i)} &= \frac{1}{2(n_x + \lambda)}, \\
z^{(i)} &= g(x^{(i)}), \\
\mu_z &= \sum_{i=-n_x}^{n_x} \omega^{(i)} z^{(i)}, \\
P_z &= \sum_{i=-n_x}^{n_x} \omega^{(i)} (z^{(i)} - \mu_z)(z^{(i)} - \mu_z)^T \\
&\qquad + (1 - \alpha^2 + \beta)(z^{(0)} - \mu_z)(z^{(0)} - \mu_z)^T,
\end{aligned}
$$

The key idea is to utilize the following form of the Kalman gain, that occurs as an intermediate step when deriving the Kalman filter in the Bayesian approach:

$$
\begin{aligned}
K_k &= P_{k|k-1}^{xy} \left(P_{k|k-1}^{yy} \right)^{-1}, \\
\hat{x}_{k|k} &= \hat{x}_{k|k-1} + K_k \left(y_k - \hat{y}_{k|k-1} \right), \\
P_{k|k} &= P_{k|k-1} - K_k P_{k|k-1}^{yy} K_k^T.
\end{aligned}
$$

That is, if somebody can provide the matrices Pxx and Pxy, the measurement update is solved. The following algorithm shows how this is done.

1. Time update: augment the state vector with the process noise, and apply the NLT to the following function:

$$
\begin{aligned}
\bar{x} &= (x_k^T, v_k^T)^T \in \mathrm{N}\left((\hat{x}_{k|k}^T, 0^T)^T, \mathrm{diag}(P_{k|k}, Q) \right) \\
z &= x_{k+1} = f(x_k, u_k) + v_k \approx \mathrm{N}\left(\hat{x}_{k+1|k}, P_{k+1|k} \right).
\end{aligned}
$$

   The time updated state and covariance come out explicitly.

2. Measurement update: augment the state vector with the measurement

noise, and apply the NLT to the following function:

$$\bar{x} = (x_k^T, e_k^T)^T \in \mathrm{N}\left((\hat{x}_{k|k}^T, 0^T)^T, \mathrm{diag}(P_{k|k}, R)\right)$$
$$z = (x_k^T, y_k^T)^T = (x_k^T, (h(x_k, u_k) + e_k)^T)^T \approx \mathrm{N}\left((\hat{x}_{k|k-1}^T, \hat{y}_{k|k-1}^T)^T, P_{k|k-1}^z\right),$$

where Pz is partitioned into the blocks Pxx, Pxy, Pyx, and Pyy , respectively. Use these to compute the Kalman gain and measurement update.

7.5 Usage

The algorithm is called with syntax

```
x=nltffilt(m,z,Property1,Value1,...)
```

where

- `m` is the NL object specifying the model.
- `z` is an input SIG object with measurements.
- `x` is an output SIG object with state estimates `xhat=x.x` and signal estimate `yhat=x.y`.

The algorithm with script notation basically works as follows:

1. Time update:
   (a) Let xbar = [x;v] = N([xhat;0];[P,0;0,Q])
   (b) Transform approximation of x(k+1) = f(x,u)+v gives xhat, P
2. Measurement update:
   (a) Let `xbar = [x;e] = N([xhat;0];[P,0;0,R])`.
   (b) Transform approximation of `z(k) = [x; y] = [x; h(x,u)+e]` provides `zhat=[xhat; yhat]` and `Pz=[Pxx Pxy; Pyx Pyy]`.
   (c) The Kalman gain is `K=Pxy*inv(Pyy)`.
   (d) Set `xhat = xhat+K*(y-yhat)` and `P = P-K*Pyy*K'`.

The transform in 1b and 2b can be chosen arbitrarily from the set of `uteval`, `tt1eval`, `tt2eval`, and `mceval` in the `ndist` object.

Note: the NL object must be a function of indexed states, so always write for instance `x(1,:)` or `x(1:end,:)`, even for scalar systems. The reason is that the state vector is augmented, so any unindexed `x` will cause errors.

User guidelines:

1. Increase state noise covariance `Q` to mitigate linearization errors in `f`
2. Increase noise covariance `R` to mitigate linearization errors in `h`
3. Avoid very large values of `P0` and `Q` (which can be used for KF and EKF)

One important difference to running the standard EKF in common for all these filters is that the initial covariance must be chosen carefully. It cannot be taken as a huge identity matrix, which works well when a Ricatti equation is used. The problem is most easily explained for the Monte Carlo method. If `P0` is large, random number all over the state space are generated and propagated by the measurement relation. Most certainly, none of these come close the observed measurement, and the problem is obvious.

7.5.1 Examples

Standard PF Example

The UKF should outperform the EKF when the second term in the Taylor expansion is not negligible. The standard particle filter example can be used to illustrate this.

```
m=exnl('pfex');
z=simulate(m,50);
zekf1=nltf(m,z,'tup','taylor1','mup','taylor1');
zekf=ekf(m,z);
zukf=nltf(m,z);
xplot(z,zukf,zekf,zekf1,'conf',90,'view','cont')
```

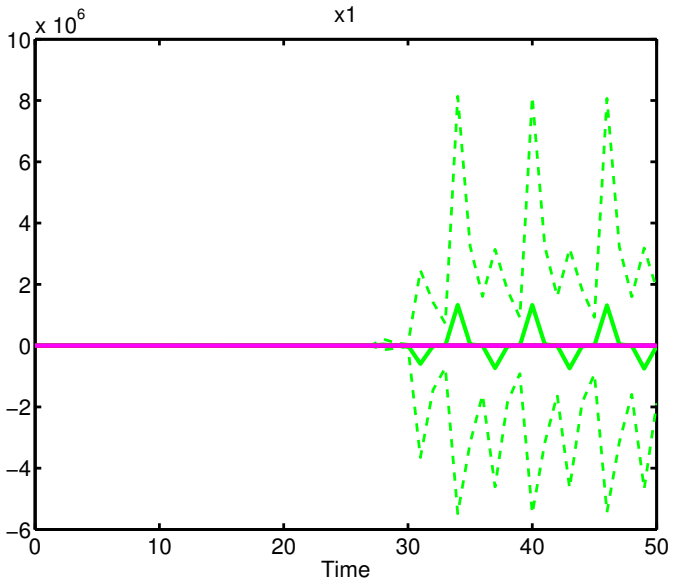


Table 7.4: `nl.nltf`

PROPERTY	VALUE	DESCRIPTION
k	k>0	0 Prediction horizon: 0 for filter (default) 1 for one-step ahead predictor
P0	[]	Initial covariance matrix Scalar value scales identity matrix Empty matrix gives a large identity matrix
x0	[]	Initial state matrix (overrides the value in m.x0) Empty matrix gives a zero vector
Q	[]	Process noise covariance (overrides m.Q) Scalar value scales m.Q
R	[]	Measurement noise covariance (overrides m.R) Scalar value scales m.R
tup	'uteval' 'tt1eval' 'tt2eval' 'mceval'	The unscented Kalman filter (UKF) The extended Kalman filter (EKF) The second order extended Kalman filter The Monte Carlo KF
mup	'uteval' 'tt1eval' 'tt2eval' 'mceval'	The unscented Kalman filter (UKF) The extended Kalman filter (EKF) The second order extended Kalman filter The Monte Carlo KF
ukftype	'ut1','ut2'|'ct'	Standard, modified UT, or cubature transform
ukfpar	[]	Parameters in UKF For ut1, par=w0 with default w0=1-n/3 For ut2, par=[beta,alpha,kappa] with default [2 1e-3 0] For ct, par=[a] with default [1]
NMC	100	Number of Monte Carlo samples for mceval

Table 7.5: *Different versions of the* UT *(counting the* CT *as a* UT *version given appropriate parameter choice) using the definition* $\lambda = \alpha^2(n_x + \kappa) - n_x$.

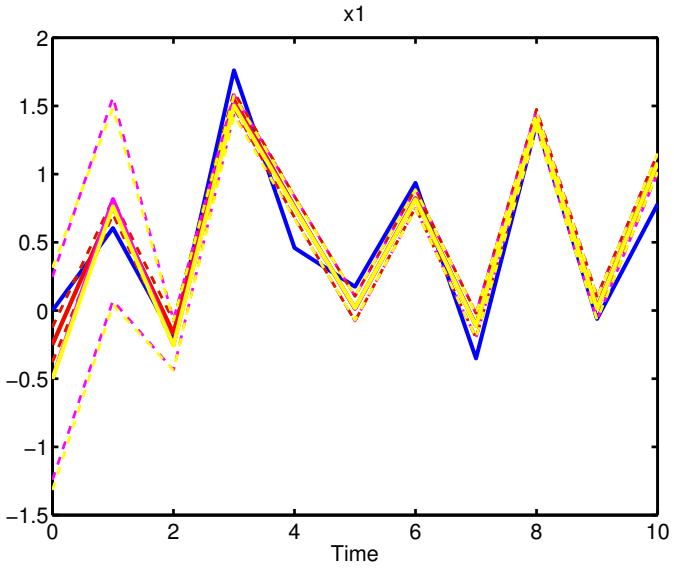
Parameter	**UT1**	**UT2**	**CT**	**DFT**
α	$\sqrt{3/n_x}$	10^{-3}	1	–
β	$3/n_x - 1$	2	0	–
κ	0	0	0	–
λ	$3 - n_x$	$10^{-6}n_x - n_x$	0	0
$\sqrt{n_x + \lambda}$	$\sqrt{3}$	$10^{-3}\sqrt{n_x}$	$\sqrt{n_x}$	$\frac{1}{a}\sqrt{n_x}$
$\omega^{(0)}$	$1 - n_x/3$	-10^6	0	0

The result of UKF is somewhat better than EKF. Both the standard and the NLT-based EKF version give identical results. That is, all variants of this method give quite consistent estimates.

Different EKF versions

Both the standard EKF and the EKF based on NLTF with TT1 options are equivalent in theory. Consider the following example which is nonlinear in both dynamics and measurements.

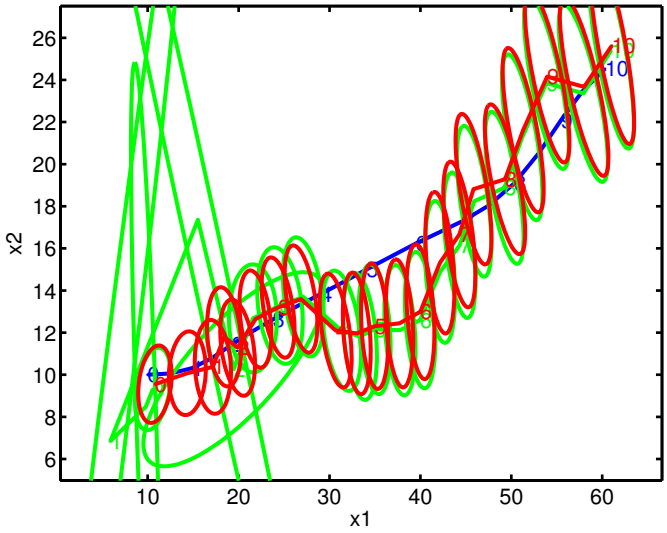
```
f='(x(1,:)-0.9).^2';
h='x(1,:)+0.5*x(1,:).^2';
m=nl(f,h,[1 0 1 0]);
m.fs=1;
m.pv=0.1;
m.pe=0.1;
m.px0=1;
m
NL object
x[k+1] = (x(1,:)-0.9).^2 + N(0,0.1)
   y = x(1,:)+0.5*x(1,:).^2 + N(0,0.1)
   x0'  = 0   + N(0,1)
  States:  x1
  Outputs: y1
z=simulate(m,10);
zekfstandard=ekf(m,z);
zukf=nltf(m,z,'tup','ut','mup','ut');
zekf=nltf(m,z,'tup','tt1','mup','tt1');
zmckf=nltf(m,z,'tup','mc','mup','mc');
xplot(z,zekfstandard,zekf,zukf,zmckf,'conf',90,'view','cont')
```



They are the same in practice also! Note that EKF is based on explicitly forming Jacobians, while NLTF just uses function evaluations ('derivative-free').

Target Tracking with Coordinated Turns

```
m=exnl('ctpv2d');  % coordinated turn model
z=simulate(m,10);  % ten seconds trajectory
zukf=nltf(m,z); % UKF state estimation
zekf=nltf(m,z,'tup','tt1','mup','tt1'); % EKF variant
xplot2(z,zukf,zekf,'conf',90);
```



7.6 Cramér-Rao Lower Bounds

Algorithm

The Cramer Rao lower bound (CRLB) is defined as the minimum covariance any unbiased estimator can achieve. The parametric CRLB for the NL model can be computed as

$$P_{k+1|k} = F_k P_{k|k} F_k^T + G_{v,k} \bar{Q}_k G_{v,k}^T$$
$$P_{k|k} = P_{k|k-1} - P_{k|k-1} H_k^T (H_k P_{k|k-1} H_k^T + \bar{R}_k)^{-1} H_k P_{k|k-1}.$$

All of F, G, H are obtained by linearizing the system around the nominal trajectory $x_{1:k}$. The state and measurement noise covariances `Q` and `R` are here replaced by the overlined matrices. For Gaussian noise, these coincide. Otherwise, `Q` and `R` are scaled with the intrinsic accuracy of the noise distribution, which is strictly smaller than one for non-Gaussian noise. Here, the gradients are defined at the true states. That is, the parametric CRLB can only be computed for certain known trajectories. The code is essentially the same as for the EKF, with the difference that the true state taken from the input SIG object is used in the linearization rather than the current estimate.

Usage

The usage of CRLB is very similar to EKF:

```
x=crlb(m,z,Property1,Value1,...)
```

The arguments are as follows:

Table 7.6: `nl.crlb`

PROPERTY	VALUE	DESCRIPTION
k	k>0 0	Prediction horizon: 0 for filter (default), 1 for one-step ahead predictor.
P0	[]	Initial covariance matrix. Scalar value scales identity matrix. Empty matrix gives a large identity matrix.
x0	[]	Initial state matrix. Empty matrix gives a zero vector.
Q	[]	Process noise covariance (overrides the value in m.Q). Scalar value scales m.Q.
R	[]	Measurement noise covariance (overrides the value in m). Scalar value scales m.R.

- `m` is a NL object defining the model.

- `z` is a SIG object defining the true state `x`. The outputs `y` and inputs `u` are not used for CRLB computation but passed to the output SIG object.

- `x` is a SIG object with covariance lower bound `Pxcrlb=x.Px` for the states, and `Pxcrlb=x.Px` for the outputs, respectively.

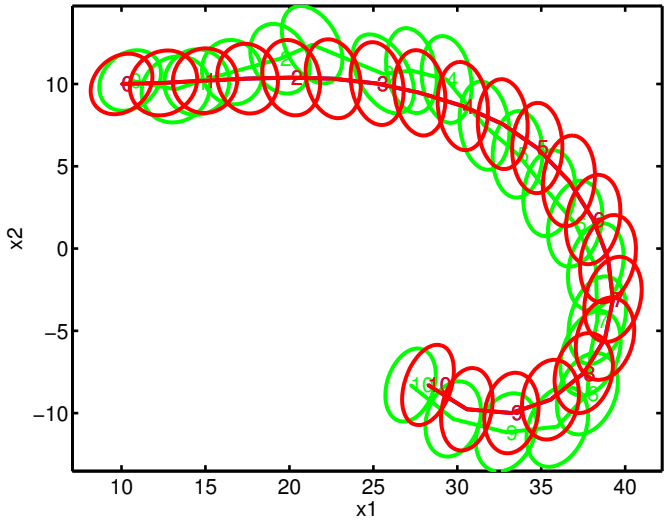
The optional parameters are summarized in Table 7.6.

7.6.1 Examples

Target Tracking

The CRLB bound is computed in the same way as the EKF state estimate, where the state trajectory in the input SIG object is used for linearization.

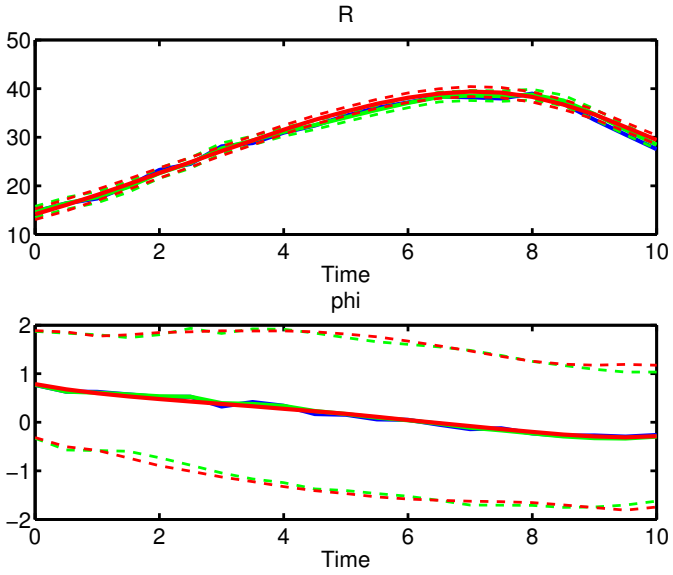
```
m=exnl('ctpv2d');
z=simulate(m,10);
zekf=ekf(m,z);
zcrlb=crlb(m,z);
xplot2(z,zekf,zcrlb,'conf',90)
```



The figure shows the lower bound on covariance as a minimum ellipsoid around the true state trajectory. In this example, the EKF performs very well, and the ellipsoids are of roughly the same size.

The CRLB method also provides an estimation or prediction bound on the output.

```
plot(z,zekf,zcrlb,'conf',90);
```

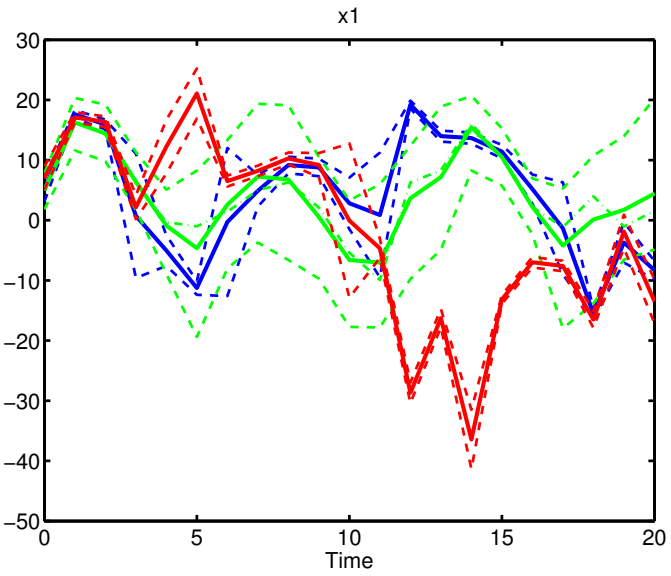


Also here, the confidence bounds are of the same thickness.

The Standard PF Example

The standard PF example illustrates a more challenging case with a severe nonlinearity, where it is harder to reach the CRLB bound.

```
m=exnl('pfex')
NL object
x[k+1] = x(1,:)/2+25*x(1,:)./(1+x(1,:).^2)+8*cos(t) + N(0,10)
   y = x(1,:).^2/20 + N(0,1)
   x0'  = 5   + N(0,5)
  States:  x1
  Outputs: y1
z=simulate(m,20);
mpf=m;
mpf.pe=10*cov(m.pe);  % Some dithering required
zekf=ekf(m,z);
zpf=pf(mpf,z,'k',1);
zcrlb=crlb(m,z);
xplot(zcrlb,zpf,zekf,'conf',90,'view','cont')
```



The CRLB confidence band is much smaller than for the PF algorithm. The EKF seemingly has a smaller confidence bound, but the bias is significant.

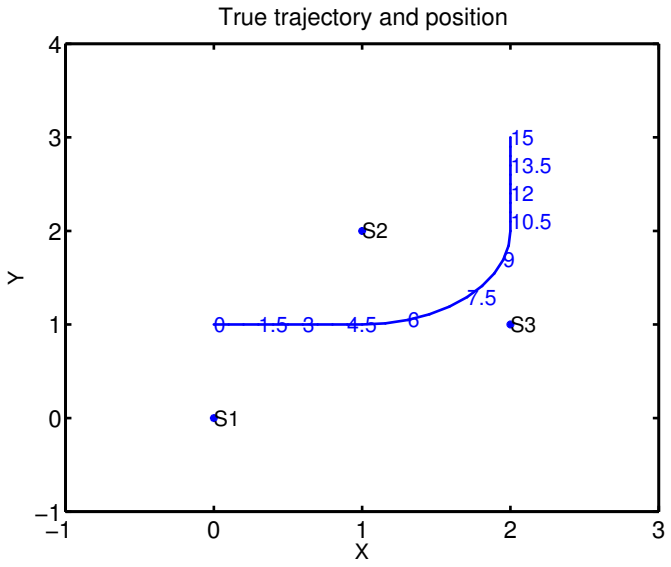
8

Application Example: Sensor Networks

Localization and target tracking in sensor networks are hot topics today in both research community and industry. The most well spread consumer products that are covered in this framework are global positioning systems (GPS) and the yellow page services in cellular phone networks.

The problem that is set up in this section consists of three sensors and one target moving in the horizontal plane. Each sensor can measure distance to the target, and by combining these a position fix can be computed. The range distance corresponds to travel time for radio signals in wireless networks as GPS and cellular phone systems.

The figure below depicts a scenario with a trajectory and a certain combination of sensor positions. First, target data and corresponding range measurements will be generated.



Four different problems are covered in the following sections:

1. Localization: Determine the position of the target from one snapshot measurement. This can be repeated for each time instant. The NLS algorithm will be used to solve the least squares fit of the three range observations for the two unknown horizontal coordinates.

2. Tracking: Use a dynamic motion model to track the target. This is the filter approach to the localization problem. A coordinated turn model will be used as motion model, and the EKF as the filter.

3. CRLB: What is the fundamental lower bound for tracking accuracy, given that the coordinated turn model is used, but independent on which algorithm is applied.

4. Mapping: Suppose that the target position is known, and the target observes range to three landmarks of partial unknown position. How to refine this landmark map along the travelled trajectory? The EKF will be applied to a state vector consisting of sensor positions.

5. Simultaneous localization and tracking (SLAM): If both the trajectory and sensor positions are unknown, a joint state vector can be used. EKF is applied to a coordinated turn model for the target states, where the state vector is augmented with the sensor positions.

The majority of the work to localize and track the target consists of setting up an appropriate model. Once this has been done, estimation and filtering

are quickly done, and the result conveniently presented using different NL methods. For that reason, the definitions of the models are presented in detailed in the following sections.

8.1 Defining a Trajectory and Range Measurements

The following lines define three critical points and a trajectory.

```
N=10;
fs=2;
phi=0:pi/N/2:pi/2;
x1=[0:1/N:1-1/N, 1+sin(phi), 2*ones(1,N)];
x2=[ones(1,N) 2-cos(phi), 2+1/N:1/N:3];
v=[1/N*fs*ones(1,N) pi/N/2*fs*ones(1,N+1) 1/N*fs*ones(1,N)];
heading=[0*ones(1,N) phi pi/2*ones(1,N)];
turnrate=[0*ones(1,N) pi/N/2*fs*ones(1,N+1) 0*ones(1,N)];
targetpos=[x1; x2];
```

The measurements are computed by defining a general range function, where both sensor locations are parameters and target position is the state vector.

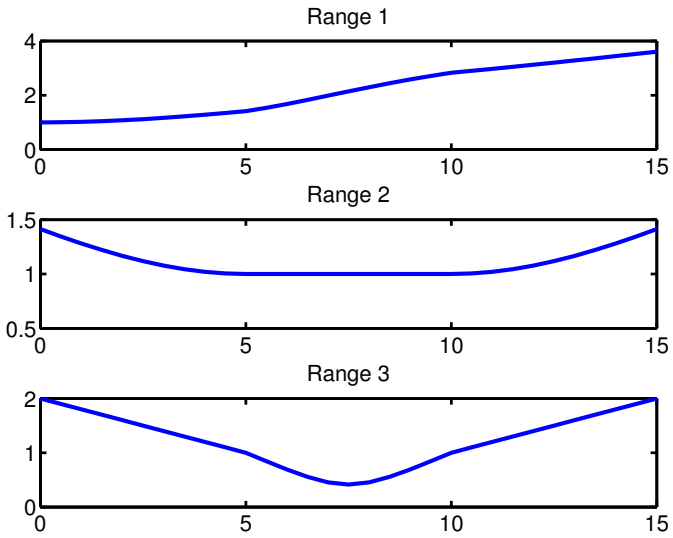
```
hstr='[sqrt((x(1,:)-th(1)).^2+(x(2,:)-th(2)).^2);sqrt((x(1,:)-th(3)).^2+(x(2,:)-th(4)).^2);sqrt((x(1,:)-t
h=inline(hstr,'t','x','u','th');
th=[0 0 1 2 2 1 ]';  % True sensor positions
y=h(0,targetpos,[],th).';
```

This will later be the measurement model in filtering. The noise-free range measurements are converted to a SIG object, and noise is added.

```
z=sig(y,fs,[],[x1' x2' v' heading' turnrate']);
z.name='Sensor network data';
z.ylabel={'Range 1','Range 2','Range 3'};
z.xlabel={'X','Y','V','Heading','Turn rate'}
SIG object with discrete time (fs = 2) stochastic state space data (no ➥
    input)
  Name:         Sensor network data
  Sizes:        N = 31,  ny = 3, nx = 5
  MC is set to: 30
  #MC samples:  0
z
SIG object with discrete time (fs = 2) stochastic state space data (no ➥
    input)
  Name:         Sensor network data
  Sizes:        N = 31,  ny = 3, nx = 5
  MC is set to: 30
  #MC samples:  0
zn=z+ndist(zeros(3,1),0.01*eye(3));
```

The data look as follows.

```
plot(z)
```



Finally, the plot shown in the beginning of this section is generated.

```
plot(th(1:2:end),th(2:2:end),'b*','linewidth',2)
hold on
for i=1:3
    text(th(2*i-1),th(2*i),['S',num2str(i)])
    text(th(2*i-1),th(2*i),['S',num2str(i)])
    text(th(2*i-1),th(2*i),['S',num2str(i)])
end
xplot2(z,'linewidth',2)
hold off
axis([-1 3 -1 4])
set(gca,'fontsize',18)
```

8.2 Target Localization using Nonlinear Least Squares

A standard GPS receiver computes its position by solving a nonlinear least squares problem. A similar problem is defined below. A static model is defined below. There are eight parameters corresponding to the 2D position of the three sensors and the target. The sensor positions are assumed known, so these values are entered to the parameter vector, while the origin is taken as initial guess for the target position.

```
h='[sqrt((th(7)-th(1)).^2+(th(8)-th(2)).^2);sqrt((th(7)-th(3)).^2+(th(8)-th(4)).^2);sqrt((th(7)-th(5)).^2
h =
[sqrt((th(7)-th(1)).^2+(th(8)-th(2)).^2);sqrt((th(7)-th(3)).^2+(th(8)-th(4)).^2);sqrt((th(7)-th(5)).^2+(t
m=nl([],h,[0 0 3 8]);
NL constructor warning: try to vectorize h for increased speed
m.th=[0 0 1 2 2 1 0 0]';  % True sensor positions and initial target ➟
    position
%m.px0=diag([0 0 0 0 0 0 10 10]); % Initial uncertainty of target position
```

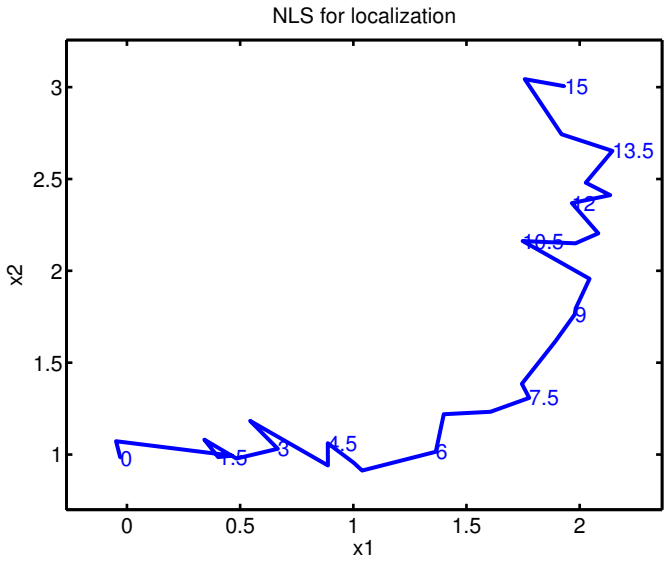
```
m.fs=fs;
m.name='Static sensor network model';
m.ylabel={'Range 1','Range 2','Range 3'};
m.thlabel={'pX(1)','pY(1)','pX(2)','pY(2)','pX(3)','pY(3)','X','Y'};
m
NL object: Static sensor network model
       / sqrt((th(7)-th(1)).^2+(th(8)-th(2)).^2) \
   y = | sqrt((th(7)-th(3)).^2+(th(8)-th(4)).^2) |
       \ sqrt((th(7)-th(5)).^2+(th(8)-th(6)).^2) /
   th'  = [0,0,1,2,2,1,0,0]
  Outputs: Range 1     Range 2     Range 3
  Param.: pX(1)     pY(1)     pX(2)     pY(2)     pX(3)     pY(3)     X ➥
          Y
```

Next, the NLS function is called by the NL method estimate. The search mask is used to tell NLS that the sensor locations are known exactly and thus these do not have to be estimated.

```
mhat=estimate(m,zn(1),'thmask',[0 0 0 0 0 0 1 1],'disp','on')
-------------------------------------------------------
Iter          Cost    Grad. norm    BT   Alg
-------------------------------------------------------
    0     1.628e+00             -     -   rgn
    1     8.673e-02     9.595e-01     1   rgn
    2     5.128e-04     3.827e-01     1   rgn
    3     7.144e-05     2.181e-02     1   rgn
    4     7.139e-05     2.741e-04     1   rgn
Relative difference in the cost function < opt.ctol.
NL object: Static sensor network model (calibrated from data)
       / sqrt((th(7)-th(1)).^2+(th(8)-th(2)).^2) \
   y = | sqrt((th(7)-th(3)).^2+(th(8)-th(4)).^2) |
       \ sqrt((th(7)-th(5)).^2+(th(8)-th(6)).^2) /
   th'  = [0,0,1,2,2,1,-0.029,0.98]
   std = [9.9e-08    9.9e-08    9.9e-08    9.9e-08    9.9e-08    9.9e-08 ➥
             0.86       0.86]
  Outputs: Range 1     Range 2     Range 3
  Param.: pX(1)     pY(1)     pX(2)     pY(2)     pX(3)     pY(3)     X ➥
          Y
```

The estimated initial position of the target is thus (0.077, 0.86), which you can compare to the true position (0,1). There are just three equations for two unknowns, so you can expect this inaccuracy. To localize the target through the complete trajectory, this procedure is repeated for each time instant in a for loop (left out below), and the result is collected to a SIG object and illustrated in a plot.

```
xhatmat(k,:)=mhat.th(7:8)';
P(k,:,:)=mhat.P(7:8,7:8);
xhat=sig(xhatmat,fs,[],xhatmat,P);
xplot2(xhat)
```

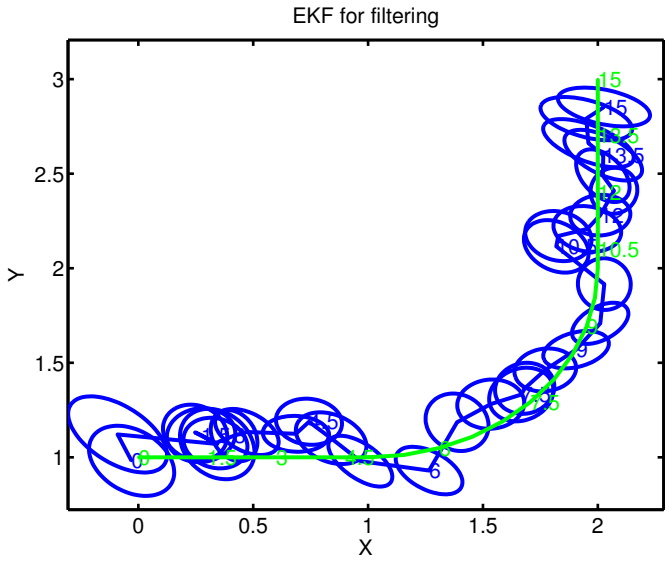


The snapshot estimates are rather noisy, and there is of course no correlation over time. Filtering as described in the next section makes use of a model to smoothen the trajectory

8.3 Target Tracking using EKF

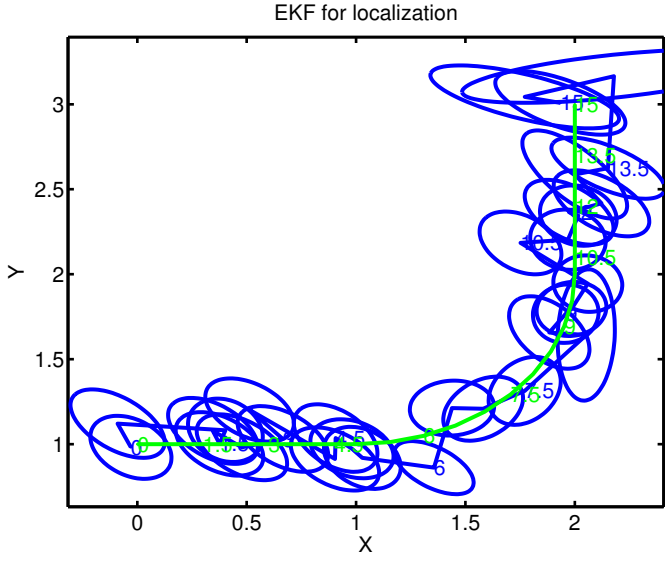
The difference between localization and tracking is basically only that a motion model is used to predict the next position. The estimated position can at each time be interpreted as an optimally weighted average between the prediction and the snapshot estimate. The standard coordinated turn polar velocity model is used again. However, the measurement relation is changed from a radar model to match the current sensor network scenario, and a new NL object is created.

```
sv=0.01; sw=0.001; sr=0.01;
m.pv=diag([0 0 sv 0 sw]);
m.pe=sr*eye(3);
xhat=ekf(m,zn);
NL.EKF warning: px0 not defined, using a default value instead
xplot2(xhat,z,'conf',90)
```



Process noise and measurement noise covariances have to be specified before the filter (here EKF) is called.

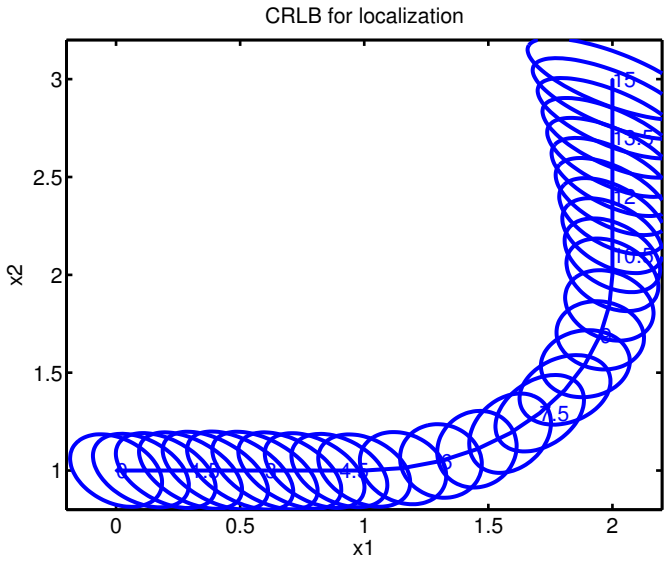
```
sv=1e3; sw=1e3; sr=0.01;
m.pv=diag([0 0 sv 0 sw]);
m.pe=sr*eye(3);
xhat=ekf(m,zn);
NL.EKF warning: px0 not defined, using a default value instead
xplot2(xhat,z,'conf',90)
```



The filter based on a dynamic motion model thus improves the NLS estimate a lot. To get a more fair comparison, the process noise can be increased to a large number to weigh down the prediction. In this way, the filtered estimate gets all information from the current measurement, and the information in past observations is neglected.

EKF here works as a very simple NLS solver, where only one step is taken along the gradient direction (corresponding approximately to the choices maxiter=1 and alg=sd), where the state prediction is as an initial guess. Thus, there is no advantage at all for using EKF to solve the NLS problem other than possibly a stringent use of the EKF method. However, one advantage is the possibility to compute the Cramer-Rao lower bound for localization.

```
xcrlb=crlb(m,zn);
xplot2(xcrlb,'conf',90)
```



The confidence ellipsoids are computated by using the true state trajectory, and the ellipsoids are placed around each true position.

8.4 Mapping

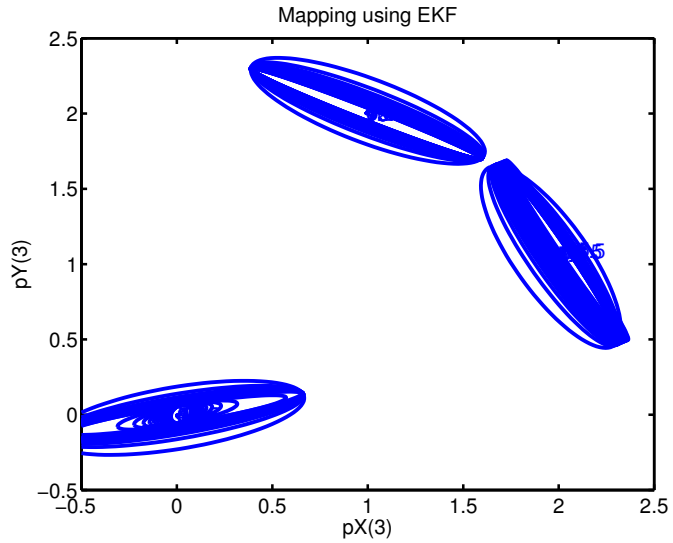
Mapping is the converse problem to tracking. It is here assumed that the target position is known at each instant of time, while the sensor locations are unknown. In a mapping applications, these corresponds to landmarks which might be added to a map on the fly in a dynamic way when they are first detected. The target position is in the model below considered to be a

known input signal. For that reason, a new data object with state position as input is defined. The sensor positions are put in the state vector.

```
f=['x(1:6,:)'];
h='[sqrt((u(1)-x(1,:)).^2+(u(2)-x(2,:)).^2);sqrt((u(1)-x(3,:)).^2+(u(2)-x(4,:)).^2);sqrt((u(1)-x(5,:)).^2
m=nl(f,h,[6 2 3 0]);
m.x0=th+0.2*randn(6,1);     % Initial sensor positions
m.px0=0.1*eye(6);           % Sensor position uncertainty
m.pv=0*eye(6)  ;            % No process noise
m.pe=1e-2*eye(3);
m.fs=mm.fs;
m.name='Mapping model for the sensor network';
m.xlabel={'pX(1)','pY(1)','pX(2)','pY(2)','pX(3)','pY(3)'};
m.ylabel={'Range 1','Range 2','Range 3'};
m
NL object: Mapping model for the sensor network
x[k+1] = x(1:6,:) + v
       / sqrt((u(1)-x(1,:)).^2+(u(2)-x(2,:)).^2) \
   y = | sqrt((u(1)-x(3,:)).^2+(u(2)-x(4,:)).^2) | + e
       \ sqrt((u(1)-x(5,:)).^2+(u(2)-x(6,:)).^2) /
   x0'  = [0.042,-0.19,1,2.1,1.8,0.96]   + ➥
        N(0,[0.1,0,0,0,0,0;0,0.1,0,0,0,0;0,0,0.1,0,0,0;0,0,0,0.1,0,0;0,0,0,0,0.1,0;0,0,0,0,0,0.1])
  States:  pX(1)     pY(1)     pX(2)     pY(2)     pX(3)     pY(3)
  Outputs: Range 1     Range 2     Range 3
  Inputs:  u1     u2
```

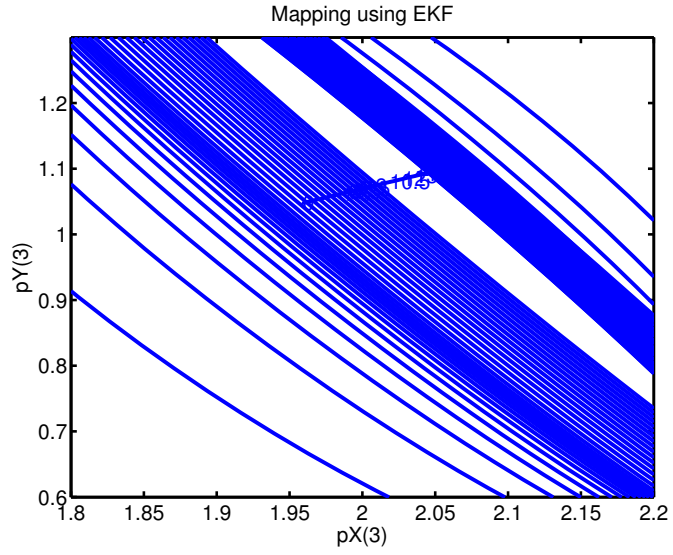
Now, the signal object is created and the EKF is called. Since each pair of states corresponds to one sensor location, the xplot2 method is applied pairwise.

```
zu=sig(y,fs,[x1' x2']);
xmap=ekf(m,zu);
xplot2(xmap,'conf',90,[1 2])
hold on
xplot2(xmap,'conf',90,[3 4])
xplot2(xmap,'conf',90,[5 6])
hold off
axis([-0.5 2.5 -0.5 2.5])
```



The sensor locations converge, but rather slowly in time. The plot below zooms in on sensor three.

```
xplot2(xmap,'conf',90,[5 6])
axis([1.8 2.2 0.6 1.3])
```



There is obviously a lack of excitation in one subspace, where range does not give much information for the given trajectory.

8.5 Simultaneous Localization and Mapping (SLAM)

The combined task of target tracking and mapping can be recast to a SLAM formulation. The model below takes the motion model from the coordinated turn, polar velocity model, where the state is augmented with the sensor locations. The model assumes that the sensor locations are fixed in time, so there is no process noise for these states. The initial distribution should reflect the initial uncertainty in both target state and sensor locations.

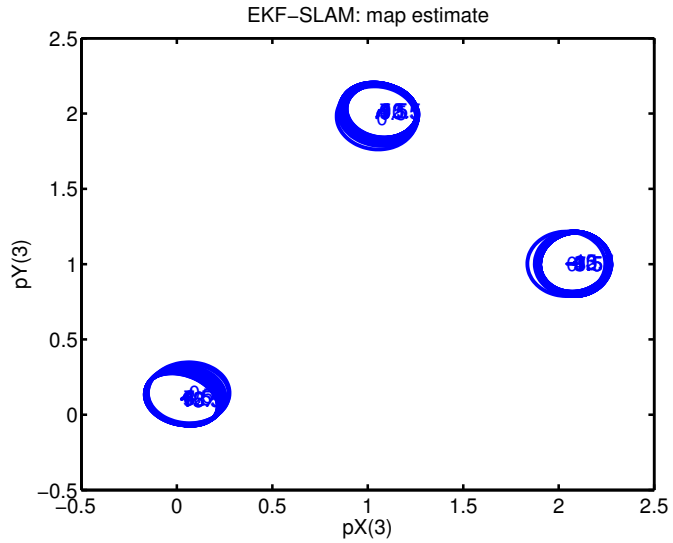
```
mm=exnl('ctpv2d');
f=['[',char(mm.f),'; x(6:11,:)]'];
h='[sqrt((x(1,:)-x(6,:)).^2+(x(2,:)-x(7,:)).^2);sqrt((x(1,:)-x(8,:)).^2+(x(2,:)-x(9,:)).^2);sqrt((x(1,:)-
m=nl(f,h,[11 0 3 0]);
m.x0=[0;1;1;0;0;th+0.1*randn(size(th))];     % Initial state
m.px0=diag([zeros(1,5), 0.01*ones(1,6)]);  % Only sensor pos uncertainty
m.pv=diag([0 0 sv 0 sw zeros(1,6)]);   % No process noise on sensor pos
m.pe=1e-1*eye(3);
m.fs=mm.fs;
m.name='SLAM model for the sensor network';
m.xlabel={mm.xlabel{:},'pX(1)','pY(1)','pX(2)','pY(2)','pX(3)','pY(3)'};
m.ylabel={'Range 1','Range 2','Range 3'};
m
NL object: SLAM model for the sensor network
x[k+1] = ➠
     [[x(1,:)+2*x(3,:)./(eps+x(5,:)).*sin((eps+x(5,:))*0.5/2).*cos(x(4,:)+x(5,:)*0.5/2);x(2,:)+2
*x(3,:)./(eps+x(5,:)).*sin((eps+x(5,:))*0.5/2).*sin(x(4,:)+(eps+x(5,:))*0.5/2);x(3,:);x(4,:)+x(5,:)*
0.5;x(5,:)]; x(6:11,:)] + v
        / sqrt((x(1,:)-x(6,:)).^2+(x(2,:)-x(7,:)).^2)   \
   y = | sqrt((x(1,:)-x(8,:)).^2+(x(2,:)-x(9,:)).^2)   | + e
        \ sqrt((x(1,:)-x(10,:)).^2+(x(2,:)-x(11,:)).^2) /
   x0'  = [0,1,1,0,0,0.061,0.16,1,2,2,1]   + ➠
        N(0,[0,0,0,0,0,0,0,0,0,0,0;0,0,0,0,0,0,0,0,0,0,0;0,0,0,0,0,0,0,0,0,0,0;0,0,0,0,0,0,0,0,0,0,0;0,0,
  States:  x1     x2     v     h     w     pX(1)     pY(1)     pX(2)     ➠
      pY(2)     pX(3)     pY(3)
  Outputs: Range 1     Range 2     Range 3
```

Applying the EKF algorithm now gives the so called EKF-SLAM, in a rather straightforward and inefficient implementation. EKF-SLAM scales badly in the number of landmarks, so structural knowledge in the model can be used to derive more efficient algorithms.

```
xslam=ekf(m,z);
xplot2(xslam)
```

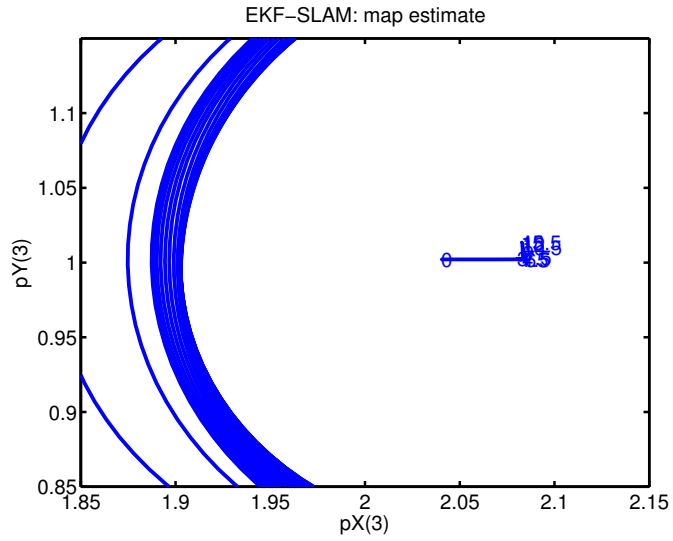
The trajectory looks worse than for the tracking case, where full knowledge of sensor location was assumed, but the estimate improves over time as the estimates of sensor positions are improved. Below, the improvement in the map is illustrated.

```
xplot2(xslam,'conf',90,[6 7])
hold on
xplot2(xslam,'conf',90,[8 9])
xplot2(xslam,'conf',90,[10 11])
hold off
axis([-0.5 2.5 -0.5 2.5])
```



As in the mapping case, the improvement is only minor over time. To learn these positions better, you need a longer trajectory in the vicinity of the sensors. The zoom below illustrates the slow convergence.

```
xplot2(xslam,'conf',90,[10 11])
axis([1.85 2.15 0.85 1.15])
```



Index