COMSOL SCRIPT

Fredrik Gustafsson

SIGNALS & SYSTEMS LAB

USER'S GUIDE

VERSION 1.1

COMSOL

How to contact COMSOL:

Benelux
COMSOL BV
Röntgenlaan 19
2719 DX Zoetermeer
The Netherlands
Phone: +31 (0) 79 363 4230
Fax: +31 (0) 79 361 4212
info@femlab.nl
www.femlab.nl

Denmark
COMSOL A/S
Diplomvej 376
2800 Kgs. Lyngby
Phone: +45 88 70 82 00
Fax: +45 88 70 80 90
info@comsol.dk
www.comsol.dk

Finland
COMSOL OY
Arabianranta 6
FIN-00560 Helsinki
Phone: +358 9 2510 400
Fax: +358 9 2510 4010
info@comsol.fi
www.comsol.fi

France
COMSOL France
WTC, 5 pl. Robert Schuman
F-38000 Grenoble
Phone: +33 (0)4 76 46 49 01
Fax: +33 (0)4 76 46 07 42
info@comsol.fr
www.comsol.fr

Germany
FEMLAB GmbH
Berliner Str. 4
D-37073 Göttingen
Phone: +49-551-99721-0
Fax: +49-551-99721-29
info@femlab.de
www.femlab.de

Italy
COMSOL S.r.l.
Via Vittorio Emanuele II, 22
25122 Brescia
Phone: +39-030-3793800
Fax: +39-030-3793899
info.it@comsol.com
www.it.comsol.com

Norway
COMSOL AS
Søndre gate 7
NO-7485 Trondheim
Phone: +47 73 84 24 00
Fax: +47 73 84 24 01
info@comsol.no
www.comsol.no

Sweden
COMSOL AB
Tegnérgatan 23
SE-111 40 Stockholm
Phone: +46 8 412 95 00
Fax: +46 8 412 95 10
info@comsol.se
www.comsol.se

Switzerland
FEMLAB GmbH
Technoparkstrasse 1
CH-8005 Zürich
Phone: +41 (0)44 445 2140
Fax: +41 (0)44 445 2141
info@femlab.ch
www.femlab.ch

United Kingdom
COMSOL Ltd.
UH Innovation Centre
College Lane
Hatfield
Hertfordshire AL10 9AB
Phone:+44-(0)-1707 284747
Fax: +44-(0)-1707 284746
info.uk@comsol.com
www.uk.comsol.com

United States
COMSOL, Inc.
1 New England Executive Park
Suite 350
Burlington, MA 01803
Phone: +1-781-273-3322
Fax: +1-781-273-6603

COMSOL, Inc.
10850 Wilshire Boulevard
Suite 800
Los Angeles, CA 90024
Phone: +1-310-441-4800
Fax: +1-310-441-0868

COMSOL, Inc.
744 Cowper Street
Palo Alto, CA 94301
Phone: +1-650-324-9935
Fax: +1-650-324-9936

info@comsol.com
www.comsol.com

For a complete list of international representatives, visit
www.comsol.com/contact

Company home page
www.comsol.com

COMSOL user forums
www.comsol.com/support/forums

Signals & Systems Lab User's Guide
© COPYRIGHT 1994–2007 by COMSOL AB. All rights reserved

Patent pending

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from COMSOL AB.

COMSOL, COMSOL Multiphysics, COMSOL Reaction Engineering Lab, and FEMLAB are registered trademarks of COMSOL AB. COMSOL Script is a trademark of COMSOL AB.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Version: October 2007 COMSOL 3.4

CONTENTS

Chapter 1: Introduction

Overview 2
What Can You Do With the Signals & Systems Lab? 2
News in Version 1.1. 4

Quick Tours 5
Signal Preprocessing and Frequency Analysis 5
Systems and Control 10
Time-Frequency Descriptions and Adaptive Filtering 13
Statistics 18

Signal Representations and Objects 22

Embedded Monte Carlo Simulations 26
Ensemble Uncertainty 26
Model Uncertainty 27

Chapter 2: Signal Processing

The SIG Object 30
Fields in the SIG Object 30
Creating SIG Objects 31
Data Preprocessing 38
Conversions to Other Objects 44
Examples of Real Signals 44
Standard Signal Examples 48

The FT Object 52
Introduction 52
Tutorial 53

Filtering and Filter Design **60**
Introduction 60
The TFTOOL Graphical User Interface 61
Filter Design 62

Spectral Analysis **69**
Introduction 69
The SPECTOOL Graphical User Interface 70
Tutorial 72

Covariance Function Analysis **79**
Introduction 79
Tutorial 80

Chapter 3: Systems and Control

Deterministic LTI Objects **86**
Creating and Displaying LTI Objects. 87
Operations on LTI Objects 95
Conversions from LTI to Other Objects and Data Types 108
Graphical Illustrations of LTI Objects 110

Application Examples **119**
Aircraft Pitch Control 119
Investment Model 127

Uncertain Systems **131**
Direct Definition of Uncertainty 131
Estimation 136

Stochastic State-Space Objects **137**
Operations on Stochastic LTI Objects 137

Nonlinear Model Objects **140**
Definition of the NL Object. 140
Examples of Model Definition and Simulation 143

Chapter 4: State Estimation and Kalman Filtering

State Estimation 152

Kalman Filtering for SS Models. 152
Extended Kalman Filtering for NL Objects 157
Particle Filtering for NL Objects 163
Nonlinear Transformation-Based Filters 171
The Cramer Rao Lower Bound (CRLB) 177
Application Example: Sensor Networks 181

Chapter 5: Model Estimation

ARX Models 196

Basic Algorithms 197
Simulation and Estimation of ARX Models 199
MIMO ARX Models. 201
Conversions of ARX Models 203
Operations on ARX Models 203

TF Models 206

Least-Squares Approach 206
DC Motor Example 207

NL Models 209

Nonlinear Least-Squares Parameter Estimation 209
Examples with Static Models 214
Step Response 216
Bouncing Ball 219
AUV Dynamics 222
Examples in Chemical Engineering 226

Chapter 6: Statistics

PDF Objects **238**

Available PDFs 238
The PDFTOOL User Interface 239
Basic Use 241
Symbolic Computations 243
Estimation 244
Multivariate Distributions 246
Defining Your Own Distributions 249

Fusion **251**

Theory 251
Examples 252

Nonlinear Transformations **254**

Algorithms 254
Examples 256

Chapter 7: Adaptive Filtering and Nonstationary Signal Processing

Time-Frequency Descriptions **262**

Introduction 262
Tutorial 262

Adaptive Filtering **267**

Introduction 267
Basic Models 267
The RARX Object 268
Basic Algorithms 268
Tutorial 270

INDEX **275**

1

Introduction

The documentation set for the Signals & Systems Lab consists of this book, the *Signals & Systems Lab User's Guide*, and the *Signals & Systems Lab Reference Guide*. Both books are available in PDF and HTML versions from the COMSOL Help Desk. The *Signals & Systems Lab User's Guide* is also available as a printed book.

Overview

What Can You Do With the Signals & Systems Lab?

The Signals & Systems Lab is an extension to COMSOL Script that adds high-level functions in the area of signals and systems. It contains tools for the following engineering disciplines:

- Data preprocessing
  - Filter design and filtering as extensions to the `filter` function in COMSOL Script
  - Windowing
  - Resampling and interpolation
  - Detrending and prefiltering
- Frequency analysis using the discrete-time Fourier transform and continuous-time Fourier transform analysis as a high-level complement to the `fft` function in COMSOL Script.
- Control system analysis:
  - Linear time-invariant (LTI) multiple-input–multiple-output (MIMO) system objects for transfer functions and stochastic state-space models.
  - Overloaded functions for LTI objects: `*`, `+`, `-`, `/`, `feedback`, and others.
  - Numerical algorithms such as Riccati equation solvers, Gramian computations, model reductions, and balanced realizations
  - Visualization tools such as Bode diagrams, Nyquist plots, root locus plots, and pole-zero plots
  - Computation of impulse responses, step responses, and general simulations
  - Kalman filtering
  - Nonlinear (NL) system simulation, calibration (parameter estimation, aka. gray box system identification) and state estimation (extended Kalman filtering, particle filtering)
  - Uncertain systems (both LTI and NL) can be defined, and the uncertainty is propagated in further processing

- Analysis of stochastic processes:
  - Spectral analysis
  - Covariance function estimation
  - Embedded Monte Carlo simulations for propagating and illustrating model uncertainty
- Model estimation by fitting transfer functions or ARMAX models and all of their special cases to signal data, with applications in spectral analysis, signal conditioning, and prediction
- Time-frequency description tools as an extension to Fourier analysis and spectral analysis for signals with time-varying characteristics
- Adaptive filtering:
  - Recursive estimation of time-varying models
  - Kalman filtering for state estimation
- Statistical analysis extending the `rand`, `erf`, and `erfinv` functions in COMSOL Script to non-Gaussian distributions:
  - Generating random numbers from a range of distributions
  - Estimating parametric distributions from data
  - Computing error functions and their inverses for hypothesis testing and confidence intervals

The Signals & Systems Lab includes the following special features:

- Customized overloaded plot functions that accept multiple arguments.
- Support of Monte Carlo simulations for representing stochastic objects. For instance, a filtering operation $y = Gu$ results in a stochastic signal y if either the input u is a stochastic process or if the filter G is uncertain.
- Graphical user interfaces with two modes:
  - Data analysis
  - Learning by examples
- Many benchmark examples plus a database of real signals with demonstrations

This introductory chapter starts off with a guided quick tour that provides illustrative examples for the different application areas in the section “Quick Tours” on page 5. Next comes an introduction to the objects and their main theoretical concepts in “Signal Representations and Objects” on page 22. Finally, the section “Embedded Monte Carlo Simulations” on page 26 explains the foundation of Monte Carlo

simulations that is used consistently as a powerful built-in feature throughout the Signals & Systems Lab. The other chapters treats one application area at a time, where the sections present the appropriate objects.

News in Version 1.1

Version 1.1 of Signals & Systems Lab contains the following new features:

- The nonlinear least squares (NLS) method.
- The NL object for representing nonlinear systems, with the following main methods:
  - **a** Simulation.
  - **b** Estimation and calibration of uncertain parameters in the model.Here, the NLS function is used.
  - **c** Filtering methods, featuring the extended Kalman filter, the unscented Kalman filter and the particle filter.
- The SS object has been extended with Kalman filtering algorithms, and supporting plot functions for state illustrations have been added to the SIG object.
- In the PDFCLASS objects, methods for fusion different estimates and approximating nonlinear transformations have been added.

Quick Tours

To help you better understand the basic ideas and to be able to explain the main concepts, this section provides some brief but typical examples from the Signals & Systems Lab organized in the following categories:

- "Signal Preprocessing and Frequency Analysis" on page 5
- "Systems and Control" on page 10
- "Time-Frequency Descriptions and Adaptive Filtering" on page 13
- "Statistics" on page 18

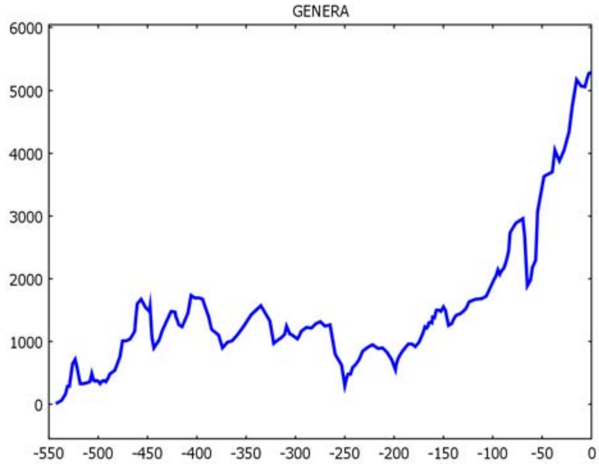
Signal Preprocessing and Frequency Analysis

The first example is based on a real data set, in which the number of genera for many time eras has been approximated from fossils. This signal is one of many real data examples in the Signals & Systems Lab.

To start with, this example loads it into the workspace and plots it with a dedicated signal plot. The `info` function displays user-specified information.

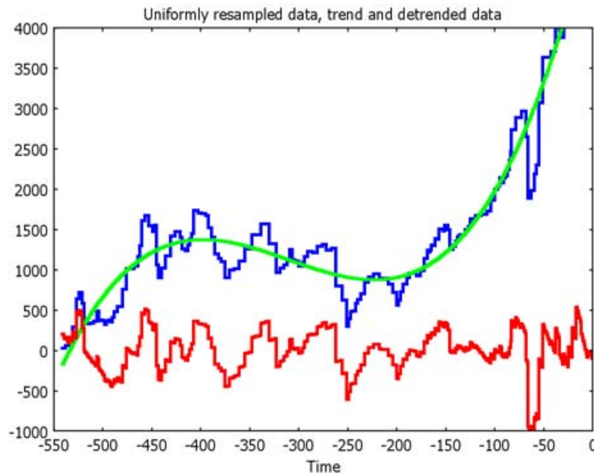
```
load genera
info(y2)
Name:    GENERA
Description:
 S3LAB Signal Database:  GENERA

 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:
```

```
   y:  # genera
Time:  Million years
plot(y2)
```



Because the eras are of different lengths, this data set is nonuniformly sampled by its very nature. Most signal-processing tools are designed based on uniform sampling, so here the code first interpolates the data points to a sampling interval of one million year. A plot then shows that there is clearly a slow trend in the data that resembles a cubic time function. For further analysis, this cubic trend is then removed from the data. This section does not include comments on each specific function that is used, so details on the `interp` and `detrend` functions follow in the tutorial and reference chapters.

```
t=-541:-1;
y3=interp(y2,t);
y3.fs=1;
[yd,ytrend,lsfit]=detrend(y3,3);
plot(y3,ytrend,yd,'Ylim',[-1000 4000])
title('Uniformly resampled data, trend and detrended data')
```

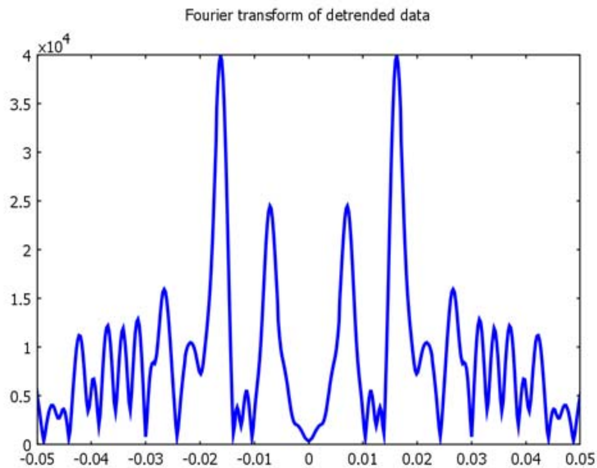


USING THE FOURIER TRANSFORM

Returning to this example, a challenging area of research is trying to correlate fluctuations in the number of genera to possible causes such as ice ages and other major events in the history of earth. The first step is to determine fundamental frequencies in the data using a Fourier transform. The fast Fourier transform (FFT) is the enabling algorithm for computer-based Fourier analysis, and the `fft` function is the workhorse for all transform-based methods in the Signals & Systems Lab. The most straightforward way to get a plot of the frequency content is to make a call similar to `plot(abs(fft(sig)))`. However, it is convenient to define a high-level function that takes care of axis scalings and zero padding to get a plot that is as informative as possible. The Signals & Systems Lab handles this in the `ft` conversion, which computes the discrete-time Fourier transform (DTFT).

It is usually advisable to window the data before frequency analysis. The following example illustrates the effect of windowed data in the Fourier-transform domain.

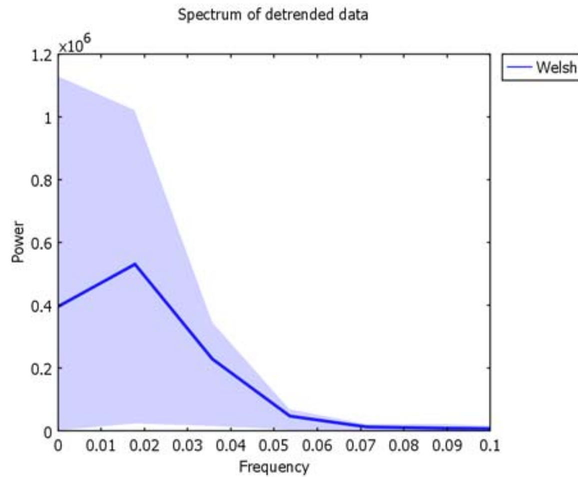
```
ydw=window(yd,'kaiser');
Ydw=ft(ydw);
plot(Ydw,'Xlim',[-0.05 0.05],'Ylim',[0 4e4])
title('Fourier transform of detrended data')
```



In the plot, the time window suppresses spurious frequency peaks caused by end transients, and the curve indicates two major fundamental frequencies in the number of genera. These correspond to cycles of approximately 65 and 145 million years.

Related to Fourier analysis is the field of spectral analysis. The basic difference is the underlying assumption in spectral analysis that the signal can be seen as a stochastic process with certain first-order (mean value) and second-order (variance) moments. The following code plots the signal's spectral estimate:

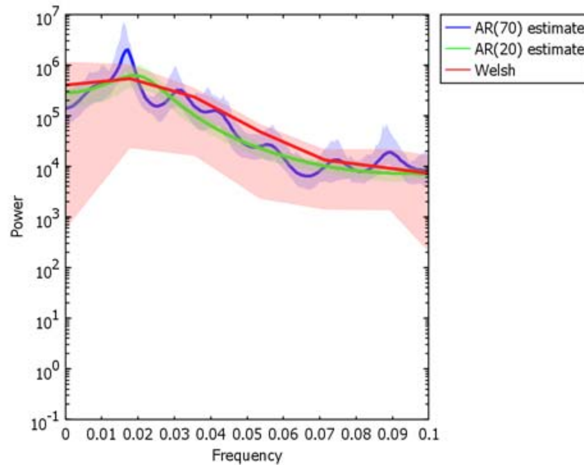
```
plot(spec(ydw),'Xlim',[0 0.1]);
title('Spectrum of detrended data')
```



The main difference as compared to Fourier analysis is that the resulting plot is a smoothed version of the FFT. In this example and using the default parameters, the time series turns out to be too short to show the two independent peaks in the spectrum. That is, the two peaks seen in the Fourier transform are blurred into one by the smoothing operations. Because of the assumption that the signal is a realization of a stochastic process, it is possible to compute a confidence region of the spectrum, which the Signals & Systems Lab generates automatically whenever possible.

A model-based approach to spectral analysis first adapts a linear time-invariant (LTI) model to describe the signal. There are many alternative models and options; this example uses autoregressive (AR) models of order 20 and 70.

```
AR20=estimate(arx(20),ydw);
AR70=estimate(arx(70),ydw);
semilogy(spec(AR70),spec(AR20),spec(ydw),'Xlim',[0 0.1]);
```

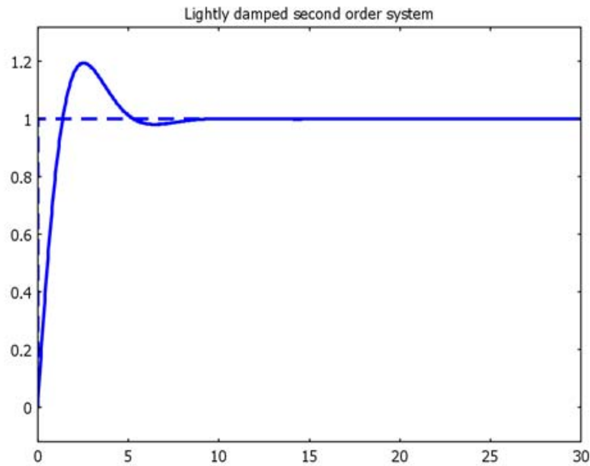


The high-order model catches the two low-frequency peaks. Again, the function automatically computes and displays the uncertainty intervals.

Systems and Control

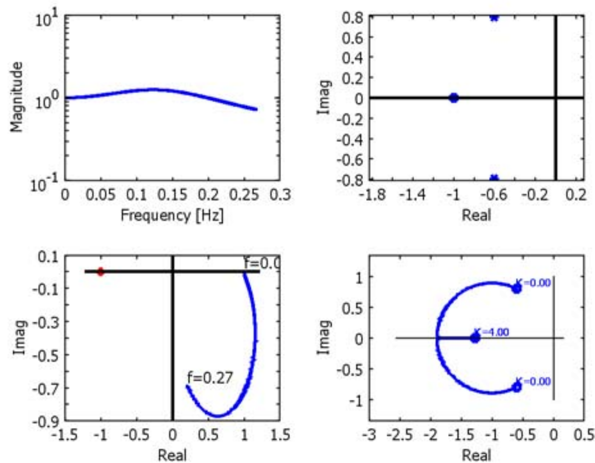
There are many ways to create and represent system models for control. The one in this section is constructed as a linear time-invariant (LTI) system from first principles in transfer-function representations. The first line in the following example defines the Laplace operator, then the following line shows how to create the transfer function using a direct definition. The system models a lightly damped system, and for subsequent plots the code gives the model a name for easier identification at later times.

```
s=tf('s');
G=(s+1)/(s^2+1.2*s+1)

                s+1
   Y(s) = ----------- U(s)
          s^2+1.2*s+1

G.name='Lightly damped second order system';
step(G);
```



Note that the printout is in the form of ASCII-formatted text. For system analysis and control design, there are many graphical ways to illustrate the system: Bode diagrams, Nyquist charts, pole-zero plots, and root loci. The following code generates these four graph types:

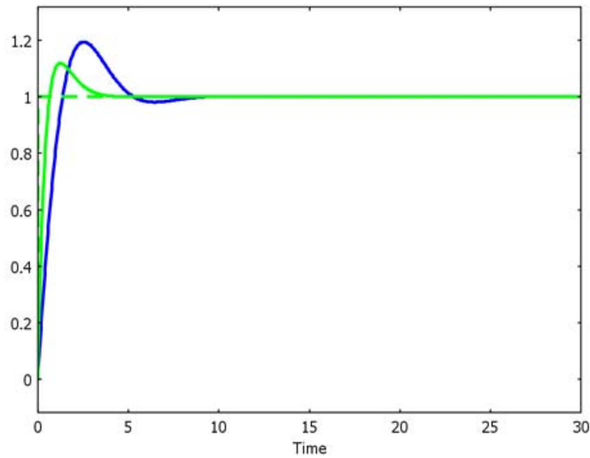
```
subplot(2,2,1)
bodeamp(G);
subplot(2,2,2)
zpplot(G);
subplot(2,2,3)
nyquist(G);
subplot(2,2,4)
rlplot(G,'Kmax',4);
```



Note how you can add properties in the input syntax—in this case the maximum gain, *K*, for the root locus is specified. There are often many property-value pairs associated with a function, and they are defined in the *Command Reference* chapter of this manual and the function's help text. The root locus indicates that a feedback with a gain of roughly 2 would stabilize the system (the poles' imaginary part decreases and the dominating pole moves further away from the origin).

The final example here computes the closed-loop system with a feedback gain of 2 and with a forward gain of 3 in order to get a unit static gain. You can simulate any LTI system using an arbitrary input signal `u` as a SIG object with the function call `simulate(G,u)`. For impulse and step responses of continuous-time systems, however, there is an analytic solution computed by `impulse(G)` and `step(G)`, respectively. The step responses of the open-loop and closed-loop systems are now compared:

```
Gc=3*feedback(G,2)

                3*s^3+6.6*s^2+6.6*s+3
      Y(s) = --------------------------- U(s)
             s^4+4.4*s^3+7.8*s^2+6.8*s+3

plot(step(G),step(Gc))
```

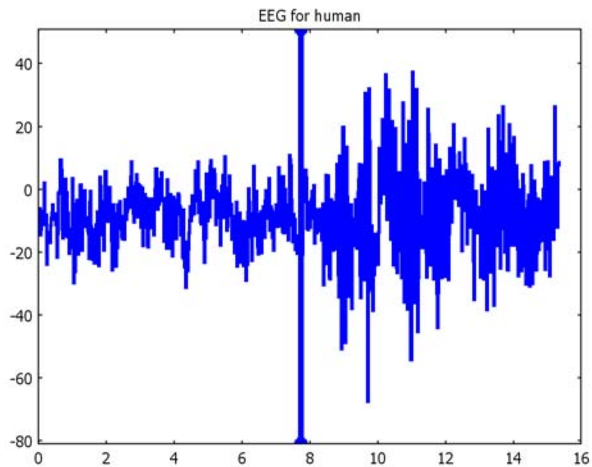


The feedback system has a faster step response and a smaller overshoot than the open-loop system.

Time-Frequency Descriptions and Adaptive Filtering

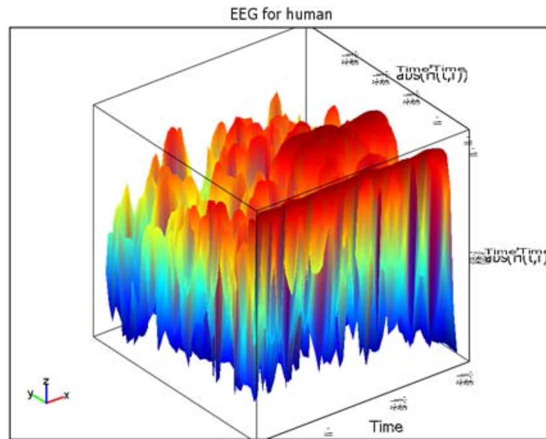
Model-based analysis of signals with time-varying characteristics is the task in adaptive filtering. The next example studies a human EEG signal, and the `info` function displays some background text about this experiment.

```
load eeg_human
info(y)
Name:    EEG for human
Signals:
   y1:  y
Time:  s
plot(y)
```



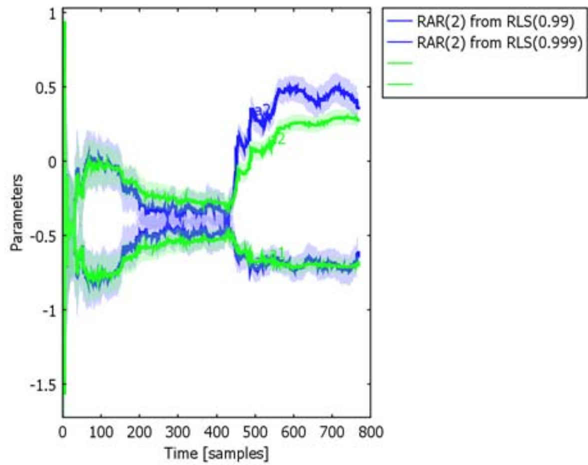
The marker at sample time 387 (7.7 s) is part of the data set. The extension of spectral and Fourier analysis into time-varying signal characteristics is called the *time-frequency description* (TFD). The following code computes the TFD as a short-time Fourier transform and illustrates it as a surface plot. The resonance frequency that appears after 8 seconds is clearly visible.

```
Yt=tfd(y);
surf(Yt)
```



As this illustration shows, something happens in the 25 Hz region shortly after the light goes on. A model-based alternative is to estimate an autoregressive model adaptively, here using the recursive least-squares algorithm with forgetting factor 0.99. This results in a time-varying AR model, which is illustrated using the `plot` method. The code compares two adaptive algorithms, one with a fast and one with a slow adaptation rate. The plot shows how the time-varying AR parameters evolve over time for these two adaptive filters.

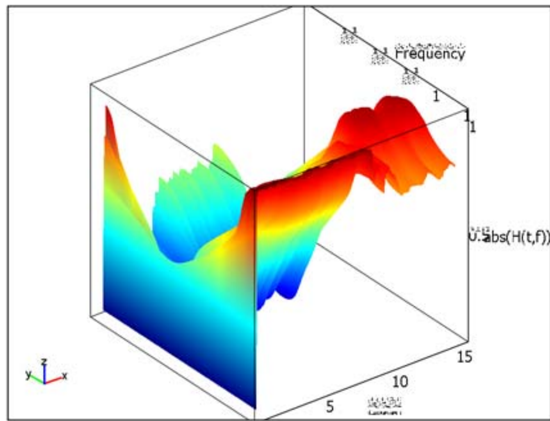
```
m1=estimate(rarx(2),y,'adg',0.99);
m2=estimate(rarx(2),y,'adg',0.999);
plot(m1,m2)
```



You would select the fast filter for further processing due to its quick response to the known excitation.

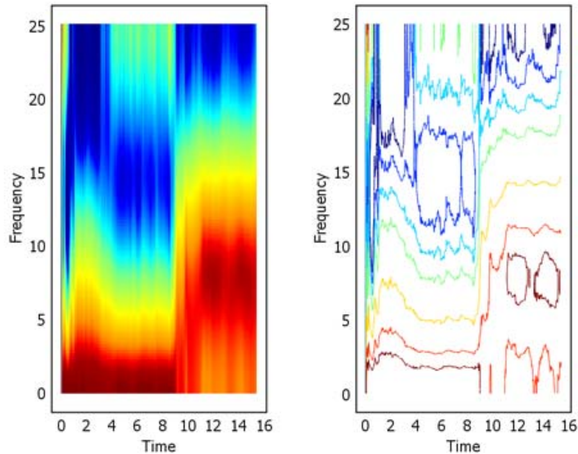
At each time, the AR model can be associated with a Fourier transform. This gives the TFD view of the RARX object `m1` that the previous code segment created.

```
Ytm=tfd(m1);
surf(Ytm)
```



In this case, the change in frequency content is more clearly visible for a model-based analysis, compared to the nonparametric Fourier-transform approach.

```
subplot(1,2,1),
image(Ytm)
subplot(1,2,2),
contour(Ytm)
```



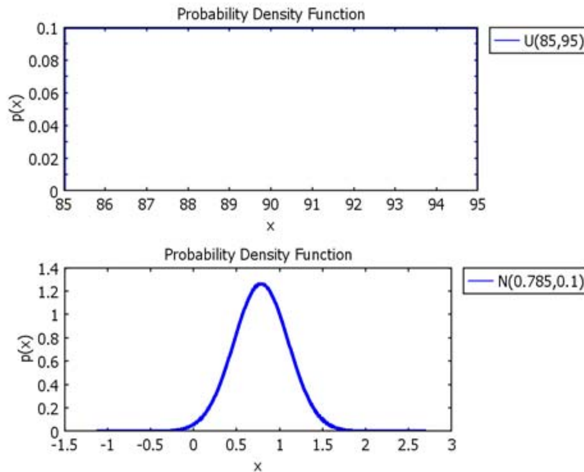
Statistics

As an application example in the sophisticated modeling of measurement error distribution, consider a positioning problem where a sensor gives rather accurate range information but poor bearing accuracy. This is typical for localization applications in cellular phone systems where distance is computed from received signal strength or propagation time delay, while the spatial resolution is obtained only from the directional antenna property. Radars work similarly, although the spatial resolution is usually much better, so this example is also relevant for many tracking applications. Further, satellite-based global positioning systems (GPS) give similar uncertainties from a single satellite.

There are plenty of possible distributions available for modeling sensor errors. This example uses a uniform (`udist`) range uncertainty and Gaussian (`ndist`) bearing error. Symbolic computations on distributions are possible, so this example simply adds a constant to each distribution for simulating data from a target at an angle of 45 degrees and a range of 90 m.

```
R=90+udist(-5,5)
U(85,95)
Phi=pi/4+ndist(0,0.1)
N(0.785,0.1)
subplot(2,1,1)
```

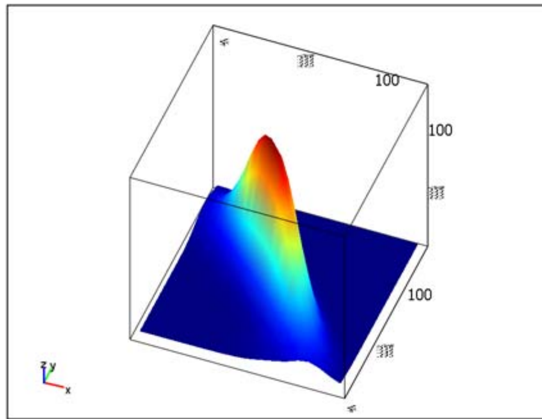
```
plot(R)
subplot(2,1,2)
plot(Phi)
```



The overloaded `plot` function illustrates the distributions. You can apply any feasible mathematical operation to the distributions. In some cases, the result belongs to a new parametric distribution, in which case the Signals & Systems Lab creates a symbolic result. This was the case earlier. In all other cases, the result is an empirical distribution represented by a large number of samples.

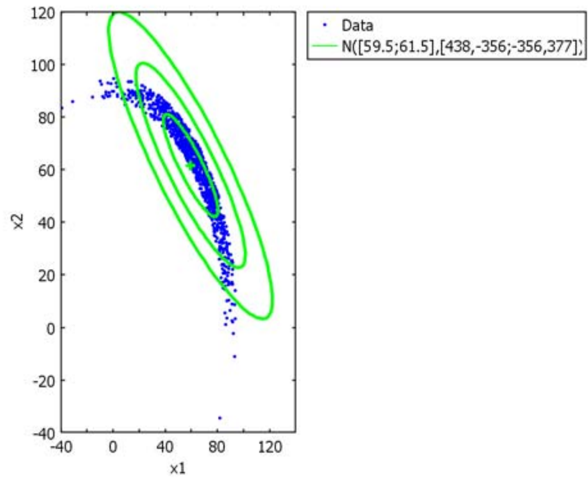
As an illustration of this, suppose that you now want to express the position distribution in Cartesian coordinates. Then simply define the two transformations and concatenate them to a stochastic vector (a multivariate statistics). A `surf` plot illustrates two dimensions of a multivariate distribution.

```
P=[R*cos(Phi);R*sin(Phi)]
Empirical data vector of size 2 with 1000 samples
surf(P,[1 2],'Xlim',[0 130],'Ylim',[0 130])
campos([400 -800 1000])
```



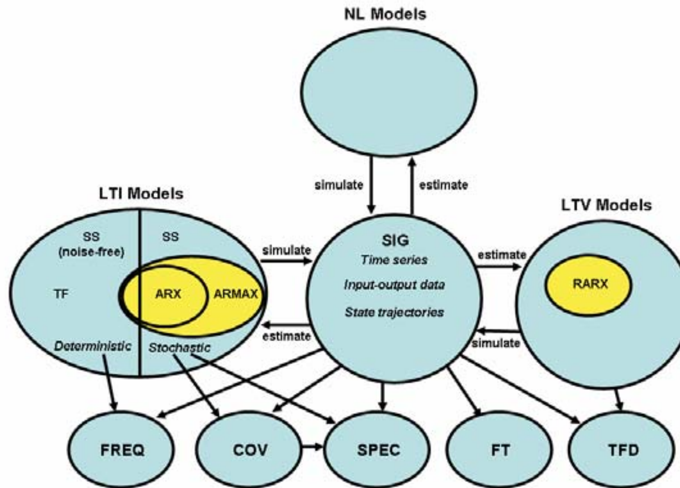
The banana shape of this distribution is more visible in a scatter plot, which the following lines of code generate. For comparison, the code fits a two-dimensional Gaussian distribution to the empirical distribution and illustrates it in the same plot.

```
Nhat=estimate(ndist,P);
plot2(P,Nhat)
```


Data
N([59.5;61.5],[438,-356;-356,377])
x1
x2
120
100
80
60
40
20
0
-20
-40
0
40
80
120

Signal Representations and Objects

One concise way to view the Signals & 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 & 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):

- *SIG object*: The signal domain represents an observed signal y. 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 the section "The SIG Object" on page 30.
- *FT object*: The frequency-domain representation that the Fourier transform (FT) object uses extends the discrete Fourier transform (DFT) computed with the fast

Fourier transform (FFT) to the discrete-time Fourier transform (DTFT). The DFT is defined by

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

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

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

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

- *LTI object*: Linear time-invariant (LTI) signal models are covered in the following LTI objects:
  - SS (state-space objects)
  - TF (transfer functions)
  - 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 as described in "Deterministic LTI Objects" on page 86. The possibility to extend SS objects to stochastic models is presented in "Stochastic State-Space Objects" on page 137. For more information about ARX, see ARX Models on page 196. 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 1-1 summarizes the objects with their structural parameters and model definitions.

TABLE 1-1: DEFINITION OF MODEL OBJECTS.

OBJECT	NN	DEFINITION
`tf`	[na, nb, nk]	Ay(t)=Bu(t-nk)
`arx`	[na, nb, nk]	Ay(t)=Bu(t-nk)+v(t)
`ss`	[nx, nu, nv, ny]	$y(t)=C(sI-A)^{-1}(Bu(t)+v(t))+Du(t)+e(t)$

- *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 & 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 (see "Graphical Illustrations of LTI Objects" on page 110). For the stochastic signal model part, see the COV and SPEC objects as discussed next.
- *COV* object: The covariance function is defined as

$$R(\tau) = \mathrm{E}(y(t)y(t-\tau)),$$

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. The covariance function object is presented in "Covariance Function Analysis" on page 79.
- *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:

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

It is related to the Fourier transform as

$$\Phi(f) = \lim_{N \to \infty} \frac{T}{N}|Y(f)|^2$$

Comparing it with Parseval's formula

$$\int y^2(t)\mathrm{d}t = \frac{1}{2\pi}\int_{-\infty}^{\infty}|Y(f)|^2 df$$

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. For further details, see "Spectral Analysis" on page 69.
- *LTV object*: A linear time-varying (LTV) signal model is similar to an LTI object except the transfer functions depend on time. An adaptive filter produces an estimate of an LTV model. This object is described in "Adaptive Filtering and Nonstationary Signal Processing" on page 261.
- *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

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

where $\omega(t-s)$ 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. See the section "Time-Frequency Descriptions" on page 262 for more information.

- *PDFCLASS object*: The amplitude distribution of a signal vector is described by its probability density function (PDF). Specific distributions such as the normal, exponential, beta, gamma, student's t, F, and χ^2 are children of the PDFCLASS with names as `ndist`, `expdist`, `betadist`, `gammadist`, `tdist`, `fdist`, `chi2dist`, and so on. These objects are examined in "PDF Objects" on page 238.

  Common tasks in statistics involve:

  - Random number generation, where the `rand` function is an overloaded method on each distribution.
  - 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.
  - Evaluation of the PDF, the cumulative distribution function (CDF), the error function (ERF), or certain moments (mean, variance, skewness, and kurtosis) of given distributions.
  - Fitting parametric distributions to empirical data.
  - Visualization of data and PDFs.

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

Embedded Monte Carlo Simulations

The basic idea behind embedded Monte Carlo simulations is that every time a stochastic variable is transformed, a number of samples of the stochastic variable undergo the same transformation. This special feature in the Signals & Systems Lab is something you as a user seldom have to bother with. It is a background process, and at any time you can ask for confidence bounds in your plots or analyze the distribution of a particular parameter of interest. The confidence bounds are approximate, conditioned on the underlying assumptions in the chain of function calls leading to the result rather than on hard bounds. Nevertheless, they give a clear hint of what information is the raw data contains and what you can conclude from the analysis. In the end, interactive analysis always boils down to getting confident conclusions.

Once Monte Carlo realizations of an object are obtained, these are propagated in subsequent operations and conversions. Such an object can be considered a stochastic variable where standard operations as `rand`, `mean` and `var` apply.

What you should be aware of is that Monte Carlo simulations take time. There is a property in all functions using Monte Carlo simulations called `MC`, which is an integer defining the number of Monte Carlo samples. The default is 30 in most functions. This is a fair tradeoff of processing time and information. Usually, when you are working interactively, the difference in time to run a transformation 31 times instead of 30 is not noticeable. The script's interpretation time is longer than the execution time for the computations.

You generally use Monte Carlo simulations in two ways:

- To represent ensemble variations over different realization of the noise processes
- To represent estimation uncertainty

Ensemble Uncertainty

When Monte Carlo samples of a stochastic signal are generated, several realizations of the driving noise are generated and simulated. For instance, for an ARMA model a nominal signal and N other realizations are simulated:

$$y[k] = \frac{C(q)}{A(q)}e[k],$$

$$y^{(i)}[k] = \frac{C(q)}{A(q)}e^{(i)}[k], i = 1, \ldots, N.$$

The resulting SIG object stores the result in the fields `y` and `yMC`, and the field MC contains the number of samples, N. The ensemble average of the signal itself is of little interest for stochastic models because the theoretical average is usually zero.

When the Signals & Systems Lab converts a signal to another representation, the conversion includes both the nominal signal and the Monte Carlo realizations. In this way, you can compute reliable confidence bounds. For instance, consider spectral estimation. The mapping from a signal to a spectral estimate is completely deterministic and can functionally be written as

$$\hat{\Phi}(f) = f(y) .$$

The Monte Carlo theory then provides an estimate of the spectrum's probability density function (PDF),

$$p(\Phi(f)) \approx \frac{1}{N} \sum_{k=1}^{N} \delta[f(y^{(i)})] .$$

Here $\delta(\theta)$ is the impulse response assigning infinite probability density to that point. It is easy to obtain a confidence bound for each frequency by sorting the Monte Carlo samples of the spectrum at that frequency.

Similarly, you can replace the spectral estimate by a covariance function, zero-poles, a frequency function (in particular amplitude and phase curves in Bode diagrams), root locus plots, and so on.

Model Uncertainty

An LTI model is characterized by its structural indices (model orders `nn`) and parameters θ. Uncertain models are characterized by an uncertain parameter vector. How is this uncertainty represented? For some cases, the parameter vector is Gaussian distributed, so the mean and covariance are sufficient quantities to store. The most essential examples are estimated AR and ARX models, which are linear operations on measurement data, where Gaussian noise or the central limit theorem can be used to justify a Gaussian distribution.

In the general case, however, Monte Carlo samples are stored inside the LTI object. The algorithm for model estimation of TF models includes two steps as the following equation shows:

$$\hat{\theta}_h = f(y(1:N)) \in N(\theta_h^o, P_h),$$
$$\hat{\theta} = g(\hat{\theta}_h).$$

The first step maps the data vector linearly on a high-dimensional model whose parameters are assumed Gaussian distributed (first equation). The second step performs a simulation and estimation, which can be seen as a deterministic mapping of the high-order parameter vector to the low-order sought parameter vector (second equation).

The Monte Carlo principle suggests that samples from the Gaussian distribution are propagated by the nonlinear mapping to get an approximative probability density function (PDF):

$$\theta_h^{(i)} \sim N(\theta_h^o, P_h),$$
$$p(\theta) \approx \frac{1}{MC} \sum_{i=1}^{MC} \delta(\theta - g(\theta_h^{(i)})).$$

One frequently used alternative is based on a first-order Taylor expansion of the nonlinear mapping, which yields an approximate expression for the covariance matrix sometimes referred to as *Gauss approximation formula*:

$$\theta = g(\theta_h) \approx g(\hat{\theta}_h) + g'_\theta(\hat{\theta}_h)(\theta_h - \hat{\theta}_h) \Rightarrow$$
$$\mathrm{Cov}(\hat{\theta}_h) \approx g'_\theta(\hat{\theta}_h)\mathrm{Cov}(\hat{\theta}_h) g'_\theta(\hat{\theta}_h)^T$$

This works very well asymptotically when the estimation error becomes negligible and thus the Taylor expansion is more accurate. For small data sizes, the Monte Carlo approach might be more appropriate. At least it avoids confidence regions of amplitude curves that cover negative values and zeros/poles outside the unit circle for ARMA model estimation. However, the most important reason to use a Monte Carlo representation of uncertainty is its ability to propagate in further operations on the object, such as connection and feedback of LTI system blocks with different kind of uncertainty. It also gives the option of specifying regions with finite support for critical parameters in a model such as $|\Delta| < 1$ in a control system.

2

Signal Processing

This chapter focuses on signal representations and signal-processing applications.

- "The SIG Object" on page 30 provides information about the SIG object, which is the central object for signal representation in the Signals & Systems Lab.
- "The FT Object" on page 52 describes how the Fourier transform is approximated and represented.
- "Filtering and Filter Design" on page 60 describes the tools for designing filters.
- "Spectral Analysis" on page 69 shows how to use the Signals & Systems Lab for spectral analysis.
- "Covariance Function Analysis" on page 79 describes how to work with the tools for covariance function analysis.

The SIG Object

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

- "Fields in the SIG Object" on page 30 provides an overview of the SIG object.
- "Creating SIG Objects" on page 31 describes how to create a SIG object from a vector.
- "Data Preprocessing" on page 38 shows how to preprocess signals and overviews the types of real and artificial signals.
- "Examples of Real Signals" on page 44 introduces the signals in the database of real signals that is included in the Signals & Systems Lab.
- "Standard Signal Examples" on page 48 presents some of the benchmark examples in the Signals & Systems Lab.

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 $y(t_k)$ (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.

Creating SIG Objects

The SIG constructor accepts inputs as summarized in Table 2-1:

TABLE 2-1: SIG CONSTRUCTOR SYNTAX

`sig(y,fs)`	Uniformly sampled time series y[k]=y(k/fs)
`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

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.

TABLE 2-2: METHODS FOR SIG SYSTEM OBJECTS

`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

The overloaded operators are summarized in Table 2-3.

TABLE 2-3: OVERLOADED OPERATORS

OPERATOR	**DESCRIPTION**	**EXAMPLE**
plus	Adds a constant, a vector, another signal, or noise from a PDFCLASS object	y=sin(t)+1+expdist(1)
minus	Subtracts a constant, a vector, another signal, or noise from a PDFCLASS object	y=sin(t)-1-expdist(1)
times	Multiplies a constant, a vector, another signal, or noise from a PDFCLASS object	y=udist(0.9,1.1)*sin(t)
rdivide	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.	y=sin(t)/2
mean/E	Returns the mean of the Monte Carlo data	y=E(Y)
std	Returns the standard deviation of the Monte Carlo data	sigma=std(Y)
var	Returns the variance of the Monte Carlo data	sigma2=var(Y)

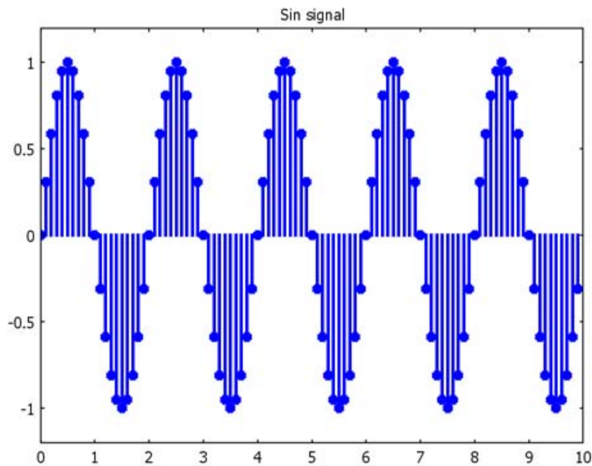
TABLE 2-3: OVERLOADED OPERATORS

rand	Returns one random SIG object or a cell array of random SIG objects	y=rand(Y,10)
fix	Removes the Monte Carlo simulations from the object	y=fix(Y)

DEFINING DISCRETE-TIME SIGNALS

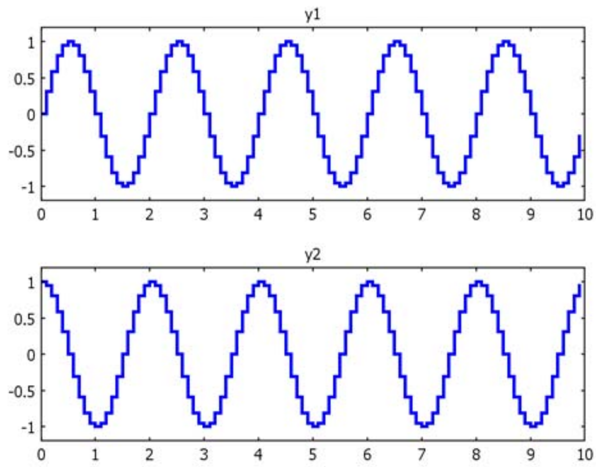
Scalar discrete-time signals are defined by a vector and the sampling frequency:

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



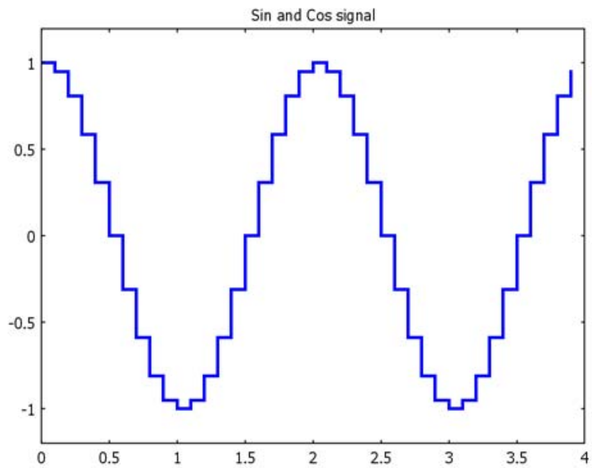
You define a multivariate signal in a similar way using a matrix where time is the first dimension:

```
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))
```



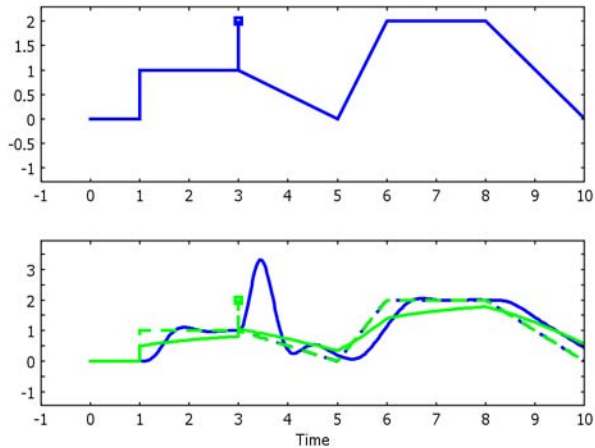
DEFINING CONTINUOUS-TIME SIGNALS

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:

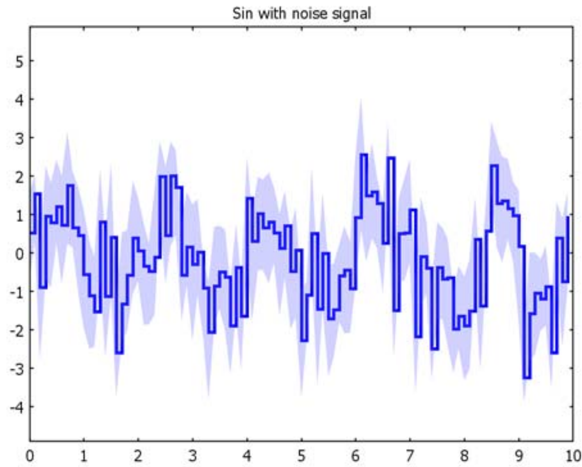
```
t=   [0 1 1 3 3 3 5 6 8 10]';
uvec=[0 0 1 1 2 1 0 2 2  0]';
u=sig(uvec,t);
G1=getfilter(4,1,'fs',NaN);
G2=tf([0.5 0.5],[1 0.5]);
y1=simulate(ss(G1),u)
SIG object with continuous time input-output state space data
  Name:        Simulation of butter filter of type lp
  Sizes:       N = 421,  ny =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



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 & Systems Lab is to use a PDFCLASS object for the stochastic part:

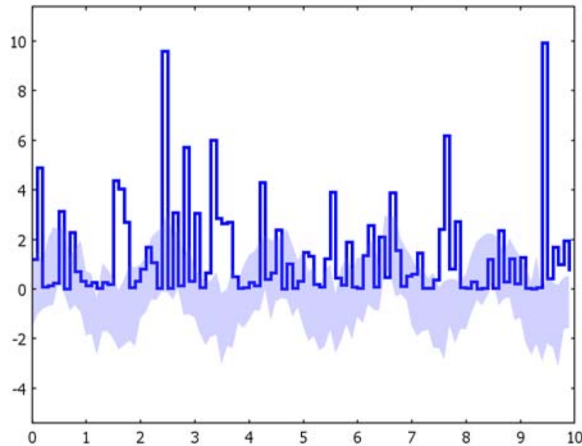
```
N=100;
fs=10;
t=(0:N-1)'/fs;
yvec=sin(pi*t);
y=sig(yvec,fs);
V=ndist(0,1);
yn=y+V;  % One nominal realization + 20 Monte Carlo realizations
yn.name='Sin with noise signal';
y.MC=0;
yn1=y+V;  % One realization
plot(yn,'conf',90)
```



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)
```



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. Examples of such operations appear n the Quick Tour in "Signal Preprocessing and Frequency Analysis" on page 5, and this section contains additional information and examples. The following table summarizes the SIG object's methods discussed in this section:

TABLE 2-4: DATA PREPROCESSING 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

To illustrate the different kind of data operations available, assume that a nonuniformly sampled signal $y(t_k)$ 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 $y(t_k)$ 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);`

- *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 $y[l] = y(lnT/m)$, $l = 1, 2,..., \text{ceil}(mN/n)$.

- As a special case of the previous operation, *decimation* decreases the sampling frequency 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);
  ```

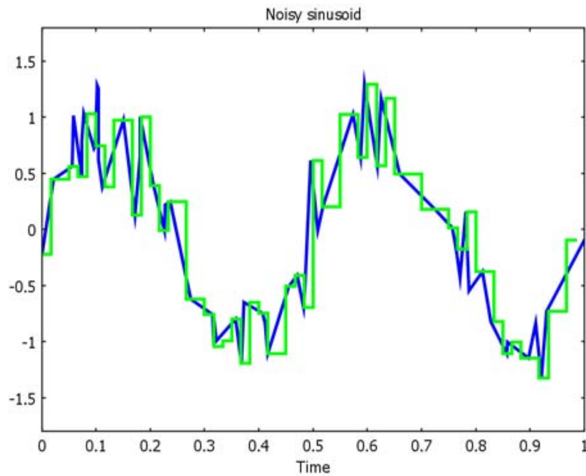
  The low-level `filter` function is called inside the TF (transfer function) method `filter`. The main difference is that the TF method filters each signal in a multivariate signal object individually.

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';
```

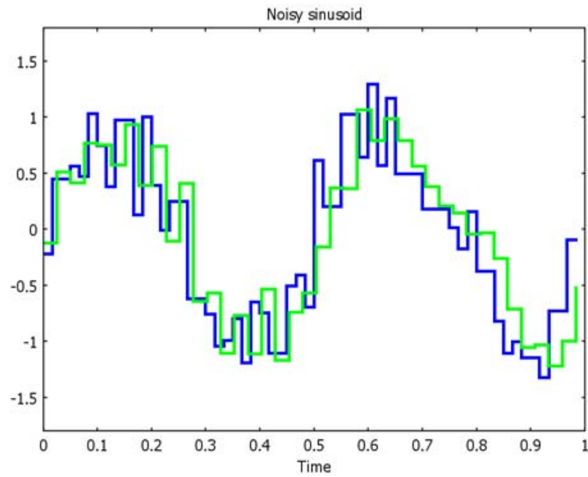
The signal is linearly interpolated at 30 equidistant time instants,

```
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 & Systems Lab automatically uses an antialias filter).

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



The signal is now prewindowed by a Hamming window,

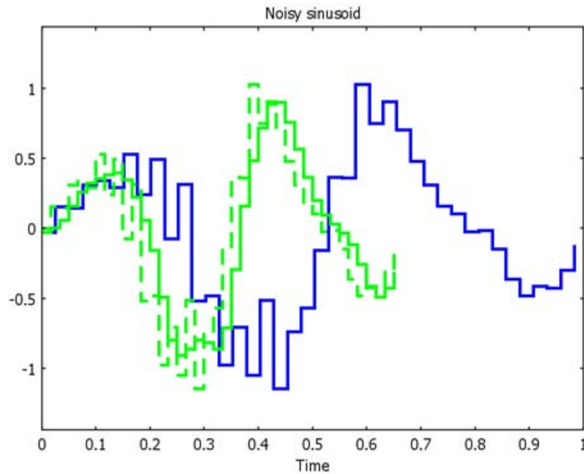
```
y3=window(y2,'kaiser');
```

and low-pass filtered by a Butterworth filter:

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

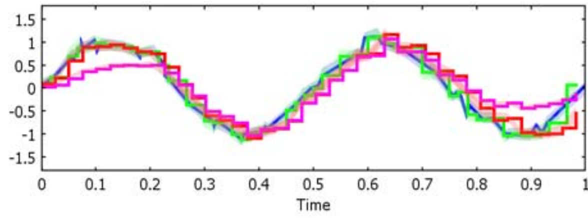
Finally, compare all the signals:

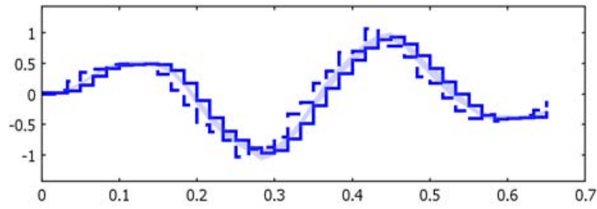
```
plot(y3,y4)
```



To illustrate the use of Monte Carlo simulations, the next code segment defines a stochastic signal. It does so by adding a PDF 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.

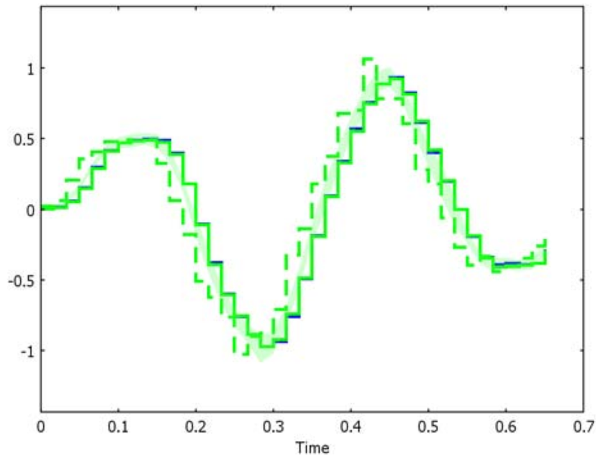
```
yt=sin(4*pi*t);
y=sig(yt,t);
y=y+0.1*ndist(0,1);
fs=N/(t(end)-t(1));
y1=sample(y,fs);
y2=resample(y1,3,2);
y3=window(y2,'kaiser');
y4=filter(G,y3);
subplot(2,1,1), plot(y,y1,y2,y3,'conf',90)
subplot(2,1,2), plot(y4,'conf',90)
```





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)
```



Conversions to Other Objects

TABLE 2-5: SIGNAL CONVERSION ALGORITHMS

`sig2ft`	Compute the Fourier transform (approximation) of a signal
`sig2covf`	Estimate the (cross-)covariance function of (the columns in) a signal
`sig2spec`	Perform spectral analysis of a signal

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

- Fourier transform (FT)
- Covariance function (COVF)
- 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 TF, SS, and ARX models are contained in the corresponding objects.

Examples of Real Signals

The Signals & Systems Lab contains a database, `dbsignal`, with real-world application data as summarized in the following table:

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.

NAME	DESCRIPTION
`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.

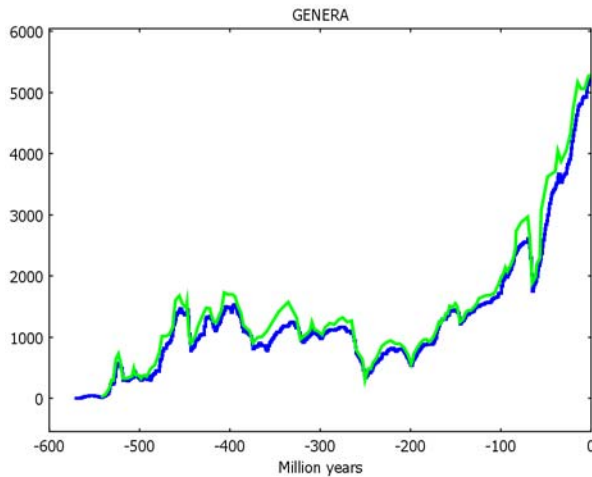
Each example contains a data file and one M-file, which in turn contains a brief explanation of the data in the help text and provides some initial illustrations of the data. By just calling the function, the Signals & Systems Lab shows both the info and a plot.

Now return to the examples in “Signal Preprocessing and Frequency Analysis” on page 5:

```
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:
   y:  # genera
Time:  Million years
```

There are two signals in the `genera` file: `y1` and `y2`. The first is the original data, which is nonuniformly sampled. You learned how to interpolate this signal in the “Signal Preprocessing and Frequency Analysis” on page 5. The second signal is resampled uniformly by stochastic resampling techniques. The following plot illustrates these signals.

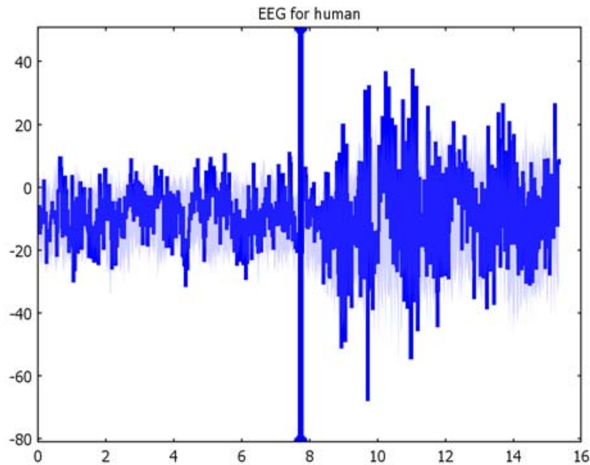
```
plot(y1,y2)
```



Note that the `sigplot` function 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:

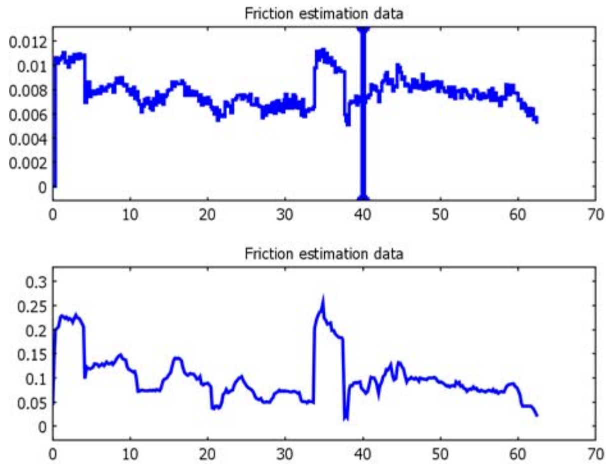
```
load eeg_human
info(y)
Name:    EEG for human
Signals:
   y1:  y
Time:  s
plot(y,'conf',90)
```



The visualization of markers in `sig.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 rhythm starting directly after the light is turned off (see “Tutorial” on page 53).

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 estimation data
Signals:
   u:  Normalized traction torque
   y:  Slip
subplot(2,1,1), plot(y(:,:,[]))  % Only output
subplot(2,1,2), uplot(y)         % Only input
```



Standard Signal Examples

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

DISCRETE-TIME SIGNALS

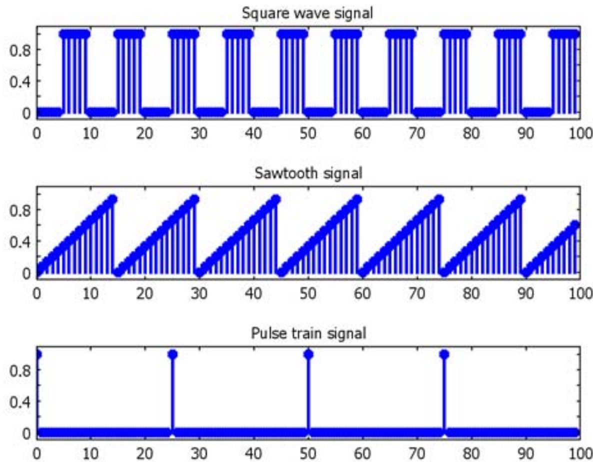
The following table summarizes the options for discrete-time signals:

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

EXAMPLE	DESCRIPTION
`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^2) with t=k*T where T=opt1 and a=opt2

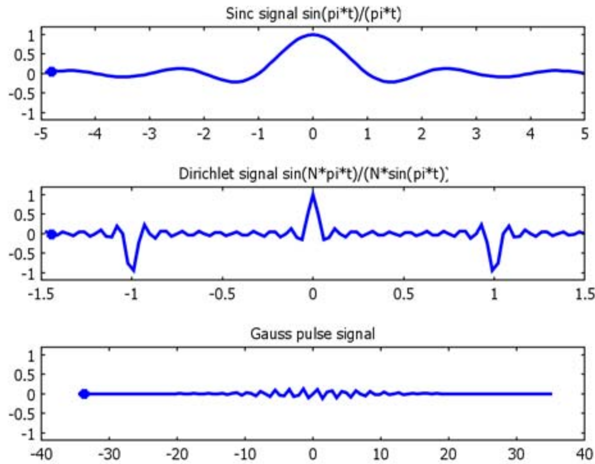
One category of signals contains periodic functions:

```
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)
```



The second kind of signals contains the following window-shaped oscillating 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)
```



There are also a few other signals that come with the Signals & Systems Lab for demonstration purposes.

`sin1`	One sinusoid in noise
`sin2`	Two sinusoids in noise
`sin2n`	Two sinusoids in low-pass filtered noise

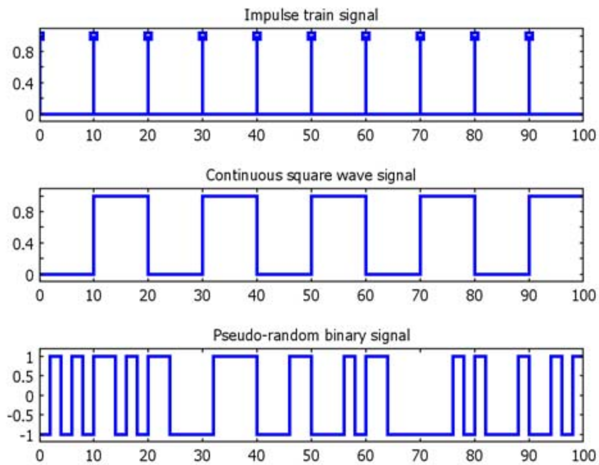
CONTINUOUS-TIME SIGNALS

The following table lists the continuous-time signals available in `getsignal`:

EXAMPLE	DESCRIPTION
`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)

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)
```



The FT Object

Introduction

You compute a Fourier Transform (FT) object from a SIG object in this way: `Y=ft(y)`. This basically involves adding zero padding and then calling the built-in function `fft` to compute the discrete Fourier transform. Also, Monte Carlo realizations of the FT are computed from the Monte Carlo realizations of the signal, if available. See the definition of `sig` on page 113 in the *Signals & Systems Lab Reference Guide* for more information.

The Fourier transform is represented by a transform vector and a frequency vector with the field names `Y` and `f`, respectively. Monte Carlo samples are stored in a matrix with field name `YMC`. The plot method uses the sampling frequency for correct axis scalings. The direct way to construct an FT object is

```
Y=ft(Y,f,YMC)
```

Here, `Y` is of size `(nf,ny)`, `f` is `nf`, and `YMC` is of size `(MC,nf,ny)`.

The usual plot variants are overloaded (see `ft.plot` on page 39 in the *Signals & Systems Lab Reference Guide* for further information).

PLOT FUNCTION	DESCRIPTION
`plot`	Plot with linear axes
`loglog`	Plot with logarithmic axes
`semilogy`	Plot with linear frequency axis and logarithmic amplitude axis
`semilogx`	Plot with logarithmic frequency axis and linear amplitude axis

You can use the overloaded operators in the following table to further control what is plotted using, for instance, `plot(angle(Y))`:

OPERATORS	DESCRIPTION
`abs`	Absolute value
`angle`	Angle, or phase
`real`	Real part
`imag`	Imaginary part

Furthermore, indexing an FT object with `Y(freqind,yind)` selects the frequency indices in `freqind` and the subsignal(s) in `yind` for multivariate signals.

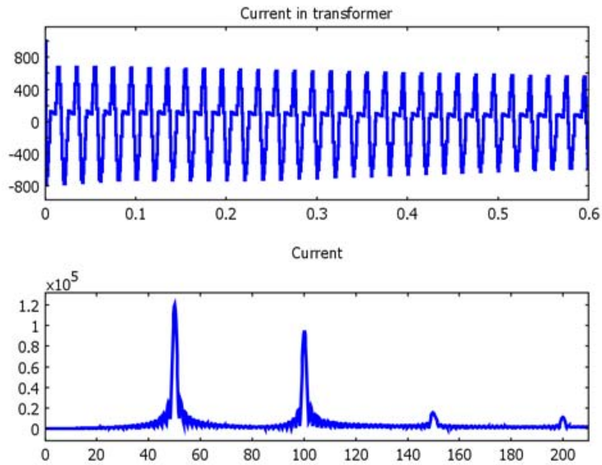
When an FT is computed from a stochastic process represented by a SIG object with Monte Carlo data, these random realizations are propagated to random realizations of the FT object contained in the field `YMC`. That is, the Fourier Transform object can be regarded as a stochastic variable at each frequency. Using the `plot` function, you can add confidence bounds or a scatter plot of the realizations. Further, the following operations are possible:

mean/E	Returns the mean of the Monte Carlo data	E(Y)
std	Returns the standard deviation of the Monte Carlo data	std(Y)
var	Returns the variance of the Monte Carlo data	var(Y)
rand	Return one random FT object or a cell array of random FT objects	rand(Y,10)
fix	Remove the Monte Carlo simulations from the object	fix(Y)

Tutorial

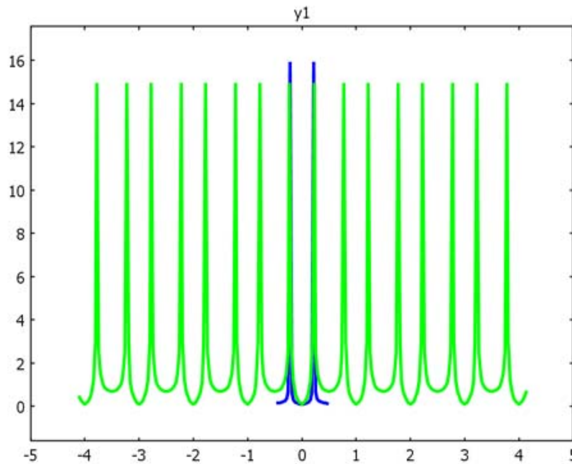
A first standard example of Fourier analysis is to find the harmonics in periodic signals. The demonstration signal `current` is used here for this purpose. The variation in current amplitude has one dominating harmonic component as the following plot reveals. Note that the frequency axis is correctly graded in Hz when you have set the sampling frequency correctly in the SIG object.

```
load current
info(y)
Name:    Current in transformer
Description:
The data describes the current in a power transformer after a
shortcut in the power network
Signals:
   y:  Current
subplot(2,1,1)
plot(y)
Y=ft(y);
subplot(2,1,2)
plot(Y,'Xlim',[0 210])
```



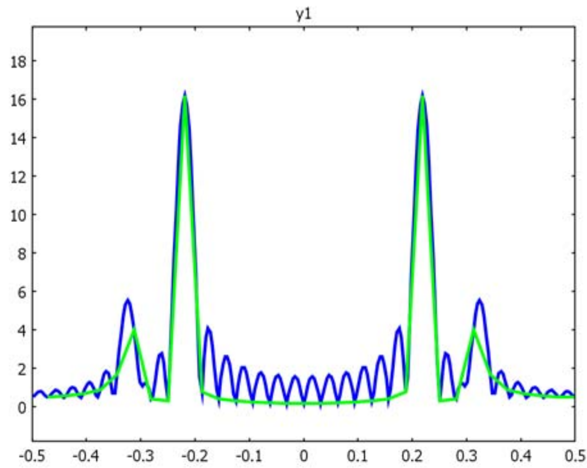
A second standard example in Fourier analysis is to find the frequencies of pure sinusoidal components in signals. The following code segment first defines a SIG object as having a pure sinusoid and then adds more frequency components and noise. It then constructs an FT object in two ways: the direct one using the FT constructor and the more convenient one based on the conversion implemented in `sig.sig2ft`:

```
t=(0:31)';
f1=0.22;
y=sin(2*pi*f1*t);
Y=fft(y);
Y1=ft(Y(1:16),(0:15)/32);     % Direct definition
Y2=ft(sig(y,t));              % Conversion
plot(Y1,Y2)
```



There is an important difference in the results. The `fft` function implements the standard DFT resulting in as many frequency grid points as there are data points. The well-known leakage phenomenon resulting from using data batches of finite length then leads to an ambiguity where the result can be difficult to interpret. A well-known remedy is to apply zero padding by trailing the signal with a large number of zeros. In this way, the discrete-time Fourier transform (DTFT) is obtained on a much denser frequency grid, revealing the window effects of the finite data size. This is in some ways a better approximation of the underlying Fourier transform. At least it is easier to interpret in the case of more frequency components. To see this, add another nearby frequency component to the signal:

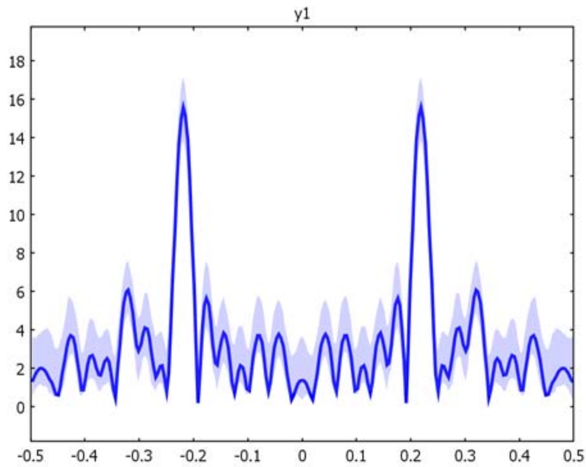
```
f1=0.22;
f2=0.32;
yvec=sin(2*pi*f1*t)+0.3*sin(2*pi*f2*t);
y=sig(yvec,1);
Y1=ft(y);
Y2=ft(y,'Nf',32);
plot(Y1,Y2)
```



Here the FT constructor computes both the DFT and the DTFT. It is easy to distinguish the weaker frequency component from the side lobes of the data window using the DTFT.

Finding signals in noise is another classical signal-processing task. Next, add Gaussian noise to the signal:

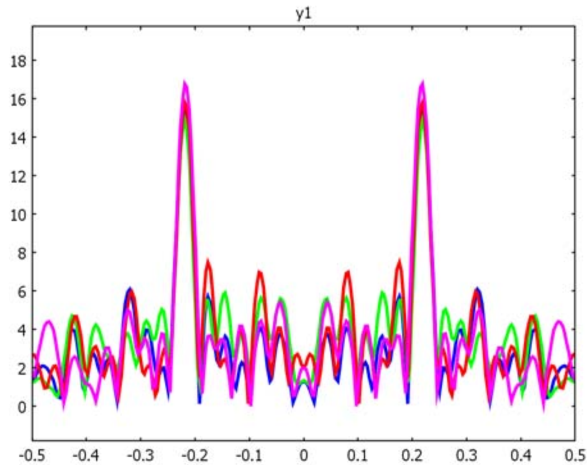
```
y.MC=100;
yn=y+ndist(0,0.1);
Yn=ft(yn);
plot(Yn)
```



Monte Carlo realizations of the signal propagates to the FT object. This example explicitly changes the number of Monte Carlo simulations to 100. The plot function illustrates the stochastic uncertainty in the result with a confidence bound. This bound covers 90% of all Fourier transforms computed from different noise realizations in the signal.

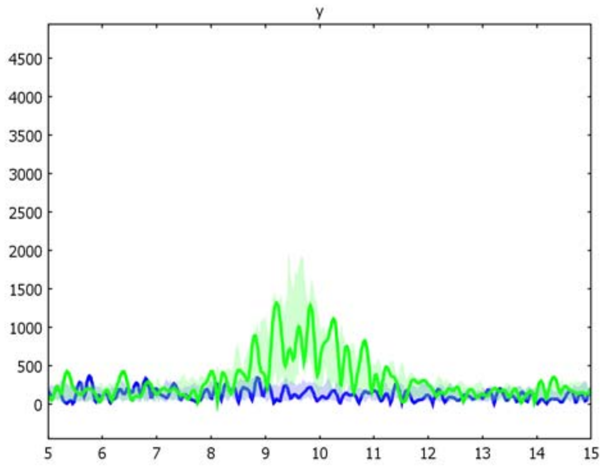
Another way use Monte Carlo samples is to compute the mean, standard deviation, and the variance of the Fourier transform. The next example generates three random FT objects and compares them to the expected value.

```
Ynrand=rand(Yn,3);
plot(E(Yn),Ynrand{:})
```



As an illustration taken from practice, consider the EEG signal presented in "Examples of Real Signals" on page 44. The 18 individuals (one signal corresponds to the nominal signal and the other 17 to Monte Carlo realizations) get an excitation at sample time 387, when a light is turned off. The hypothesis is that there is a 10 Hz rest rhythm in the dark, whose magnitude indicates certain diseases. The confidence bound of the two segments in data show a significant difference.

```
load eeg_human
Y1=ft(y(1:387));
Y2=ft(y(388:end));
plot(Y1,Y2,'Xlim',[5 15])
```


y
4500
4000
3500
3000
2500
2000
1500
1000
500
0
5
6
7
8
9
10
11
12
13
14
15

Filtering and Filter Design

Introduction

The filtering part of the Signals & Systems Lab covers the following aspects:

- Approximations of ideal filters of these types: low-pass (LP), high-pass (HP), band-pass (BP) and band-stop (BS). You can do this either by computing a transfer function in the time domain followed by filtering, or in the frequency domain by zeroing the transform values in the stop band.
- Implementation of noncausal filters. This includes using an arbitrary filter as a zero-phase filter by forward-backward filtering, and stable implementation of arbitrary filters (with poles anywhere).
- Conversion between continuous-time and discrete-time filters.
- Conversion between transfer function, state-space model, and zero-pole gain representations of a given filter.

The Signals & Systems Lab in its current version does not cover realization aspects of filters and relies on the standard direct form IIR structure implemented in `filter`. However, the software covers some useful noncausal implementations as summarized in the following table:

TABLE 2-6: NONCAUSAL IMPLEMENTATIONS

LOW-LEVEL CODE	TF METHODS	DESCRIPTION
`y=filter(b,a,u)`	`ysig=filter(tf(b,a),sig(u))`	Basic filtering
`y=filtfilt(b,a,u)`	`ysig=filtfilt(tf(b,a),sig(u))`	Noncausal forward-backward filtering
`y=ncfilter(b,a,u)`	`ysig=ncfilter(tf(b,a),sig(u)`	Noncausal stable implementation of arbitrary filters

A filter is denoted $H(s)$ in the continuous time and $H(z)$ in the discrete-time domains, respectively. They can be represented by their numerator polynomial `b` and denominator polynomial `a`. However, the recommended use is to define a TF object for the filters or the corresponding state-space SS object (see “Deterministic LTI

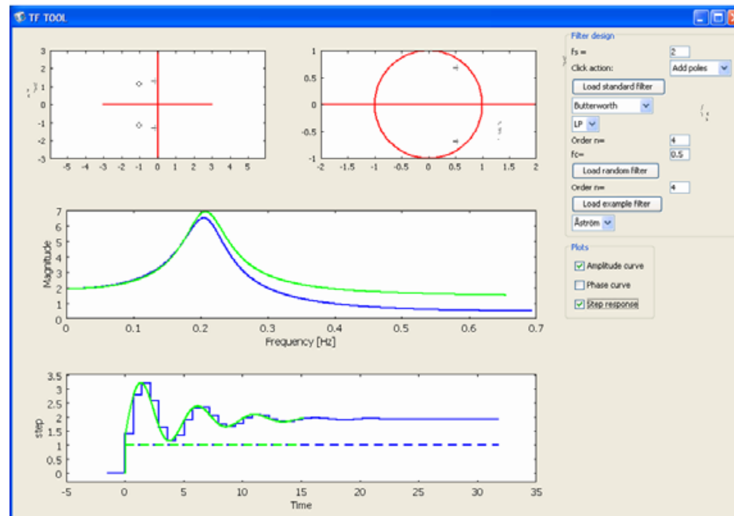
Objects” on page 86). The main reasons for using these methods rather than the functions are when

- The signal is multivariate or the filter is MIMO
- The signal is stochastic and contains Monte Carlo realizations
- The filter is uncertain
- Information about the signal (name, time scale, markers, and so on) has been defined

The following sections contain examples of how to use these functions.

The TFTOOL Graphical User Interface

The graphical user interface `tftool` has, as do the other graphical tools, two modes: Analysis and Training.



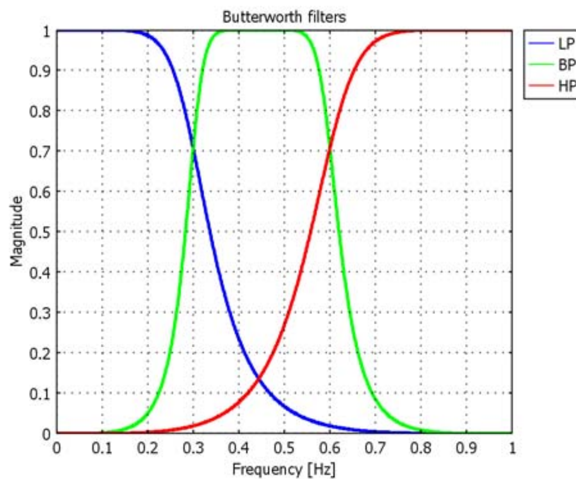
In the Analysis mode, you provide the transfer-function polynomials as inputs in this way: `tftool(b,a,fs)`. The GUI then illustrates the poles and zeros in both the domain of the filter and its complement (continuous time or discrete time). When you start the GUI, the default view shows the complex plane for the poles and zeros of $H(s)$ and $H(z)$, respectively. You can compare the Bode diagram and step response in the two domains by clicking the appropriate check box. You can then move the zeros and poles and investigate sensitivity in the parameters and how they influence the transfer function’s properties.

In the Training mode, you design a filter yourself by placing zeros and poles in your choice of domain. You can mix the design by adding, for instance, the first pole in the continuous-time domain and the next one in the discrete-time domain.

Filter Design

The following command sequence generates and plots a filter bank with one low-pass, one band-pass, and one high-pass filter:

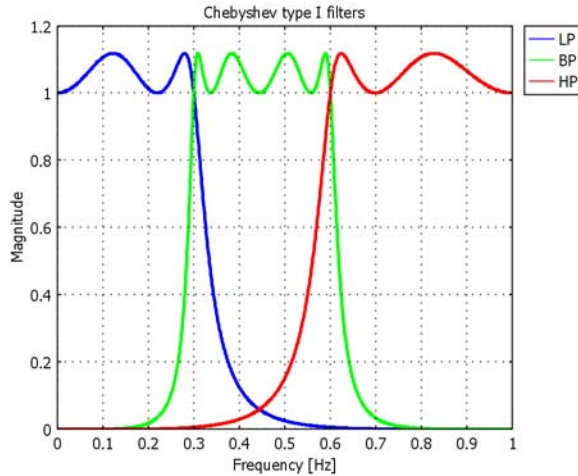
```
g1=getfilter(4,0.3,'type','LP');
g2=getfilter(4,[0.3 0.6],'type','BP');
g3=getfilter(4,0.6,'type','HP');
plot(g1,g2,g3,'plottype','plot')
legend('LP','BP','HP')
title('Butterworth filters')
grid('on')
```



The default filter type is Butterworth. Chebyshev filters of type I with ripple parameter 0.5 work analogously:

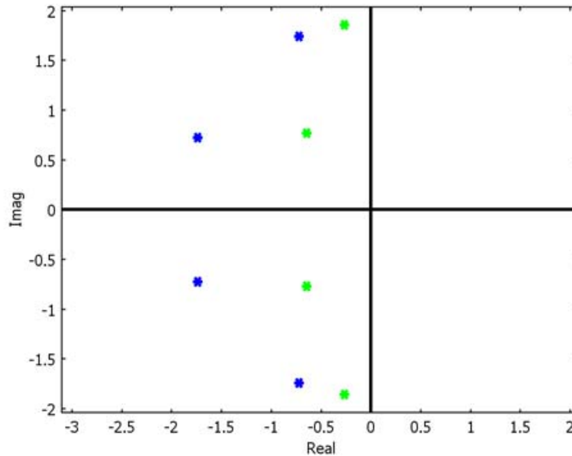
```
g1=getfilter(4,0.3,'type','LP','alg','cheby1','ripple',0.5);
g2=getfilter(4,[0.3
0.6],'type','BP','alg','cheby1','ripple',0.5);
g3=getfilter(4,0.6,'type','HP','alg','cheby1','ripple',0.5);
plot(g1,g2,g3,'plottype','plot')
legend('LP','BP','HP')
```

```
title('Chebyshev type I filters')
grid('on')
```



You create filters in the continuous-time domain. To illustrate the design process, use a zero-pole plot of the continuous time LP filters:

```
gbutter=getfilter(4,0.3,'type','LP','alg','butter','fs',NaN);
gcheby=getfilter(4,0.3,'type','LP','alg','cheby1','fs',NaN);
zpplot(gbutter,gcheby)
```



Basically, the algorithm (Butterworth or Chebyshev) provides a formula for how to position the continuous-time poles. Then a bilinear transformation `gd = c2d(gc,fs,'bilinear')` converts the filter to discrete time.

ZERO-PHASE FILTERING

It is possible to implement any discrete-time filter $H(z)$ as a zero-phase filter using the relation

$$H_{\text{zero - phase}} = H(z)H(1/z) = |H(z)|^2$$

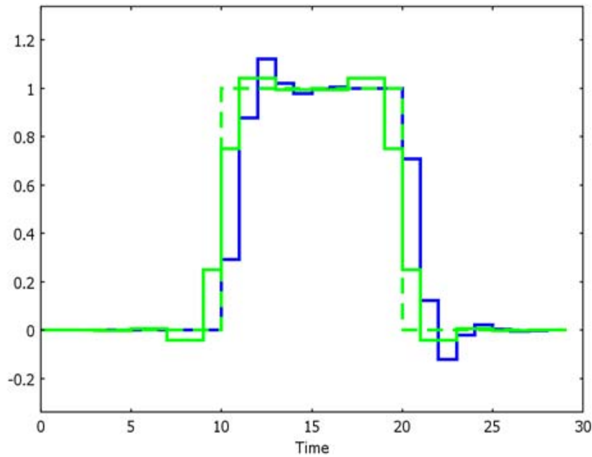
The last equality holds on the unit circle and shows that the resulting filter is real-valued and thus has zero phase. The trick in realizing this filter is to use $H(z)$ first forward and then backward in time ($1/z$ becomes z when time is reversed). The basic code to achieve this is

```
x=filter(b,a,u);
xr=x(end:-1:1);
yr=filter(b,a,xr);
y=yr(end:-1:1);
```

This usually works quite well. However, the order of forward and backward filtering is arbitrary, and it affects the transients. The function `filtfilt` also includes code to take care of nonzero initial conditions, which also removes the ambiguity of order of time reversal.

As an illustration, consider the low-pass filtering of a square wave. The method `filtfilt` in the TF object invokes the `filtfilt` function in the same way as the method `filter` invokes the `filter` function.

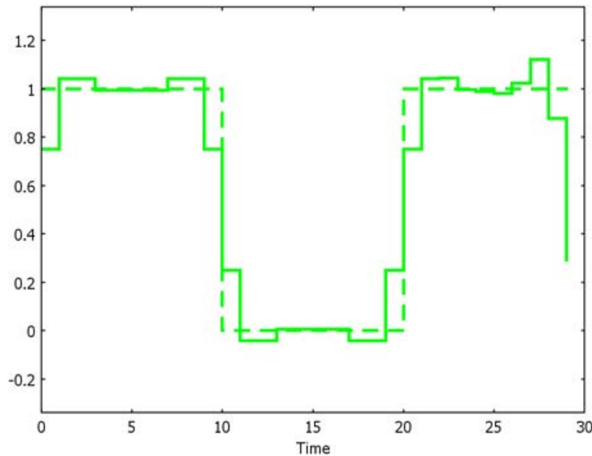
```
G=getfilter(2,0.5);
u=sig([zeros(10,1);ones(10,1);zeros(10,1)],1);
y1=filter(G,u);
y2=filtfilt(G,u);
plot(y1,y2)
```



The `filter` function gives a signal waveform that is delayed compared to the square wave. The delay is caused by the phase delay in the filter. The zero-phase filter gives a symmetric result.

If the square wave starts and ends with one instead of zero, the initial conditions in the filter become important.

```
u=sig([ones(10,1);zeros(10,1);ones(10,1)],1);
y1=filtfilt(G,u);
y2=filtfilt(G,u,5);
plot(y1,y2)
```



The extra argument `5` in the call for `y2` starts an estimation procedure that fits the initial conditions to get a symmetric result independent of the order of forward and backward filtering. Note that the transient of the blue curve is larger at the end in this case when filtering first forward and then backward in time. The reverse order would give the reverse behavior. The green curve is perfectly symmetric, however.

NONCAUSAL FILTERS

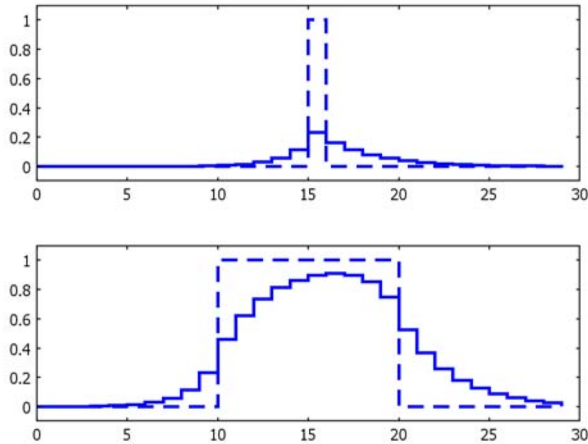
All filters are stable when implemented correctly. Given any transfer function, you can factorize it into one minimum phase and one maximum phase rational function,

$$H(z) = \frac{b(z)}{a(z)} = H_f(z)H_b(1/z) = \frac{b_f(z)b_r(z)}{a_f(z)a_r(z)}$$

The minimum-phase filter is implemented forward in time; the result is time reversed and then filtered with the maximum phase filter with all poles and zeros reflected in the unit circle (or imaginary axis for continuous-time filters), and finally the result is time reversed again.The code structure for this is very similar to the implementation of `filtfilt`. In fact, `filtfilt` is called after the factorization as `y=filtfilt(bf,af,br,ar,u,M)`. The factorization of the zeros is ambiguous because it does not affect the result (except for transient effects) or the stability of the filter, but `ncfilter` uses the minimum-phase principle.

The following example shows the realization of a filter with poles in 0.7 and 2 when applied to a pulse and a square wave:

```
b=-0.3;
a=poly([0.7 2]);
G=tf(b,a,1);
u1=sig([zeros(15,1);1;zeros(14,1)],1);
y1=ncfilter(G,u1);
subplot(2,1,1),
plot(y1)
u2=sig([zeros(10,1);ones(10,1);zeros(10,1)],1);
y2=ncfilter(G,u2);
subplot(2,1,2),
plot(y2)
```



WIENER FILTERING

The functions `filtfilt` and `ncfilter` are useful when implementing noncausal Wiener filters, and this section provides an example. First, a Wiener filter concerns the signal-in-noise problem, and the approach assumes known spectral models for signal and noise. Consider a signal defined by an AR(1) model

$$s(t) = \frac{0.6}{1 - 0.8q^{-1}} w(t)$$

with an observation in white noise $y(t) = s(t) + e(t)$. The Wiener filter to estimate the signal from observations is given by

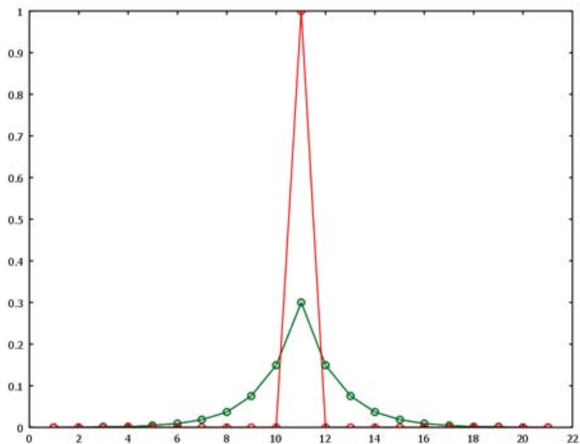
$$H(z) = \frac{\Phi_{sy}(z)}{\Phi_{yy}(z)} = \frac{\Phi_{ss}(z)}{\Phi_{ss}(z)+1} = \frac{-0.45z^{-1}}{1-2.5z^{-1}+z^{-2}} = \frac{\sqrt{0.225}\cdot\sqrt{0.225}}{(1-0.5z^{-1})(1-0.5z)}$$

The involved calculations are straightforward but quite tedious even in this simple case. You can identify the last expression as a zero-phase form of the first factor, which is a stable causal filter (a pole inside the unit circle). For such transfer functions you can use `filtfilt`:

```
a=[1 -0.5];
b=[sqrt(0.225) 0];
u=[zeros(10,1);1;zeros(10,1)];
y1=filtfilt(b,a,u);
```

A more straightforward way to realize a Wiener filter is based on the second to last expression in the equation. Using `ncfilter`, you do not need to factorize and symmetrize the expression, or even care whether the poles are inside or outside the unit circle.

```
a=[1 -2.5 1];
b=[0 -0.45 0];
y2=ncfilter(b,a,u);
plot([y1 y2 u],'-o')
```



Spectral Analysis

Introduction

The spectrum is defined as the Fourier transform of the covariance function:

$$\Phi(f) = \mathrm{FT}[R(\tau)]$$

The covariance function, in turn, is defined only for stationary stochastic processes:

$$R(\tau) = E[s(t)s(t-\tau)]$$

The *periodogram* is the basic input to spectral-estimation methods, and it can be equivalently defined in two different ways:

$$\hat{\Phi}(f) = \mathrm{FT}[\hat{R}(\tau)],\ \hat{R}(\tau) = \frac{1}{N}\sum_k s[k]s[k-\tau],$$

$$\hat{\Phi}(f) = \frac{T}{N}|TDFT[s[k]]|^2.$$

The three methods to smooth the periodogram are:

- Windowing the covariance-function estimate using a standard window of size M and type that you choose from the options in `window`. This is called the *Blackman-Tukey's method*.
- Segmenting the data into M segments, computing the periodogram on each one, and then averaging. This is referred to as the *Welch method*.
- Direct smoothing using a low-pass filter approximation and using `filtfilt` to perform noncausal zero-phase low-pass filtering to avoid frequency shifts in the spectral estimate. The low-pass filter is a running average of M samples, and after `filtfilt` i it becomes a triangular averaging window.

The design parameter you tune to trade off resolution to noise reduction is basically the same for all three methods: the number of elements in averaging, the size of the window, and the number of segments are all related. The design parameter M is therefore of the same order for all methods, but the results are not exactly the same.

Alternatively, if a known model generates the stationary process as filtered white noise, then the spectrum is given by

$$s[k] = H(q)e[k] \Rightarrow \Phi(f) = \sigma_e^2 |H(e^{i2\pi f}|^2 .$$

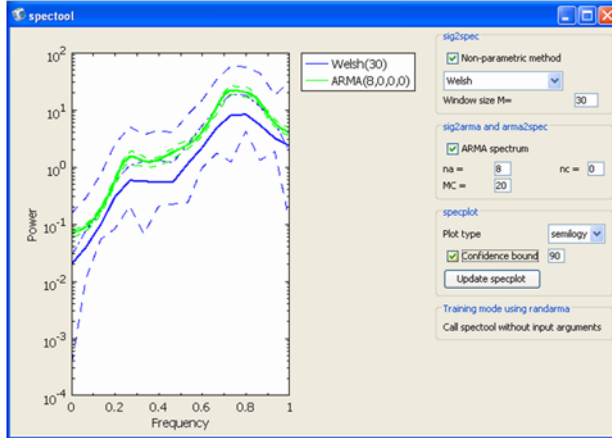
This opens up the possibilities for model-based approaches, where you estimate a model from the signal and then convert it to a spectrum.

When you compute the spectrum from a stochastic process represented by a SIG object that has Monte Carlo data, these random realizations are propagated to random realizations of the SPEC object contained in the field `PhiMC`. A similar situation occurs when you convert an uncertain model to a spectrum—each sample of the uncertain model is converted to a spectrum. In both cases you can regard the spectrum as a stochastic variable at each frequency. The `plot` function allows you to add confidence bounds or a scatter plot of the realizations. Further, the following operations are possible:

mean/E	Returns the mean of the Monte Carlo data	Phi=E(PHI)
std	Returns the standard deviation of the Monte Carlo data	Phi=std(PHI)
var	Returns the variance of the Monte Carlo data	Phi=var(PHI)
rand	Return one random SPEC object or a cell array of random SPEC objects	Phi=rand(PHI,10)
fix	Remove the Monte Carlo simulations from the object	Phi=fix(PHI)

The SPECTOOL Graphical User Interface

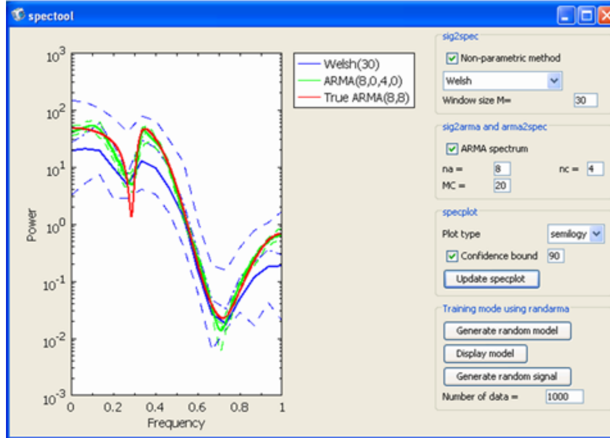
It is convenient to use a graphical user interface to apply and compare the transform- and model-based approaches to spectral analysis. If you have a signal `s`, type `spectool(s)`, and the following window appears after selecting the **ARMA spectrum** and **Confidence bounds** check boxes.



The Signals & Systems Lab can presently compute confidence bounds for Fourier transform-based methods only for the Welch method. It uses the variation between the different segments to estimate the confidence interval.

The confidence bound for the model is always much more narrow, which depends on the quite restrictive prior assumption that a model from the chosen structure (here AR(8)) can generate the signal. If this is the case, the model-based approach gives very accurate result, otherwise the bias in the estimate can be large.

If you do not provide any input arguments to `spectool`, it starts in the Training mode. Here you can generate random models and the realizations of random processes of specified length.



Tutorial

CONSTRUCTION

Spectral analysis attempts to decide how much energy in a signal that belongs to each frequency. To undertake such a study, first generate a stochastic stationary signal using an ARMA model at random.

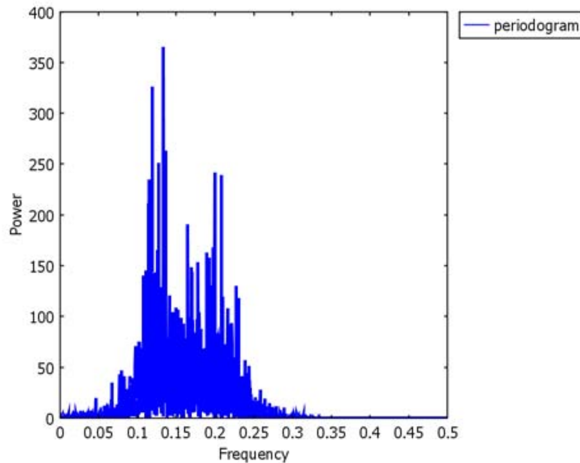
```
N=4096;
fs=2;
m0=rand(arx(8));
y=simulate(m0,N);
```

It is possible to compute the periodogram directly using the squared magnitude of the FFT:

```
Y=fft(y.y);
Y=Y(1:N/2+1);
f=(0:N/2)'/N*fs;
Phihathat=abs(Y).^2;
plot(f,Phihathat)
```

The alternative is to use the `spec` object constructor and the `plot` method:

```
Phihathat=spec(y,'method','periodogram');
plot(Phihathat);
```



The periodogram contains too much detail and must be smoothed if it is to clearly reveal the underlying spectrum. One standard method of doing so is Blackman-Tukey's method. It averages locally in the periodogram by windowing an estimate of the covariance function using a standard data window of width M. This is the basic design parameter to trade off details against averaging.

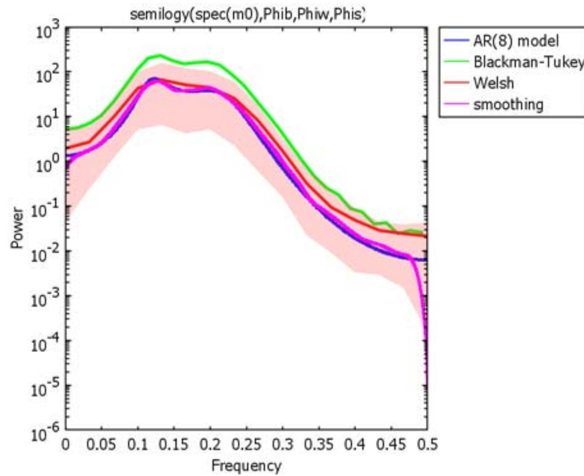
The other standard method is the Welch method. It first splits the signal into possibly overlapping segments, each one prewindowed. It then computes the periodogram over each segment and finally averages them. The different periodograms can be interpreted as Monte Carlo samples of the spectrum, so the Welch method also provides decent uncertainty regions.

A natural but nonstandard method is the direct smoothing of the periodogram. This method uses the `filtfilt` function to low-pass filter the periodogram.

The following code computes the different estimates for the same smoothing parameter, then it plots and compares all estimates together. The output in this case is a structure with a field `Phi` that contains the squared magnitude of the FFT as shown earlier. You can plot the periodogram by calling `spec` without output arguments.

```
M=30;
Phib=spec(y,'M',M,'method','blackman');
Phiw=spec(y,'M',M,'method','welch');
Phis=spec(y,'M',M,'method','smoothing');
```

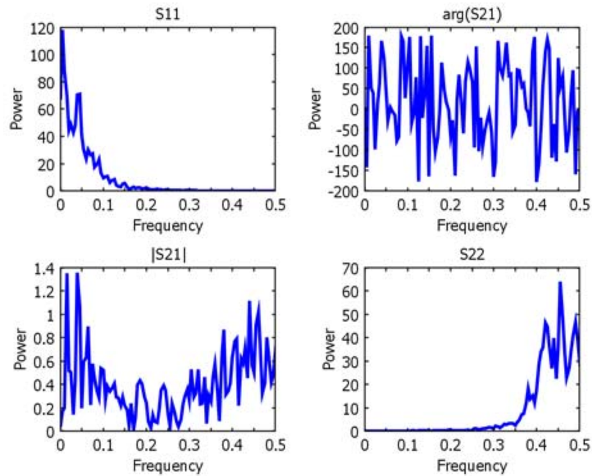
```
semilogy(spec(m0),Phib,Phiw,Phis);
```



MULTIVARIATE SIGNALS

A spectrum is defined as the Fourier transform of the covariance function of a signal. For multivariate signals, the covariance function is a square matrix of the same size as the signal dimension. The main diagonal $R_{ii}(t)$ is a symmetric function in t, so the Fourier transform is real-valued—that is, the diagonal elements $\Phi_{ii}(f)$ of the spectrum are all real-valued. The cross spectra corresponding to the cross-covariance functions are, however, complex-valued. The convention in the plot method is to plot the magnitude curves under the main diagonal and the phase curves above the main diagonal, as the following example illustrates:

```
ms=arx(4);
m1=rand(ms);
m2=rand(ms);
z1=simulate(m1,2000);
z2=simulate(m2,2000);
z=[z1 z2]; %MO
S=spec(z);
plot(S);
```

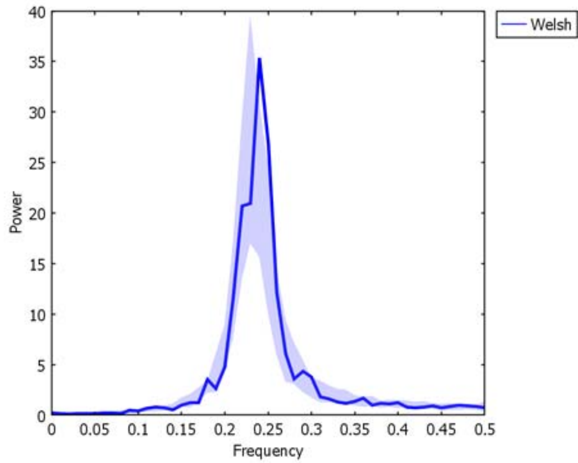


Because the two signals are generated independently, the cross spectrum shows no significant deviation from zero.

MONTE CARLO SIMULATIONS

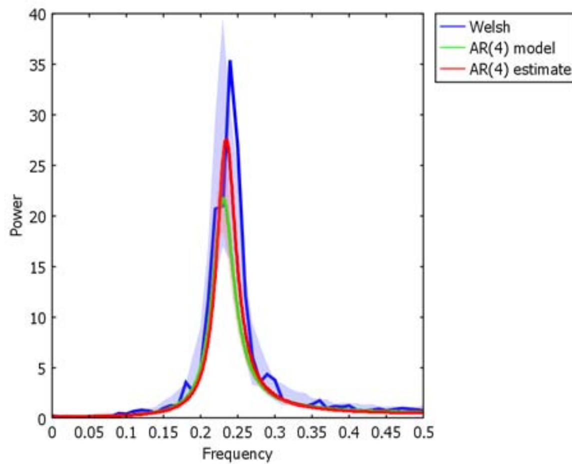
You can compute a spectrum from an ensemble of signal realizations by using the Monte Carlo simulation feature in the Signals & Systems Lab. To illustrate this, generate a random AR(4) model just as before, but explicitly change the field `MC` to 30. In this way, the `simulate` function generates a nominal signal and 30 other realizations corresponding to different realizations of the underlying noise. If the field `yMC` in the SIG object is not empty, the SPEC constructor (which calls `sig.sig2spec`) estimates one spectrum for each realization. The spectrum can now be considered as a stochastic variable, for which it is possible to compute the mean, standard deviation, and confidence intervals. The following example adds a confidence interval to the plot:

```
ms=arx(4);
m0=rand(ms);
m0.MC=30;
z=simulate(m0,1000);
S=spec(z);
plot(S);
```



When estimating a model from this data set, the model's uncertainty is represented by the different models that are estimated from each signal realization. The algorithm converts each model to one spectrum and computes the corresponding confidence bound.

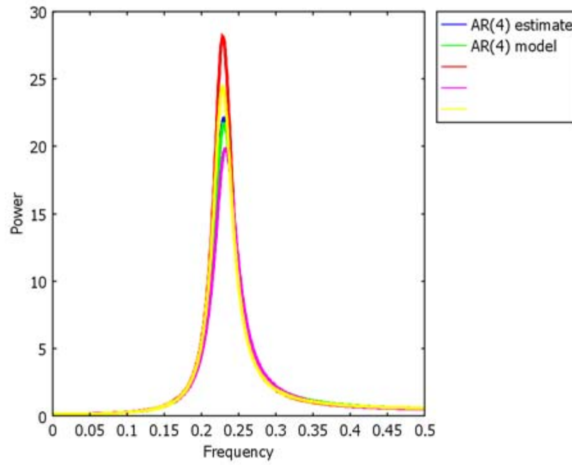
```
mhat=estimate(ms,z);
plot(S,m0,mhat);
```



Note here that only the first argument in the spectrum list to the plot must be of the correct object form (in order to find the correct plot method); the other ones are automatically converted.

Because you can now consider the estimated model-based spectrum now as a stochastic variable, it is allowed to apply standard operations such as `rand` and `E/mean`.

```
Shat=spec(mhat);
Srand=rand(Shat,3);
plot(E(Shat),m0,Srand{:});
```



Note that the spectrum's mean value, computed from 30 models estimated from 30 signal realizations, is very close to the true spectrum.

Covariance Function Analysis

Introduction

The covariance function describes how a stochastic signal correlates with itself, and it is defined for stationary stochastic processes as:

$$R(\tau) = E[s(t)s(t-\tau)]$$

The covariance function is closely related to spectral analysis because the spectrum is defined as the Fourier transform of the covariance function.

$$\Phi(f) = \mathrm{FT}[R(\tau)]$$

The Signals & Systems Lab represents the covariance function in the COVF object for discrete-time signals. The direct construction uses the syntax

```
R=covf(R,tau,RMC)
```

where each column of `R` contains one covariance function, `tau` is the vector of time lags, and `RMC` contains Monte Carlo realizations of the covariance function. Here, `R` equals `(ntau,ny*ny)`, `tau` is `(ntau,1)`, and `RMC` is `(MC,ntau,ny*ny)`.

Normally the covariance function is estimated from a signal using the unbiased estimate

$$\hat{R}_{ij}(\tau) = \frac{1}{N-|\tau|}\sum_{t=1}^{N-|\tau|} y_i(t)y_j(t-\tau)$$

For coherence in syntax throughout the Signals & Systems Lab, there are three different ways to do the same thing, as the following table summarizes:

`c=estimate(covf,y,Property1,Value1,...)`	Explicit call
`c=covf(y,Property1,Value1,...)`	Implicit call
`c=sig2covf(y,Property1,Value1,...)`	Direct low-level call

Alternatively, it is possible to compute the covariance function from a stochastic signal model, which can be certain or uncertain. In the latter case, MC data are generated in `RMC`. For stochastic state-space models, the deterministic part is neglected. The theoretical covariance function is computed using a state-space realization using the following algorithm:

1. R(0) = C * Pibar * C' + R
2. R(tau) = C * A^tau * Pibar * C' + C * A^(tau-1) * S, tau=1,2,...,taumax, where Pibar is the controllability Gramian (see `ss.gram` on page 155 in the *Signals & Systems Lab Reference Guide*)

You can invoke this algorithm in the following two ways:

`c=ss2covf(m,Property1,Value1,...)`	Explicit call
`c=covf(m,Property1,Value1,...)`	Implicit call

When the covariance function is computed from a stochastic process represented by a SIG object that has Monte Carlo data, these random realizations are propagated to random realizations of the COV object contained in the `RMC` field. A similar situation occurs when an uncertain model is converted to a covariance function. Each sample of the uncertain model is converted to covariance function. In both cases, the covariance function can be regarded as a stochastic variable for each time lag. The `plot` function allows you to add confidence bounds or a scatter plot of the realizations. Further, the following operations are possible:

mean/E	Returns the mean of the Monte Carlo data	c=E(C)
std	Returns the standard deviation of the Monte Carlo data	sigma=std(C)
var	Returns the variance of the Monte Carlo data	sigma2=var(C)
rand	Return one random COV object or a cell array of random COV objects	c=rand(C,10)
fix	Remove the Monte Carlo simulations from the object	c=fix(C)

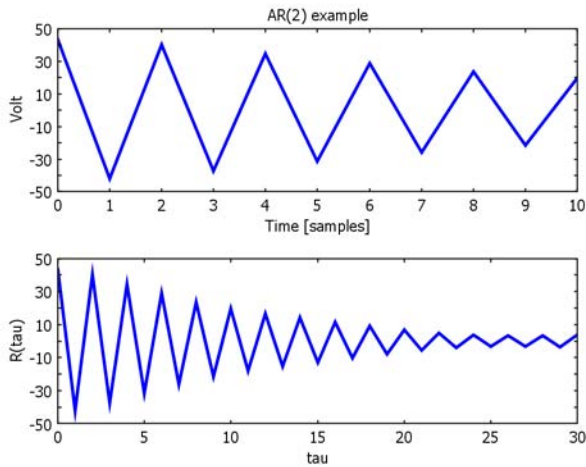
Tutorial

CONSTRUCTION

To illustrate the different ways to obtain a covariance function object, you here use a random AR(4) model to simulate data. From this data, compute the covariance function by direct definition and in the implicit way as implemented in the SIG object.

```
N=1000;
y=filter(1,[1 1.6 0.64],randn(N,1));
for tau=0:10;
    R(1+tau)=sum(y(1:N-tau)'*y(1+tau:N))/(N-tau);
    R(1+tau)=sum(y(1:N-tau)'*y(1+tau:N))/(N-tau);
```

```
    R(1+tau)=sum(y(1:N-tau)'*y(1+tau:N))/(N-tau);
    R(1+tau)=sum(y(1:N-tau)'*y(1+tau:N))/(N-tau);
    R(1+tau)=sum(y(1:N-tau)'*y(1+tau:N))/(N-tau);
    R(1+tau)=sum(y(1:N-tau)'*y(1+tau:N))/(N-tau);
    R(1+tau)=sum(y(1:N-tau)'*y(1+tau:N))/(N-tau);
    R(1+tau)=sum(y(1:N-tau)'*y(1+tau:N))/(N-tau);
    R(1+tau)=sum(y(1:N-tau)'*y(1+tau:N))/(N-tau);
    R(1+tau)=sum(y(1:N-tau)'*y(1+tau:N))/(N-tau);
    R(1+tau)=sum(y(1:N-tau)'*y(1+tau:N))/(N-tau);
end
c1=covf(R,0:10);
c1.name='AR(2) example';
c1.tlabel='Time [samples]';
c1.ylabel='Volt';
ysig=sig(y);
c2=covf(ysig);
subplot(2,1,1), plot(c1)
subplot(2,1,2), plot(c2)
```

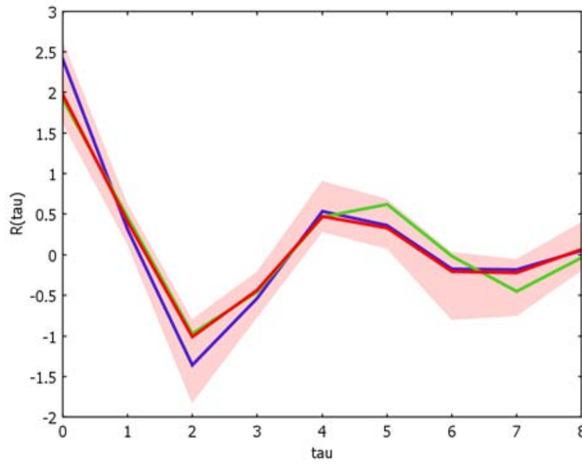


The default maximum time lag is set to 30, and this is the main difference in the plots shown here.

For comparison, plot the covariance function of the model as the "ground truth." To illustrate how to add uncertainty bounds, an AR(4) model is also estimated to the data, and its corresponding covariance function gets a confidence bound.

```
m0=rand(arx(4));
y=simulate(m0,100);
```

```
mhat=estimate(arx(4),y);
csighat=covf(y);
cmhat=covf(mhat);
c0=covf(m0);
plot(c0,csighat,cmhat)
set(gca,'Xlim',[0 8])
```



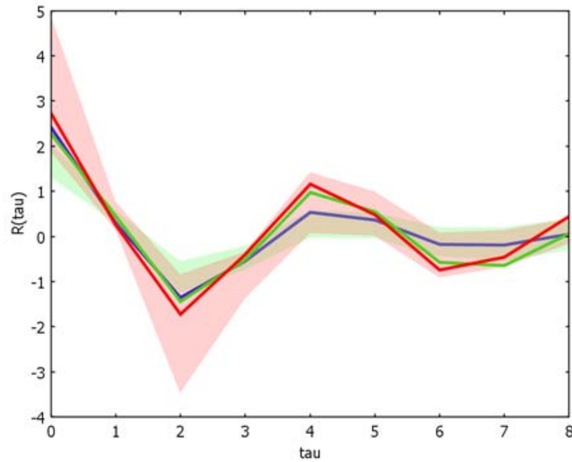
The bound shown here comes from the uncertain parameters in the model.

MONTE CARLO SIMULATIONS

Another kind of uncertainty is obtained from Monte Carlo simulations. By setting the MC field in the model object to 30, `simulate` automatically provides 30 realizations of the signal. These realizations are all converted to a covariance function by `covf(y)`, and you can compute a confidence interval (the blue area in the following plot). The model in this case is estimated from each of the 30 realizations, and the uncertainty is now represented by 30 parameter vectors rather than the covariance matrix of the parameter vector, as was the case previously. Again, each of these 30 models is converted to a covariance function, and a you can find a confidence bound (the red area in the following plot).

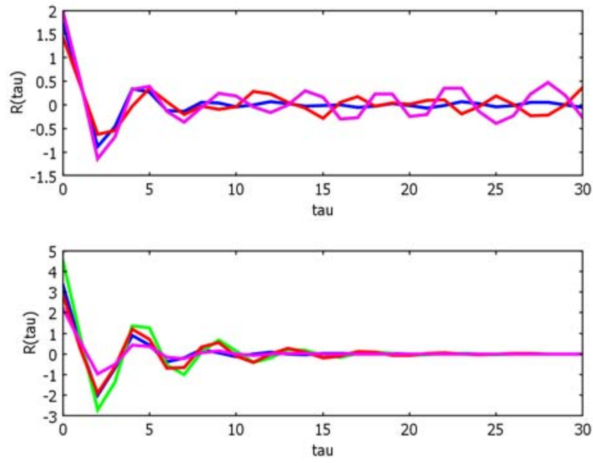
```
m0.MC=30;
yMC=simulate(m0,100);
mhatMC=estimate(arx(4),yMC);
csighatMC=covf(yMC);
cmhatMC=covf(mhatMC);
```

```
c0=covf(m0);
plot(c0,csighatMC,cmhatMC)
set(gca,'Xlim',[0 8])
```



Because you can consider the estimated covariance function as a stochastic variable, it is permissible to use standard operations such as `rand` and `E/mean`.

```
csigrand=rand(csighatMC,3);
cmrand=rand(cmhatMC,3);
subplot(2,1,1), plot(E(csighatMC),csigrand{:})
subplot(2,1,2), plot(E(cmhatMC),cmrand{:})
```



Note that the mean values of the covariance function, computed from 30 models estimated from 30 realizations of the signal (first subplot) and directly from 30 realizations of the signal (second subplot), should be very close to the true spectrum.

3

Systems and Control

The central theme in this chapter is on models for representing systems. The focus in this version of the Signals & Systems Lab is on linear time-invariant (LTI) models. To represent linear time-varying (LTV) models, use the RARX model. The LTI models are represented as transfer functions (TF) or the more general state-space model (SS). The latter includes a stochastic framework. Both have full support for multiple-input multiple-output (MIMO) systems.

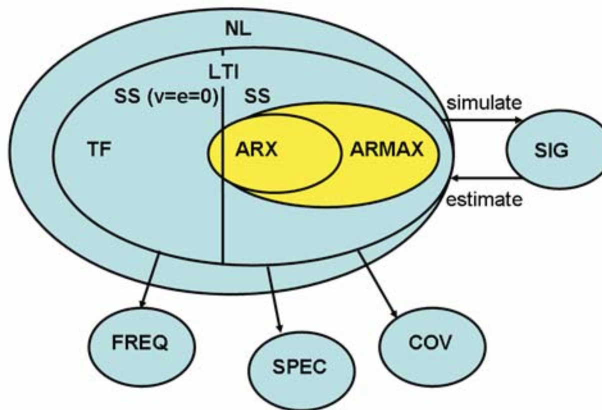
- For a presentation of the TF and nonstochastic SS objects, see “Deterministic LTI Objects” on page 86. This section describes how to generate models and use the visualization tools.
- “Application Examples” on page 119 describes a couple of application examples.
- “Uncertain Systems” on page 131 describes how to obtain uncertain systems, how the Signals & Systems Lab represents them, and how to visualize uncertainty.
- For a discussion about the specific features of the stochastic state-space model, see “Stochastic State-Space Objects” on page 137. The dominating theme here is on Kalman filtering for state estimation.

Deterministic LTI Objects

Models are central to many applications for the Signals & Systems Lab, and linear time-invariant (LTI) models are the dominating class of models. This class contains both linear filters and stochastic models. You can use LTI objects for filtering, simulation, and estimation, as well as graphical illustrations and system analysis. The Signals & Systems Lab represents LTI systems using objects.

The main realizations of LTI systems are state-space models and transfer functions, as defined in the objects `ss` and `tf`, respectively. The focus here is to model dynamic relations between deterministic known input signals and observed output signals according to the input-output relation $y(t) = Gu(t)$. Later chapters extend the `ss` object and introduce the `arx` and `armax` objects to handle stochastic system models of the kind $y(t) = Gu(t) + He(t)$ for stochastic input-output systems and $y(t) = He(t)$ for time series.

The LTI object is the central object in the Signals & Systems Lab as the following figure illustrates.



You can think of an LTI object G as a structure with certain field names. The fields are protected, but you can access them for reading by `G.<fieldname>` using the "dot syntax" for accessing the fields. The main advantage of using the object class is that it

makes it possible to overload certain natural operations. Such overloaded functions (methods) include:

- A display function (`display`) used whenever you request a workspace printout.
- A plot function (`ltiplot`) invoked when you type, for example, `plot(G)`, `bode(G)`.
- A simulation function (`simulate`) to produce a `SIG` object.
- An estimation function (`estimate`) to produce a TF model from a `SIG` object.
- A filter function invoked when you type `kalman(G,y)` for estimating and predicting system state.
- Operators for basic model operations such as `+`, `-`, `*`, `/`, and `feedback`.

A distinguishing feature of the LTI object is the support for uncertain systems. This means that certain parameters in the model can be defined to belong to a stochastic probability density function. You can also obtain this kind of uncertain system by estimation, where all parameters are uncertain in the general case.

Creating and Displaying LTI Objects

Table 3-1 summarizes different ways to construct a model.

TABLE 3-1: BASIC WAYS TO CONSTRUCT A MODEL

`tf`	TF constructor and conversion from SS objects
`ss`	SS constructor and conversion from TF objects
`eye`	Generate a unitary model
`zeros`	Generate a zero model
`ones`	Generate an all one model
`rand`	Generate a random model
`estimate`	Estimate a model from data

while Table 3-2 lists the functions for displaying models;

TABLE 3-2: MODEL PRESENTATIONS

`plot`	Invoke ltiplot to create standard illustrations of LTI systems
`display`	Give an ASCII-formatted printout
`info`	Print out user-specified information about model name and signal labels
`size`	Return or print out the sizes nn
`tex`	Create LaTeX code

In this section you find the underlying mathematical notation and some illustrative script examples.

CONTINUOUS-TIME MODELS

A continuous-time transfer function is defined as

$$Y(s) = G(s)u(t) = \frac{b(s)}{a(s)}U(s)$$
$$= \frac{1}{s^{n_k}} \frac{b(1)s^{n_a} + \ldots + b(n_b)s^{n_a - n_b + 1}}{s^{n_a} + a(1)s^{n_a - 1} + \ldots + a(n_a)} U(s)$$

or as a differential equation

$$y^{(n_a)}(t) + a(1)y^{(n_a - 1)}(t) + \ldots + a(n_a)y(t) =$$
$$b(1)u^{(n_a - n_k)}(t) + \ldots + b(n_b)u^{(n_a - n_b - n_k + 1)}(t).$$

Here, s is the Laplace transform operator that in the time domain corresponds to differentiation, $s \sim d/dt$. To illustrate how to generate a simple TF object, first define the Laplace operator and then use it to define a second-order system:

```
s=tf('s')


   Y(s) = s U(s)


G=1/(s^2+s+2)

              1
   Y(s) = ------- U(s)
          s^2+s+2
```

The overloaded `display` function generates an ASCII- formatted printout. The direct way to define this system is to enter the polynomials into the TF constructor directly using `G=tf([0 0 1],[1 1 2],0)`, where the last interval denotes the sampling frequency, which by convention is `NaN` for continuous-time systems. This is the default value, if the third argument is omitted.

The numerator b and denominator a polynomials can be arbitrary vectors of arbitrary lengths. However, it is good practice to fill up with zeros to equal length. TF otherwise fills up with zeros from the left. This corresponds to polynomials in descending powers of s and z, respectively.

To summarize, the Signals & Systems Lab applies the following conventions when using the syntax `tf(b,a,fs)`:

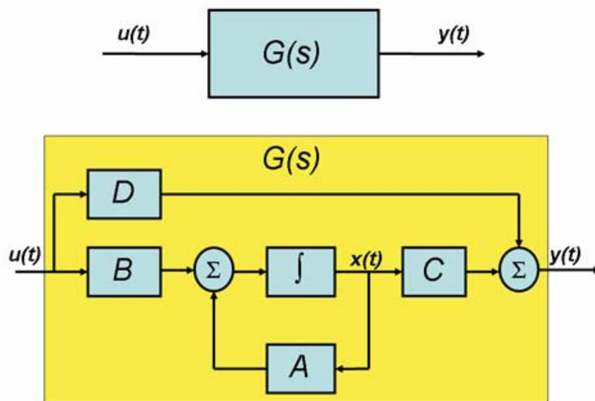
- The sampling frequency is given in Hertz for discrete time systems, and is by convention `NaN` for continuous time systems (default if `fs` is omitted).
- Coefficient $a(1)$ preceding $y(t)$ is always one.
- $b(1)$ is nonzero.
- If you specify `b` and `a` of different lengths, the shorter one is extended with zeros from the right. Use `b,a` of equal size to avoid problems.
- `nk` is the relative degree, so `nk>=0` for causal systems, and `nk<0` for noncausal systems.

STATE-SPACE MODELS

The corresponding state-space (SS) realization of a transfer function is related to the TF object by

$$
\begin{aligned}
(A, B, C, D) \leftrightarrow G(s) &= C(sI - A)^{-1}B + D, \\
\dot{x}(t) &= Ax(t) + Bu(t), \\
y(t) &= Cx(t) + Du(t).
\end{aligned}
$$

To convert between SS and TF forms, use `Gss=ss(Gtf)` and `Gtf=tf(Gss)`. The previous formula shows the implementation of the conversion from SS to TF. The conversion from TF to SS is not unique. The Signals & Systems Lab offers several canonical forms as options (see `tf2ss` on page 223 in the *Signals & Systems Lab Reference Guide*). The following block diagram illustrates the state-space realization:



The vector of polynomial coefficients `(b,a)` and the state-space matrices `(A,B,C,D)` are sufficient information to specify the input-output relation between $u(t)$ and $y(t)$.

The SS constructor transforms the model in the previous example to a state-space model by

```
ss(G)
              / -1  -2 \       / 1 \
  d/dt x(t) = \  1   0 / x(t) + \ 0 / u(t)

y(t) = (0  1) x(t) + (0) u(t)
```

Again, the Signals & Systems Lab displays an ASCII-formatted printout. In the next step, define this model as a state-space model directly. The first problem then is that the Laplace operator is noncausal and does not have a state-space realization. This is one main difference between TF and SS objects: Only the former supports noncausal systems. For that reason, define the integration operator rather then the differentiation operator and then create the model using integrations instead:

```
sinv=ss('int')
d/dt x(t) = 0 x(t) + 1 u(t)

y(t) = 1 x(t) + 0 u(t)


Gss=sinv^2/(1+sinv+2*sinv^2)
              / 0  1   0   0  0 \        / 0 \
              | 0  0  -1  -2  0 |        | 1 |
```

```
  d/dt x(t) = | 0  0  -1  -2  0 | x(t) + | 1 | u(t)
              | 0  0   0   0  1 |        | 0 |
              \ 0  0  -1  -2  0 /        \ 1 /

y(t) = (1  0  0  0  0) x(t) + (0) u(t)


tf(Gss)

               1
   Y(s) = --------- U(s)
          s^2+1*s+2
```

Note that repeated low-level operations do not necessarily lead to a minimal realization, which in this case is a second-order state-space model. However, the transformation to TF validates the correctness of the model.

The alternative here is to define the state-space model directly using

```
ss([-1 -2;1 0],[1;0],[0 1],0)
```

MIMO SYSTEMS

State-space models include both SISO and MIMO models without any change in notation (`B` is (`nx`, `nu`), `C` is (`ny`, `nx`) and `D` is (`ny nu`), respectively). For transfer functions, the convention used in the Signals & Systems Lab is that the $b(s)$ polynomial is matrix valued, while $a(s)$ is kept scalar and equal to `poly(A)` for the state-space model. For instance, a system with two inputs and two outputs has the transfer function

$$Y(s) = G(s)U(s) = \frac{1}{a(s)}\begin{bmatrix} b_{11}(s) & b_{12}(s) \\ b_{21}(s) & b_{22}(s) \end{bmatrix} U(s)$$

where, for instance, $b_{12}(s)/a(s)$ is the transfer function from input 2 to output 1.

You can define a MIMO model just as you create matrices, which the following section explains in more detail:

```
G22=[1/(s^2+s+2) 1/s; 1/(s+2) (s+1)/(s+2)]

                 s^3+4*s^2+4*s
   Y1(s) = ------------------------- U1(s)
           s(s^4+5*s^3+10*s^2+12*s+8)

            s^4+5*s^3+10*s^2+12*s+8
   Y1(s) = ------------------------- U2(s)
           s(s^4+5*s^3+10*s^2+12*s+8)
```

```
            s^4+3*s^3+4*s^2+4*s
Y2(s) = ------------------------- U1(s)
         s(s^4+5*s^3+10*s^2+12*s+8)

         s^5+4*s^4+7*s^3+8*s^2+4*s
Y2(s) = ------------------------- U2(s)
         s(s^4+5*s^3+10*s^2+12*s+8)
```

Note that all I/O relations appear on a common denominator.

In summary, you build up MIMO systems just as you build matrices in COMSOL Script. The main difference is that you have to define an empty structure for some of the most common functions. To do this, following these steps:

```
SSstruc=ss([3 2 0 2])
Unspecified SS model with 3 states, 2 inputs, 0 process noise
dimensions, and 2 outputs.
TFstruc=tf([2 2 0 2 2])
Unspecified TF model with na=2, nb= 2, nk= 0 and with 2 inputs and
2 outputs.
```

From these structures you can, for instance, apply `eye` and `rand`:

```
eye(SSstruc)
             /1  0\
  y(t) =  +  \0  1/  u(t)


rand(TFstruc)

            s(s+1.4)
Y1(s) = --------------- U1(s)
         s^2+0.96*s+0.17

            s(s+0.58)
Y1(s) = --------------- U2(s)
         s^2+0.96*s+0.17

            s(s+1.5)
Y2(s) = --------------- U1(s)
         s^2+0.96*s+0.17

            s(s+1.1)
Y2(s) = --------------- U2(s)
         s^2+0.96*s+0.17
```

The `size` function returns the size of an object as the sizes of all vector sorted “in order of appearance,” which coincides with alphabetical order for TF objects. For instance,

```
size(G22);
na = 4
nb = 6
nk = 0
nu = 2
ny = 2
```

DISCRETE-TIME SYSTEMS

The definition of discrete-time systems is completely analogous to the one of continuous-time systems, but you replace s with the time-shift operator q defined as $qu(t) = u(t+1)$. More often, however, the definition uses a backward shift formalism,

$$\begin{aligned} y(t) &= G(q^{-1})u(t) = \frac{b(q^{-1})}{a(q^{-1})}u(t) \\ &= \frac{b(1)q^{-n_k} + \ldots + b(n_b)q^{-n_k-n_b+1}}{1 + a(1)q^{-1} + \ldots + a(n_a)q^{-n_a}}u(t) \end{aligned}$$

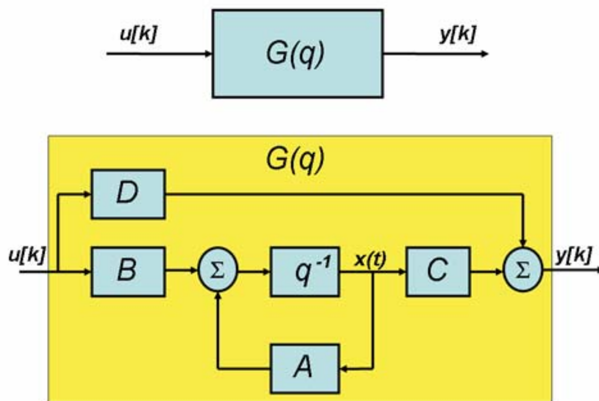
which leads to the difference equation

$$\begin{aligned} y(t) + a(1)y(t-1) + \ldots + a(n_a)y(t-n_a) &= \\ b(1)u(t-n_k) + \ldots + b(n_b)u(t-n_k-n_b). \end{aligned}$$

The relation between transfer-function and state-space realizations is now

$$\begin{aligned} (A, B, C, D) \leftrightarrow G(q^{-1}) &= C(qI-A)^{-1}B + D \\ x(t+1) &= Ax(t) + Bu(t), \\ y(t) &= Cx(t) + Du(t). \end{aligned}$$

illustrated in the following block diagram:



ZEROS AND POLES

Another useful representation is the zero-pole-gain `zpk` form or input-output relations. For discrete-time models, the transfer function can be rewritten as

$$G(q) = kq^{-n_k}\frac{(1-z(1)q^{-1})(1-z(2)q^{-1})\ldots(1-z(n_b)q^{-1})}{(1-p(1)q^{-1})(1-p(2)q^{-1})\ldots(1-p(n_a)q^{-1})}$$

and similarly for continuous-time systems. For the example system, you get

```
[z,p,k]=zpk(G)
z =
        Inf        Inf
p =
    -0.5000 +    1.3229i    -0.5000 -    1.3229i
k =
  1
```

TEX SUPPORT

For documentation and presentation, there is an option to generate models formatted as TeX code.

```
tex(G)
ans =
  \begin{eqnarray*}
   Y(z) &=& \frac{1}{z^2+z+2}  U(z)
  \end{eqnarray*}
```

The primary use for such code is in LaTeX documents. However, filters (freeware or shareware) for other word processors are available:

- TexPoint: freeware for PowerPoint
- LaImport: FrameMaker
- TeX2Word: Word
- LaTeX2rtf: RTF documents
- TeX4ht: HTML or XML hypertext documents

For instance, this document includes the TF and SS models above to produce

$$Y(s) = \frac{1}{s^2 + s + 2} U(s)$$

and

$$\dot{x}(t) = \begin{bmatrix} -1.0 & -2.0 \\ 1.0 & 0.0 \end{bmatrix} x(t) + \begin{bmatrix} 1.0 \\ 0.0 \end{bmatrix} u(t)$$
$$y(t) = \begin{bmatrix} 0.0 & 1.0 \end{bmatrix} x(t)$$

respectively. There are optional arguments for controlling number of decimals and LaTeX environments.

Operations on LTI Objects

Table 3-3 summarizes the available operations on LTI objects (both SS and TF):

TABLE 3-3: OPERATIONS ON DETERMINISTIC LTI OBJECTS

`plus`	+G	Unitary plus
`uminus`	-G	Unitary minus
`plus`	G1+G2	Parallel connection with summation at output
`minus`	G1-G2	Parallel connection with difference at output
`power`	G.^n	Repeated multiplication
`mpower`	G^n	Repeated multiplication
`inv`	inv(G)	Inverse of square systems
`eq`	G1==G2	Test for equality
`mrdivide`	1/G	Right inverse of a system
`mldivide`	G\1	Left inverse of a system

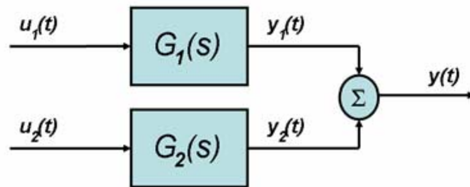
TABLE 3-3: OPERATIONS ON DETERMINISTIC LTI OBJECTS

`mtimes`	G1*G2	Series connection of two models
`times`	G1.*G2	Elementwise multiplication of two models
`feedback`	feedback(G1,G2)	Feedback connection of two systems
`diag`	diag(G1,G2,...)	Append independent models
`append`	append(G1,G2)	Append independent models
`ctranspose`	G.	Reverse the inputs with outputs
`transpose`	G	Reverse the inputs with outputs
`horzcat`	[G1 G2]	Horizontal concatenation
`vertcat`	[G1;G2]	Vertical concatenation
`arrayread`	G(i,j)	Pick out subsystems by indexing

Several of the arithmetic operations such as +, -, `*`, / , and `feedback` operate on two systems. In all cases, you can enter a matrix as one of the arguments, where the matrix corresponds to a static system that scales the inputs and outputs. You have to check that the dimensions of the two systems make sense for the operation you want to use, otherwise the Signals & Systems Lab returns an error message. The operations and their requirements on the systems are described in detail below.

ADDITION (PARALLEL CONNECTION)

The sum of two systems is defined as a parallel connection,



where the total output is the sum of the respective output of each system according to the following relations:

$$
\begin{aligned}
y &= y_1 + y_2, \\
u &= u_1 = u_2 \Rightarrow \\
G &= G_1 + G_2, \\
(A_1, B_1, C_1, D_1) + (A_2, B_2, C_2, D_2) &= (A, B, C, D)
\end{aligned}
$$

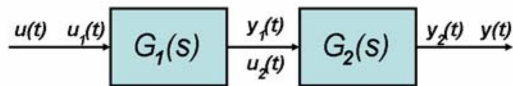
$$
\left(\begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix}, \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}, [C_1,\ C_2], D_1 + D_2 \right)
$$

Both systems must have the same number of inputs and outputs. The following code shows a simple example:

```
G=1/(s^2+s+2)

             1
   Y(s) = ------- U(s)
          s^2+s+2

G + 1/(s+3)

             s^2+2*s+5
   Y(s) = --------------- U(s)
          s^3+4*s^2+5*s+6
```

MULTIPLICATION (SERIES CONNECTION)

Multiplication corresponds to series connection,



and the following equations provide the definition:

$$
\begin{aligned}
y_1 &= u_2 \Rightarrow \\
G &= G_1 * G_2, \\
(A_1, B_1, C_1, D_1) * (A_2, B_2, C_2, D_2) &= (A, B, C, D)
\end{aligned}
$$

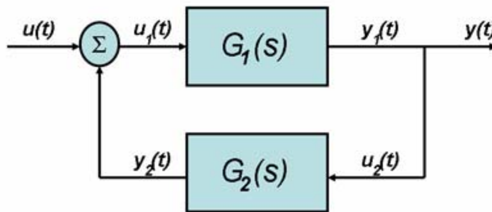
$$
\left(\begin{bmatrix} A_1 & 0 \\ B_2 C_1 & A_2 \end{bmatrix}, \begin{bmatrix} B_1 \\ B_2 D_1 \end{bmatrix}, [D_2 C_1,\ C_2], D_2 D_1 \right)
$$

Here, the number of outputs from the first system must equal the number of inputs from the second system. An example similar to the previous one is:

```
G * 1/(s+3)

                  1
  Y(s) = --------------- U(s)
         s^3+4*s^2+5*s+6
```

FEEDBACK

The following figure shows the structure of a feedback system.



This is based on the assumption that the number of inputs of the first system is equal to the number of outputs of the second system, and the other way around. The closed-loop system is quite easy to derive when there is no direct terms involved, so both transfer functions are strictly proper, and the D matrices of the corresponding state-space models are both zero. The closed-loop system is then

$$u_1 = u - y_2,$$
$$u_2 = y_1 \Rightarrow$$
$$G_c = \frac{G_1}{1 + G_2 G_1},$$
$$\text{feedback}((A_1, B_1, C_1, 0), (A_2, B_2, C_2, 0)) = (A_c, B_c, C_c, D_c)$$
$$\left(\begin{bmatrix} A_1 & -B_1 C_2 \\ B_2 C_1 & A_2 \end{bmatrix}, \begin{bmatrix} B_1 \\ 0 \end{bmatrix}, [C_1 \ 0], 0\right)$$

With direct terms, the state-space matrices become more complicated, and the following general equation shows the closed-loop system incorporating possible algebraic loops:

$$\text{feedback}((A_1, B_1, C_1, D_1), (A_2, B_2, C_2, D_2)) = (A_c, B_c, C_c, D_c),$$

$$A_c = \begin{bmatrix} A_1 - B_1(X^+ D_2)C_1 & -B_1(X^+ C_2) \\ B_2[I - D_1(X^+ D_2)]C_1 & A_2 - B_2 D_1(X^+ C_2) \end{bmatrix},$$

$$B_c = \begin{bmatrix} B_1/X \\ B_2 D_1/X \end{bmatrix},$$

$$C_c = [(I - D_1(X^+ D_2))C_1 - D_1(X^+ C_2)]],$$

$$D_c = D_1/X,$$

$$X = I + D_2 D_1.$$

The Signals & Systems Lab only allows SISO systems for feedback of TF objects. Use the SS method in the MIMO case.

Unit feedback of the example system gives:

```
Gc=feedback(G,1)

                   s^2+s+2
   Y(s) = --------------------- U(s)
          s^4+2*s^3+6*s^2+5*s+6
```

There is no automatic simplification of the expressions. This chapter has already included a couple of examples of overparameterized results. The function `minreal` is handy in such situations. It performs pole-zero cancellations for TF objects, and gives:

```
minreal(Gc)

              1
   Y(s) = --------- U(s)
          s^2+1*s+3
```

For SS objects the software uses a more advanced method (see "Balanced Realization and Model Truncations" on page 105).

INVERSION

Inversion of square systems, or the right inverse for systems with more inputs than outputs, is computed by

$$n_u \geq n_y \Rightarrow$$
$$G * G^+ = I_{n_u}$$
$$(A, B, C, D)^+ = (A - \mathrm{BD}^+ C), BD^+, -D^+ C, D^+)$$

A necessary and sufficient condition for the right inverse to exist is that $\mathrm{rank}(D) = n_y$. For transfer functions, this is only implemented in the SISO case, where the numerator and denominator simply swap places. This is the reason for the SISO limitations in the `feedback` function. SS objects have no such limitations.

APPEND AND DIAG

Appending two systems means that they are packed into one model. This gives a two-input two-output model from two SISO models, for instance. The definition is

$$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix},$$
$$y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \Rightarrow$$
$$G_c = \begin{bmatrix} G_1 & 0 \\ 0 & G_2 \end{bmatrix},$$
$$\mathrm{append}((A_1, B_1, C_1, D_1), (A_2, B_2, C_2, D_2)) = (A, B, C, D)$$
$$\left(\begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix}, \begin{bmatrix} B_1 & 0 \\ 0 & B_2 \end{bmatrix}, \begin{bmatrix} C_1 & 0 \\ 0 & C_2 \end{bmatrix}, \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix} \right)$$

To define a block-diagonal MIMO system, type `s=diag(s1,s2,s3,s4)`, for example. This function calls `diag` recursively. The reverse operation, `s=diag(s)`, gives a diagonal system where the output *j* depends only on input *j*.

TRANSPOSITION

The transpose operation (`ctranspose`, `'`) has a particular physical meaning as the complex conjugate of the transfer function,

$$G'(s) = G^T(-s),$$
$$G'(z) = G^T(1/z),$$
$$(A, B, C, D)' = (A^T, C^T, B^T, D^T).$$

CONCATENATION

Systems can be concatenated in the same way as matrices of complex numbers to create MIMO systems. Rows and columns get somewhat different physical meaning. First, column concatenation is defined as

$$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix},$$

$$y = Gu = G_1 u_1 + G_2 u_2 \Rightarrow$$

$$G = [G_1, G_2]$$

$$[(A_1, B_1, C_1, D_1), (A_2, B_2, C_2, D_2)] = (A, B, C, D)$$

$$\left(\begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix}, \begin{bmatrix} B_1 & 0 \\ 0 & B_2 \end{bmatrix}, [C_1,\ C_2], [D_1,\ D_2] \right)$$

and row concatenation is defined as

$$y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix},$$

$$u = u_1 = u_2 \Rightarrow$$

$$G = \begin{bmatrix} G_1 \\ G_2 \end{bmatrix},$$

$$[(A_1, B_1, C_1, D_1); (A_2, B_2, C_2, D_2)] = (A, B, C, D)$$

$$\left(\begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix}, \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}, \begin{bmatrix} C_1 & 0 \\ 0 & C_2 \end{bmatrix}, \begin{bmatrix} D_1 \\ D_2 \end{bmatrix} \right)$$

SYSTEM ANALYSIS AND MODEL CONVERSIONS

You can convert an LTI model to another LTI model using one of the functions in Table 3-4.

TABLE 3-4: CONVERSIONS BETWEEN LTI MODELS

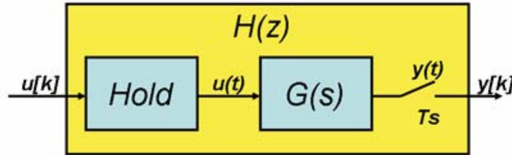
`c2d`	Convert a continuous-time TF to a discrete-time TF
`d2c`	Convert a discrete-time TF to a continuous-time TF
`minreal`	Compute a minimal realization

TABLE 3-4: CONVERSIONS BETWEEN LTI MODELS

modred	Compute a reduced-order model using a balanced realization
balreal	Compute the balanced realization

DISCRETIZATION

The figure below illustrates how a continuous-time system can be discretized with a hold circuit and a sampling to give a discrete-time system.



The derivation of discretization formulas using this approach is based on the solution to a continuous-time state-space model

$$\begin{aligned} \dot{x} &= Ax + Bu, \\ y &= Cx + Du. \end{aligned}$$

whose solution is given by

$$\begin{aligned} x(t+T) &= e^{AT}x(t) + \oint_0^T e^{A\tau}Bu(t+T-\tau)d\tau \\ &= e^{AT}x(t) + \oint_0^T e^{A\tau}d\tau Bu(t). \end{aligned}$$

The second equality follows if the input is piecewise constant, which the zero-order hold sampling assumes. From this equation, the discrete-time state-space matrices are readily identified. The exponential function is defined by its series expansion, from which it is possible to compute the integral as

$$\begin{aligned} e^{AT} &= I + AT + A^2\frac{T^2}{2!} + A^3\frac{T^3}{3!} + \dots, \\ \oint_0^T e^{A\tau}d\tau &= IT + A\frac{T^2}{2!} + A^2\frac{T^3}{3!} + \dots. \end{aligned}$$

From this relation, you can compute the discrete-time state-space model in one step by computing the matrix exponential of an augmented A matrix:

$$\exp\left(\begin{bmatrix} AT & BT \\ 0 & 0 \end{bmatrix}\right) = \begin{bmatrix} A_d & B_d \\ 0 & 0 \end{bmatrix}$$

There are many ways to compute the matrix exponential. One direct method based on the definition and a clever scaling uses the following equality:

$$e^A = (e^{A/2^j})^{2^j}$$

The idea is that the outer power is computed by squaring the exponential matrix j times, where j is determined to be large enough to get a rapid decay in the Taylor expansion.

As an illustration, consider a model of a DC motor,

```
Gdc=ss([0 2; 0 0],[0;1],[1 0],0,0)
            / 0  2 \       / 0 \
  x[k+1] = \ 0  0 / x[k] + \ 1 / u[k]

y[k] = (1  0) x[k] + (0) u[k]


GdcZOH=c2d(Gdc,0.5)
            / 1  4 \       / 4 \
  x[k+1] = \ 0  1 / x[k] + \ 2 / u[k]

y[k] = (1  0) x[k] + (0) u[k]
```

This model has an idempotent A matrix (the product when multiplied by itself sufficiently many times—in this case, 2 times—becomes zero), so the Taylor expansion includes only two terms. That is, the result above is easy to verify by hand.

First-order hold sampling is derived analogously. The computations are based on the general solution, where the input is interpolated between the sampling times.

Bilinear transformation is another way to convert systems between continuous and discrete time. The bilinear transformation is defined by the following mapping between the left half plane and the unit circle,

$$s = \frac{2}{T}\frac{z-1}{z+1},$$

$$z = \frac{2/T+s}{2/T-s}.$$

For state-space models, it gives the discrete-time model

$$\frac{2}{T}\frac{z-1}{z+1}x(t) \approx sx(t) = Ax + Bu.$$

which can be rewritten in the standard form with

$$H = (I_{n_x} - T/2A)^{-1},$$

$$A_d = H(I_{n_x} + T/2A),$$

$$B_d = T/2HB,$$

$$C_d = CH,$$

$$D_d = D + C_d.$$

For the DC motor,

```
GdcBIL=c2d(Gdc,0.5,'bil')
              / 2  -1 \        / 1 \
  x[k+1] = \ 1   0 / x[k] + \ 0 / u[k]

y[k] = (8  0) x[k] + (2) u[k]
```

the discrete-time model becomes somewhat different compared to the zero-order hold (ZOH) assumption. The conversion back to continuous time is given by

$$A_c = 2/T(I_{n_x} + A)^{-1},$$

$$\dot{H} = (I_{n_x} - T/2A),$$

$$B_c = 2/T\dot{H}B,$$

$$C_c = C\dot{H},$$

$$D_c = D - CB.$$

For transfer functions, only the bilinear transformation is available. The substitution is given by a linear transformation of the parameter vectors. If they are first padded with zeros to equal length, there is a transformation matrix that only depends on the sampling interval for which the row vectors obey the linear relation

$$b_d = b_c W,$$
$$a_d = a_c W.$$

SYSTEM ANALYSIS

System analysis includes the functions in the table below:

TABLE 3-5: SYSTEM ANALYSIS TOOLS

ss.ctrb	Compute the controllability matrix
ss.obsv	Compute the observability matrix
ss.gram	Computes the controllability (observability) Gramian
ss.lqe	Solve the continuous-time stationary Riccati equation
ss.dlqe	Solve the discrete-time stationary Riccati equation

These are useful macros in, for example, the model conversions in the next section.

BALANCED REALIZATION AND MODEL TRUNCATIONS

Numerical sensitivity is a critical issue for poorly damped or large systems. The best way to avoid problems with finite precision numerics is to use a balanced realization of a state-space model.

The following algorithm computes a balanced realization:

1 Solve the Lyaponov function $AP + PA^T + BB^T = 0$ for P using `P=gram(s,'c')`.
2 Solve the Lyaponov function $A^T Q + QA + C^T C = 0$ for Q using `Q=gram(s,'o')`.
3 Compute the factorization $Q = R^T R$.
4 Compute the SVD $RPR^T = U\Sigma^2 U^T$.
5 Compute the transformation matrix $T = \Sigma^{-1/2} U^T R$.
6 The balanced realization is given by
$(A_b, B_b, C_b, D_b) = (TAT^{-1}, TB, CT^{-1}, D)$.

To avoid problems, compute the balanced realization `s=balreal(s)` after each operation. The Signals & Systems Lab does not do this automatically, because this operation destroys the structure of the states, which is sometimes not something you want. For instance, from physical modeling of a DC motor, the first state represents the angle, and the second state represents the angular speed. After transforming the state vector, this physical meaning gets lost.

The following example uses a random system:

```
Grand=rand(ss([3,1,0,1],1))
```

```
              / -0.89  1  0 \         / -1.8 \
  x[k+1] = |  0.58  0  1 | x[k] + | 0.84 | u[k]
              \  0.52  0  0 /         \  0.5 /

y[k] = (1  0  0) x[k] + (1) u[k]


balreal(Grand)
              /    -0.93    0.1  0.0078 \         / -1.8 \
  x[k+1] = |     -0.1   -0.72   0.062 | x[k] + |  1.2 | u[k]
              \ -0.0078  0.062    0.76 /         \ 0.11 /

y[k] = (1.8  1.2  0.11) x[k] + (1) u[k]
```

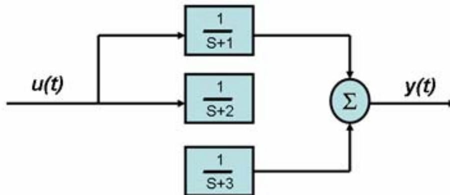
A certain degree of symmetry always characterizes the balanced realization.

A balanced realization is also useful for model approximations. The balanced realization algorithm automatically sorts the states in order of importance, so you obtain an n:th order approximation by simply truncating the model at order n. To achieve this, use `s=modred(s,n)`.

```
modred(Grand)
              / -0.93    0.1 \         / -1.8 \
  x[k+1] = \  -0.1  -0.72 / x[k] + \  1.2 / u[k]

y[k] = (1.8  1.2) x[k] + (1) u[k]
```

The algorithm automatically chooses a second-order approximation in this case. The neglected dynamics is insignificant. You can force a certain model order by a second argument to `modred`.

For a conceptually good example for understanding dynamics that can be truncated from an input-output relation, consider the block diagram given below.



The third state is not observable, and the second state is not controllable. Presuming that these states are stable, a model can omit them once the transients are gone. This is obvious in this simple case, and you can use `modred` here as well as in more complex

cases. The following code provides the state-space model corresponding to this block diagram:

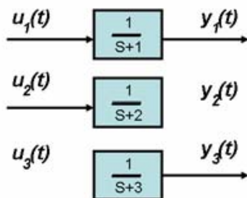
```
Gmax=ss(diag([-1 -2 -3]),[1;1;0],[1 0 1],0,0)
            / -1   0   0 \        / 1 \
  x[k+1] = |  0  -2   0 | x[k] + | 1 | u[k]
            \  0   0  -3 /        \ 0 /

y[k] = (1  0  1) x[k] + (0) u[k]
```

Its reduced model is

```
modred(Gmax)
x[k+1] = -1 x[k] + 1 u[k]

y[k] = 1 x[k] + 0 u[k]
```

Further, the first step in computing a minimal realization is to remove states that do not contribute to the input-output relation. The algorithm achieves this by truncating at the order where all singular values (Σ in the algorithm earlier in this section) are zero. The second step involves searching for inputs that do no affect the output, and outputs that are not affected by the inputs, and removing these from the system. This is the procedure that `s=minreal(s)` uses.

In the previous example, `modred` succeeded in removing uncontrollable and unobservable states. Consider now the modified MIMO system below:



In this case, `modred` obtains a minimal state but keeps the ambiguous inputs and outputs:

```
G33max=ss(diag([-1 -2 -3]),diag([1;1;0]),diag([1 0
1]),zeros(3),0);
modred(G33max)
x[k+1] = -1 x[k] + (1  0  0) u[k]

          /1\          /0  0  0\
```

```
y[k] = | 0 | x[k] + | 0  0  0 | u[k]
       \0/          \0  0  0/
```

This is the main reason for the second step in `minreal`, which removes these signals:

```
minreal(G33max)
x[k+1] = -1 x[k] + 1 u[k]

y[k] = 1 x[k] + 0 u[k]
```

For transfer functions, only `minreal` is implemented. This algorithm systematically searches for common poles and zeros computed in `zpk` and then cancels these.

Conversions from LTI to Other Objects and Data Types

Table 3-6 provides a summary of the different ways to convert LTI models to other objects and data types:

TABLE 3-6: CONVERSION FROM LTI TO OTHER OBJECTS AND DATA TYPES

`freq`	Compute frequency domain response G(f)
`zpk`	Compute the zeros, poles, and gain
`impulse`	Compute the analytic impulse response
`step`	Compute the analytic step response
`simulate`	Simulate a signal y=G(u) from an LTI state-space model

FREQUENCY RESPONSE

The frequency response is computed straightforwardly for each input and output channel as

$$G(f) = C((i2\pi f)I - A)^{-1}B + D = \frac{1}{(i2\pi f)^{n_k}}\frac{b(1)(i2\pi f)^{n_a} + \ldots + b(n_b)(i2\pi f)^{n_a - n_b + 1}}{(i2\pi f)^{n_a} + a(1)(i2\pi f)^{n_a - 1} + \ldots + a(n_a)}$$

for continuous-time systems and

$$G(f) = C(e^{i2\pi f}I - A)^{-1}B + D = e^{-i2\pi f n_k}\frac{b(1) + \ldots + b(n_b)e^{-i2\pi f n_b}}{1 + a(1)e^{-i2\pi f} + \ldots + a(n_a)e^{-i2\pi f}n_a}$$

for discrete-time systems.

ZEROS AND POLES

The computation of zeros and poles uses `roots` for TF objects. For state-space models, `zpk` uses `eig` to compute the poles for numerical reasons.

SIMULATION

Discrete-Time Models

The workhorses of all simulations and filtering operations is the `filter` function for TF models and the `dlsim` function for SS models. Use this function for simulating discrete-time TF objects. For state-space models, a straightforward time recursion is used. In this way the algorithm preserves the numerical conditioning of a balanced realization.

Continuous-Time Models

In theory, it is possible to compute the solution to the differential equations of a state-space model analytically if you know the intersample behavior of the input. This is utilized for computing the impulse response function

$$x(t) = e^{At}$$

In the same way, the step response is computed as

$$x(t) = \oint_0^t e^{A\tau} d\tau B$$

In the general case of a simulation, the Signals & Systems Lab samples the system using `c2d`, where the sampling interval is much smaller than the time constant that the dominating pole determines. Then, the software performs a discrete-time simulation.

There is a special procedure when the input signal contains discontinuities such as

- Steps
- Impulses

The SIG object shows how these discontinuities are represented and constructed. In these cases, the Signals & Systems Lab uses the following algorithm:

1. Segment the input signal where the steps and impulses are located at the borders.
2. Adjust the basic sampling interval slightly to get a regular grid over each segment.
3. Approximate impulses with short pulses with the same area.
4. Use the final state of the filter in each segment as initial state in the next segment.

In this way, the algorithm synchronizes the output with the input discontinuities.

Graphical Illustrations of LTI Objects

You can view an LTI object in different ways using the following plots:

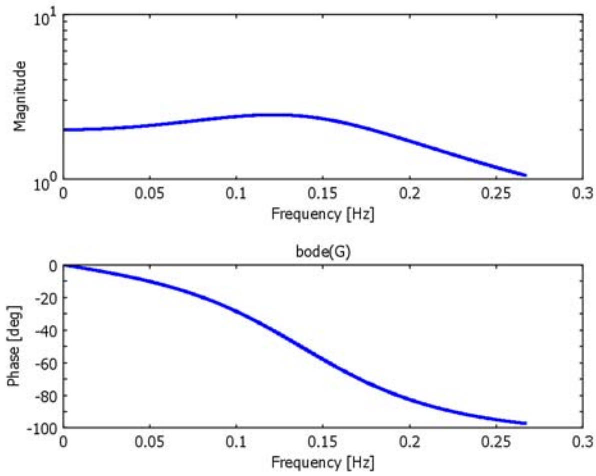
- Bode diagram of amplitude and phase as a function of frequency f of $G(f)$. There are options that you can use to plot only the amplitude, only the phase, or both.
- Nyquist curve, where the plot shows $G(f)$ as a complex function.
- Pole-zero plots, which show the poles and zeros of $G(s)$ in the complex plane.
- Root locus for $G(s)$, which is a plot of the poles of the closed-loop system, using a constant feedback K, as a function of K. For SISO TF objects, the poles are the roots of the equation $A(s) + KB(s) = 0$. For MIMO state-space models, the closed loop poles are determined by the eigenvalues to the matrix $A - B(1/kI + D)^{-1}C$.

TABLE 3-7: VIEWING OPTIONS OF LTI OBJECTS

`bode(G1,G2,...)`	Bode plot of amplitude and phase curves
`bodeamp(G1,G2,...)`	Bode diagram of phase curve
`bodephase(G1,G2,...)`	Bode diagram of phase curve
`nyquist(G1,G2,...)`	Nyquist curve
`zpplot(G1,G2,...)`	Zero-pole plot
`rlplot(G1,G2,...)`	Root locus plot

You can use the same notation for continuous and discrete-time systems—just replace s with q above. For MIMO systems, the plots appear in a subplot array with `ny` rows and `nu` columns. It is possible to set the most common properties of standard plots such as `Xlim`, `Ylim`, `fontsize`, `linewidth`, and `axis`. Also, you can specify the color (or color order for multiple LTI object inputs) using the property `col`, which is a vector with one letter color abbreviations such as `b` for blue, `k` for black, and `r` for red.

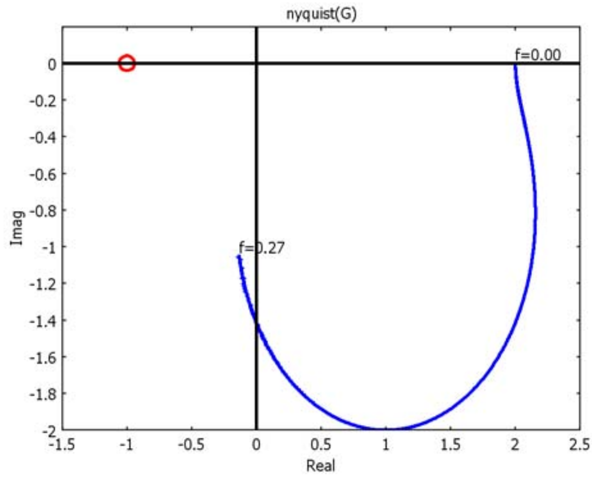
For illustration, consider a slightly damped second-order system:



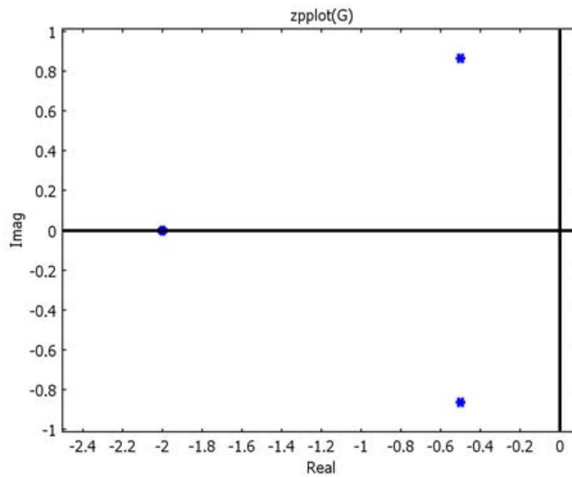
The bode function plots the amplitude and phase curve of the frequency function $G(f)$.

Special functions are available for plotting only amplitude (bodeamp) and phase (bodephase).

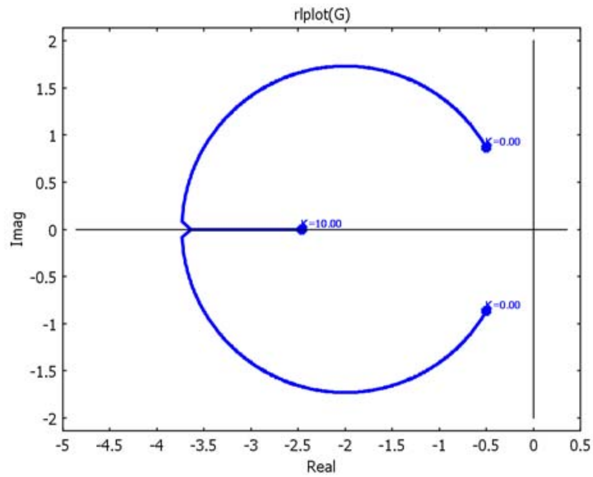
The Nyquist curve illustrates the frequency function $G(f)$ in the complex plane.



The function marks the start and end frequency in the plot. The zero-pole plot displays the poles and zeros in the complex plane.

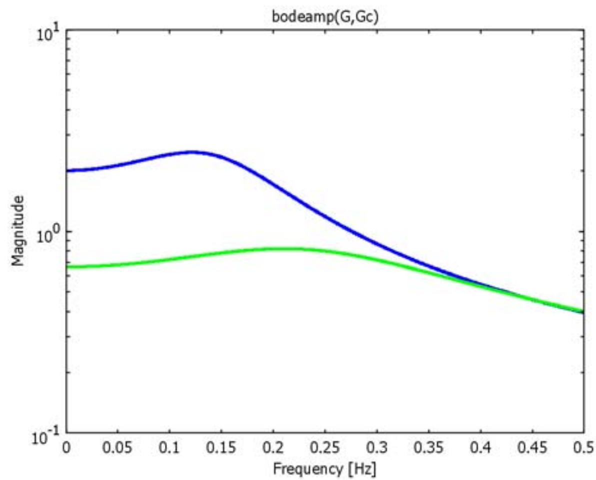


Finally, the root locus plot illustrates how the poles are moving as a function of the feedback gain, when you apply a constant feedback.



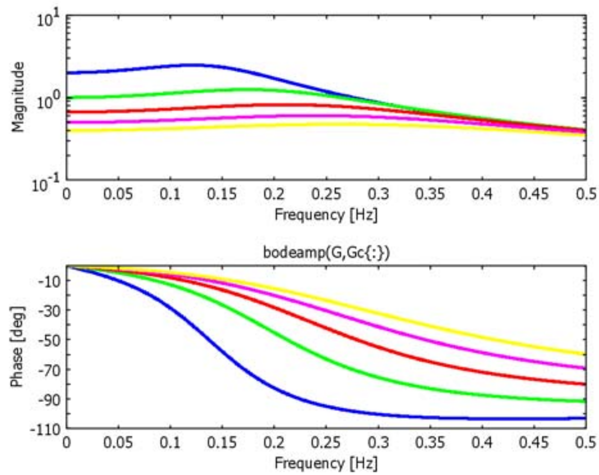
The root locus indicates that a proportional feedback attenuates the oscillatory modes. First, try with a unit feedback and compare the Bode plot with the open-loop system. In the Bode plot, you can specify the frequency range using the `fmax` property.

```
Gc=feedback(G,1);
bodeamp(G,Gc,'fmax',0.5)
```



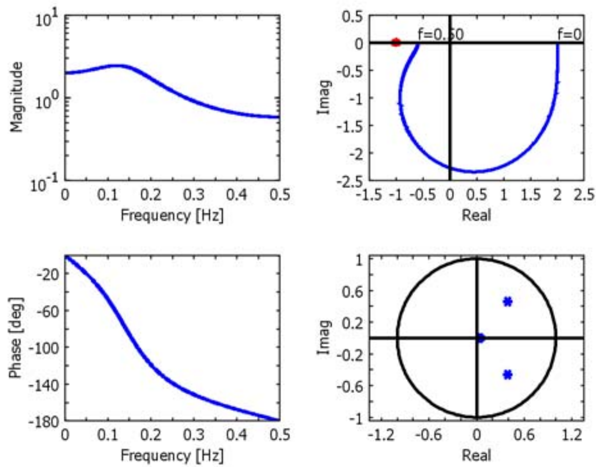
All plot functions accept arrays of LTI systems, so you can define several feedback systems simultaneously using a simple `for` loop:

```
K=[0.5:0.5:2];
for k=1:length(K)
    Gc2{k}=feedback(G,K(k));
    Gc2{k}=feedback(G,K(k));
    Gc2{k}=feedback(G,K(k));
    Gc2{k}=feedback(G,K(k));
end
bode(G,Gc2{:},'fmax',0.5)
```



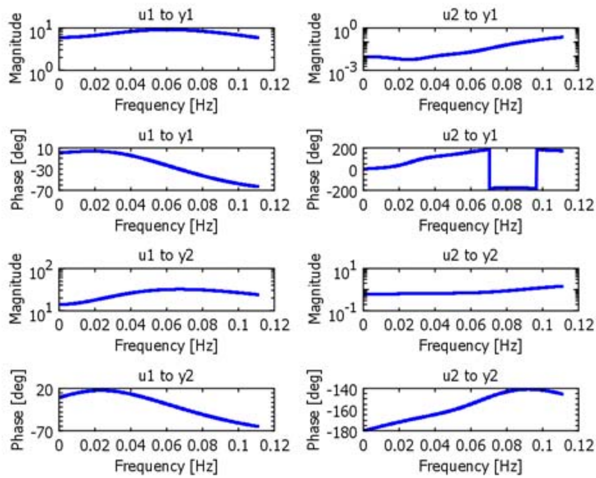
If you prefer discrete-time system analysis, first sample the system. The following code gives an overview of system properties:

```
Gd=c2d(G,1);
subplot(2,2,1)
bodeamp(Gd)
subplot(2,2,3)
bodephase(Gd)
subplot(2,2,2)
nyquist(Gd)
subplot(2,2,4)
zpplot(Gd)
```

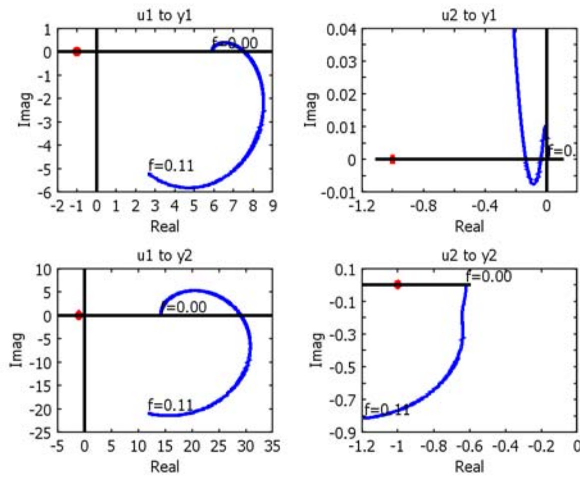


Now, consider a randomly generated MIMO system. The Bode diagram then becomes a set of subplots with one plot for each input-output pair.

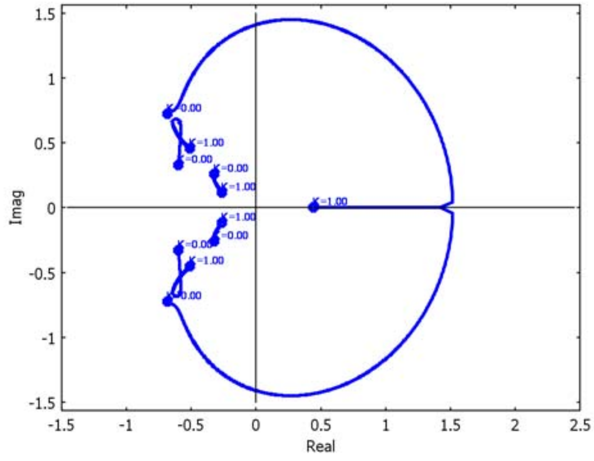
```
Grand=rand(ss([6,2,0,2]));
bode(Grand)
```



Similarly, the following plot shows the Nyquist array:



The root locus, however, has the same number of poles (six in this case) independently of the number of inputs and outputs. Note that root loci assume feedback with K times the identity matrix, so the MIMO system has to be square with the same number of inputs and outputs.


Imag
Real
K=0.00
K=1.00

Application Examples

Aircraft Pitch Control

This example is based on the AIRC case study of J. Maciejowski, ***Multi-variable control design***. You can load the state-space model from this study into the Signals & Systems Lab by:

```
M=exlti('AIRC')
               / 0       0    1.1      0      -1 \        /
               | 0  -0.054  -0.17      0    0.07 |        |
  d/dt x(t) =  | 0       0      0      1       0 | x(t) + |
               | 0   0.049      0  -0.86      -1 |        |
               \ 0   -0.29      0    1.1  -0.069 /        \

      0  0       0 \
  -0.12  1       0 |
      0  0       0 | u(t)
    4.4  0    -1.7 |
    1.6  0  -0.073 /

          /1  0  0  0  0\          /0  0  0\
  y(t) = | 0  1  0  0  0 | x(t) + | 0  0  0 | u(t)
          \0  0  1  0  0/          \0  0  0/
```

To generate a LaTeX-formatted description, use `tex(M)`:

$$\dot{x}(t) = \begin{bmatrix} 0.00 & 0.00 & 1.13 & 0.00 & -1.00 \\ 0.00 & -0.05 & -0.17 & 0.00 & 0.07 \\ 0.00 & 0.00 & 0.00 & 1.00 & 0.00 \\ 0.00 & 0.05 & 0.00 & -0.86 & -1.01 \\ 0.00 & -0.29 & 0.00 & 1.05 & -0.07 \end{bmatrix} x(t) + \begin{bmatrix} 0.00 & 0.00 & 0.00 \\ -0.12 & 1.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 4.42 & 0.00 & -1.67 \\ 1.57 & 0.00 & -0.07 \end{bmatrix} u(t)$$

$$y(t) = \begin{bmatrix} 1.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 1.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 1.00 & 0.00 & 0.00 \end{bmatrix} x(t)$$

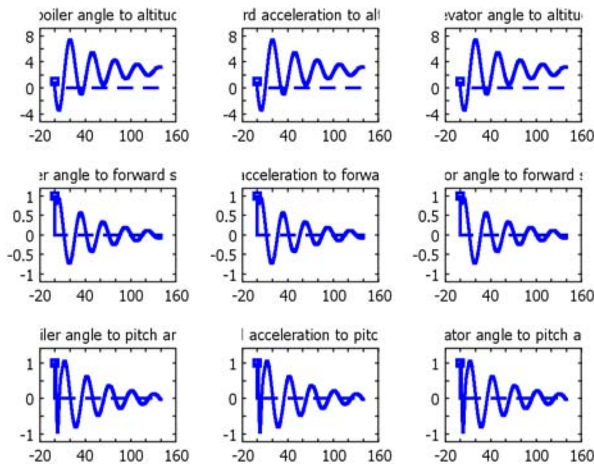
The inputs, outputs, and states have the following physical interpretations:

```
info(M)
Name:    AIRC
Description:
Case study in the book of Maciejowski, see Appendix A.1
```

```
Inputs:
   u1:  spoiler angle
   u2:  forward acceleration
   u3:  elevator angle
Outputs:
   y1:  altitude
   y2:  forward speed
   y3:  pitch angle
States:
   x1:  altitude
   x2:  forward speed
   x3:  pitch angle
   x4:  pitch rate
   x5:  vertical speed
```

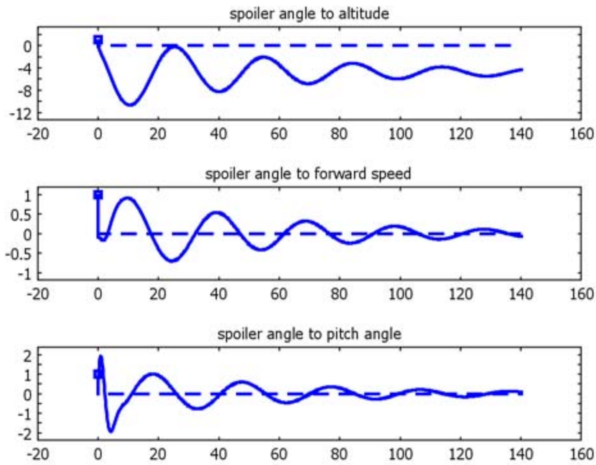
The function `info` prints out the user specified fields `name`, `xlabel`, `ulabel`, `ylabel`, and `desc`, respectively.

```
impulse(M);
```



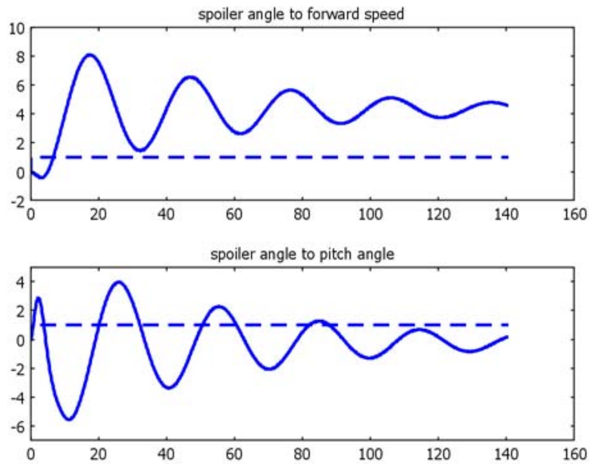
The convention is that the impulse comes in all input dimensions, which is perhaps not so practically interesting. It is more relevant to check the impulse response from a certain input, for instance from the spoiler angle:

```
impulse(M(:,1));
```



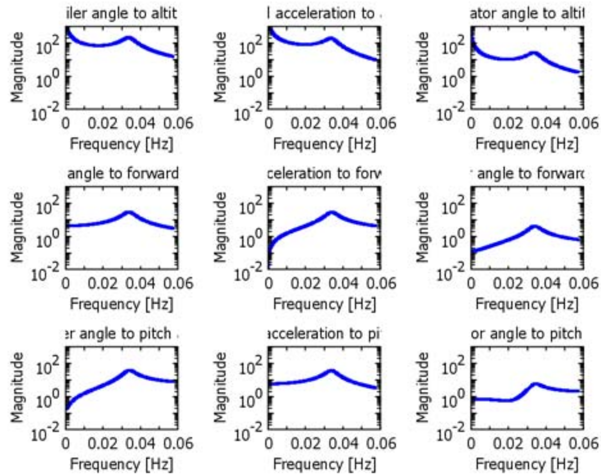
Similarly, the step response from spoiler angle to forward speed and pitch angle is given by:

```
step(M(2:3,1));
```



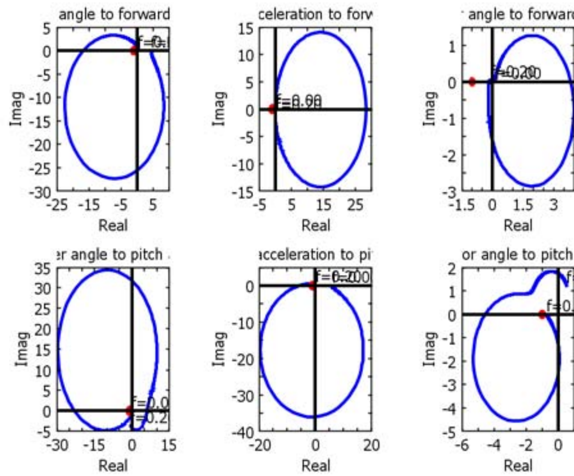
The system is clearly poorly damped and marginally stable, because the altitude includes an integrator. To design a controller, first compute the Bode amplitude curve of the system:

```
bodeamp(M,'Ylim',[1e-2 1e3])
```



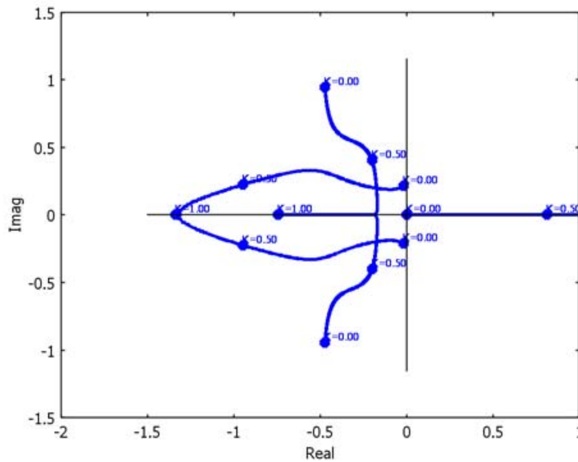
The Nyquist array is, excluding the altitude output:

```
nyquist(M(2:3,:),'fmax',0.2)
```



The root-locus plot reveals that a simple P-controller cannot stabilize the system, because the pole at the origin moves into the right half plane.

```
rlplot(M,'Xlim',[-2 1],'Ylim',[-1.5,1.5],'Kmax',1,'Kgrid',[1 500
1000])
```



For state-space control approaches, first verify the controllability and observability of the system:

```
rank(obsv(M))
ans =
5
rank(ctrb(M))
ans =
5
```

The following example object provides one possible feedback design:

```
K=exlti('KAIRC');
info(K)
Name:    AIRC controller
Description:
Design by Maciejowski, see eq (4.171)-(4.174) in his book
Inputs:
   u1:  altitude
   u2:  forward speed
   u3:  pitch angle
Outputs:
   y1:  spoiler angle
   y2:  forward acceleration
   y3:  elevator angle
```

which in formatted form is

$$\dot{x}(t) = \begin{bmatrix} 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 1.10 & 0.00 & 0.28 & -10.72 & 0.00 \\ -0.69 & -0.00 & 0.32 & 0.00 & -10.72 \end{bmatrix} x(t) + \begin{bmatrix} 1.00 & 0.00 & 0.00 \\ 0.00 & 1.00 & 0.00 \\ 0.00 & 0.00 & 1.00 \\ 2.19 & 0.00 & 0.56 \\ -1.39 & -0.00 & 0.65 \end{bmatrix} u(t)$$

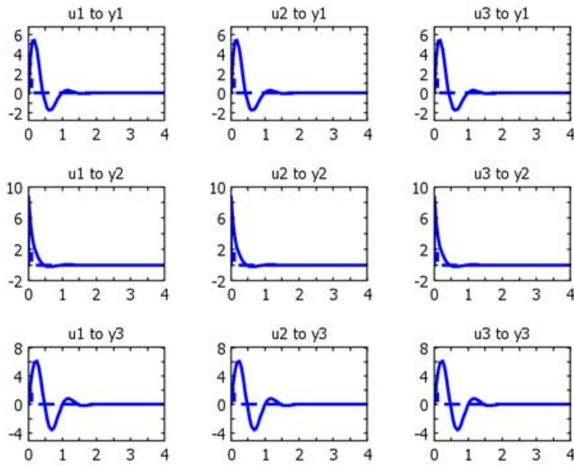
$$y(t) = \begin{bmatrix} -29.38 & 0.02 & 1.81 & 164.61 & -94.96 \\ -3.52 & 5.00 & -0.26 & 20.89 & -11.19 \\ -78.02 & -0.00 & -33.74 & 743.90 & 232.05 \end{bmatrix} x(t) + \begin{bmatrix} -58.76 & 0.03 & -3.61 \\ -7.04 & 10.00 & -0.51 \\ -156.03 & -0.01 & -67.48 \end{bmatrix} u(t)$$

Use `feedback` to compute the closed-loop system:

```
Mc=feedback(K*M,eye(3));
```

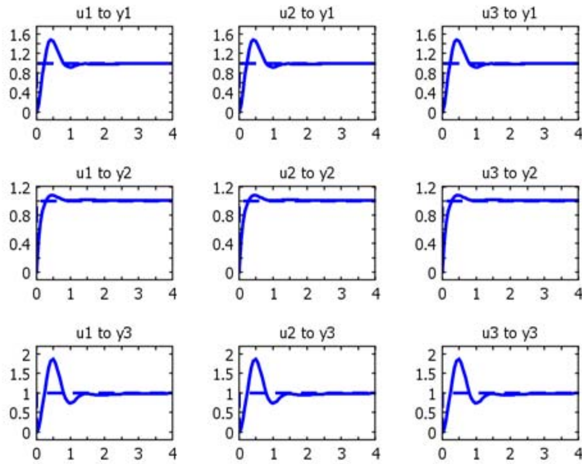
Then compute and plot the closed-loop impulse and step responses using

```
zimpulse=impulse(Mc);
plot(zimpulse,'Xlim',[0 4])
```



and

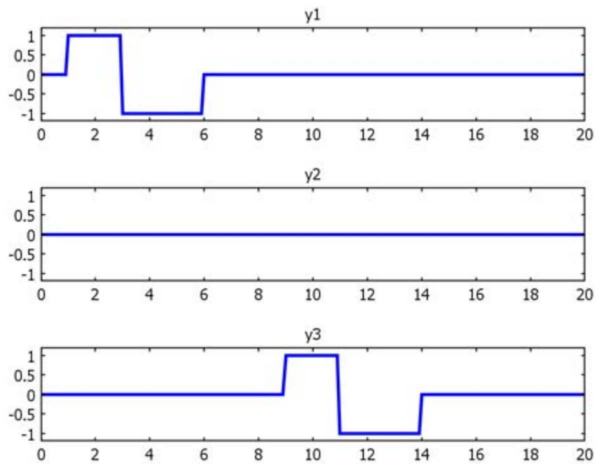
```
zstep=step(Mc);
plot(zstep,'Xlim',[0 4])
```



respectively.

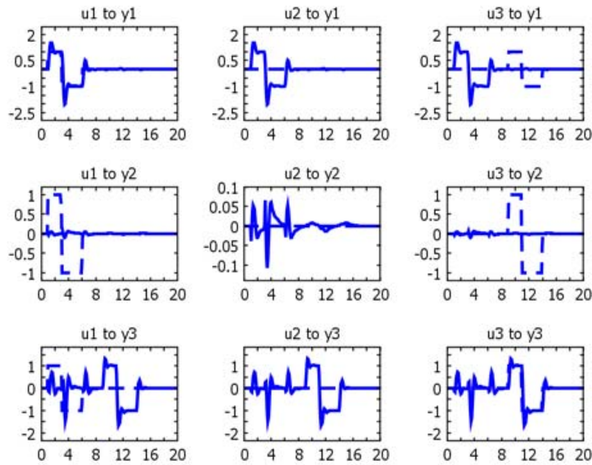
To simulate a test case, you must first specify an input sequence. Here, assume that the system is excited by steps up and down, first in then spoiler angle and then in the elevator angle.

```
u=zeros(14,3);
u(3:6,1)=[1 1 -1 -1];   % Steps down and up in spoiler angle
u(9:12,3)=[1 1 -1 -1];  % Steps down and up in elevator angle
t=[0 0.9 1 2.9 3 5.9 6 8.9 9 10.9 11 13.9 14 20]';
zin=sig(u,t);
plot(zin)
```



The simulation function accepts nonuniformly sampled input but assumes piecewise linear input signals for continuous-time systems (the simulation uses first-order hold fast sampling internally). The simulation looks as follows:

```
zout=simulate(Mc,zin);
plot(zout)
```



The closed-loop system reacts correctly and quickly to the input maneuvers. The cross-couplings are probably sufficiently small.

Investment Model

Consider the following investment offer: Each share gives interest every bank day. You have the choice how much of the interest is cashed, the other part is re-invested. A distinguishing condition in this offer is that the money that is invested stays only for six months. After that, the investment is gone, but its interest is left and working for six month per reinvestment. Based on recent data, the average interest is 2% per bank day. Given these information and assuming that the interest stays at this extreme level, the following example shows how to create a model and use it for deciding how much to reinvest.

A model with months as time scale will be based on the following assumptions:

- There are 20 bank days per month, which gives an interest rate of 50% ($1.02^{20}=1.5$) per month (aggregation of states).
- The capital each month lives for size months (decimation of sampling interval).

These two assumptions lead to a model with six states. Each state k keeps track of the capital of age k months. The first state contains the interest return, $u(t)$ is a pulse

excitation corresponding to the initial investment, and $y(t)$ is the total capital giving interest.

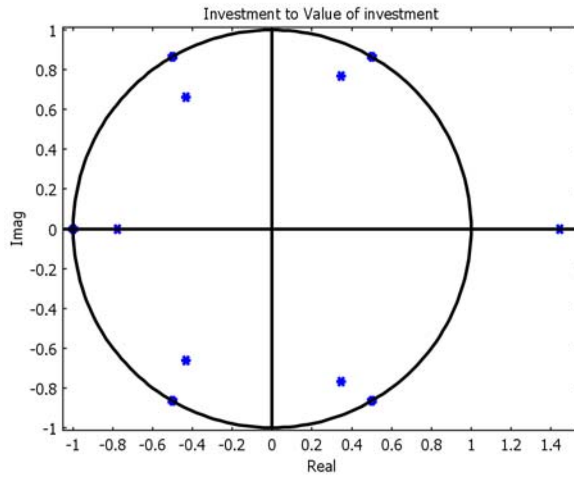
```
A=[0.5*ones(1,6);eye(5) zeros(5,1)];
B=[1;zeros(5,1)];
C=ones(1,6);
m=ss(A,B,C,0,1)
           / 0.5  0.5  0.5  0.5  0.5  0.5 \        / 1 \
           |   1    0    0    0    0    0 |        | 0 |
           |   0    1    0    0    0    0 |        | 0 |
  x[k+1] = |   0    0    1    0    0    0 | x[k] + | 0 | u[k]
           |   0    0    0    1    0    0 |        | 0 |
           \   0    0    0    0    1    0 /        \ 0 /

y[k] = (1  1  1  1  1  1) x[k] + (0) u[k]


m.name='Off-shore investment model';
m.ylabel{1}='Value of investment';
m.ulabel{1}='Investment';
m.tlabel='Time [months]';
m.xlabel={'Capital of age 1','Capital of age 2','Capital of age
3','Capital of age 4','Capital of age 5','Capital of age 6'};
```

For curiosity, the transfer function of this model is

```
G=tf(m)

                1*z^5+1*z^4+1*z^3+1*z^2+1*z+1
   Y(z) = -------------------------------------------- U(z)
          z^6-0.5*z^5-0.5*z^4-0.5*z^3-0.5*z^2-0.5*z-0.5
```

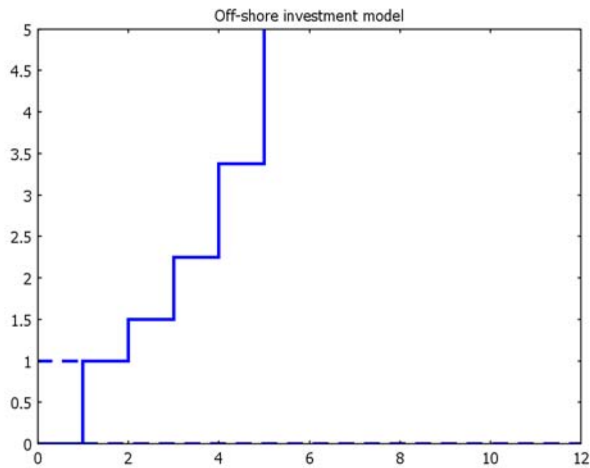
The following function plots the poles and zeros:

```
zpplot(m)
```



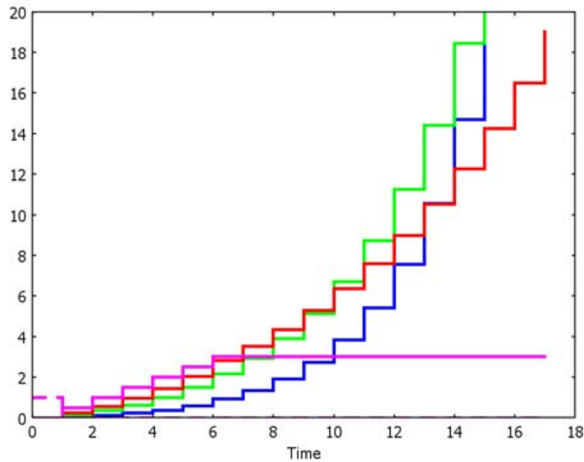
There is a pole outside the unit circle, which indicates that the capital grows to infinity according to the model. The impulse response, corresponding to an initial investment of one unit confirms this.

```
r0=impulse(m);
staircase(r0,'Xlim',[0 12],'Ylim',[0 5]);
```



The interesting question is what the return on investment is, not how much capital there is on the account. This is a feedback control problem. The model assumes that all interest is reinvested. If you cash 25% each day, the first row of the A matrix must be decreased by 0.25. The accumulated return is the sum of the capital each month ($y(t)$) times 25%. The next lines compute the return in cash corresponding to cashing 5%, 15%, and 25% each month, respectively.

```
m.ylabel={'Return'};
S=ss('sum');
rates=[0.05 0.15 0.25 0.5];
for k=1:length(rates)
   mc{k}=rates(k)*S*feedback(m,rates(k));
   accreturn{k}=impulse(mc{k},18);
   mc{k}=rates(k)*S*feedback(m,rates(k));
   accreturn{k}=impulse(mc{k},18);
   mc{k}=rates(k)*S*feedback(m,rates(k));
   accreturn{k}=impulse(mc{k},18);
   mc{k}=rates(k)*S*feedback(m,rates(k));
   accreturn{k}=impulse(mc{k},18);
end
staircase(accreturn{:},'Xlim',[0 18],'Ylim',[0 20]);
```



The solution to the control problem depends on what to expect from the company. If it does not exist after half a year, the optimal solution is to cash all interest, which gives back 3 units (50% times six months).

Uncertain Systems

This section illustrates the approaches to obtaining and analyzing uncertain systems. At the same time, it shows you can use system analysis tools to get a feeling for the system uncertainty and further to gain confidence in design of, for instance, control systems.

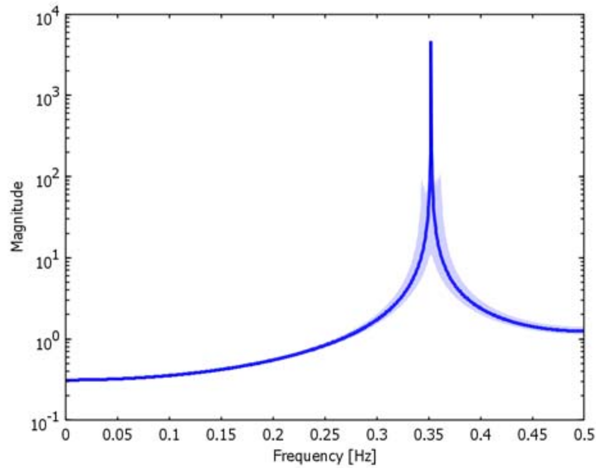
There are basically two ways to obtain uncertain systems:

- By setting coefficients in the model to a distribution from the PDFCLASS (see the section "Direct Definition of Uncertainty" below). This is useful in robustness analysis, for instance, where it is of interest to evaluate the sensitivity to certain model parameter.
- Uncertainty from model estimation (see "Estimation" on page 136). This is essential in system design based on estimated models ("identification for control"), where the estimation uncertainty is crucial for evaluating the design.

Direct Definition of Uncertainty

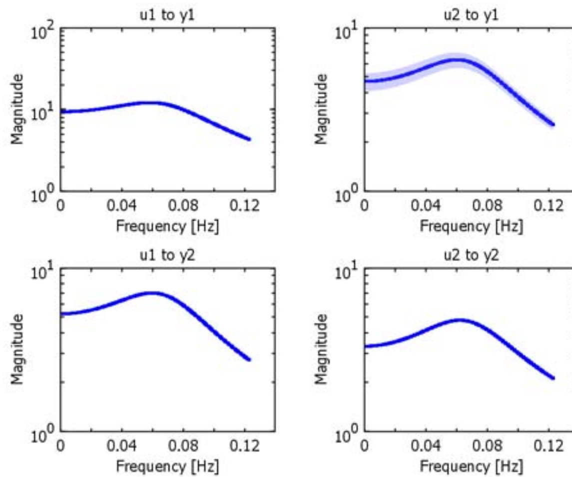
The method `uncertain` is available for setting parameters in a transfer function to a probability density function (PDF). You can use any of the PDFs that are available in the Signals & Systems Lab, or you can create one yourself. The following example illustrates how to change the parameter $a(2)$, which affects the pole angle of a resonant system, from 1.2 to a uniform distribution. The nominal system in the result gets `a(2)=E(udist(1.1,1.3)=1.2`; that is, it does not change. The Monte Carlo samples of the system in the field `Gu.sysMC` get random values of $a(2)$ taken from this distribution. This uncertainty is propagated to the default plot method (Bode amplitude) and all other subsequent model operations and visualization tools.

```
G=tf(1,[1 1.2 1],1);
Gu=uncertain(G,'a(2)',udist(1.1,1.3),100)

                 1
   Y(z) = ----------- U(z)
          z^2+1.2*z+1

plot(Gu)
```



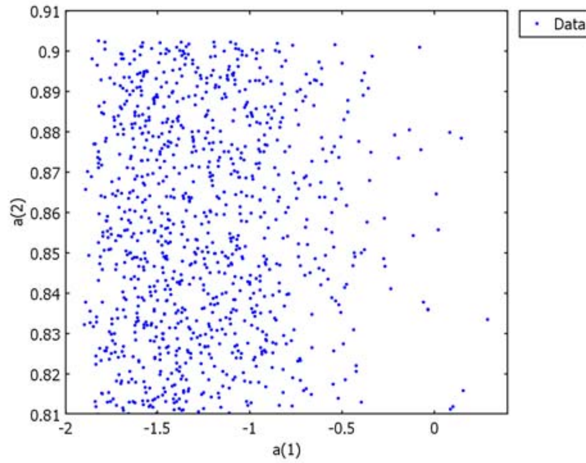
Using a multivariate statistical distribution, you can set correlated parameters simultaneously. The following example shows how to make a random MIMO(2,2) object uncertain by changing two numerator coefficients:

```
X=ndist([1;1],0.01*[1 0.8;0.8 1]);
G=rand(tf([2 2 1 2 2]));
Gu=uncertain(G,'b(1,1,1) b(1,2,2)',X,100)

               s+2
   Y1(s) = -------------- U1(s)
           s^2+0.4*s+0.21

               s+1
   Y1(s) = -------------- U2(s)
           s^2+0.4*s+0.21

              s+1.1
   Y2(s) = -------------- U1(s)
           s^2+0.4*s+0.21

              s+0.71
   Y2(s) = -------------- U2(s)
           s^2+0.4*s+0.21

plot(Gu)
```



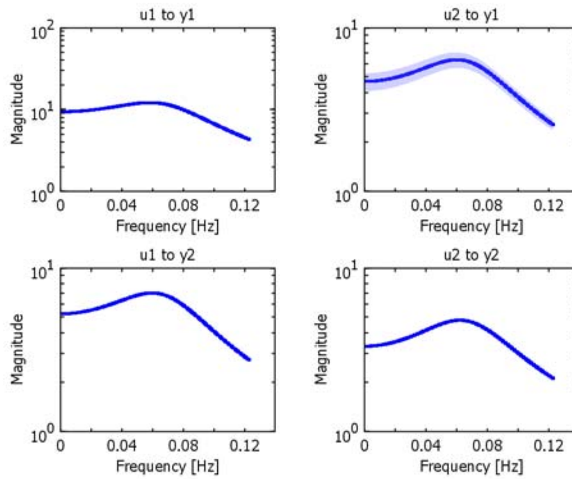
For a more practical example of correlated coefficients, assume that the complex-conjugated pole pair in a second-order resonant discrete-time system has an uncertain location. The model uses a uniform distribution to describe the radius to the origin and a Gaussian distribution with a relatively large uncertainty to describe the angle. This might correspond to a spring-damper system, where the damper coefficient is known quite well, while the spring constant varies over a large range. The code mixes the two independent stochastic variables representing pole radius and angle to a two-dimensional stochastic variable Z, which corresponds to the theoretical relation between pole location and denominator polynomial coefficients. A plot shows the two-dimensional distribution of $a(1)$ and $a(2)$.

```
r=udist(0.9,0.95);   % Pole distance to origin
phi=ndist(pi/4,0.1); % Pole angle
Z=[-2*r*cos(phi); r.^2];
Z.xlabel={'a(1)','a(2)'};
plot2(Z)
```



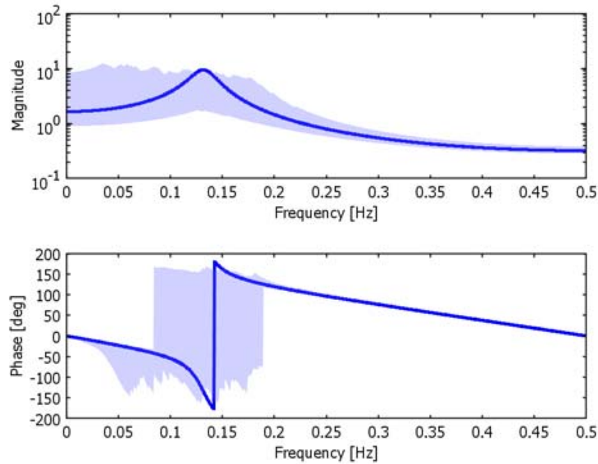
The correlation between the coefficients might not be that obvious from this plot. However, after having introduced this uncertainty to the transfer function, a scatter plot of the poles shows a banana-shaped distribution.

```
G=tf(1,[1 1 1],1);   % Arbitrary a(2) and a(3)
Gu=uncertain(G,'a(2) a(3)',Z,100)

                 1
   Y(z) = -------------- U(z)
          z^2-1.2*z+0.86

zpplot(Gu,'scatter','on')
```



The Bode diagram in the following figure reveals that system parameters as amplitude and phase margin are very uncertain and distributed over a large range of possible values.

```
bode(Gu)
```



The uncertainty representation can at any time be removed using the fix method. This is one action that can be taken to speed up simulations and other processing based on the LTI object.

Estimation

Estimation as system identification of black-box models, where all parameters are free parameters, naturally leads to an uncertain system, where uncertainty is represented by either the covariance matrix of the parameter vector, or by Monte Carlo samples of the parameter vector, depending on which model structure and function that is used for estimation. This is described in detail in ARX Models on page 196 and TF Models on page 206.

Stochastic State-Space Objects

It is possible to model stochastic systems using stochastic state-space objects (SS objects), which allow for *process noise* $v(t)$ and *measurement noise* $e(t)$:

$$
\begin{aligned}
x(t+1) &= Ax(t) + Bu(t) + v(t), \\
y(t) &= Cx(t) + Du(t) + e(t), \\
&\mathrm{Cov}(x(0)) = P_0,\ E(x(0)) = x_0, \\
\mathrm{Cov}(v(t)) &= Q,\ \mathrm{Cov}(e(t)) = R,\ \mathrm{Cov}(v(t), e(t)) = S.
\end{aligned}
$$

The matrix triplet (Q, R, and S) summarizes the stochastic properties. The stochastic SS object is similar to the deterministic one, and this section describes the main extensions. Basically, you must complement all operations with rules for the noise models. Another main difference is state estimation using Kalman filters.

Operations on Stochastic LTI Objects

A state-space model provides a general description of stochastic systems. Not all overloaded functions for the deterministic LTI objects apply to the stochastic ones. For instance, it is not possible to defined an exact inverse. This section provides a summary of the methods that support stochastic objects and the rules for the Q, R, and S matrices (you can treat the deterministic part just as in “Operations on LTI Objects” on page 95).

PLUS

Parallel connection leads to the following rule:

$$
\begin{aligned}
(A_1, B_1, C_1, D_1, Q_1, R_1, S_1) + (A_2, B_2, C_2, D_2, Q_2, R_2, S_2) &= \\
(A, B, C, D, Q, R, S), \\
Q &= \begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix}, \\
R &= R_1 + R_2, \\
S &= \begin{bmatrix} S_1 \\ S_2 \end{bmatrix}.
\end{aligned}
$$

MULTIPLICATION

Series connections result in the following system:

$$(A_1, B_1, C_1, D_1, Q_1, R_1, S_1)*(A_2, B_2, C_2, D_2, Q_2, R_2, S_2) = (A, B, C, D, Q, R, S),$$

$$Q = \begin{bmatrix} Q_1 & S_1 B_2^T \\ B_2 S_1^T & Q_2 + B_2 R_1 B_2^T \end{bmatrix},$$

$$R = R_2 + D_2 R_1 D_2^T,$$

$$S = \begin{bmatrix} S_1 D_2^T \\ S_2 + B_2 S_1 D_2^T \end{bmatrix}.$$

APPEND

Appending two systems is straightforwardly implemented as:

$$\text{append}((A_1, B_1, C_1, D_1, Q_1, R_1, S_1), (A_2, B_2, C_2, D_2, Q_2, R_2, S_2)) = (A, B, C, D, Q, R, S),$$

$$Q = \begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix},$$

$$R = \begin{bmatrix} R_1 & 0 \\ 0 & R_2 \end{bmatrix},$$

$$S = \begin{bmatrix} S_1 & 0 \\ 0 & S_2 \end{bmatrix}.$$

FEEDBACK

Feedback is the algebraically most complicated case, and the rule is as follows:

$$\text{feedback}((A_1, B_1, C_1, D_1, Q_1, R_1, S_1), (A_2, B_2, C_2, D_2, Q_2, R_2, S_2)) = (A, B, C, D, Q, R, S),$$

$$Q = \begin{bmatrix} Q_1 + B_1 D_2 R_1 D_1^T - S_1 D_1^T + B_1 R_2 B_1^T & S_1 B_2^T - B_1 D_2 R_1 B_2^T - B_1 S_2^T + B_1 R_2 D_2^T \\ B_2 S_1^T - B_2 R_1 D_1^T - S_2 B_1^T + B_2 D_1 R_2 B_1^T & Q_2 + B_2 D_1 R_1 D_2^T + B_2 R_1 B_2^T - S_2 D_2^T \end{bmatrix},$$

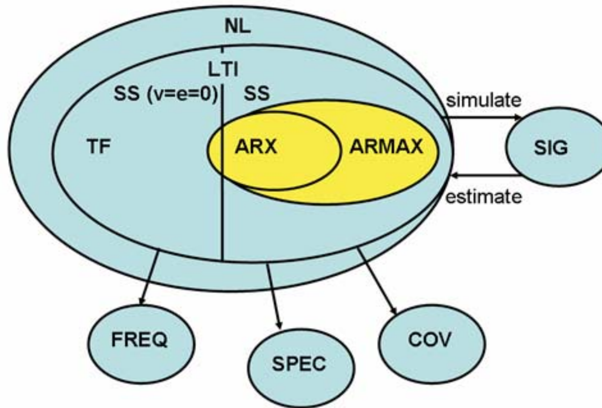
$$R = R_1 + D_1 R_2 D_1^T,$$

$$S = \begin{bmatrix} S_1 - B_1 D_2 R_1 - B_1 R_2 D_1^T \\ B_2 R_1 - S_2 D_1^T + B_2 D_1 R_2 D_1^T \end{bmatrix}.$$

Nonlinear Model Objects

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

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

and in discrete time

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

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. Example: `f='-th*x^2';`
  - An inline function. Example: `f=inline('-x^2','t','x','u','th');`
  - An M-file. Example:

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

  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 apparently trivial information must be provided by the user, since it is hard to unambiguously 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 used 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.
- `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 below.

TABLE 3-8: NL OBJECT METHODS

METHOD	DESCRIPTION
`ARRAYREAD`	Used to pick out subsystems by indexing. Ex: m(2,:) picks out the dynamics from all inputs 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 DASPK
`NL2SS`	Returns a linearized state space model
`EKF`	Implements the extended Kalman filter for state estimation
`GENFILT`	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

The filtering methods are described in detail in Chapter 4, “State Estimation and Kalman Filtering.”

The most fundamental usage of the NL objects is illustrated with a couple of examples:

- The van der Pol system illustrates definition of a second 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.

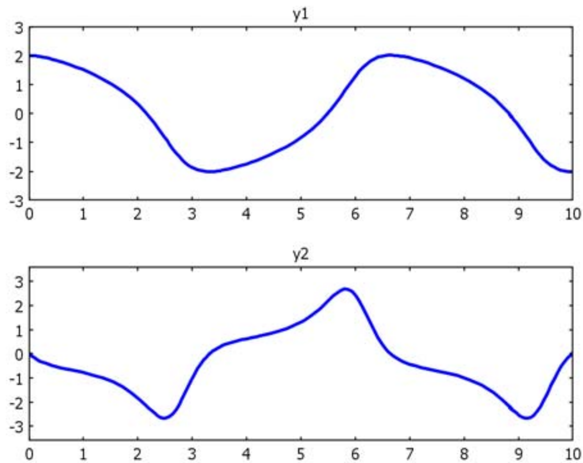
- NL objects as provided after conversion from LTI objects.
- A first-order discrete-time nonlinear model used in many papers on particle filtering.

Examples of Model Definition and Simulation

VAN DER POL SYSTEM

First, a nonlinear dynamic system known as the van der Pol equations are 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')`.

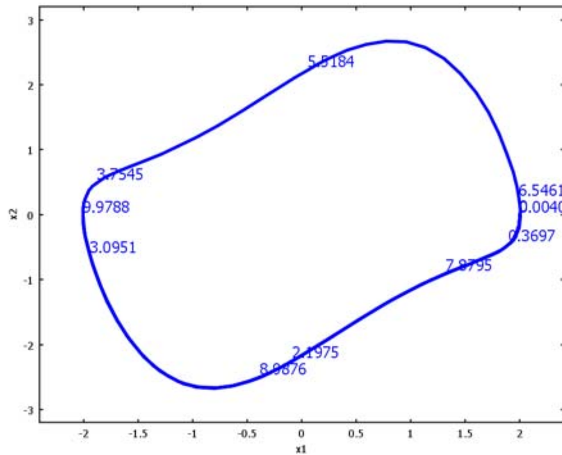
```
f='[x(2);(1-x(1)^2)*x(2)-x(1)]';
h='x';
m=nl(f,h,[2 0 2 0]);
NL constructor warning: try to vectorize f for increased speed
m.name='Van der Pol system';
m.x0=[2;0];
m
NL object: Van der Pol system
 dx/dt = [x(2);(1-x(1)^2)*x(2)-x(1)]
     y = x
   x0' = [2         0]

y=simulate(m,10);
plot(y)
```



The constructor checks if the sizes in `nl` 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 evaluated, 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(m,10); toc

Elapsed time:  0.031 s

tic, simulate(m2,10); toc

Elapsed time:  0.047 s
```

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

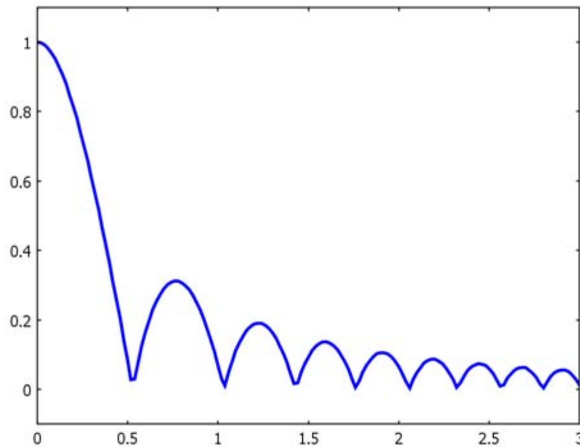
Another reason to use the notation is for the `genfilt` method, where an implicit state augmentation forces the user to specify state indices explicitly. For this purpose, it is also important to avoid the single colon operator and to replace any `end` with `nx`.

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')`).

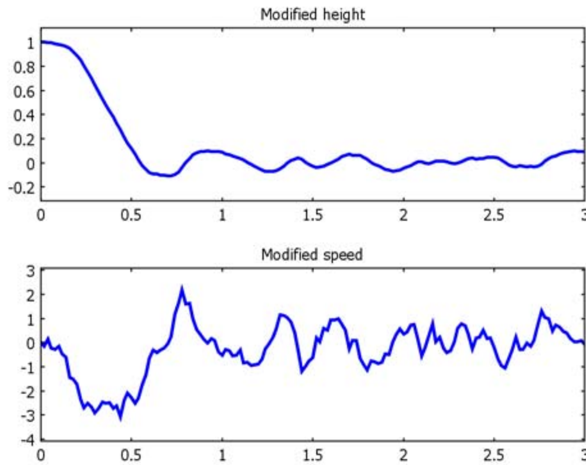
```
f='[x(2,:);-th(1)*x(2,:).^2.*sign(x(2,:))-9.8.*sign(x(1,:))]';
h='abs(x(1,:))';
m=nl(f,h,[2 0 1 1]);
m.x0=[1;0];
m.th=1;
m.name='Bouncing ball';
m.xlabel={'Modified height','Modified speed'};
m.ylabel='Height';
m.thlabel='Air drag';
m
NL object: Bouncing ball
 dx/dt = [x(2,:);-th(1)*x(2,:).^2.*sign(x(2,:))-9.8.*sign(x(1,:))]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

  States:  Modified height     Modified speed
  Outputs: Height
  Param.: Air drag
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
 dx/dt = [x(2,:);-th(1)*x(2,:).^2.*sign(x(2,:))-9.8.*sign(x(1,:))]
+ N([0;0],[0,0;0,0.1])
      y = abs(x(1,:)) + N(0,0.1)
    x0' = [1          0]
    th' = [1]

   States:  Modified height     Modified speed
   Outputs: Height
   Param.: Air drag
y=simulate(m,0:0.02:3);
xplot(y)
```

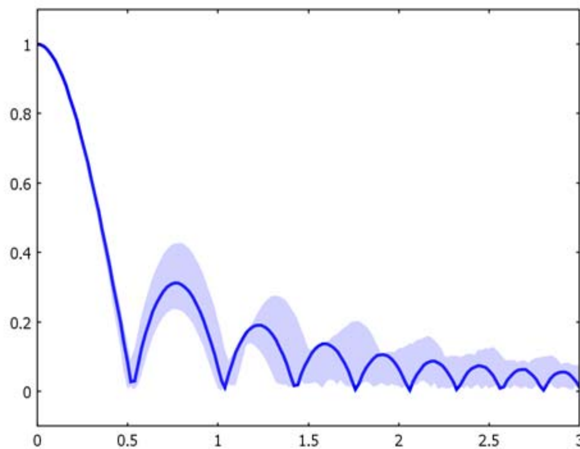


Note that the process and measurement noise are printed out symbolically by the `display` function.

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
 dx/dt = [x(2,:);-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.: Air drag
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).

The section “Bouncing Ball” on page 219 illustrates how to calibrate the air drag parameter from observations of the ball.

CONVERSION FROM LTI OBJECTS

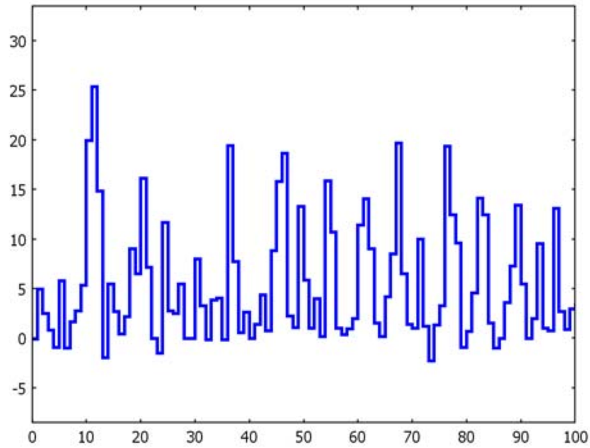
Since the class of LTI models is a special case of NL models, any LTI object (SS or TF) can be converted to an NL object:

```
nl(rand(ss([2 1 0 1])))
NL object
 dx/dt = [-0.2846946691911659 1 ; -0.028584710363888835
0]*x(1:2,:)+[-0.14216243543734491 ; 0.11469697363152677]*u(1:1,:)
+ N([0;0],[0,0;0,0])
     y = [1 0]*x(1:2,:)+1*u(1:1,:) + N(0,0)
   x0' = [0         0]
```

A BENCHMARK EXAMPLE FOR 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 available as `m=exnl('vdp')`).

```
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(t+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)

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 `fs` is set, the meaning of `f` changes from continuous to discrete time.

State estimation in this model is treated in “The Standard Example” on page 166.

4

State Estimation and Kalman Filtering

This chapter describes the functionality for state estimation and Kalman filtering.

State Estimation

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.

Kalman Filtering for SS Models

ALGORITHM

For a linear state space model defined by

$$\begin{aligned} x_{k+1} &= A_k x_k + B_{u,k} u_k + B_{u,k} v_k, \\ y_k &= C_k x_k + D_k u_k + e_k, \\ \mathrm{Cov}(x_0) = P_0,&\; E(x_0) = \hat{x}_{1|0}, \\ \mathrm{Cov}(v_k) = Q_k, \mathrm{Cov}(e_k) &= R_k, \mathrm{Cov}(v_k, e_k) = 0. \end{aligned}$$

the optimal linear filter is given by the Kalman filter (KF) recursions

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

The inputs to the KF are the SS object and a SIG object containing the observations y_k and possible an input u_k, and the outputs are the state estimate $x_{k|k}$ and its covariance matrix $P_{k|k}$. There is also a possibility to predict future states $x_{k+m|k}, P_{k+m|k}$ with the m-step ahead predictor, or to compute the smoothed estimate using the

complete observation record $x_{k|N}$, $P_{k|N}$. The corresponding output estimate and covariance are also computed. All these quantities are packed into a SIG object, where also the signal labels inherited from the model are assigned.

USAGE

Call the KF with

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

The arguments are as follows:

- `m` is a SS 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.

TABLE 4-1: KALMAN FUNCTION PROPERTIES

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, generally k>0 gives xhat(t+k|t) and 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.

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 SS object for filtering.

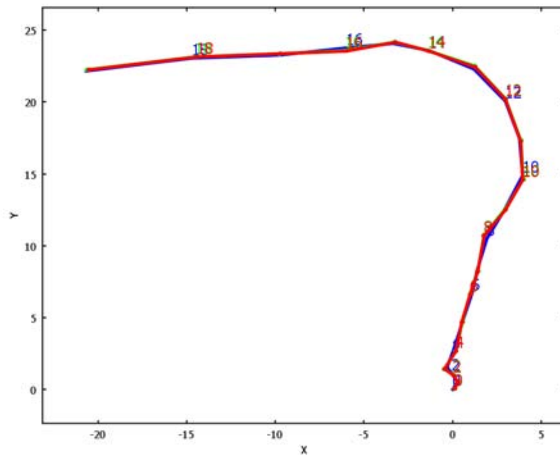
```
m=exlti('CV2D');  % Pre-defined model, or...
T=1;
A=[1 0 T 0; 0 1 0 T; 0 0 1 0; 0 0 0 1];
B=[T^2/2 0; 0 T^2/2; T 0; 0 T];
C=[1 0 0 0; 0 1 0 0];
R=0.01*eye(2);
m=ss(A,[],C,[],B*B',R,1/T);
m.xlabel={'X','Y','vX','vY'};
m.ylabel={'X','Y'};
m.name='Constant velocity motion model';
m
           / 1  0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

First, simulate 20 samples.

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

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

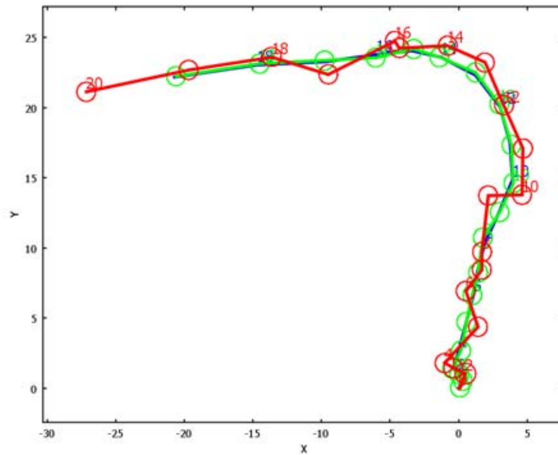
```
xhat10=kalman(m,z,'alg',1,'k',0);
xhat20=kalman(m,z,'alg',2,'k',0);
xhat40=kalman(m,z,'alg',4,'k',0);
xplot2(z,xhat20,xhat40,'conf',99,[1 2])
```



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.

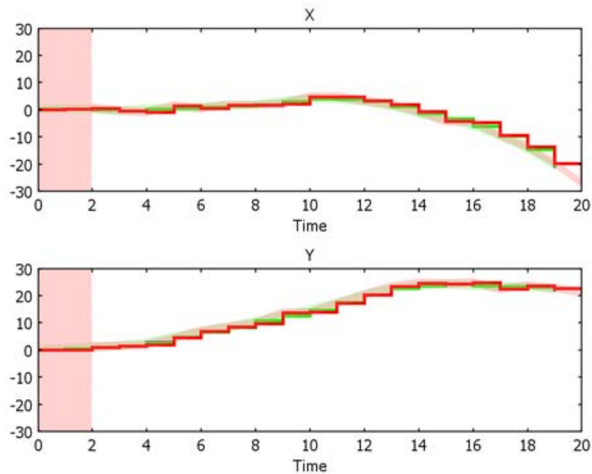
The time-varying and stationary Kalman filter can be used for one-step ahead prediction.

```
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 is predicted.

```
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.

ALGORITHM

An NL object of a nonlinear time-varying model

$$x_{k+1} = f(k, x_k, u_k; \theta) + v_k,$$
$$y_k = h(k, x_k, u_k, e_k; \theta).$$

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 can be used. In total, the EKF implements the following recursion (where some of the arguments to f and h are dropped for simplicity):

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

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 co variance `R` to compensate for higher order nonlinearities in the measurement equation.

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

The optional parameters are summarized in the table below.

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.

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.

TARGET TRACKING

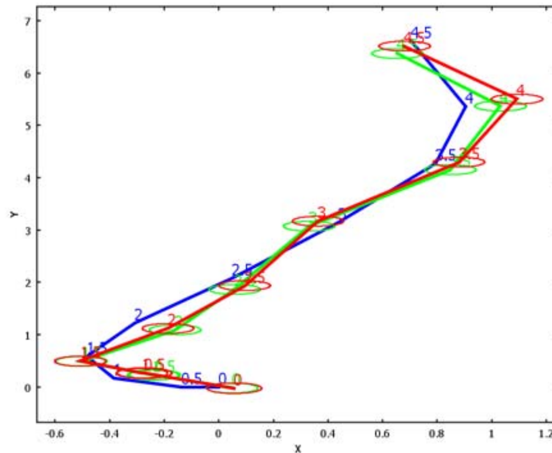
Since SS is a special case of NL objects, the Kalman filter is a kind of special case of the EKF method. To illustrate this, let us return to the previous tracking example. All SS objects can be converted to an NL object.

```
mss=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      0.063          0\
             |     0     0.016          0      0.063 |
Q = Cov(v) = | 0.063         0       0.25          0 |
              \    0     0.063          0       0.25/

             / 0.01         0\
R = Cov(e) = \    0      0.01/

mnl=nl(mss)
NL object: Constant velocity motion model
x(t+1) = [1 0 0.5 0 ; 0 1 0 0.5 ; 0 0 1 0 ; 0 0 0 1]*x(1:4,:) +
N([0;0;0;0],[0.0156,0,0.0625,0;0,0.0156,0,0.0625;0.0625,0,0.25,0;
0,0.0625,0,0.25])
     y = [1 0 0 0 ; 0 1 0 0]*x(1:4,:) + N([0;0],[0.01,0;0,0.01])
   x0' = [0          0          0          0]

  States:  X     Y     vX     vY
  Outputs: X     Y
```

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

```
z=simulate(mss,10);
zhat1=kalman(mss,z);
zhat2=ekf(mnl,z);
xplot2(z,zhat1,zhat2,'conf',90)
```



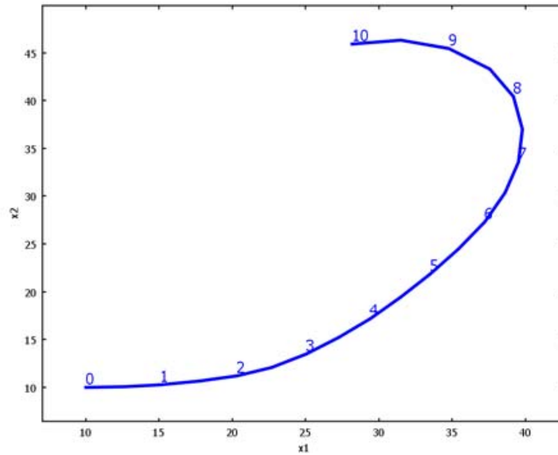
Except for some numerical differences, the results are comparable.

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(t+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,:)] +
N([0;0;0;0;0],[0,0,0,0,0;0,0,0,0,0;0,0,0.1,0,0;0,0,0,0,0;0,0,0,0,
0.01])
     y = [sqrt(x(1,:).^2+x(2,:).^2);atan2(x(2,:),x(1,:))] +
N([0;0],[1,0;0,0.003])
   x0' = [10         10          5          0        0.1] +
N(0,[10,0,0,0,0;0,10,0,0,0;0,0,1e+002,0,0;0,0,0,10,0;0,0,0,0,1])

  States:  x1     x2     v     h     w
  Outputs: R     phi
```

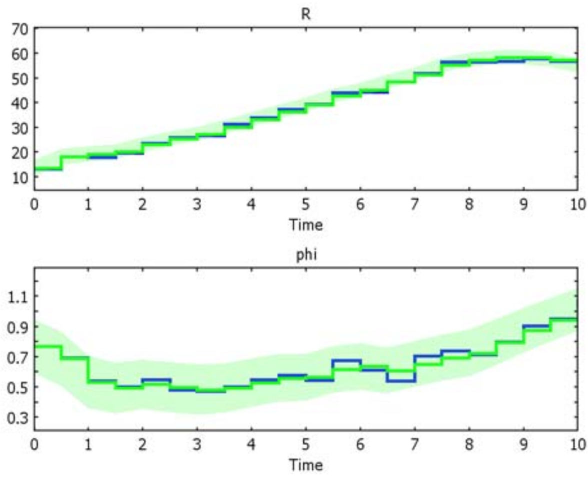
The measurement model assumes range and bearing sensors as in for instance a radar. These are nonlinear functions of the state vector. Also, the state dynamics is nonlinear. The model is simulated, then the first two states (`ind=[1 2]` is default) are plotted.

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



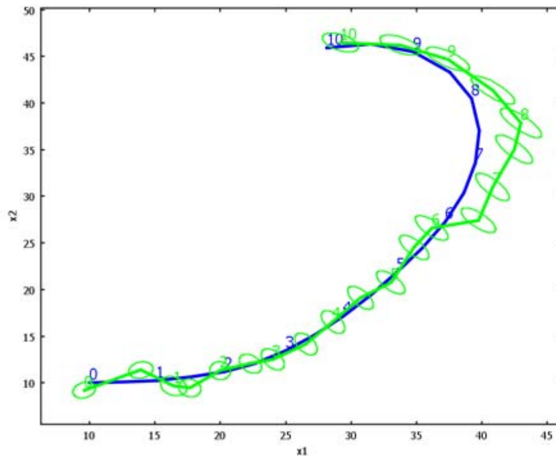
The EKF is applied, and the estimated output is compared to the measured output with a confidence interval.

```
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])
```



ALGORITHM

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

$$x_{k+1} = f(k, x_k, u_k; \theta) + v_k,$$
$$y_k = h(k, x_k, u_k, e_k; \theta).$$

consists of the following recursions:

$$p(x_t|y_{1:t}) = \frac{p(y_t|x_t)p(x_t|y_{1:t-1})}{p(y_t|y_{1:t-1})}$$
$$p(x_{t+1}|y_{1:t}) = \oint p(x_{t+1}|x_t)p(x_t|y_{1:t})dx_t$$
$$p(y_t|y_{1:t-1}) = \oint p(y_t|x_t)p(x_t|y_{1:t-1})dx_t$$

The particle filter (PF) approximates the infinite dimensional integrals by stochastic sampling techniques, which leads to a (perhaps surprisingly simple) numerical solution.The general PF algorithm consists of the following recursion:

Design parameters: Number of particles N. Importance resampling threshold N_{th} $N_{th} = N$ for SIR). Resampling algorithm (default is simple resampling). Proposal density for prediction (default is the state dynamics).

Initialization: Take N state vectors (particles) at random, according to the state prior distribution.

1 *Measurement update*: For each particle, update the weights by the likelihood function

$$w_k^i = w_{k-1}^i \frac{p(y_k|x_k^i)p(x_k^i|x_{k-1}^i)}{q(x_k^i|x_{k-1}^i, y_k)} = w_{k-1}^i p(y_k|x_k^i),$$

The latter expression holds for

$$q(x_k^i|x_{k-1}^i, y_k) = p(x_k^i|x_{k-1}^i).$$

2 *Normalization*: make sure the weights sum to one.

3 *Estimation*: approximate the conditional mean with

$$\hat{x}_k \approx \sum_{i=1}^{N} w_k^i x_k^i .$$

4 *Resampling*:

- *Bayesian bootstrap or Sampling Importance Resampling* (SIR): Take N samples with replacement with probability according to the weights.
- *Importance sampling:* Only resample when the efficient number of particles

$$N_{\text{eff}} = \frac{1}{\sum_i (w_k^i)^2} < N_{\text{th}} .$$

is smaller than a threshold.

5 *Time update or prediction*: Generate samples from a proposal distribution

$$x_{k+1}^i \in q(x_k | x_{k-1}^i, y_k) .$$

The dynamic equation is the standard proposal, in which case prediction is a simulation

$$v_k^i \sim p_{v_k},$$
$$x_{k+1}^i = f(x_k^i, v_k^i).$$

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.

USAGE

The PF is called with

```
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 the table below.

TABLE 4-2: OPTIONAL PF PARAMETERS

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 states x(ind)

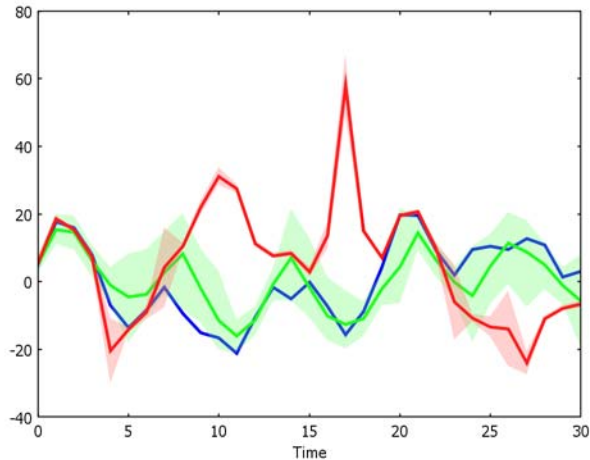
THE STANDARD EXAMPLE

The seminal paper by Neil Gordon in 1993 used a simple first-order example as illustration, and this example has been revisited many times since then in literature. The model has already been presented, and is part of the model example library:

```
m=exnl('pfex')
NL object
x(t+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)
```

The model is simulated and the state is estimated with both the EKF and PF. Note that some dithering is required.

```
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 particle filter gives good estimates with a reliable confidence region which covers the true state, whereas the EKF does neither deliver good estimates nor a useful confidence bound. Further, the EKF is more prone to diverge in longer simulations.

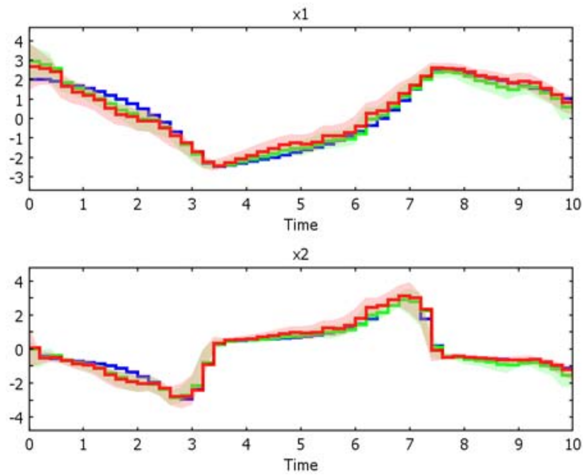
TRACKING IN THE VAN DER POL SYSTEM

The van der Pol system was used to illustrate the NL constructor, and a discrete counterpart will be used here for state tracking. The van der Pol equation is discretized using Euler sampling with a sampling frequency of 5 Hz.

```
m=exnl('vdpdisc')
NL object: Discretized van der Pol system (Euler Ts=0.2)
x(t+1) =
[x(1,:)+0.2*x(2,:);x(2,:)+0.2*((1-x(1,:).^2).*x(2,:)-x(1,:))] +
N([0;0],[0,0;0,0])
     y = x + N([0;0],[1,0;0,1])
   x0' = [2          0] + N(0,[10,0;0,10])
```

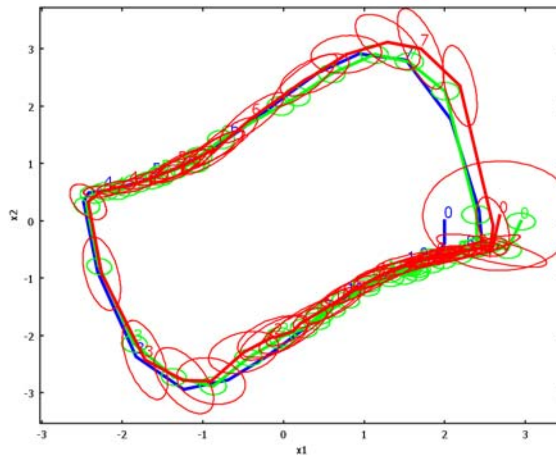
The system is simulated, and the EKF and PF (with dithering) state estimates are computed and compared.

```
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 with confidence ellipses are generated next.

```
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.

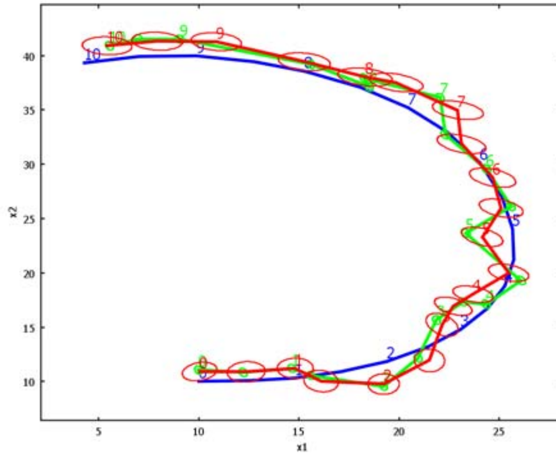
TARGET TRACKING

The target tracking example is here revisited.

```
m=exnl('ctpv2d')
NL object: Coordinated turn model with polar velocity
x(t+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,:)] +
N([0;0;0;0;0],[0,0,0,0,0;0,0,0,0,0;0,0,0.1,0,0;0,0,0,0,0;0,0,0,0,
0.01])
     y = [sqrt(x(1,:).^2+x(2,:).^2);atan2(x(2,:),x(1,:))] +
N([0;0],[1,0;0,0.003])
   x0' = [10        10         5          0        0.1] +
N(0,[10,0,0,0,0;0,10,0,0,0;0,0,1e+002,0,0;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.

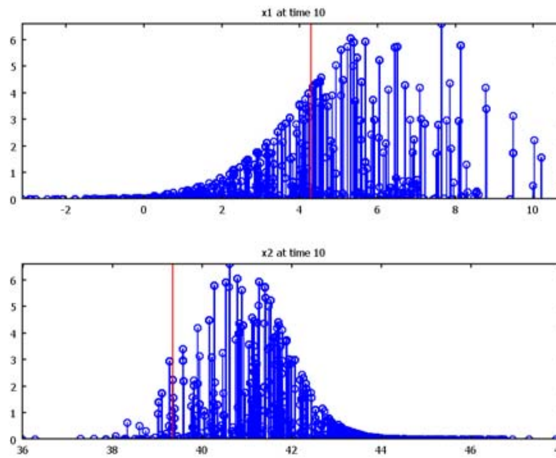
```
z=simulate(m,10);
zekf=ekf(m,z);
mpf=m;
mpf.pv=20*cov(m.pv);  % Dithering required for the PF
zpf=pf(mpf,z,'Np',1000);
xplot2(z,zpf,zekf,'conf',90)
```



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.

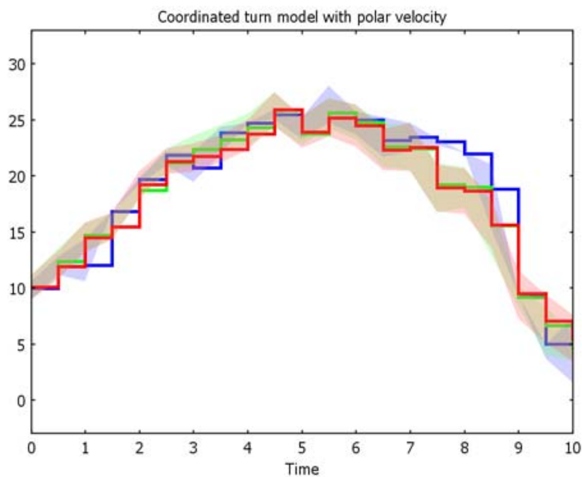
```
zpf=pf(mpf,z,'animate','on','Np',1000,'ind',[1 2]);
```



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 =
     0.8440     7.6870    79.3750
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.

Nonlinear Transformation-Based 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.

ALGORITHMS

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

$$X \in N(m_x, P_x) \Rightarrow Z = g(X) \approx N(m_z, P_z)$$

The following options exist:

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

$$\begin{aligned} m_z &= g(\hat{x}), \\ P_z &= [g'_i(\hat{x})P(g'_j(\hat{x}))^T]_{ij} \end{aligned}$$

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

$$\begin{aligned} m_z &= g(\hat{x}) + \frac{1}{2}[\operatorname{tr}(g''_i(\hat{x})P)]_i, \\ P_z &= \left[g'_i(\hat{x})P(g'_j(\hat{x}))^T + \frac{1}{2}\operatorname{tr}(Pg''_i(\hat{x})Pg''_j(\hat{x}))\right]_{ij} \end{aligned}$$

3 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 transformation, and the mean and covariance are fitted to these points.

$$\begin{aligned} x^0 &= \hat{x}, \\ x^{\pm i} &= \hat{x} \pm \sqrt{n_x + \lambda} P^{1/2}_{:,i}, \quad i = 1, 2, \ldots, n_x \\ z^i &= g(x^i), \quad i = -n_x, \ldots, n_x \\ m_z &= \frac{\lambda}{n_x + \lambda} z^0 + \sum_{i=\pm 1}^{\pm n_x} \frac{1}{2(n_x + \lambda)} z^i, \\ P_z &= \left(\frac{\lambda}{n_x + \lambda} + (1 - \alpha^2 + \beta)\right)(z^0 - \mathrm{E}(z))(z^0 - \mathrm{E}(z))^T \\ &\quad + \sum_{i=\pm 1}^{\pm n_x} \frac{1}{2(n_x + \lambda)}(z^i - \mathrm{E}(z))(z^i - \mathrm{E}(z))^T. \end{aligned}$$

4 MC in `ndist.mceval`: Generate N random points, transform these, and fit mean and covariance to these points.

$$x^i \in N(m_x, P_x),$$
$$z^i = g(x^i),$$
$$m_z = \frac{1}{N}\sum_{i=1}^{N} z^i,$$
$$P_z = \frac{1}{N}\sum_{i=1}^{N} (z^i - m_z))(z^i - m_z)^T.$$

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:

$$K_k = P^{xy}_{k|k-1}(P^{yy}_{k|k-1})^{-1},$$
$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(y_k - \hat{y}_{k|k-1}),$$
$$P_{k|k} = P_{k|k-1} - K_k P^{yy}_{k|k-1} K_k^T.$$

That is, if somebody can provide the matrices P^{xx} and P^{xy}, 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:

$$\dot{x} = (x_k^T, v_k^T)^T \in N((\hat{x}^T_{k|k}, 0^T)^T, \quad \text{diag}(P_{k|k}, Q)),$$
$$z = x_{k+1} = f(x_k, u_k) + v_k \approx N(\hat{x}_{k+1|k}, P_{k+1|k}).$$

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:

$$\dot{x} = (x_k^T, e_k^T)^T \in N((\hat{x}^T_{k|k}, 0^T)^T, \quad \text{diag}(P_{k|k}, R)),$$
$$z = (x_k^T, y_k^T)^T = (x_k^T, (h(x_k, u_k) + e_k)^T)^T \approx N((\hat{x}^T_{k|k-1}, \hat{y}^T_{k|k-1})^T, \quad P^z_{k|k-1}),$$

where P^z is partitioned into the blocks P^{xx}, P^{xy}, P^{yx}, and P^{yy}. Use these to compute the Kalman gain and measurement update.

USAGE

The algorithm is called with syntax

```
x=genfilt(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. `xhat = xhat+K*(y-yhat)` and `P = P-K*Pyy*K'`.

The transform in 1b and 2b can be chosen arbitrarily using `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:nx,:)` (avoid using `end`), even for scalar systems. The reason is that the state vector is augmented, so any unindexed `x` will cause errors.

TABLE 4-3: NL OBJECT PROPERTIES

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

TABLE 4-3: NL OBJECT PROPERTIES

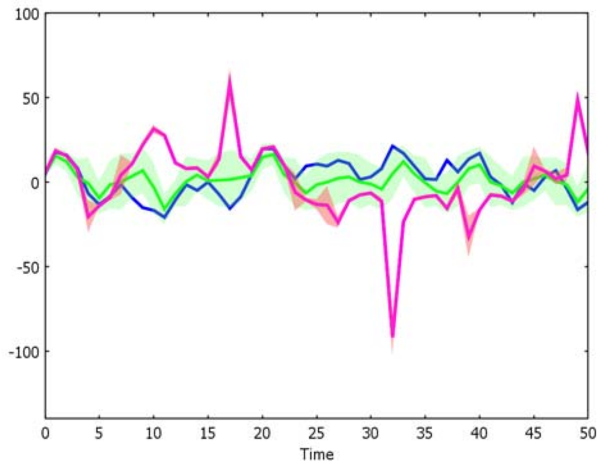
PROPERTY	VALUE	DESCRIPTION
Q	{[]}	Process noise covariance (overrides m.Q). Scalar value scales m.Q
R	{[]}	Measurement noise covariance (overrides m.R). Scalar value scales m.R
tup		Nonlinear transformation in the time update
	{'ut'}	The unscented Kalman filter (UKF) based on the uteval method of the ndist object
	'tt1'	Variant of the EKF, based on a first order Taylor expansion in the tt1eval method of the ndist object
	'tt2'	Second-order corrected EKF, based on a second-order Taylor expansion in the tt1eval method of the ndist object
	'mc'	Monte Carlo version of the EKF, based on Monte Carlo sampling in the mceval method of the ndist object
mup		Nonlinear transformation in the measurement update
	{'ut'}	The unscented Kalman filter (UKF) based on the uteval method of the ndist object
	'tt1'	Variant of the EKF, based on a first order Taylor expansion in the tt1eval method of the ndist object
	'tt2'	Second-order corrected EKF, based on a second-order Taylor expansion in the tt1eval method of the ndist object
	'mc'	Monte Carlo version of the EKF, based on Monte Carlo sampling in the mceval method of the ndist object
ukftype	{'std'}|'wan'	Standard or WAN unscented transform
ukfpar	{[]}	Parameters in UKF
		For std, par=w0 {w0=1-n/3}
		For wan, par=[beta,alpha,kappa] {[2 1e-3 0]}
NMC	{100}	Number of Monte Carlo samples for mceval

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.

EXAMPLES

The UKF should outperform the EKF when the second term in the Taylor expansion is not negligible. The standard example is one such system.

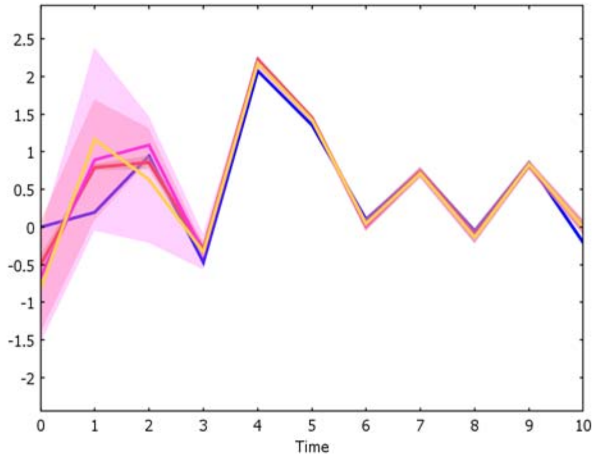
```
m=exnl('pfex');
z=simulate(m,50);
zekf1=nltf(m,z,'tup','taylor1','mup','taylor1');
zekf=ekf(m,z);
zukf=nltf(m,z);
xplot(z,zukf,zekf,zekf1,'conf',90,'view','cont')
```



The result of UKF is somewhat better than EKF. Both the standard and the NLT-based EKF version give identical results.

```
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(t+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)

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')
```



That is, all variants of this method give quite consistent estimates.

The Cramer Rao Lower Bound (CRLB)

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

$$
\begin{aligned}
P_{k+1|k} &= A_k P_{k|k} A_k^T + \bar{Q}_k, \\
P_{k+1|k+1} &= P_{k+1|k} - P_{k+1|k} C_k^T (C_{k+1|k} C_k^T + \bar{R}_k)^{-1} C_{k+1|k}, \\
A_k^T &= \frac{\partial}{\partial x_k} f^T(x_k), \\
C_k^T &= \frac{\partial}{\partial x_k} h^T(x_k).
\end{aligned}
$$

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:

- `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 the table below.

TABLE 4-4: OPTIONAL CRLB PARAMETERS

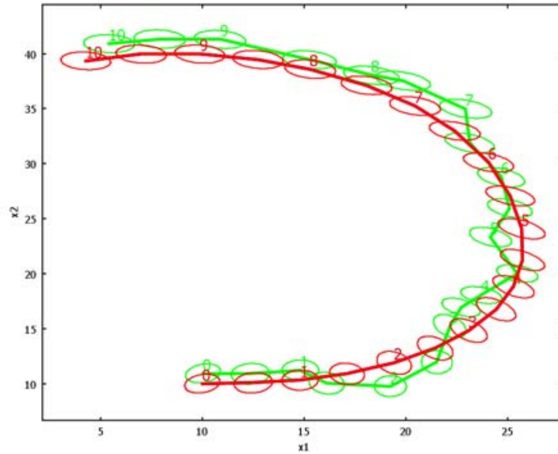
PROPERTY	VALUE	DESCRIPTION
k	k>0 {0}	Prediction horizon: 0 for filter (default), 1 for one-step ahead predictor.
P0	{[]}	Initial covariance matrix. Scalar value scales identity matrix. Empty matrix gives a large identity matrix.
x0	{[]}	Initial state matrix. Empty matrix gives a zero vector.
Q	{[]}	Process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

TARGET TRACKING EXAMPLE

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');
```

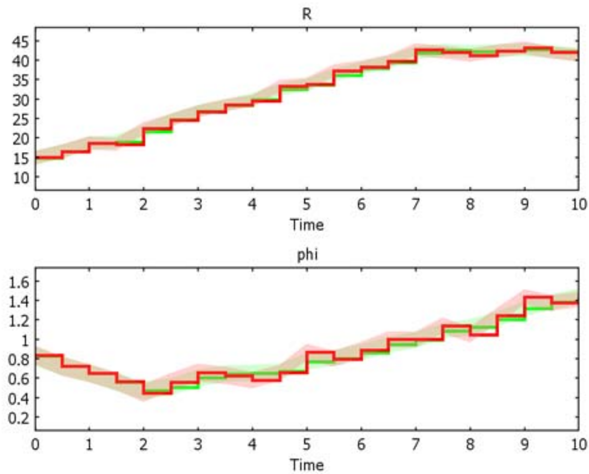
```
z=simulate(m,10);
zekf=ekf(m,z);
zcrlb=crlb(m,z);
xplot2(z,zekf,zcrlb,'conf',90)
```



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);
```

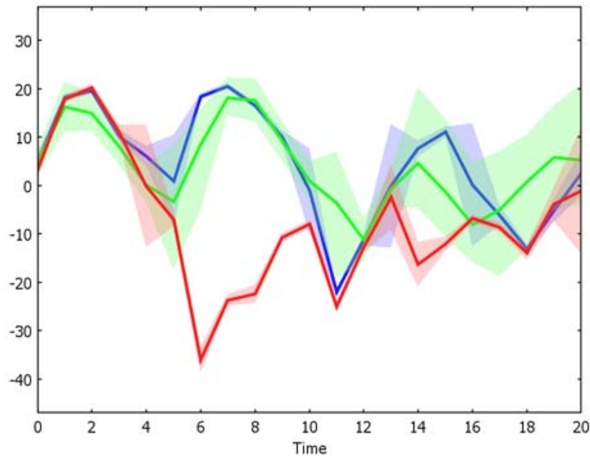


Also here, the confidence bounds are of the same thickness.

THE PF STANDARD 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(t+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)

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')
```



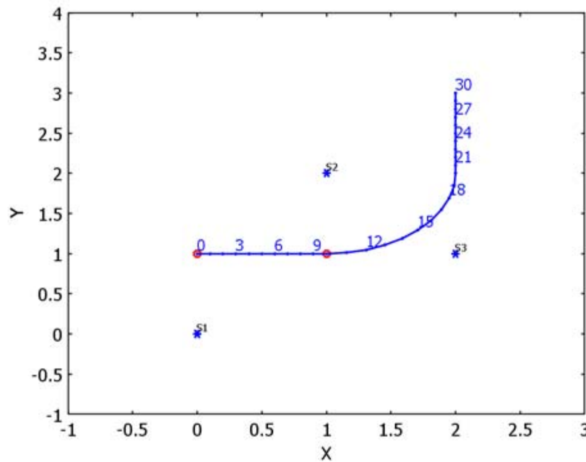
The CRLB confidence band is much smaller than for the PF algorithm. The EKF seemingly has a smaller confidence bound, but the bias is significant.

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 setup 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:

- *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.
- *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.
- *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.
- *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.
- *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.

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,:)-th(5)).^2+(x(2,:)-th(6)).^
2)]';
h=inline(hstr,'t','x','u','th');
th=[0 0 1 2 2 1 ]';  % True sensor positions
y=h(0,targetpos,[],th).';
```

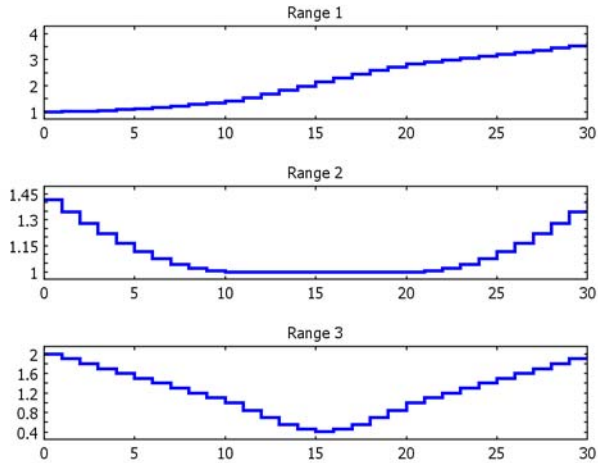
This will later be the measurement model in filtering.

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)
```

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+(th(8)-th(6)).^2)]'
h =
```

```
[sqrt((th(7)-th(1)).^2+(th(8)-th(2)).^2);sqrt((th(7)-th(3)).^2+(t
h(8)-th(4)).^2);sqrt((th(7)-th(5)).^2+(th(8)-th(6)).^2)]
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.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
     y =
[sqrt((th(7)-th(1)).^2+(th(8)-th(2)).^2);sqrt((th(7)-th(3)).^2+(t
h(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.

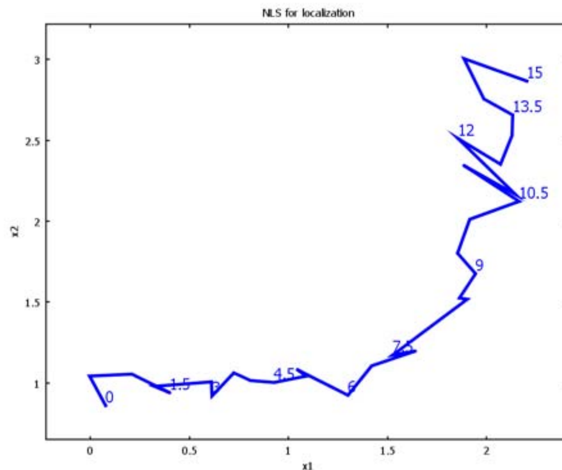
```
mhat=estimate(m,zn(1),'thmask',[0 0 0 0 0 0 1 1],'disp','on')
-----------------------------------------------------
Iter#          Cost    Grad. norm   BT#   Alg
-----------------------------------------------------
    0    1.441e+000             -     -   rgn
    1    3.219e-002    1.022e+000     1   rgn
    2    1.326e-003    2.226e-001     1   rgn
    3    1.162e-003    1.202e-002     1   rgn
    4    1.162e-003    4.482e-004     1   rgn
Relative difference in the cost function < opt.ctol.
NL object: Static sensor network model (calibrated from data)
     y =
[sqrt((th(7)-th(1)).^2+(th(8)-th(2)).^2);sqrt((th(7)-th(3)).^2+(t
h(8)-th(4)).^2);sqrt((th(7)-th(5)).^2+(th(8)-th(6)).^2)] +
N([0;0;0],[0.00034,0.00046,-0.00026;0.00046,0.000623,-0.000352;-0
.00026,-0.000352,0.000199])
   th' = [0         0         1         2         2         1
0.077       0.86]
    std = [1.6e-015   1.6e-015   1.6e-015   1.6e-015   1.6e-015
1.6e-015   1.4e-008   1.3e-008]

   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.

TARGET TRACKING USING EKF

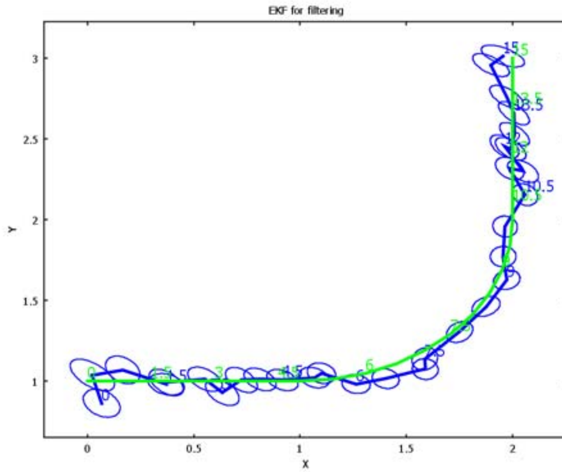
The difference between localization and tracking is basically only that a motion model is used to predict the next position. The estimated position can at each time be interpreted as an optimally weighted average between the prediction and the snapshot estimate. The standard coordinated turn polar velocity 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.

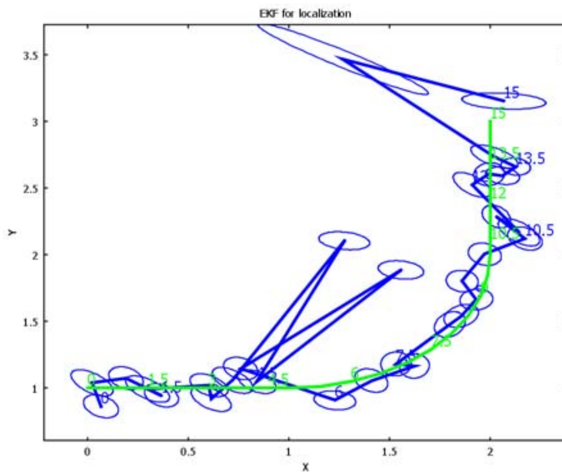
```
mm=exnl('ctpv2d');
h='[sqrt((x(1,:)-th(1)).^2+(x(2,:)-th(2)).^2);sqrt((x(1,:)-th(3))
.^2+(x(2,:)-th(4)).^2);sqrt((x(1,:)-th(5)).^2+(x(2,:)-th(6)).^2)]
';
m=nl(mm.f,h,[mm.nn(1:2) 3 6]);
m.x0=[0;1;1;0;0];     % Initial state
m.th=[0 0 1 2 2 1]';  % True sensor positions
m.fs=mm.fs;
m.name='Sensor network';
m.xlabel=mm.xlabel;
m.ylabel={'Range 1','Range 2','Range 3'};
m.thlabel={'pX(1)','pY(1)','pX(2)','pY(2)','pX(3)','pY(3)'}
NL object: Sensor network
x(t+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,:)]
     y =
[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,:)-th(5)).^2+(x(2,:)-th(6)).^2)]
   x0' = [0         1         1         0         0]
   th' = [0         0         1         2         2         1]

  States:  x1     x2     v     h     w
  Outputs: Range 1     Range 2     Range 3
  Param.: pX(1)     pY(1)     pX(2)     pY(2)     pX(3)     pY(3)
```

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

```
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);
xplot2(xhat,z,'conf',90)
```



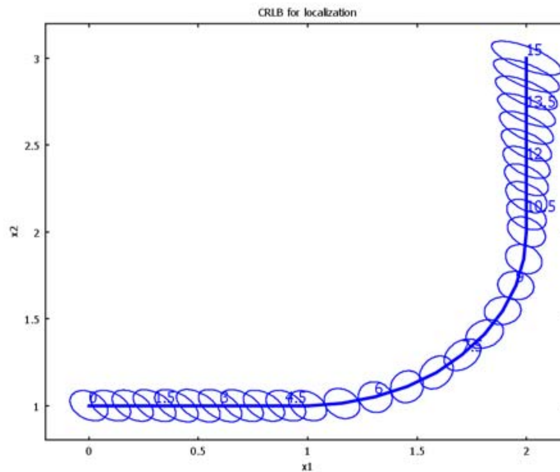
The filter based on a dynamic motion model thus improves the NLS estimate a lot. To get a more fair comparison, the process noise can be increased to a large number to weigh down the prediction. In this way, the filtered estimate gets all information from the current measurement, and the information 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 computed by using the true state trajectory, and the ellipsoids are placed around each true position.

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.

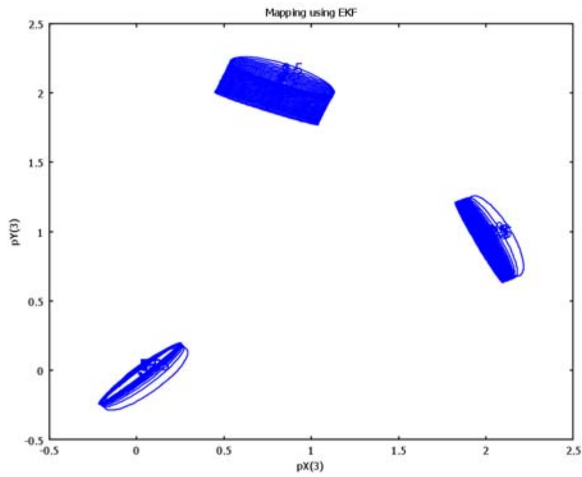
The target position is in the model below considered to be a known input signal. For that reason, a new data object with state position as input is defined. The sensor positions are put in the state vector.

```
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+(u(2)-x(6,:)).^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(t+1) = x(1:6,:) +
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])
     y =
[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+(u(2)-x(6,:)).^2)] +
N([0;0;0],[0.01,0,0;0,0.01,0;0,0,0.01])
   x0' = [0.19      -0.2      0.77       1.9       2.3       1.1]
+
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
```

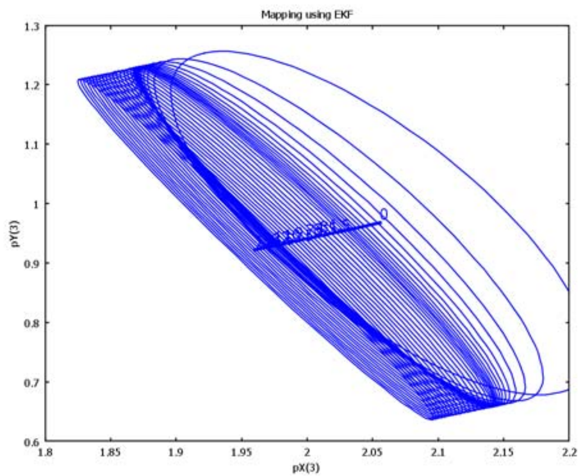
Now, the signal object is created and the EKF is called. Since each pair of states corresponds to one sensor location, the `xplot2` method is applied pairwise.

```
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.

SIMULTANEOUS MAPPING AND LOCALIZATION (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.

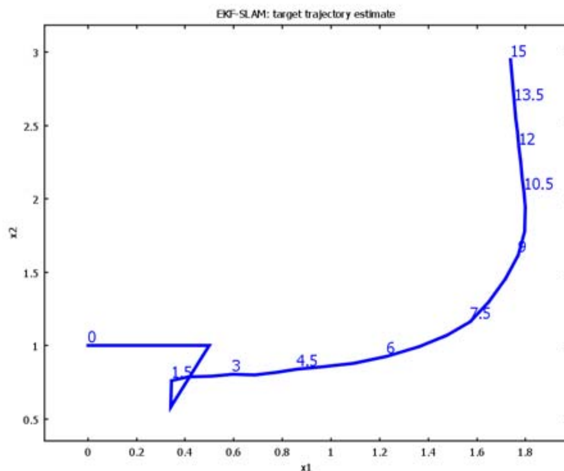
```
mm=exnl('ctpv2d');
f=['[',char(mm.f),'; x(6:11,:)]'];
h='[sqrt((x(1,:)-x(6,:)).^2+(x(2,:)-x(7,:)).^2);sqrt((x(1,:)-x(8,
:)).^2+(x(2,:)-x(9,:)).^2);sqrt((x(1,:)-x(10,:)).^2+(x(2,:)-x(11,
:)).^2)]';
m=nl(f,h,[11 0 3 0]);
m.x0=[0;1;1;0;0;th+0.1*randn(size(th))];     % Initial state
m.px0=diag([zeros(1,5), 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)','p
Y(3)'};
m.ylabel={'Range 1','Range 2','Range 3'};
m
NL object: SLAM model for the sensor network
x(t+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,:)] +
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;0,0,1e+003,0,0,0,0,0,0,0,0;0,0,0,0,0,0,0,0,0,0,0;0,0,0,0,1e
+003,0,0,0,0,0,0;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;0,0,0,0,0,0,0,0,0,0,0;0,0,0,0,0,0,0,0,0,0,0;0,0
,0,0,0,0,0,0,0,0,0])
     y =
[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,:)-x(10,:)).^2+(x(2,:)-x(11,:))
.^2)] + N([0;0;0],[0.1,0,0;0,0.1,0;0,0,0.1])
   x0' = [0        1        1        0        0     -0.21
-0.1       0.77        1.8        1.9       0.96] +
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;0,0,0,0,0,0,0,0,0,0,0;0,0,0,0,0,0,0.01,
0,0,0,0,0;0,0,0,0,0,0,0.01,0,0,0,0;0,0,0,0,0,0,0,0.01,0,0,0;0,0,0
,0,0,0,0,0,0.01,0,0;0,0,0,0,0,0,0,0,0.01,0;0,0,0,0,0,0,0,0,0,0,
0.01])
```

```
  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
```

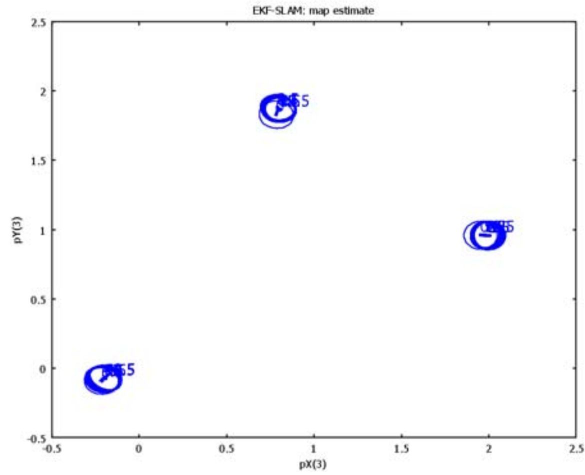
Applying the EKF algorithm now gives the so called EKF-SLAM, in a rather straightforward and inefficient implementation. EKF-SLAM scales badly in the number of landmarks, so structural knowledge in the model can be used to derive more efficient algorithms.

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



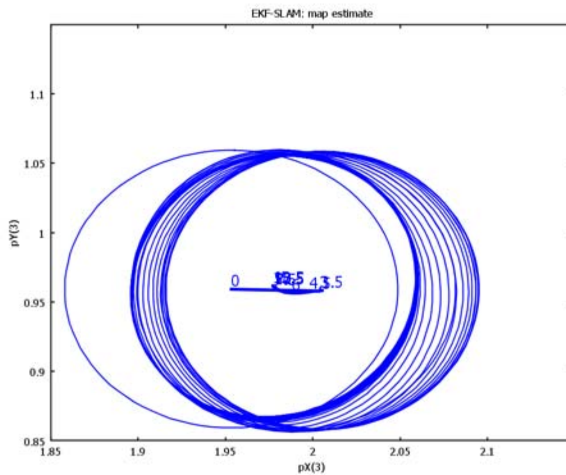
The trajectory looks worse than for the tracking case, where full knowledge of sensor location was assumed, but the estimate improves over time as the estimates of sensor positions are improved. Below, the improvement in the map is illustrated.

```
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])
```



5

Model Estimation

This chapter describes how to estimate models from data. The focus in the Signals & Systems Lab is on models that are linear in the parameters. Such models are described in “ARX Models” on page 196. This class of models include autoregressive models that are important for model-based signal processing applications and time-series analysis. Such applications have already been described in the context of spectral analysis and covariance function estimation in “Spectral Analysis” on page 69. This chapter focuses on how the underlying algorithms work and, most importantly, how to interpret the results.

ARX Models

The discrete-time transfer function is extended to stochastic systems via the ARMAX models, where the triplet (a, b, and c) of polynomials define the model

$$a(q^{-1})y(t) = b(q^{-1})u(t) + c(q^{-1})v(t) .$$

The ARX model includes the special cases when $c = 1$, which are important for estimation because the estimation problem is linear in the parameters and explicit least-squares solutions exist. MIMO models are supported. The special cases of AR ($b = 1$) and Finite Impulse Response (FIR) have dedicated objects with simplified constructors. Otherwise, they inherit all methods from the ARX object.

Table 5-1 contains a list the constructors and main methods.

TABLE 5-1: METHODS FOR THE ARX OBJECT

`arx(nn)`	Constructor of ARX structure
`arx(b,a,fs)`	Constructor of ARX model parameters, implicitly also defining the structure
`arx(nn,th,P)`	Constructor of uncertain ARX model defined by a parameter vector and associated covariance matrix P
`arx(nn,thMC)`	Constructor of uncertain ARX model defined by a parameter vector Monte Carlo samples
`ar(na), ar(a)`	Simplified constructor for AR models
`fir(nb),fir(b)`	Simplified constructor for FIR models
`display`	The overloaded display function gives an ASCII-formatted printout
`tex`	LaTeX code for displaying the model
`symbolic`	Return a symbolic string expression for the ARX structure
`size`	Return the sizes of the model structure nn
`info`	Display user specified information about the signal names
`estimate`	Estimate an ARX model of specified structure from a signal SIG object
`rand`	Return a random ARX model of specified structure
`arrayread`	Pick out subsystems from MIMO systems. Example: G2=G(:,2);

TABLE 5-2: FIELDS IN THE ARX OBJECT

`b`	Matrix of dimension (*ny, nb, nu*) with numerator polynomials of order *nb*, where each entry specifies the numerator polynomial from input *i* to output *j*.
`a`	Common denominator polynomial for all input-output channels
`th`	Parameter vector of free parameters in *b* and *a*
`P`	Covariance matrix of *th*
`nn`	Structure parameters [na nb nk nu ny]
`MC`	Number of Monte Carlo samples
`pe`	Noise variance (Gaussian assumption) or PDF object (default 1)
`fs`	User-specified sampling frequency (default 1)
`name`	User-specified name
`marker`	Time instants of interest
`tlabel`	Time label
`ylabel`	User-specified names on output signals
`ulabel`	User-specified names on input signals
`markerlabel`	Label for the marker
`method`	For estimated models, the method and design parameters are saved here
`desc`	User-specified description of the signal

Basic Algorithms

This section gives a brief introduction to the basic algorithms inside `sig2lti`. The *least-squares* (LS) framework starts with writing the model structure as a linear regression

$$y(t) = \varphi^T(t)\theta + e(t).$$

The solution to the following LS optimization problem

$$V_N(\theta) = \frac{1}{N}\sum_{t=1}^{N}(y(t)-\varphi^T(t)\theta)^2 = \frac{1}{N}\sum_{t=1}^{N}\varepsilon(t,\theta)^2,$$
$$\hat{\theta}_N = \text{argmin}(\,)_\theta V_N(\theta).$$

then becomes

$$\hat{\theta}_N = R_N^{-1} f_N = \left[\sum_{t=1}^{N} \varphi(t)\varphi^T(t) \right]^{-1} \sum_{t=1}^{N} \varphi(t) y(t)$$

The noise variance is estimated as

$$\hat{\sigma}_e^2 = V_N(\hat{\theta}_N) = \frac{1}{N} \sum_{t=1}^{N} [y(t) - \varphi^T(t)\hat{\theta}_N]^2 .$$

The stochastic properties of these estimates are summarized as

$$\mathrm{Cov}(\hat{\theta}_N) = E(\hat{\theta}_N - \theta_0)(\hat{\theta}_N - \theta_0)^T \approx \frac{1}{N}\lambda_0 \dot{R}_n^{-1} ,$$

$$\mathrm{Var}\left(\hat{\sigma}_e^2\right) = \mathrm{E}(\hat{\sigma}_e^2 - \sigma_e^2)^2 \approx \frac{1}{N}(\mathrm{E}e^4(t) - \sigma_e^2) ,$$

$$\sqrt{N}(\hat{\theta}_N - \theta_0) \in \mathrm{Asn}(0, \lambda_0 \dot{R}_n^{-1}) .$$

The covariance matrix of the parameter vector is one of the key quantities for validation.

It is possible to write AR and ARX models as linear regressions. For instance, an AR model corresponds to the following regressor and parameter vector:

$$y(t) + a_1 y(t-1) + \ldots + a_{ny}(t-n) = e(t) ,$$

$$\varphi(t) = (-y(t-1), -y(t-2), \ldots, -y(t-n))^T$$

$$\theta = (a_1, a_2, \ldots, a_n)^T.$$

The linear regression framework does not cover ARMA models. The procedure chosen here is to estimate a high-order AR model in the first step,

$$y[k] = \frac{C(q)}{A(q)} e[k] \approx \frac{1}{A_h(q)} e[k]$$

Any ARMA model can be arbitrarily well approximated with a high-order AR model. However, zeros in $C(q)$ close the unit circle can require very high model orders.

Assuming that the AR order is high enough, AR estimation gives an estimate and a covariance matrix. Further, the estimate is asymptotically Gaussian distributed.

The algorithm then generates an arbitrary signal $u[k]$, simulates the output from the estimated high-order AR model, and finally estimates the *ARMA model* in an ARX model framework. The rationale for this is that the simulation is noise free, and because the input to the simulation is known, the sought transfer function fits the ARX model structure. This is a nonlinear mapping of a high-dimensional parameter vector to a low-order parameter vector consisting of the parameters in $A(q)$ and $C(q)$,

$$\hat{\theta}_h \in N(\theta_h^o, P_h),$$
$$\hat{\theta} = f(\hat{\theta}_h^i).$$

To get a stochastic description, the nonlinear mapping is repeated in a Monte Carlo fashion, taking samples from the assumed Gaussian distribution of the high-order model.

$$\theta_h^i \in N(\hat{\theta}_h, P_h),$$
$$\theta \sim \frac{1}{N}\sum_{i=1}^{N} \delta[\theta - f(\theta_h^i)].$$

where $\delta(\theta)$ is the impulse function.

The principle for TF and ARMAX models is the same. Start with a high-order ARX model, simulate it for a known noise signal, and finally identify a sought structure as a low-order ARX model.

Simulation and Estimation of ARX Models

You specify an ARX model by its structure and its parameters. The structural parameters are essential for estimation and also to generate random models. To specify an ARX model, use a vector as input to the ARX constructor:

```
mstruc=arx([2 2 1])
Unspecified FIR(2,1)
```

You can generate a random example model from this structure:

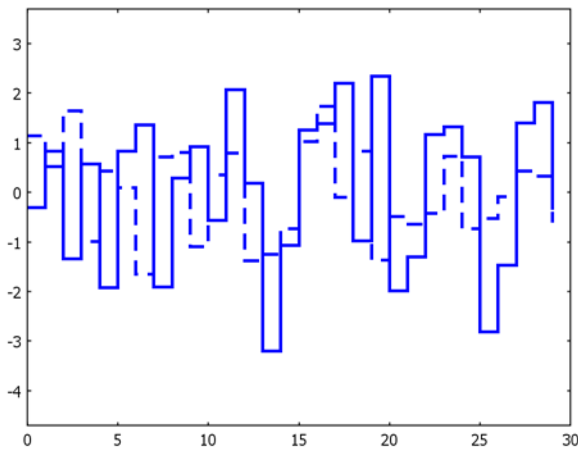
```
m0=rand(mstruc)
Discrete time ARX(2,2,1) model (fs=1):
(q^2-0.0181*q+0.269) y(t) = (q-0.361) u(t) + e(t)
```

The polynomials are taken from random roots inside the unit circle, with one fraction being complex-conjugated root pairs and the other one real-valued roots. Simulating the model results in a SIG object:

```
z=simulate(m0,100)
SIG object with discrete time (fs = 1) input-output data
  Sizes:       N = 100,  ny = 1, nu = 1
```

The overloaded `plot` function for SIG objects illustrates the result:

```
plot(z(1:30))
```



Use the `estimate` function to estimate a model from the same structure from the simulated data:

```
mhat=estimate(mstruc,z)
Discrete time ARX(2,2,1) model (fs=1):
(q^2-0.0882*q+0.295) y(t) = (1.21*q-0.424) u(t) + e(t)

Parameter vector and uncertainties [std=sqrt(P(i,i)]
    -0.0882  0.295  1.21    -0.424
    0.0877   0.058  0.0983  0.148
```

Note that you specify the structure as the first input argument, so that the Signals & Systems Lab finds the correct `estimate` method belonging to the ARX class. The display function shows the transfer function on polynomial form as usual. For

estimated models, however, it also displays the parameter vector with standard deviations.

MIMO ARX Models

You obtain a MIMO structure by including the input and output dimensions to the structure parameter vector. The `display` function confirms the chosen MIMO structure:

```
mstruc=arx([2 2 1 2 2])
Unspecified MIMO(2,2)  FIR(2,1)
```

The following example repeats the steps from the previous section for the MIMO structure. First generate a random model:

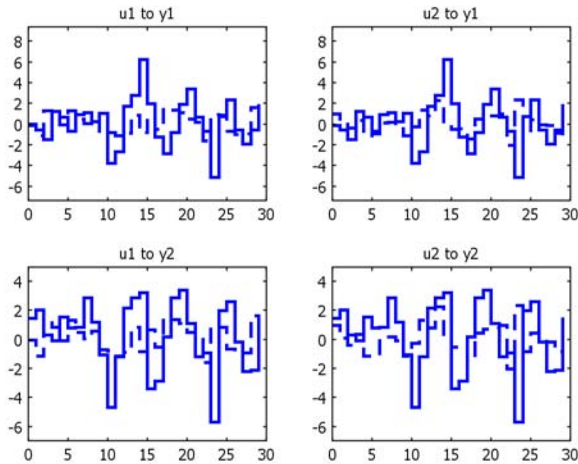
```
m0=rand(mstruc)
Discrete time MIMO(2,2)  ARX(2,2,1) model (fs=1):
(q^2-0.613*q+0.485) y1(t) = (q-0.576) u1(t) + e1(t)
(q^2-0.613*q+0.485) y1(t) = (q-0.0723) u2(t) + e1(t)
(q^2-0.613*q+0.485) y2(t) = (q-0.35) u1(t) + e2(t)
(q^2-0.613*q+0.485) y2(t) = (q-0.755) u2(t) + e2(t)
```

Remember the convention for MIMO ARX models that the $a(q)$ polynomial is the same for all input-output channels. Next simulate for 100 data points (using the white noise default input signal):

```
z=simulate(m0,100)
SIG object with discrete time (fs = 1) input-output data
  Sizes:       N = 100,  ny = 2, nu = 2
```

The next command plots the first 30 samples:

```
plot(z(1:30))
```



Estimation from these simulated data gives the result:

```
mhat=estimate(mstruc,z)
Discrete time MIMO(2,2)  ARX(2,2,1) model (fs=1):
(q^2-0.614*q+0.48) y1(t) = (1.01*q-0.587) u1(t) + e1(t)
(q^2-0.614*q+0.48) y1(t) = (0.958*q+0.00382) u2(t) + e1(t)
(q^2-0.614*q+0.48) y2(t) = (0.95*q-0.336) u1(t) + e2(t)
(q^2-0.614*q+0.48) y2(t) = (0.928*q-0.856) u2(t) + e2(t)

Parameter vector and uncertainties [std=sqrt(P(i,i)]

Columns 1-60:
    -0.614  0.48    1.01    -0.587  0.958  0.00382  0.95    -0
    0.0517  0.0359  0.0993  0.115   0.106  0.118    0.0984  0.

Columns 61-79:
  .336  0.928  -0.856
  109   0.104  0.114
```

The sizes of the estimated model are as specified:

```
size(mhat);
na = 2
nb = 2
nk = 1
nu = 2
ny = 2
```

You obtain a symbolic string summary by:

```
symbolic(mhat)
ans =
  MIMO(2,2)  ARX(2,2,1)
```

Array operations allow picking out subsystems from MIMO systems:

```
mhat(2,2)
Discrete time ARX(2,2,1) model (fs=1):
(q^2-0.614*q+0.48) y(t) = (0.928*q-0.856) u(t) + e(t)

Parameter vector and uncertainties [std=sqrt(P(i,i)]
    -0.614  0.48    0.928   -0.856
    0.0517  0.0359  0.0984  0.109
```

Conversions of ARX Models

Table 5-3 summarizes low-level conversions from ARX objects to other objects. These functions are called by the respective constructor.

TABLE 5-3: LOW-LEVEL CONVERSIONS FROM ARX OBJECTS

`arx2tf`	Extract the transfer function B/A of an ARX model
`arx2spec`	Compute the spectrum of the AR part of and ARX model
`arx2cov`	Compute the covariance of the AR part of and ARX model
`arx2freq`	Compute the frequency function B/A of an ARX model
`arx2ss`	Exact conversion to state-space object

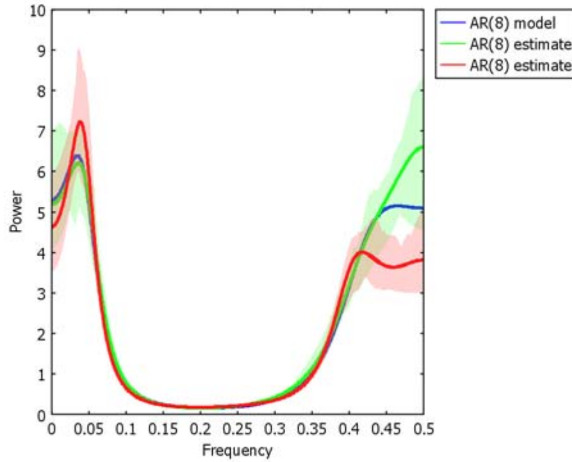
Operations on ARX Models

There are only a few overloaded functions for ARX objects as opposed to SS and TF objects. The reason is that connections and other operations seldom lead to a new ARX model. The solution in such cases is to transform the ARX model to a stochastic state-space model, as described in the previous section.

As an illustration, the following code converts a random AR(8) model to state-space form, simulates data from both models, and estimates an AR(8) model to each one. The spectra indicate that both models simulate the same stochastic process (but with different realizations).

```
m0=rand(ar(8));
m0ss=ss(m0);
y=simulate(m0,1000);
yss=simulate(m0ss,1000);
```

```
mhat=estimate(ar(8),y);
mhatss=estimate(ar(8),yss);
plot(spec(m0),mhat,mhatss);
```



The estimated MIMO model derived earlier becomes a state-space model on innovation form (there is only one noise source, so process noise $v(t)$ and measurement noise $e(t)$ are the same, and S shows perfect correlation):

```
ss(mhat)
          / 0.61  -0.48     0      0 \        / 1  0 \
          |    1      0     0      0 |        | 0  0 |
x[k+1] =  |    0      0  0.61  -0.48 | x[k] + | 0  1 | u[k]
          \    0      0     1      0 /        \ 0  0 /


+ v[k]


        /    1  -0.59  0.96  0.0038\        /0  0\
y[k] =  \0.95  -0.34  0.93   -0.86/  x[k] + \0  0/  u[k] +


e[k]


             /0.93         0          0          0\
            |    0         0          0          0 |
```

```
Q = Cov(v) = |    0          0       0.93          0 |
              \   0          0          0          0/

                /0.93           0\
               |    0           0 |
S = Cov(v,e) = |    0        0.93 |
                \   0           0/
             / 0.93           0\
R = Cov(e) = \    0        0.93/
```

TF Models

The ARX model is often quite efficient in predicting the output of a system. In the case of fast sampling, the output changes very slowly between the samples, and the ARX model includes the trivial model that just repeats the last observed output. The identified ARX model is thus always better than guessing on the previous value. This is not true for the TF model, which just gets access to the system input.

The TF object is used in so called output error model identification, where the dynamic relation between the input and output is in focus rather than modelling the process and measurement noise. This is something that is recommended if the model is going to be used for simulation or control purposes.

Least-Squares Approach

The algorithm to estimate a TF model is based on a multistep least-squares (LS) approach, that includes the following steps:

1. Estimate a high-order FIR model using LS with a user-provided set of input-output data as a SIG object. This step calls `arx.estimate`.
2. In the second step, simulate the high-order FIR model without noise using the same input as available in the data.
3. Then estimate the low-order ARX model using LS, using `arx.estimate` again, this time with the sought number of poles and zeros.
4. Finally, convert the ARX model into the corresponding TF model with the same `b` and `a` polynomials.
5. For uncertainty representation, use one of the following schemes:
   - **a** If Monte Carlo realizations of the output signal are provided in the SIG object, repeat the preceding four steps a for each of them, and create the cell array `sysMC`.
   - **b** Otherwise, the algorithm assumes that
   - The true system is contained in the high-order FIR model
   - The estimate can be considered Gaussian distributed

   The latter is true for Gaussian noise and asymptotically otherwise. Monte Carlo simulations convert the Gaussian high-order estimate to a sample-based representation of the non-Gaussian distribution of the low-order ARX estimate. More precisely, the uncertainty is then represented by taking random parameter

vectors from the high-order FIR model and repeating steps 2 to 4 above to obtain the cell array `sysMC`.

DC Motor Example

As an illustration, the following example uses a standard LTI object for a DC motor in `exlti`. Then create a model structure with the same structure as the simulated system:

```
N=500;
m0=exlti('tf2d')

           0.68*z-0.34
   Y(z) = ------------- U(z)
          z^2-1.4*z+0.74

ms=tf(m0.nn);
ms.fs=m0.fs
Unspecified TF model with na=2, nb= 2, nk= 1 and with 1 inputs and
1 outputs.
```

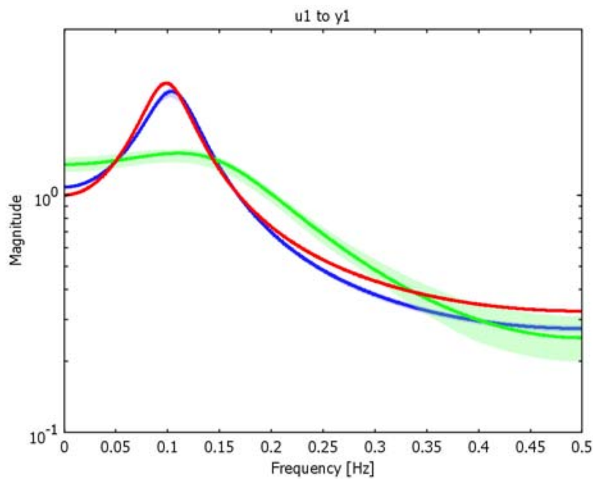
Because an ARX model has the same structural indices, it is easy to estimate an ARX model for comparison. And because the following data are noise-free, both model classes are equivalent and return the simulated transfer function:

```
u=getsignal('prbs',N,3,0.3);
y=simulate(m0,u);
mhat1=estimate(arx(m0.nn),y);
mhat1tf=tf(mhat1)

           0.68*z-0.34
   Y(z) = ------------- U(z)
          z^2-1.4*z+0.74

mhat=estimate(ms,y)

           0.68*z-0.33
   Y(z) = ------------- U(z)
          z^2-1.4*z+0.74
```

To get a more realistic example, add noise. You can do this in two ways: Either add one realization of the noise to the signal or ask for an ensemble of MC different realization by adding an object from a PDFCLASS object. The following code illustrates the former option:

```
yn=y+0.5*randn(N,1);  % One realization -> estimation uncertainty
in mhat
```

```
yn=y+0.5*ndist(0,1);  % MC realizations -> ensemble uncertainty in
mhat
mhat1=estimate(arx(m0.nn),yn);
mhat1.MC=30;        % Set MC for tf conversion
mhat1tf=tf(mhat1)

             0.66*z+0.15
   Y(z) = --------------- U(z)
           z^2-0.73*z+0.33

mhat=estimate(ms,yn)

            0.62*z-0.21
   Y(z) = -------------- U(z)
           z^2-1.3*z+0.72

bodeamp(mhat,mhat1tf,m0,'Ylim',[0.1 5])
```



Note that the described algorithm implemented in `tf.estimate` is more robust to additive noise than the one in `arx.estimate`. The model class represented by a transfer function with additive noise is often referred to as an *output-error model*.

NL Models

Estimation of NL object is in the literature referred to as gray-box system identification. It allows the user to specify a physical model structure with uncertain parameters, which are then calibrated from observed data. As special cases, the estimate method of the NL object can be used to solve the nonlinear least squares problem, and to fit parametric static models (as solutions to differential equations) to data, which is related to curve fitting.

Nonlinear Least-Squares Parameter Estimation

PROBLEM DEFINITION AND NOTATION

The nonlinear least-squares (NLS) algorithm implements various versions of the Gauss-Newton method for parameter estimation. A general problem formulation is to estimate the initial state and parameter vector in the model

$$y = H(x_0, \theta, u, e, v)$$

based on observations of y and u. Special cases include pure optimization

$$0 = H(\theta, e)$$

and data fitting

$$y = H(\theta, e)$$

The least-squares framework is appropriate whenever H is a vector. These problems can all be recast to the general nonlinear model object NL in either discrete

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

or continuous time

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

There are basically three uses of `nls`:

1 `[m,res]=nls(h)` for pure optimization, where `h` is an inline function with `th` as a parameter.

2 `[m,res]=nls(h,y)` for data fitting, where `h` is either an inline function or a structure where one field is called `h` and contains an inline function with `th` as a parameter. The advantage with the latter usage is that more information about the problem can be provided in other fields. In all cases, `y` is a SIG object.

3 `[m,res]=nls(m,y)` for model calibration. Here `m` is an NL object, and all of its specified parameters and initial states are estimated by default using the data in the SIG object `y`. This is also known as gray-box identification, as opposed to black-box identification where standard model structures as ARX have to be used. In gray-box identification, the model might be partially known, and the physical parameters are identified directly. The term model calibration is used here to stress that quite a good initial value of the parameters is generally needed for the algorithms to converge. More details on these issue follow in "Algorithm" on page 211.

In general, the NLS parameter estimation problem can always be specified as an NL object `m=nl(f,h,nn)`, and the parameters are estimated with the NL method `mhat=estimate(m,z)`, or equivalently, `mhat=nls(m,z)`.

The general solution can be stated as computing the minimizing argument of the sum of least squares

$$\eta = (x_0^T, \theta^T)^T$$

$$\hat{\eta} = \text{argmin}(\,)_\eta V(\eta, y),$$

$$V(\eta) = \frac{1}{2}\sum_{t=1}^{N} \varepsilon^T(t_k, \eta)\varepsilon(t_k, \eta)$$

The parameter vector η does not have to include all parameters and initial states, but rather a subset of them. The residual ε is defined component-wise as

$$\varepsilon(t_k, \eta) = y(t_k) - \hat{y}(t_k, \eta) \;\;,$$

or more compactly as the vector

$$\varepsilon(\eta) = (\varepsilon^T(t_1, \eta), \quad \varepsilon^T(t_N, \eta))^T .$$

With this vector notation, the loss function can be compactly expressed as

$$V(\eta) = \frac{1}{2}\varepsilon^T(\eta)\varepsilon(\eta).$$

The gradient of the residual

$$J(\eta) = \frac{\partial}{\partial\eta}\varepsilon^T(\eta) = -\frac{\partial}{\partial\eta}\hat{y}^T(\eta)$$

is central in the NLS method.

ALGORITHM

With these definitions, the Newton-Raphson method for NLS can be stated as iterating

$$\eta^{i+1} = \eta^i - \mu\left(\frac{\partial^2 V(\eta^i)}{\partial\eta^2}\right)^{-1}\frac{\partial}{\partial\eta}V(\eta^i)$$

until convergence. The Jacobian and Hessian above are given by

$$\frac{\partial}{\partial\eta}V(\eta) = \sum_{k=1}^{N}\frac{\partial}{\partial\eta}\varepsilon^T(t_k, \eta)\varepsilon(t_k, \eta) = J(\eta)\varepsilon(\eta)$$

and

$$\frac{\partial^2 V(\eta)}{\partial\eta\partial\eta^T} = J(\eta)J^T(\eta) - \frac{\partial}{\partial\eta}J(\eta)\varepsilon(\eta),$$

respectively. The Gauss-Newton method approximates the Newton-Raphson method with the logical assumption that the second term of the Hessian is an order of magnitude smaller than the first term,

$$H(\eta) \approx J(\eta)J^T(\eta).$$

This leads to the following NLS algorithm as implemented in `nls`:

1. Initialize the parameter vector $\eta = (\theta, x_0)$ using the values of `x0` and `th` in the NL object.
2. Iterate in i:

$$\eta^{i+1} = \eta^i - \mu^j(J(\eta^i)J^T(\eta^i))^{-1}J(\eta^i)\varepsilon(\eta^i)$$

3. In each iteration, check if the cost function has decreased. Otherwise, half the step size μ until either the cost function decreases or `maxhalf` iterations in the line search is reached.
4. Continue until j = `maxiter` or the relative change in cost function is smaller than `ctol`, or the maximum element in the gradient J is smaller than `gtol`.

There is some support for entering a symbolic gradient J to the NL object for pure estimation problems, where $J = dh/d\theta$. Otherwise, a numeric gradient is computed according to

$$\frac{\partial}{\partial \eta_i}\hat{y}(t_k, \eta) \approx \frac{\hat{y}(t_k, \eta + \mathrm{he}_i) - \hat{y}(t_k, \eta - \mathrm{he}_i)}{2h},$$

which is evaluated for each unit vector.

The internal parameters in the algorithm are controlled by property-value pairs for both usages `mhat=estimate(m,z,Property1,Value1,...)` and `mhat=nls(m,z,Property1,Value1,...)`. The property-value pairs are listed in the table below.

TABLE 5-4: PROPERTY-VALUE PAIRS FOR THE NLS ALGORITHM

PROPERTY	VALUE{DEFAULT}	DESCRIPTION
`thmask`	{ones(1,nth)}	Binary search mask for parameter vector
`x0mask`	{ones(1,nx)}	Binary search mask for initial state vector
`x0`	Cell array	In case z is a cell with multiple data sets, different known initial conditions can be set. x0{i} is the initial state for z{i}
`alg`		Optimization algorithm
	{'gn'}	Gauss-Newton
	'rgn'	Robust Gauss-Newton, where the Hessian *H=JJ'* is robustified by adding a small identity matrix.
	'lm'	Levenberg-Marquardt, where the line search is replaced by a region search.
	'sd'	steepest-descent, where the Hessian is replaced with the identity matrix.
`disp`	{0}|1	Display status of the iterations
`maxiter`	{50}	Maximum number of iterations in search direction
`maxhalf`	{50}	Maximum number of iterations in the line search.
`gtol`	{1e-4}	Tolerance for the gradient.

TABLE 5-4: PROPERTY-VALUE PAIRS FOR THE NLS ALGORITHM

PROPERTY	VALUE{DEFAULT}	DESCRIPTION
`ctol`	{1e-4}	Minimum relative decrease in the cost function before the search is terminated.
`svtol`	{1e-4}	Lower bound for the singular values of the Jacobian in robust Gauss-Newton
`numgrad`	(0)|1	Force a numerical computation of the gradient even if a gradient m.J is specified

The direct call to NLS enables a second output structure `[mhat,res]=nls(m,z)`, which gives access to internal variables, as summarized in the following table.

TABLE 5-5: INTERNAL NLS FIELDS

FIELD NAME	DESCRIPTION
res.TH	Parameter values at each iteration
res.V	Value of the cost function at each iteration
res.dV	The gradient at each iteration
res.m	The obtained model at each iteration as a cell array
res.sl	The step sizes at each iteration
res.sol	The obtained solution (thhat)
res.term	Text string with cause of termination
res.P	Covariance (estimated) for the parameters
res.Rhat	Covariance (estimated) for the measurements

The examples in the following section illustrate the following aspects:

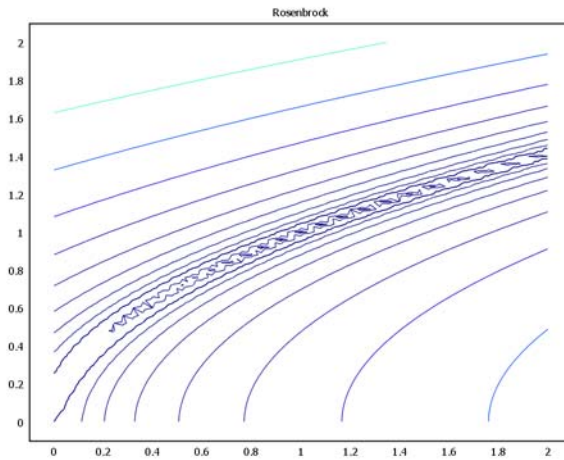
- Rosenbrock's test example: direct inline call to NLS, call with a structure and providing initial state and symbolic gradient.
- Step response of first order dynamics: Curve fitting using an NL object to get the physical parameters, time constant and static gain, of step response. Propagating model uncertainty to simulations. Fusion information from different data sets.
- Bouncing ball: parameter estimation and joint parameter and initial state estimation.
- AUV: as an application example, the steering dynamics of an autonomous underwater vehicle (AUV) is calibrated.

ROSENBROCK'S TEST EXAMPLE

The following example is a benchmark example in optimization.

```
h=inline('[10*(th(2)-th(1)^2);1-th(1)]','th')
h =
Inline function:  [10*(th(2)-th(1)^2);1-th(1)]
thopt=[1;1];
```

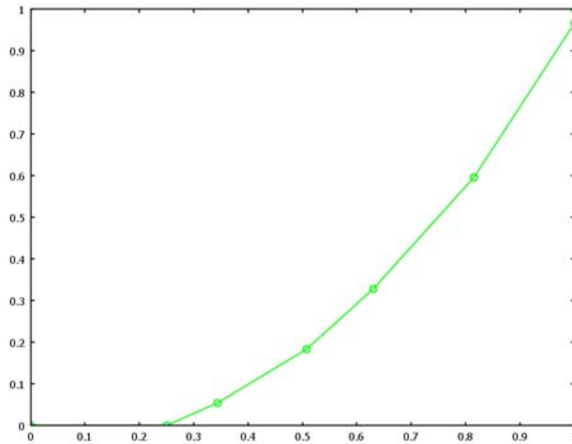
A contour plot reveals the difficulty to find the global optimum on the narrow and slightly skewed ridge of the function.



The most direct usage of `nls` just uses the inline function as input argument.

```
[m,res]=nls(h);
res.sol
ans =
  1
  1
res.TH
ans =

Columns 1-7:
        0    0.2500    0.3438    0.5078    0.6309    0.8154
1
         0         0    0.0547    0.1833    0.3270    0.5953
0.9659
```

```
Columns 8-9:
          1          1
          1          1
plot(res.TH(1,:),res.TH(2,:),'g-o')
```

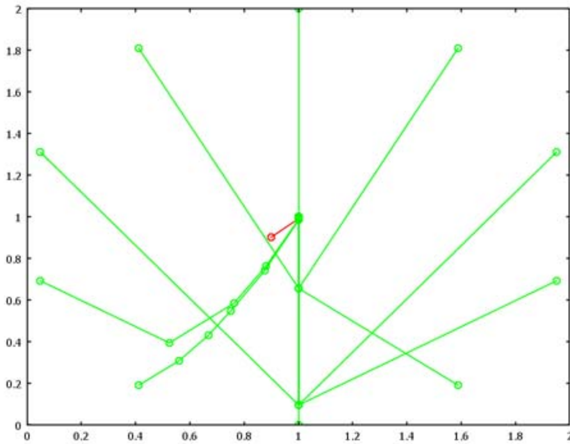


The parameter vector converges from the origin to the global optimum at (1, 1) in nine iterations.

An alterative is to define a structure which also allows to specify the initial value of the parameter vector.

```
m.h=h;
m.th=[0.9;0.9];
[mhat,res]=nls(m);
hold on
plot(res.TH(1,:),res.TH(2,:),'r-o')
```

Repeating the call to `nls` from different initial positions illustrate the fast convergence to the ridge and the slower convergence along the ridge.



To speed up computations, you can specify the gradient in the structure. This avoids the numerical differentiation used by default.

```
tic,m=nls(m);toc

Elapsed time:  0.016 s

m.J=inline('[-20*th(1) -1; 10 0]','th');
tic,m=nls(m);toc

Elapsed time:  0.015 s
```

Step Response

CURVE FITTING

The second class of problems handled by `nls` fits a parametric model to samples of a signal or, as here, a curve. The curve can in this case be interpreted as the step response of the first order system with unknown gain and time constant. The problem is here defined using the NL object.

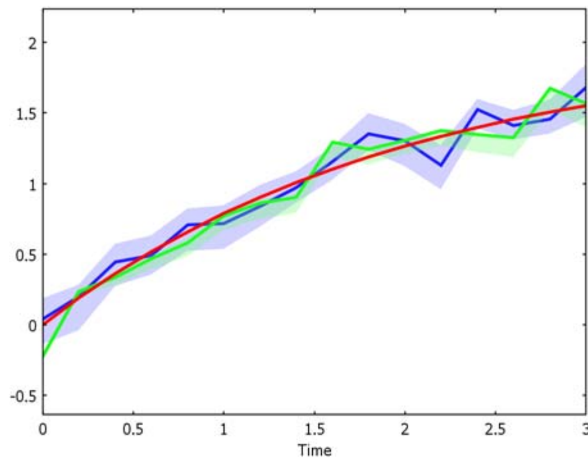
```
th0=[2;0.5];
t=(0:0.2:3)';
```

```
h='th(1)*(1-exp(-th(2)*t))';
m=nl('[]',h,[0 0 1 2]);
m.th=[1;1];
m0=m;
m0.th=th0;
y=simulate(m0,t)+ndist(0,0.01);
mhat=nls(m,y),  %Equivalent to mhat=estimate(m,y)
NL object
 dx/dt = []
     y = th(1)*(1-exp(-th(2)*t)) + N(0,0.0082)
   x0' = []
   th' = [2.1        0.47]
   std = [0.24       0.091]
```

The display function prints out the standard deviation of the estimated parameters, while the full covariance matrix is used internally.

The true model and the estimated model can be compared to the signal used for estimation by plotting the simulated curves. Note that the estimation signal is a stochastic signal providing a confidence bound, and that also the estimated model gives a uncertain curve. However, these confidence bounds are conceptually completely different, since the first one originates from the ensemble behavior of the noise realization, and the second one stems from estimation uncertainty.

```
y0=simulate(m0,t);
yhat=simulate(mhat,t);
plot(y,yhat,y0,'conf',90)
```



The confidence bounds essentially overlap each other and the noise free signal.

FUSION FROM DIFFERENT DATA SETS

Suppose that several data sets are available in the previous application. This can be interpreted as if several step responses have been collected.

```
y2=simulate(m0,t)+ndist(0,0.01);
mhat1=estimate(m,y)
NL object:  (calibrated from data)
 dx/dt = []
     y = th(1)*(1-exp(-th(2)*t)) + N(0,0.0082)
   x0' = []
   th' = [2.1       0.47]
   std = [0.24      0.091]

mhat2=estimate(m,y2)
NL object:  (calibrated from data)
 dx/dt = []
     y = th(1)*(1-exp(-th(2)*t)) + N(0,0.01)
   x0' = []
   th' = [1.9       0.52]
   std = [0.22       0.11]
```

`nls` gives essentially the same estimation accuracy for each data set. However, an improved estimate can be achieved by fusing the information from the two data sets. One general option is to use the first estimated model as initial model in the second call using the second data set. The estimate method then takes care of combining the initial covariance with the new estimation covariance. This is done using the sensor fusion formula, which is optimal for Gaussian independent estimates.

```
mhatfusion=estimate(mhat1,y2)
NL object:  (calibrated from data) (calibrated from data)
 dx/dt = []
     y = th(1)*(1-exp(-th(2)*t)) + N(0,0.01)
   x0' = []
   th' = [1.9       0.53]
   std = [0.14      0.059]
```

The standard deviation has decreased a factor of two. An alternative way to do fusion, which is more robust to the Gaussian implicit assumption, is to enter both signal sets as a cell array to `nls`.

```
mhat=estimate(m,{y,y2})
NL object:  (calibrated from data)
 dx/dt = []
     y = th(1)*(1-exp(-th(2)*t)) + N(0,0.00948)
   x0' = []
```

```
th' = [2        0.5]
std = [0.17      0.071]
```

The data sets might be of different size and even have different sampling times. Another advantage of this second alternative is that it usually achieves a faster convergence in the NLS algorithm, because more accurate Jacobians and Hessians are computed.

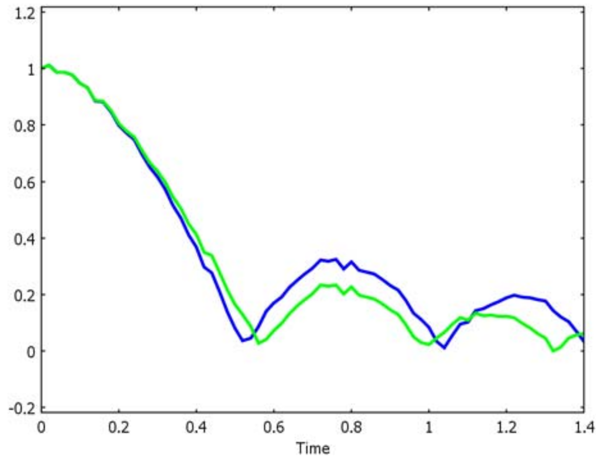
Bouncing Ball

Consider again the bouncing ball defined as the NL object below. We will in this section assume that a bouncing ball has been observed during a short time interval, and the height over ground is measured with a small measurement noise. There is no process noise disturbing the trajectory here.

```
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.pv=[];  % Q=cov(v)=0
m.pe=1e-4;   % R=cov(e)=1
m
NL object: Bouncing ball
 dx/dt = [x(2,:);-th(1)*x(2,:).^2.*sign(x(2,:))-9.8.*sign(x(1,:))]
     y = abs(x(1,:)) + N(0,0.0001)
   x0' = [1          0]
   th' = [1]

  States:  Modified height     Modified speed
  Outputs: Height
  Param.: Air drag
```

Assume that the air drag of the ball is unknown, and a nominal value of 1.2 is used. It is now possible to compare the actual and nominal bounce trajectories:

```
minit=m;
minit.th=1.5;
yinit=simulate(minit,0:0.02:1.4);
y=simulate(m,0:0.02:1.4);
plot(y,yinit)
```

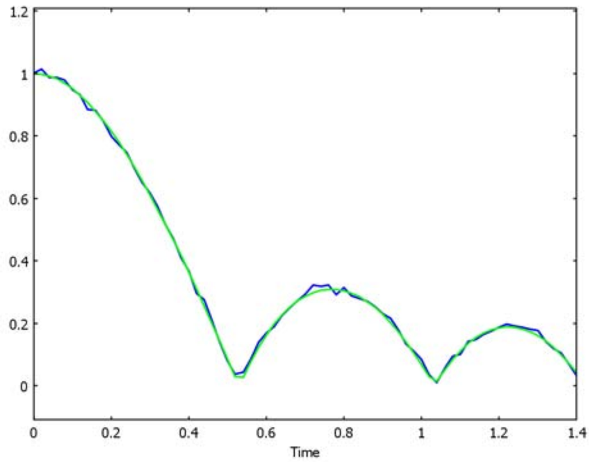


The NLS algorithm invoked by the `estimate` method of the NL object can be used to change the nominal value to obtain the best possible fit of simulated and observed trajectory.

```
mhat=estimate(minit,y,'x0mask',[0;0])
NL object: Bouncing ball (calibrated from data)
 dx/dt = [x(2,:);-th(1)*x(2,:).^2.*sign(x(2,:))-9.8.*sign(x(1,:))]
      y = abs(x(1,:)) + N(0,8.7e-005)
    x0' = [1          0]
    th' = [1]
    std = [8.4e-005]

   States:  Modified height     Modified speed
   Outputs: Height
   Param.: Air drag
```

The air drag is thus accurately estimated. Note also that `nls` also has estimated the measurement noise variance.

```
mhat.pe=[];
yhat=simulate(mhat,0:0.02:1.4);
plot(y,yhat,'conf',99)
```




```
[mhat,res]=estimate(minit,y,'disp','on','x0mask',[0;0]);
------------------------------------------------------
Iter#          Cost     Grad. norm   BT#   Alg
------------------------------------------------------
    0    3.350e-001             -     -    rgn
    1    2.288e-001    4.841e-001     1    rgn
    2    1.123e-001    5.651e-001     1    rgn
    3    8.542e-002    2.943e-001     1    rgn
    4    4.433e-002    2.055e-001     1    rgn
    5    2.272e-002    2.994e-001     1    rgn
    6    8.335e-003    2.729e-001     1    rgn
    7    7.979e-003    5.758e-002     1    rgn
    8    6.488e-003    8.258e-002     1    rgn
    9    6.176e-003    3.261e-002     2    rgn
   10    6.174e-003    1.369e-001     1    rgn
Relative difference in the cost function < opt.ctol.
res.TH
ans =

Columns 1-7:
    1.5000    1.3640    1.2250    1.1811    1.1200    1.0767
1.0269

Columns 8-11:
     1.0205     1.0129     1.0111     1.0111
```

The obtained fit is perfect, and the confidence bound is very narrow and not visible.

The initial state when the ball is observed first might be unknown as well. Assuming an incorrect initial state, the NLS algorithm can be asked to jointly estimate the initial state and air drag (which is done by default).

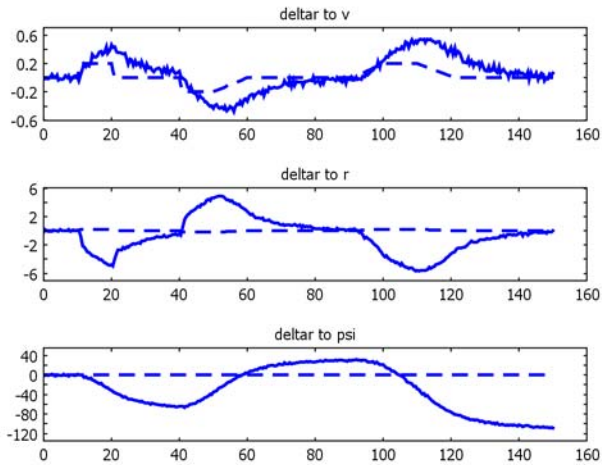
```
minit.x0=[0.5; 0.1];
mhat=estimate(minit,y,'x0mask',[1;1])
NL object: Bouncing ball (calibrated from data)
 dx/dt = [x(2,:);-th(1)*x(2,:).^2.*sign(x(2,:))-9.8.*sign(x(1,:))]
      y = abs(x(1,:)) + N(0,0.000143)
   x0' = [1   -0.0071] + N(0,[7.4e-009,2.9e-007;2.9e-007,7.4e-005])
    th' = [1]
    std = [0.00017]

   States:  Modified height     Modified speed
   Outputs: Height
   Param.: Air drag
```

The accuracy is here almost the same as when the initial state was known. The display function now also shows the uncertainty in the estimated initial state.

AUV Dynamics

This example illustrates a realistic application concerning an autonomous underwater vehicle (AUV). a linearized steering model is known from the literature, and a simulated trajectory is available as a test signal.

```
load auv
y
SIG object with continuous time input-output state space data
  Name:        AUV trajectory
  Sizes:       N = 301,  ny = 3, nu = 1, nx = 3
  MC is set to: 30
  #MC samples:  30
Movie file:  auv.wmv
plot(y)
```



This signal has an associated animation, which you start using `play(y)`. The animation shows the complete trajectory, assuming a constant speed and depth. Also the rudder deflection is animated. The following figure shows the last frame of the animation:

The NL object provided as an example contains a linearized model calibrated and validated to field test data.

```
m=exnl('auv')
NL object: AUV steering dynamics
 dx/dt = [th(1) th(2) 0; th(3) th(4) 0; 0 1 0]*x+[th(5); th(6);0]*u
     y = x + N([0;0;0],[0.001,0,0;0,0.01,0;0,0,1])
   x0' = [0          0           0]
   th' = [1.5       0.19        -45       -5.4        1.5        -36]
```

```
States:  v     r     psi
Outputs: v     r     psi
Inputs:  deltar
Param.: a11     a12     a21     a22     b1     b2
```

Because the model is linear, the state-space model and transfer function, respectively, can be retrieved. All NL objects can be linearized around a working point using `nl2ss`, and in this case, any working point can be used because the model is already linear.

```
mlinext=nl2ss(m,y(1));
mlin=mlinext(:,1)
              / 1.5  0.19  0 \        / 1.5 \
  d/dt x(t) = | -45  -5.4  0 | x(t) + | -36 | u(t)
              \   0     1  0 /        \   0 /

         /1  0  0\          /0\
  y(t) = | 0  1  0 | x(t) + | 0 | u(t)
         \0  0  1/          \0/


sscode=tex(mlin);
tfcode=tex(tf(mlin));
```

The LaTeX output provides the following formatted state-space model

$$\dot{x}(t) = \begin{bmatrix} 1.5 & 0.2 & 0.0 \\ -45.0 & -5.4 & 0.0 \\ 0.0 & 1.0 & 0.0 \end{bmatrix} x(t) + \begin{bmatrix} 1.5 \\ -35.6 \\ 0.0 \end{bmatrix} u(t)$$

$$y(t) = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} x(t)$$

and transfer function

$$Y_1(z) = \frac{1.5 \cdot z^2 + 1.2 \cdot z}{z^1(z^2 + 3.8 \cdot z + 0.36)} U(z)$$

$$Y_2(z) = \frac{-36 \cdot z^2 - 13 \cdot z}{z^1(z^2 + 3.8 \cdot z + 0.36)} U(z)$$

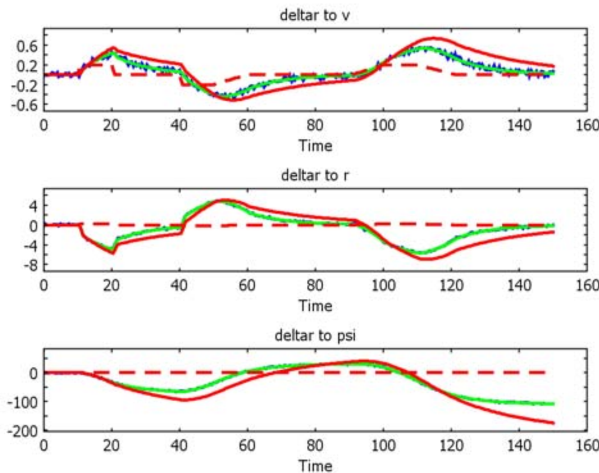
$$Y_3(z) = \frac{-36 \cdot z - 13}{z^1(z^2 + 3.8 \cdot z + 0.36)} U(z)$$

The advantage of using the NL object for linear systems is the possibility to represent partially unknown parameters in the model. In this case, there are four parameters in

the A matrix corresponding the unknown inertia of the AUV and two unknown parameters corresponding to the gain from rudder deflection to torque forces. However, there are six known parameters (fives 0 and one 1) corresponding to the integrator from yaw rate to yaw angle.

Suppose now that one parameter is uncertain and needs calibration. The coefficient a_{21} is changed one unit in the model used as initial guess in the NLS algorithm.

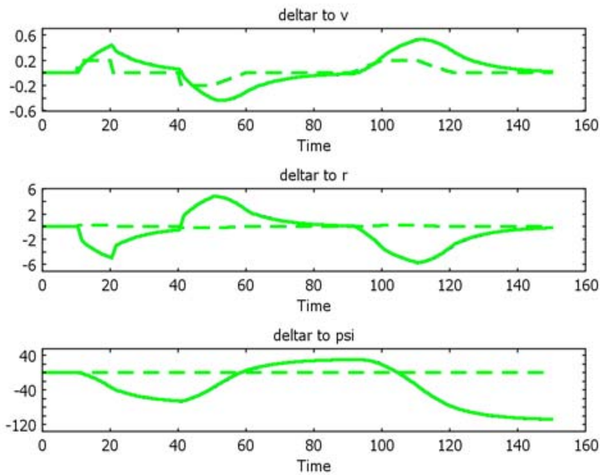
```
minit=m;
minit.pe=[];
minit.th=m.th+[0 0 1 0 0 0]';
yinit=simulate(minit,u2y(y));
plot(y,x2y(y),yinit)
```



The `x2y` method is used for extracting the state from the simulation. The figure illustrates that this quite small a change in one parameter affects the simulations quite a lot. The original data set can now be used to calibrate this value.

```
mhat=estimate(minit,y,'thmask',[0 0 1 0 0 0],'x0mask',[0 0 0])
NL object: AUV steering dynamics (calibrated from data)
 dx/dt = [th(1) th(2) 0; th(3) th(4) 0; 0 1 0]*x+[th(5); th(6);0]*u
      y = x +
N([0;0;0],[0.000918,0.000444,0.000912;0.000444,0.0104,-0.00265;0.
000912,-0.00265,0.951])
   x0' = [0          0          0]
   th' = [1.5       0.19       -45       -5.4       1.5       -36]
```

```
   std = [2.9e-010   2.9e-010     0.0029   2.9e-010   2.9e-010
2.9e-010]

  States:  v     r     psi
  Outputs: v     r     psi
  Inputs:  deltar
  Param.: a11     a12     a21     a22     b1     b2
yhat=simulate(mhat,u2y(y));
plot(x2y(y),x2y(yhat),'conf',99)
```



The parameter is estimated accurately with a quite small standard deviation. The corresponding confidence region in a simulation is very narrow and not visible in the plot above.

Despite this good estimation result, a very accurate initial model is needed because the region of global convergence is generally concentrated to a small region around the true parameters. That is, do not view NLS as a magical system identification algorithm that can start from any initial value and still converge.

Examples in Chemical Engineering

ISOTHERMAL PLUG FLOW REACTOR

A plug flow reactor is a cylindrical tube where the chemicals are inserted in one end of the cylinder and the products (and unreacted chemicals) come out at the other end.

Hence, the reaction takes place in the cylinder as the reactants move from the inlet side towards the outlet side.The first chemical reaction is an isothermal one (the temperature in the reactor is constant) and the second one nonisothermal (the temperature in the reactor varies during the reaction). Both examples are borrowed from Finlayson (2006), *Introduction to Chemical Engineering Computing*, page 118 and 121. The experiments are performed by first simulating the reaction and extracting measurements, with or without measurement noise. Using these measurements, the given model structure and an initial parameter guess, the parameter values are estimated using the method of NLS.

A somewhat simplified model of the isothermal plug flow reactor (IPFR) process is given below.

$$
\begin{aligned}
u\frac{\mathrm{d}C_A}{\mathrm{d}z} &= -2(kC_A(z) + C_A(z)^2), \\
y(z_k) &= C_A(z_k) + e(z_k), \\
e_k &\in N(0, R_k).
\end{aligned}
$$

C_A denotes the concentration, and z is the depth in the tube, corresponding to time in a standard model. That is, $t = z$ and $\theta = (k, u)$ in the NL model object. The model is predefined as a demonstration example:

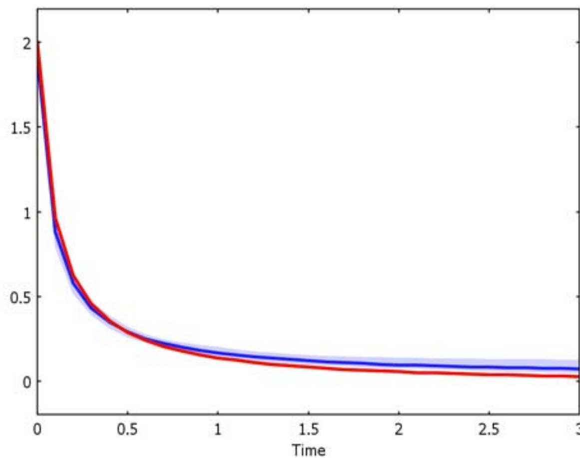
```
m=exnl('ipfr')
  Example from rate1 on page 119 in
     Introduction to chemical engineering computing
     Bruce A. Finlayson
     Wiley, 2006.
  Example modified by removing nuisance concentrations CB and CC
  Original example not identifiable of u and k, so linear term
included
NL object: Isothermal plug flow reactor
 dx/dt = -(2*th(1)/th(2))*x - (2/th(2))*x.^2
     y = x
   x0' = [2]
   th' = [0.1        0.4]
```

The model is simulated for a tube of length 3 units and measurements each 0.1 unit are assumed. The model is perturbed in the parameters, and the model used in the simulation is estimated using the simulated data and perturbed model as initial guess.

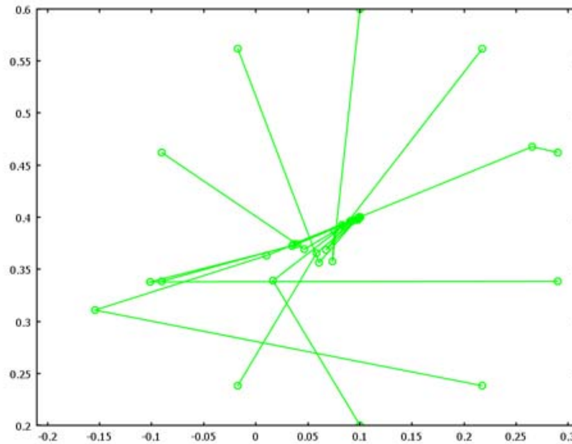
```
t=0:0.1:3;
y=simulate(m,t);
yn=y+ndist(0,0.01);
minit=m;
minit.th=[0.2;0.3];
```

```
mhat=estimate(minit,yn)
NL object: Isothermal plug flow reactor (calibrated from data)
 dx/dt = -(2*th(1)/th(2))*x - (2/th(2))*x.^2
     y = x + N(0,0.012)
   x0' = [1.9] + N(0,0.011)
   th' = [-0.036       0.32]
   std = [0.053      0.055]

mhat.pe=[];
yhat=simulate(mhat,t);
plot(yhat,y,x2y(y),'conf',90)
```

The parameter k is apparently not significant in the model because its standard deviation is larger than the estimated parameter. That is, the quadratic term dominates the linear term in the dynamics. The noise variance is correctly estimated, which indirectly validates the estimated model. The simulation also supports the estimated model and its confidence bound.



To check local identifiability, a number of different initial values are used in NLS, and the convergence path for each one is plotted below.



NLS converges at least for all points at a Euclidean distance of 0.5 from the true parameters.

NONISOTHERMAL PLUG FLOW REACTOR

In this example the previous model is extended to account for the heat transfer as well as the mass balance, when SO_2 is oxidized to form SO_3. The state vector $x = (X,T)$ includes concentration X of SO_2 and temperature T, and 12 parameters.

$$\frac{dX}{dz} = \theta_1 \frac{X\sqrt{1-\theta_2(1-X)} - \theta_3(1-X)/K_{eq}}{(k_1 + k_2(1-X))^2},$$

$$\frac{dT}{dz} = \theta_4(T-\theta_5) + \theta_6 \frac{X\sqrt{1-\theta_2(1-X)} - \theta_3(1-X)/K_{eq}}{(k_1 + k_2(1-X))^2},$$

$$y(z_k) = X(z_k) + e(z_k),$$

$$e_k \in N(0, R_k).$$

The model is predefined as an example.

```
m=exnl('nipfr')
  Example from rateSO2 on page 121 in
     Introduction to chemical engineering computing
     Bruce A. Finlayson
     Wiley, 2006.
```

```
NL constructor warning: try to vectorize f for increased speed
NL constructor warning: try to vectorize h for increased speed
NL object: Non-isothermal plug flow reactor
 dx/dt = nipfr(t,x,u,th)
     y = x(1)
   x0' = [1   6.7e+002]
  th' = [-50     0.17      2.2     -4.1  6.7e+002    1e+004
-15   1.1e+004       -1.3   2.3e+003        -11   1.2e+004]
```

The dynamic model is not well suited as an inline function, so `f` is represented as an M-file in this example. The M-file uses local variables where local parameters in the model are defined.

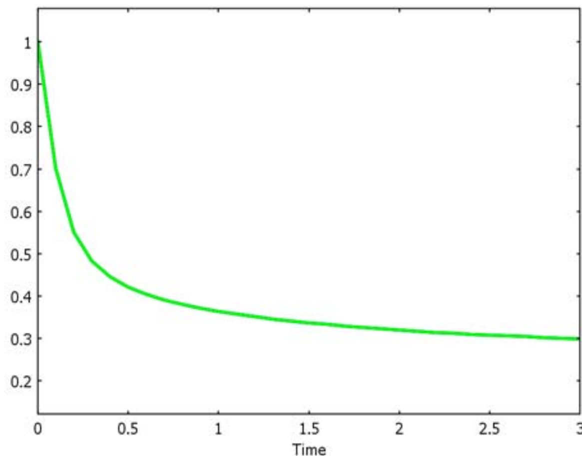
```
type nipfr

function dx=nipfr(t,x,u,th)
X=x(1,:);
T=x(2,:);

k1=exp(th(7) + th(8)./T);
k2=exp(th(9) + th(10)./T);
Keq=th(11)+th(12)./T;
Rn=X.*sqrt(1-th(2).*(1-X)) - th(3)*(1-X)/Keq;
Rd=(k1+k2*(1-X)).^2;
R=Rn/Rd;

dx1=th(1)*R;
dx2=th(4)*(T-th(5)) + th(6)*R;
dx=[dx1;dx2];
```

The model is simulated without noise (the many parameters are hardly possible to estimate from one single experiment if measurement noise is added), the parameter is perturbed about 10% with a Gaussian random disturbance, and the perturbed model is used as initial guess in NLS.

```
t=0:0.1:3;
y=simulate(m,t);
minit=m;
minit.th=m.th+0.01*diag(sqrt(abs(m.th)))*randn(size(m.th));
mhat=estimate(minit,y,'disp',1)
-----------------------------------------------------
Iter#          Cost    Grad. norm   BT#   Alg
-----------------------------------------------------
    0    2.442e-004             -     -   rgn
    1    1.524e-008    2.053e-002     1   rgn
    2    4.157e-012    1.635e-004     1   rgn
Relative difference in the cost function < opt.ctol.
NL object: Non-isothermal plug flow reactor (calibrated from data)
```

```
 dx/dt = nipfr(t,x,u,th)
     y = x(1) + N(0,1.34e-013)
   x0' = [1   6.7e+002] +
N(0,[1.2e-013,-1.3e-011;-1.3e-011,2.9e-009])
  th' = [-50       0.2       2.2      -4.1   6.7e+002    1e+004
-15    1.1e+004        -1.3    2.3e+003         -11    1.2e+004]
  std = [0.047    0.0067    0.0067    0.0016     0.075    0.00023
0.00019     0.0005    0.00057     0.0016       0.01    9.3e-006]

yhat=simulate(mhat,t);
plot(yhat,y,'conf',90)
```

The resulting model is accurately estimated, and the only uncertainty comes from numerical problems.



TEMPERATURE DEPENDENT REACTION WITH MULTIPLE DATA SETS

A typical chemical reaction shows an exponential decay, where the time constant depends on temperature. The model below shows a generic model for such a case.

```
m=exnl('reaction')
NL constructor warning: try to vectorize f for increased speed
NL object: Temperature depending reaction
 dx/dt = [-th(1)*10000*exp(-th(2)*10000/(8.31441*u(1,:)))*x(1,:)]
     y = [x(1,:)]
   x0' = [1]
   th' = [5         2]

  States:  Concentration
```

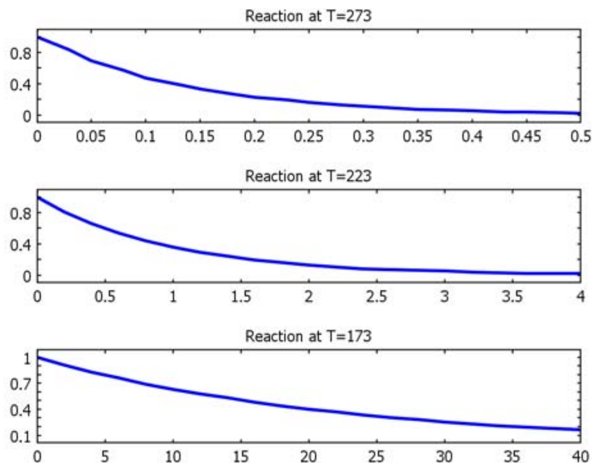
```
  Outputs: Measured concentration
  Inputs:  Temperature
```

The temperature is a known variable in the dynamics, and should thus be treated as a system input u.

The two parameters $\theta(1)$ and $\theta(2)$ are not identifiable from one single experiment, since the exponential is just a constant factor for each parameter $\theta(2)$ when $u(1)$ is constant. However, if two data sets with different temperature are available, then both parameters should be possible to estimate.

Three data sets for three different temperatures are available as a demo.

```
load reaction
z1
SIG object with continuous time input-output data
  Name:        Reaction at T=273
  Sizes:       N = 21,  ny = 1, nu = 1
  MC is set to: 30
  #MC samples:  0
subplot(3,1,1), yplot(z1)
subplot(3,1,2), yplot(z2)
subplot(3,1,3), yplot(z3)
```



Note the different time scales of the subplots. First, try to calibrate the nominal model using only the first data set.

```
x0mask = [0];
```

```
mhat1 = estimate(m,z1,'disp','on','x0mask',x0mask);
-----------------------------------------------------
Iter#          Cost    Grad. norm   BT#   Alg
-----------------------------------------------------
    0    1.604e-003             -     -   rgn
    1    1.203e-003    1.018e-001     1   rgn
    2    1.203e-003    1.158e-003     1   rgn
Relative difference in the cost function < opt.ctol.
mhat1
NL object: Temperature depending reaction (calibrated from data)
 dx/dt = [-th(1)*10000*exp(-th(2)*10000/(8.31441*u(1,:)))*x(1,:)]
     y = [x(1,:)] + N(0,5.73e-005)
   x0' = [1]
   th' = [5          2]
   std = [6.7e-005     0.0015]

  States:  Concentration
  Outputs: Measured concentration
  Inputs:  Temperature
```

The model appears to be successfully estimated, because the standard deviations are much smaller than the absolute value of the corresponding parameter. However, the standard deviation does not reflect the high correlation between the parameters. The following lines reveal that they are in fact perfectly correlated.

```
mhat1.P
ans =
   4.5e-009  -1.0e-007
  -1.0e-007   2.2e-006
eig(mhat1.P)
ans =
   2.2e-020
   2.2e-006
```

There are two ways to calibrate the parameters from multiple data sets:

1. Call the `estimate` method recursively in the number of data sets, where the previously estimated model is used as initial model at each call.
2. Call the `estimate` method or `nls` function once, with all data sets in a cell array. This is the preferred way due to estimation accuracy and speed.

The first alternative is tested below.

```
tic,
mhat1 = estimate(m,z1,'x0mask',x0mask);
mhat2 = estimate(mhat1,z2,'x0mask',x0mask);
mhat3 = estimate(mhat2,z3,'x0mask',x0mask);
toc,
```

```
Elapsed time:  57.375 s

mhat3
NL object: Temperature depending reaction (calibrated from data)
(calibrated from data) (calibrated from data)
 dx/dt = [-th(1)*10000*exp(-th(2)*10000/(8.31441*u(1,:)))*x(1,:)]
     y = [x(1,:)] + N(0,6.9e-006)
   x0' = [1]
   th' = [4.8         2]
   std = [0.076     0.0025]

  States:  Concentration
  Outputs: Measured concentration
  Inputs:  Temperature
```

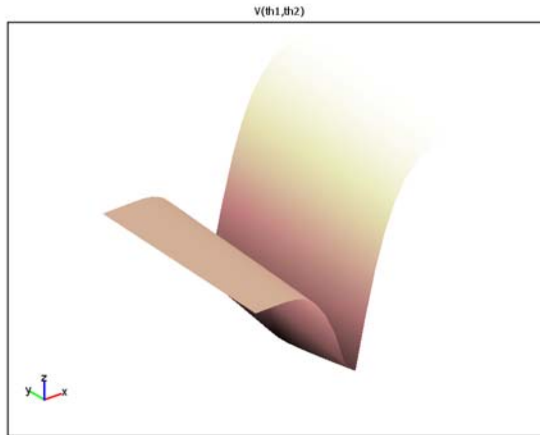
Note that the numerical values of the parameters have changed from the ones obtained from `z1`. The second alternative returns a similar estimate:

```
tic,
mhat = estimate(m,{z1,z2,z3},'disp','on','x0mask',x0mask);
-----------------------------------------------------
Iter#          Cost    Grad. norm   BT#   Alg
-----------------------------------------------------
    0    1.953e-003             -     -   rgn
    1    1.574e-003    7.734e-002     1   rgn
    2    1.568e-003    3.135e-002     1   rgn
Relative difference in the cost function < opt.ctol.
toc

Elapsed time:  2.750 s

mhat
NL object: Temperature depending reaction (calibrated from data)
 dx/dt = [-th(1)*10000*exp(-th(2)*10000/(8.31441*u(1,:)))*x(1,:)]
     y = [x(1,:)] + N(0,2.49e-005)
   x0' = [1]
   th' = [4.8         2]
   std = [0.06     0.0021]

  States:  Concentration
  Outputs: Measured concentration
  Inputs:  Temperature
```

Note, in particular, that it runs 20 times faster.

Finally, because the parameter vector is only two-dimensional, the NLS cost function can be computed by looping over a grid of parameter values. The surface plot below

illustrates the numerical difficulties in finding the minimum, because there is a very narrow but curved ridge in the cost function.



6

Statistics

This chapter describes the objects relating to statistical distributions, and their methods for data analysis, random number generation and hypothesis testing, to mention a few applications. The use of distributions to define stochastic signals and uncertain models is explained in their respective chapter.

PDF Objects

This section is organized as follows:

- "Available PDFs" on page 238 presents the available objects and their basic methods.
- "The PDFTOOL User Interface" on page 239 presents the GUI.
- "Basic Use" on page 241 gives a tutorial on how to use the objects in script mode.
- "Symbolic Computations" on page 243 gives examples of implemented symbolic computations.
- "Estimation" on page 244 explains how parametric distributions can be fit to data.
- "Multivariate Distributions" on page 246 extends the univariate distributions described before, and provides some examples.
- "Defining Your Own Distributions" on page 249 describes how you can extend the family of distributions yourself.

Available PDFs

The Signals & Systems Lab includes the following functions for probability density functions (PDFs):

FUNCTION	PURPOSE
`pdfclass`	The parent of all PDF classes, where default operations for each class are given.
`ndist(mu,P)`	The normal, or Gaussian, distribution
`udist(a,b)`	The uniform distribution
`expdist(mu)`	The exponential distribution
`tdist(n)`	Student's t distribution
`gammadist(a,b)`	The gamma distribution
`betadist(a,b)`	The beta distribution
`fdist(d1,d2)`	The F-distribution

You can use the PDF objects for the following typical problems in statistics:

- Evaluating the probability density function (PDF).
- Computing the cumulative distribution function (CDF).

- Computing the inverse cumulative distribution function, also called the error function (erf).
- Random number generation.
- Returning the first four moments: mean, variance, skewness, and kurtosis.

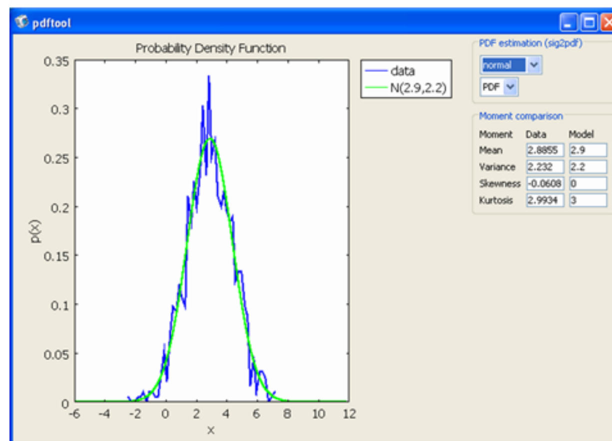
The Signals & Systems Lab supports the following PDFs:

- uniform: $U(min,max)$
- exponential: $exp(\mu)$
- normal: $N(\mu,var)$.
- multivariate: $N(\mu,P)$
- chi2: $chi2(n)$, where n is an integer.
- Student: $t(n)$, where n is an integer.
- F: $F(d1,d2)$, where $d1$ and $d2$ are positive integers.
- gamma: $\Gamma(a,b)$, where $a>0$ is a shape and $b>0$ is a scale parameter.
- beta: $\beta(a,b)$.

The PDFTOOL User Interface

To analyze the amplitude distribution of a given vector and scalar signal in a SIG object, call the `pdftool` graphical user interface (GUI) with one input argument.

```
pdftool(y)
```



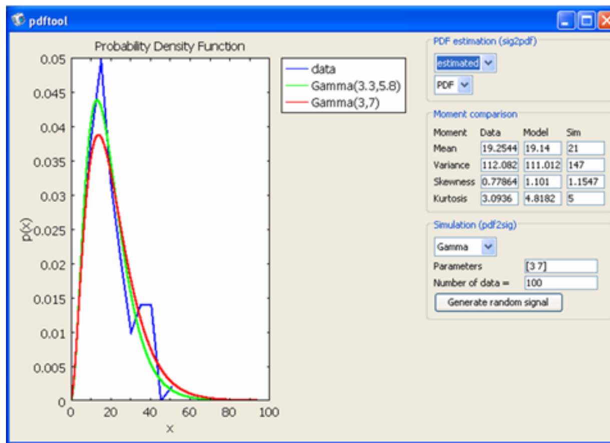
The GUI displays the first four moments and plots a histogram obtained by convolving a Gaussian smoothing kernel around each observation. It also estimates the parameters in a Gaussian distribution that gives a perfect fit of the first two moments, and plots the parametric probability density function. You can change the following settings:

- The parametric distribution, according to the available objects in the PDFCLASS. Type `list(pdfclass)` for a complete list. You can also choose Optimize that chooses the distribution that gives the best fit of the cumulative density function (CDF).
- The type of plot shown among the options PDF, CDF, ERF (error function), ERFINV (inverse error function).

You can use the fit of the histogram and parametric distribution as well as the different moments to validate the chosen distribution.

To start in training mode, call `pdftool` without input arguments:

```
pdftool
```



In this mode you can:

- Choose a distribution from the available ones in PDFCLASS
- Set the parameters in this distribution
- Simulate a number of data
- Generate new realizations to examine sensitivity

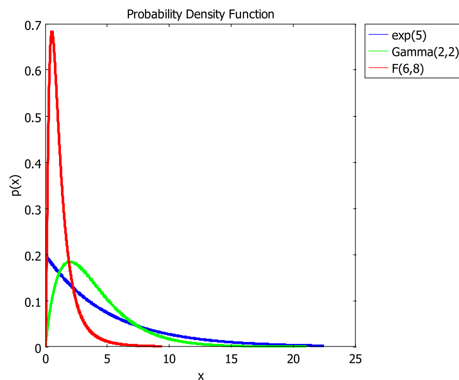
The other choices are the same as in the analysis mode.

Basic Use

This section describes how you define a distribution, plot the probability density function (PDF) or cumulative density function (CDF), generate random numbers, convert a vector of random numbers to an empirical distribution, and compute theoretical and empirical moments.

You define distributions in a rather symbolical way:

```
X1=expdist(5)
exp(5)
X2=gammadist(2,2)
Gamma(2,2)
X3=fdist(6,8)
F(6,8)
plot(X1,X2,X3)
```

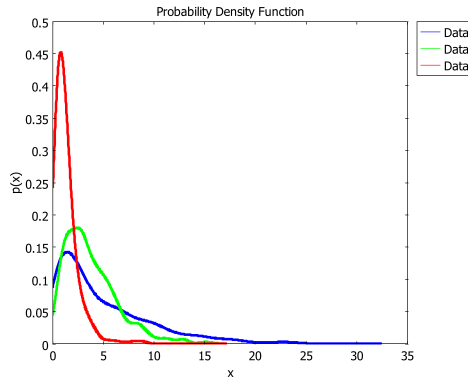


The overloaded `plot` function, as well as `cdfplot` and `erfplot`, allows multiple input arguments.

Next, generate random numbers from these distributions. Each PDF family is its own object, so `rand` is a method specific to each object. Many PDF families use the default algorithm defined in `pdfclass`, which numerically inverts the cumulative density function. The following code converts the vectors of random numbers to empirical distributions and uses `plot` to show an empirical estimate of the PDF from the simulated data.

```
x1=rand(X1,1000,1);
x2=rand(X2,1000,1);
x3=rand(X3,1000,1);
Y1=empdist(x1);
```

```
Y2=empdist(x2);
Y3=empdist(x3);
plot(Y1,Y2,Y3)
```

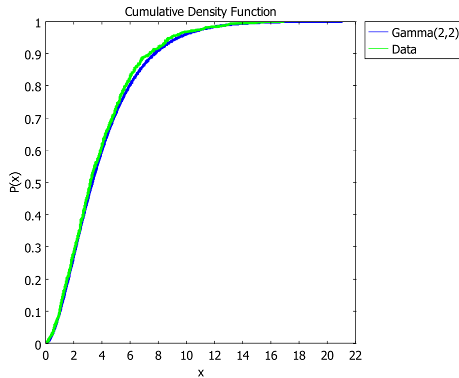


Use the following function calls to compute theoretical and empirical moments, respectively:

```
disp([E(X2) var(X2) skew(X2) kurt(X2)])
          4          8     1.4142          3
disp([E(Y2) var(Y2) skew(Y2) kurt(Y2)])
     3.8322     7.2930     1.3376     2.3162
```

To compute confidence intervals, use the error function and, in particular, its inverse. The following code plots the cumulative density function (CDF), computes the threshold value where 0.99 of the probability mass is found, and tests the ratio of random numbers actually smaller than this threshold. Finally, the integral of the CDF up to $x = 10$ is found using `erf`.

```
cdfplot(X2,Y2)
h=erfinv(X2,0.99)
h =
13.1946
length(find(x2<h))/length(x2)
ans =
0.9930
erfinv(X2,[0.005 0.995])
ans =
     0.1965
    14.7099
erf(X2,10)
ans =
0.9601
```



The definition of the threshold h is such that $P(X < h) = 0.99$, and thus it defines a one-sided 99% confidence interval. To compute a two-sided interval, split the error probability into two halves, as previous example's last call to `erfinv` shows.

Symbolic Computations

In some special cases, the PDF classes are preserved under specific operations. Some of these are implemented in the Signals & Systems Lab. Linear operations on Gaussian distributions is the most important case:

```
p1=ndist(1,2)
N(1,2)
p2=2*p1+3
N(5,8)
```

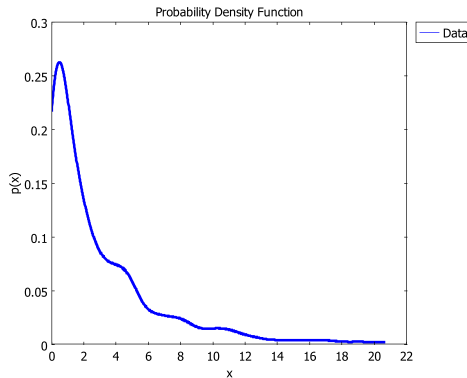
Other cases concern addition and multiplication of two Gamma distributions, linear operations on uniform distributions, and addition of chi-square distributions:

```
X=expdist(2)
exp(2)
Y=3*X
exp(6)
X1=gammadist(3,5);
X2=gammadist(7,5);
Y=X1+X2
Gamma(10,5)
X1=chi2dist(3);
X2=chi2dist(5);
Y=X1+X2
chi2(8)
X=udist(2,4);
```

```
Y=1-2*X
Empirical data vector of size 1 with 1000 samples
```

All other operations result in nonstandard distributions, which the Signals & Systems Lab automatically represents with empirical distributions. That is, as soon as an operation results in a nonstandard distribution, the software generates random numbers and applies the operation elementwise to the random numbers. The number of samples is critical for performance. It is stored as the field MC in each PDF object, and you can change this manually from its default value.

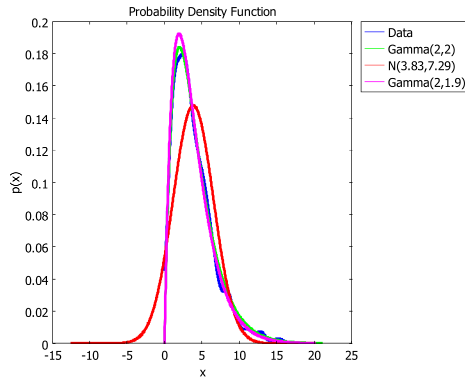
```
X=ndist(0,1);
Y=3*X^2
Empirical data vector of size 1 with 1000 samples
plot(Y)
```



Estimation

A standard problem in statistics is to estimate the parameters in a distribution to fit collected data. The following code shows how to fit a normal and exponential distribution to data. The plot illustrates the theoretical distribution, the empirical data, and the two estimates.

```
X2hatgauss=estimate(ndist,Y2)
N(3.83,7.29)
X2hatgamma=estimate(gammadist,Y2)
Gamma(2,1.9)
plot(Y2,X2,X2hatgauss,X2hatgamma)
```

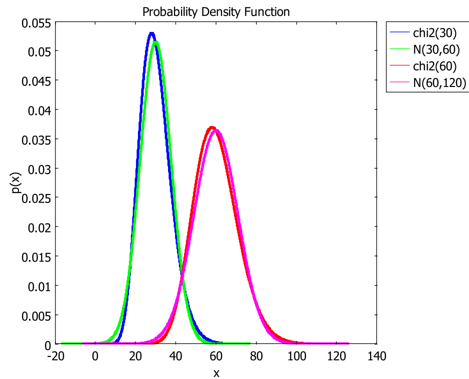


Moreover, you can also estimate the type of distribution. This works as follows: the estimate method of `pdfclass` takes an array of valid distributions as second input, loops through the distributions, estimates the parameters in each one, and compares how well the estimated distribution fits the empirical CDF. The following lines first list all available distributions and then estimate the parameters in a subset of them.

```
l=list(pdfclass)
l =

Columns 1-7  'betadist'  'chi2dist'  'expdist'  'fdist'  'gammadist'
'ndist'  'tdist'

Column 8     'udist'
X2hat=estimate(Y2,{l{2:4}})
Warning: d1<0 estimated, changed to 1.
chi2(4)
```

You can also use the estimation facility to test the validity of the central limit theorem in specific cases. Take for instance the chi-square distribution, which converges to the normal distribution when the number of freedoms tends to infinity:

```
X1=chi2dist(30);
Y1=estimate(ndist,X1);
X2=chi2dist(60);
Y2=estimate(ndist,X2);
plot(X1,Y1,X2,Y2)
```



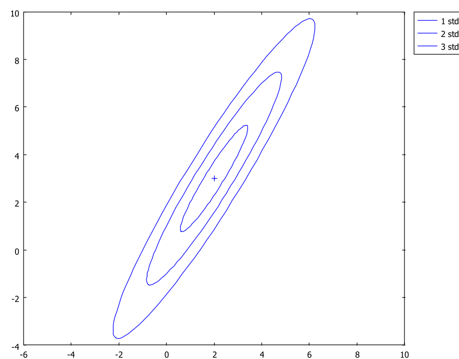
Multivariate Distributions

You can define normal and empirical distributions as *multivariate distributions*. A Gaussian vector obeys certain rules under linear transformations:

```
X=ndist(ones(2,1),eye(2))
N([1;1],[1,0;0,1])
Y=[1 1;1 2]*X
N([2;3],[2,3;3,5])
```

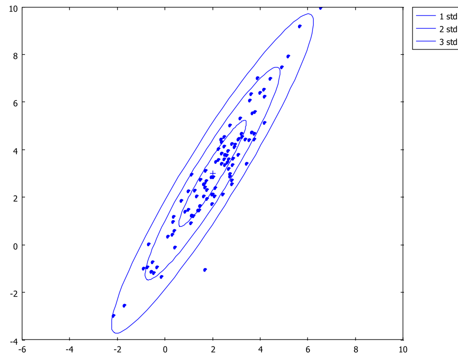
The stochastic vector Y is correlated, which a contour plot reveals:

```
contour(Y)
```



Use the `rand` function to generate random samples. Note that the dimension is a matrix with as many columns as there are elements in the random vector. That is, you cannot request an arbitrarily-sized matrix with random numbers. The following contour plot illustrates random vectors:

```
y=rand(Y,100);
hold on
plot(y(:,1),y(:,2),'.')
hold off
```



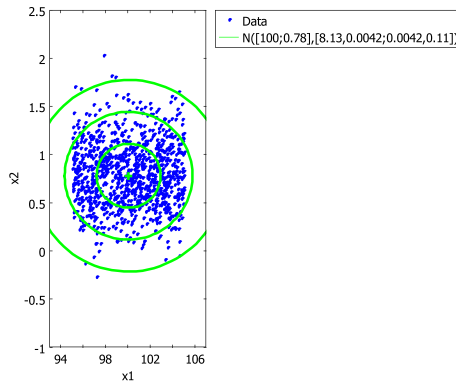
You can use these samples to illustrate the use of an empirical multivariate distribution. Here, use it to estimate the covariance:

```
Z=empdist(y);
cov(Z)
ans =
     2.3456     3.4515
     3.4515     5.6250
```

As a more application-oriented example, consider a radar that measures distance R with high accuracy and bearing θ with quite poor resolution. The range gating implies a uniform distribution on range, while you can assume that the bearing error is Gaussian:

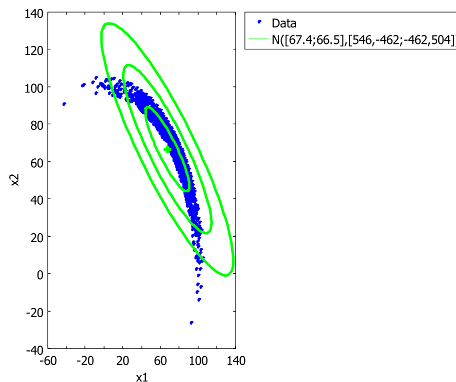
```
R=udist(95,105);
TH=ndist(pi/4,0.1);
Z=[R;TH]
Empirical data vector of size 2 with 1000 samples
Z.xlabel={'R','Theta'};
cov(Z)
ans =
     8.1254     0.0042
     0.0042     0.1099
```

```
Zg=estimate(ndist,Z);
plot2(Z,Zg,'Xlim',[93 107])
```



For tracking and surveillance applications, the Cartesian position is of interest. Therefore, calculate a stochastic position vector Y and illustrate it together with a Gaussian approximation:

```
Y=[R.*cos(TH);R.*sin(TH)]
Empirical data vector of size 2 with 1000 samples
Y.xlabel={'X1','X2'};
Yg=estimate(ndist,Y);
plot2(Y,Yg)
```



The correlation between the two coordinates is obvious: The scatter points are more or less oriented on a circle with radius 100. How does it work? To get correct correlation properties in the empirical distribution, it is important that you have the

same random numbers each time you use a stochastic variable in an expression. To make sure that the random numbers are the same, use a unique state variable to each random variable or vector, similar to the state in the built-in `rand` and `randn` functions. In contrast to these functions, however, this state does not change during a call to `rand` for the reasons indicated above.

```
R.state
ans =
550120
rand(R,3)
ans =
    95.9512
   104.0740
   100.2471
rand(R,3)
ans =
    95.9512
   104.0740
   100.2471
```

If you need a new independent stochastic variable or vector, make a new copy the stochastic variable. This changes the public field `state` automatically. Or, you can change the state manually. The following lines of code show an example:

```
R2=udist(R);
R2.state
ans =
726761
rand(R2,3)
ans =
   103.6843
    96.3236
    97.9701
```

Defining Your Own Distributions

You can rather easily define new PDF objects, using the file `chi2dist.csl`, for example, as a template:

```
class chi2dist extends pdfclass
%CHI2DIST defines the chi2 distribution chi2(d)
%   Reference: http://en.wikipedia.org/wiki/
Chi-square_distribution

private d
public MC

function chi2dist(d);
```

```
% Constructor
MC=1000; % Default number
if nargin==0
    d=[];
elseif nargin==1
    if ~isnumericscalar(d) | d<=0 | round(d)~=d
        error('d must be a positive integer')
    end
else
    error('Syntax: chi2dist(d) or chi2dist')
end
```

The first line defines the class and points to the parent object `pdfclass`. Many basic methods are inherited from this class, such as numerical algorithms for computing the CDF and random number generator. The essential methods you have to write yourself is the constructor, which should be able to accept zero arguments, a display function, a symbolic function (which is very similar to the display function), the moments, and the PDF method `pdf`. The GUI automatically detects any new children of `pdfclass` and updates the list of PDFs to simulate. If you also use the moments to solve for the PDF parameters in a method called `estimate`, the estimation facility in the GUI works as well.

For more information about classes and methods in COMSOL Script, see "User-Defined Classes" on page 275 in the *COMSOL Script User's Guide*.

Fusion

Theory

Fusion concerns the problem of combining two different estimates represented by their unbiased mean and covariance matrices:

$$E(\hat{x}_1) = E(\hat{x}_2) = x,$$
$$\mathrm{cov}(\hat{x}_1) = P_1,$$
$$\mathrm{cov}(\hat{x}_2) = P_2.$$

If these estimates are independent, so $P_{12} = 0$, then the sensor fusion formula applies:

$$P = (P_1^{-1} + P_2^{-1})^{-1},$$
$$\hat{x} = P(P_1^{-1}\hat{x}_1 + P_2^{-1}\hat{x}_2).$$

This formula applies even when the covariances are singular, if the inverses are replaced by pseudo-inverses. This is useful in cases where inference algorithms are to be distributed in space or time, and incomplete data have to be used in each partial inference step. This method is implemented in `pdfclass.fusion`, where the result is returned as an `ndist` object. However, the normal distribution is only correct if each input is normally distributed.

If there is a known correlation, the Kalman filter formula applies (see "Nonlinear Transformation-Based Filters" on page 171),

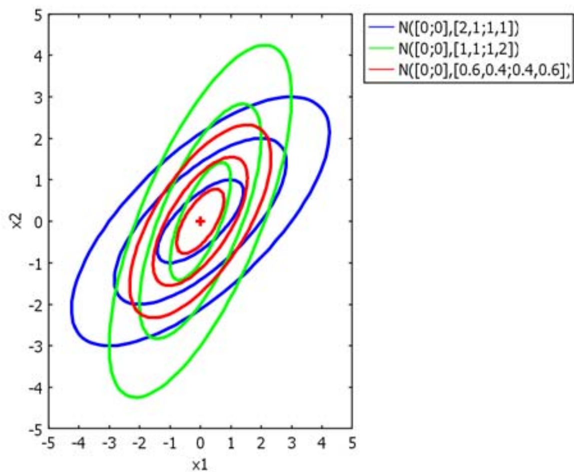
$$K = P_{12}P_2^{-1}$$
$$P = P_1 - KP_2K^T$$
$$\hat{x} = \hat{x}_1 + K(\hat{x}_2 - \hat{x}_1).$$

In the general case of unknown cross correlation, covariance intersection methods can be used. These are based on a worst case assumption, so they provide an upper bound on the covariance from a Kalman filter based fusion. This algorithm is available as `pdfclass.safefusion`.

Examples

As a test case, two Gaussian two-dimensional variables with zero mean and different nondiagonal covariance matrices are to be fused. First, assuming independence, the result looks as below.

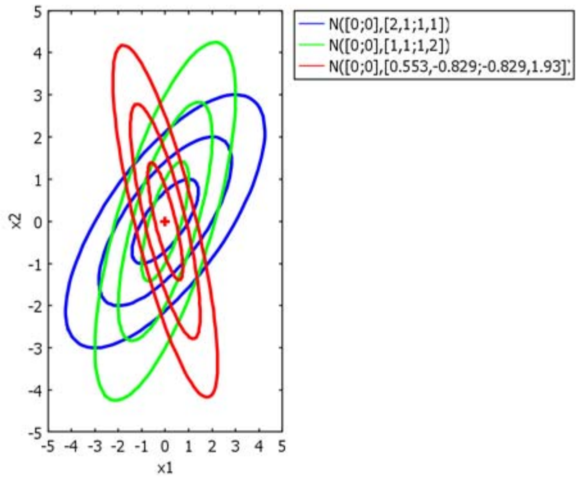
```
X1=ndist([0;0],[2 1;1 1]);
X2=ndist([0;0],[1 1;1 2]);
X=fusion(X1,X2)
N([0;0],[0.6,0.4;0.4,0.6])
plot2(X1,X2,X)
```



The fused covariance ellipsoid is inside both of the corresponding ellipsoids representing the input estimates, which is a characteristic property of optimal fusion. The output estimate improves strictly in all directions.

The following lines illustrate the conservative approach when there is an unknown correlation between the input distributions.

```
X=safefusion(X1,X2)
N([0;0],[0.553,-0.829;-0.829,1.93])
plot2(X1,X2,X)
```



The fused covariance is here considerably larger.

Nonlinear Transformations

Nonlinear transformations (NLT) of stochastic variables from parametric distributions seldom lead to a new parametric distribution. The standard option in the `pdfclass` is to represent the new distribution with Monte Carlo samples. Here, another alternative based is described that apply to the `ndist` object.

Algorithms

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

$$X \in N(m_x, P_x) \Rightarrow Z = g(X) \approx N(m_z, P_z)$$

The following methods are available:

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

$$m_z = g(\hat{x}),$$
$$P_z = [g'_i(\hat{x})P(g'_j(\hat{x}))^T]_{ij}$$

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

$$m_z = g(\hat{x}) + \frac{1}{2}[\mathrm{tr}(g''_i(\hat{x})P)]_i,$$
$$P_z = \left[g'_i(\hat{x})P(g'_j(\hat{x}))^T + \frac{1}{2}\mathrm{tr}(Pg''_i(\hat{x})Pg''_j(\hat{x}))\right]_{ij}$$

3. 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 transformation, and the mean and covariance are fitted to these points.

$$x^0 = \hat{x},$$
$$x^{\pm i} = \hat{x} \pm \sqrt{n_x + \lambda} P^{1/2}_{:,i}, \quad i = 1, 2, \ldots, n_x$$
$$z^i = g(x^i), \quad i = -n_x, \ldots, n_x$$
$$m_z = \frac{\lambda}{n_x + \lambda} z^0 + \sum_{i=\pm 1}^{\pm n_x} \frac{1}{2(n_x + \lambda)} z^i,$$
$$P_z = \left(\frac{\lambda}{n_x + \lambda} + (1 - \alpha^2 + \beta)\right)(z^0 - E(z))(z^0 - E(z))^T$$
$$+ \sum_{i=\pm 1}^{\pm n_x} \frac{1}{2(n_x + \lambda)} (z^i - E(z))(z^i - E(z))^T.$$

4 MC in `ndist.mceval`: Generate N random points, transform these, and fit mean and covariance to these points.

$$x^i \in N(m_x, P_x),$$
$$z^i = g(x^i),$$
$$m_z = \frac{1}{N} \sum_{i=1}^{N} z^i,$$
$$P_z = \frac{1}{N} \sum_{i=1}^{N} (z^i - m_z))(z^i - m_z)^T.$$

Comments:

- The Monte Carlo method should be the most reliable in general, but requires some tuning of the number of samples and it also requires most computations.
- The unscented transformation is more computational efficient than the Monte Carlo method in that a fixed and rather small number $2n_x$+1 samples are needed. Another advantage compared to the Taylor-based methods is that only function evaluations are needed.
- The first-order Taylor expansion is the classical approach in the spirit of try simple things first.
- The second-order Taylor expansion compensates for the second-order moment, so the mean and covariance should be correct in the answer. This nice property is often claimed also for the UT when the input is Gaussian.

Examples

Consider first a NLT where the first- and second-order moments can be computed analytically.

$$X \in N(\mu, \sigma^2),$$
$$Z = (X+1)^2 = X^2 + 2X + 1,$$
$$m_z = \sigma^2 + 2\mu + 1,$$
$$P_z = 4\sigma^3.$$

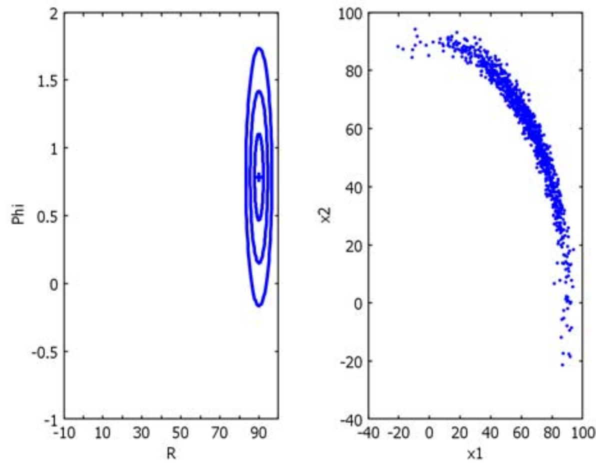
The numerical results of the different approximations are summarized below:

```
X=ndist(0,1)
N(0,1)
h=inline('(x+1).^2');
Ymc = mceval(X,h)
N(2.01,4.32)
Ymc = mceval(X,h,1000)
N(1.97,5.94)
Ytt1=tt1eval(X,h)
N(1,4)
Yut = uteval(X,h)
N(2,6)
```

The theoretical results for mean and variance according to the formula above are 2 and 6, respectively. The UT is in this case superior, because it delivers the most accurate answer with the default design parameters.

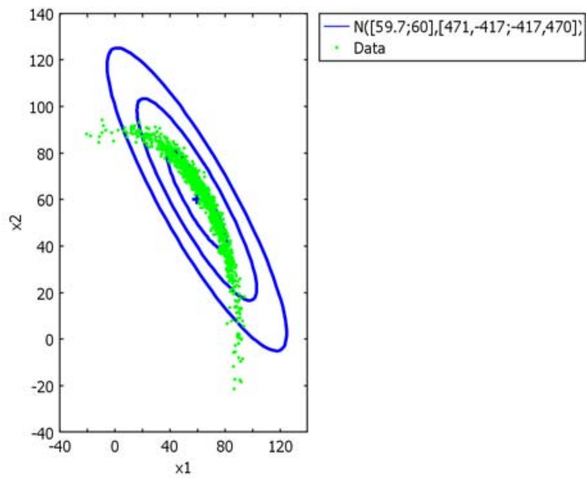
A more challenging example with implications for radar-based tracking applications is the conversion from range and bearing to Cartesian coordinates. The following lines defines the stochastic measurements and the NLT.

```
R=90+ndist(0,5);
Phi=pi/4+ndist(0,0.1);
subplot(1,2,1)
Z=[R;Phi];
Z.xlabel={'R','Phi'};
plot2(Z,'legend',[])
subplot(1,2,2)
plot2([R*cos(Phi);R*sin(Phi)],'legend',[])
```



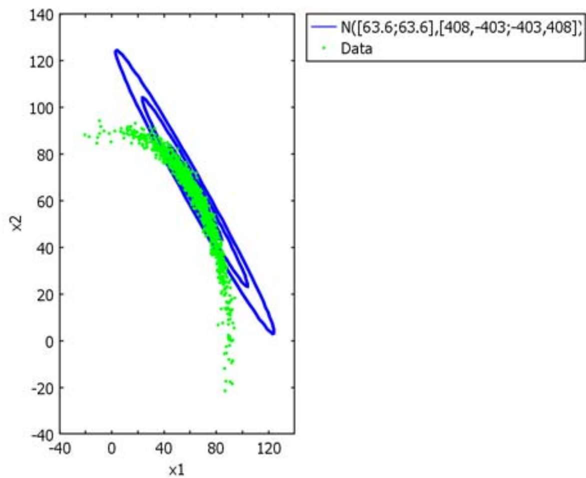
The banana-shaped samples are clearly from a non-Gaussian distribution. The question is, which Gaussian distribution catches the first-order and second-order moments? The Monte Carlo method fits mean and covariance to the samples in the right plot, and the result is depicted below.

```
h=inline('[x(1,:).*cos(x(2,:)); x(1,:).*sin(x(2,:))]');
Nmc=mceval([R;Phi],h);
plot2(Nmc,[R*cos(Phi);R*sin(Phi)])
```



The first-order Taylor expansion is unreliable in this case.

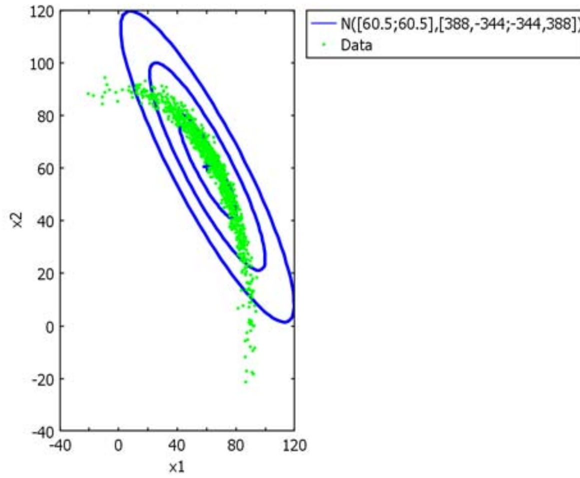
```
h=inline('[x(1,:).*cos(x(2,:)); x(1,:).*sin(x(2,:))]');
Ntt1=tt1eval([R;Phi],h);
plot2(Ntt1,[R*cos(Phi);R*sin(Phi)])
```



Comparing these plots, the TT1 approximation gives a positive bias in range and too small a variance in range, while the bearing uncertainty is well represented.

The UT gives a result very similar to the MC method, and thus works quite well in this case as well.

```
h=inline('[x(1,:).*cos(x(2,:)); x(1,:).*sin(x(2,:))]');
Nut=uteval([R;Phi],h);
plot2(Nut,[R*cos(Phi);R*sin(Phi)])
```



7

Adaptive Filtering and Nonstationary Signal Processing

The Fourier transform assumes periodic signals, and spectral analysis requires stationary signals. For signals whose properties vary in time, the generalization is here referred to as time-frequency descriptions (TFD). The current version of the Signals & Systems Lab covers short-time Fourier transforms and smoothed version in the TFD object. Model-based approaches are straightforward extensions of model estimation, where the parameters in the model are allowed to vary with time. The general class is labeled Linear Time-Varying (LTV), as an extension to LTI models. The only nonempty object in this class is the RARX object, which recursively estimates an ARX model.

The main sections are:

- “Time-Frequency Descriptions” on page 262.
- “Adaptive Filtering” on page 267.

Time-Frequency Descriptions

Introduction

For nonstationary signals, the counterpart to a spectral description is a time-frequency description (TFD). Basically, a time-varying spectrum can be defined as the squared magnitude of a short-time Fourier transform

$$\mathrm{TFD}(f,t) = \frac{T}{S}\left|\sum_{k=0}^{N} w[t-k]s[k]e^{-i2\pi fk}\right|^2 .$$

If the sliding window $w[k]$ is constant, then the TFD coincides with the periodogram used in spectral estimation.

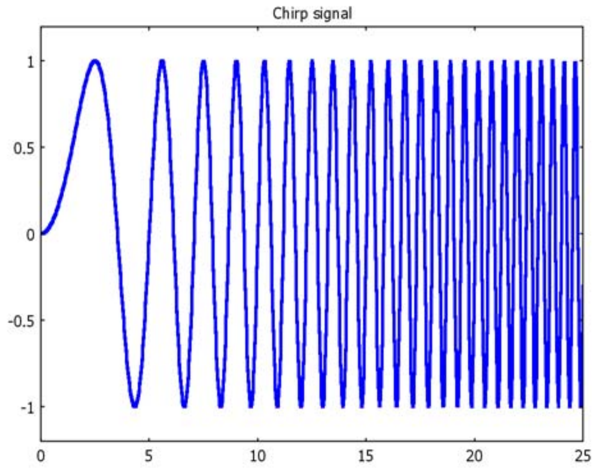
The TFD constructor from SIG objects calls `sig2tfd`. You can here choose the window type $w[k]$ from the options in `getwindow` and its size S. The size, S, of the sliding window decides the time-frequency resolution tradeoff. The larger S, the better frequency resolution at the price of decreased time resolution. You also decide on the amount of overlap of each window, which basically corresponds to the density of the grid on time t in the computed TFD. The window size S determines the frequency grid, so the resolution is $2\pi/ST$.

Tutorial

NONPARAMETRIC APPROACHES

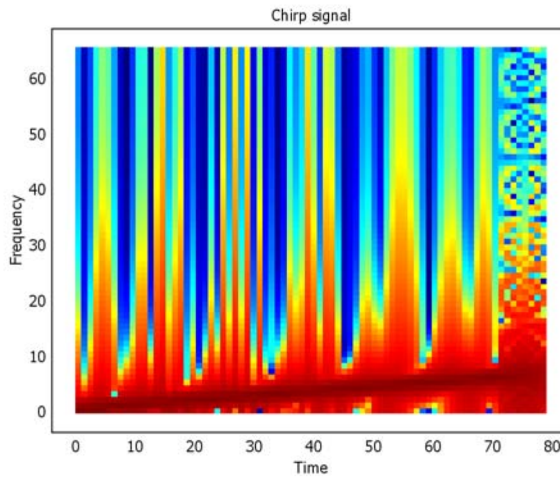
Consider the chirp signal obtained by the default values in `getsignal`:

```
y=getsignal('chirp');
plot(y);
```



The frequency of a chirp signal increases linearly in time, and to find this property is the task of a TFD. You compute the TFD using the `tfd` constructor and illustrate it with the `plot` method for the TFD object:

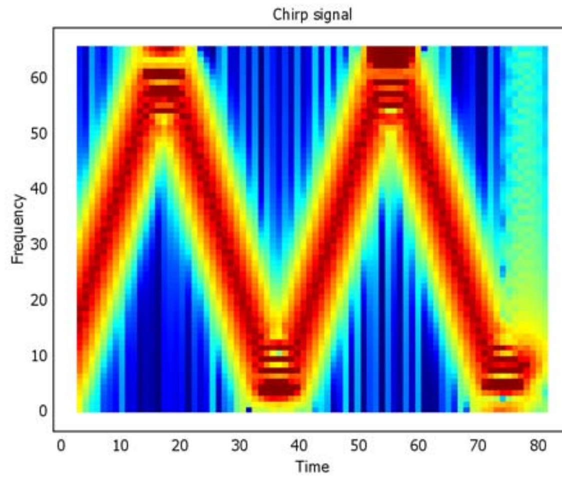
```
Yt=tfd(y);
plot(Yt);
```



The linear frequency increase is clearly visible from the image plot.

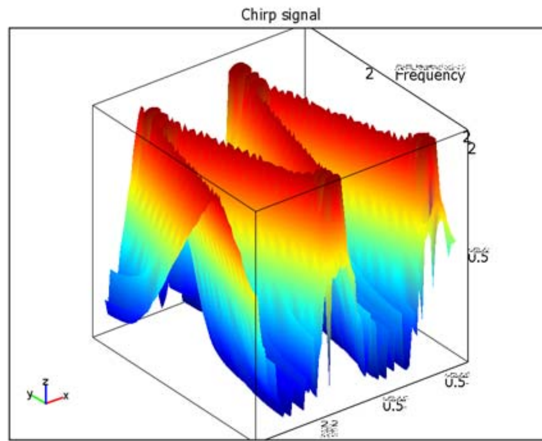
The next example illustrates aliasing in the signal. When the momentary frequency becomes larger than the Nyquist frequency, it is folded to a lower frequency—the alias effect.

```
y=getsignal('chirp',1024,0.16);
Yt=tfd(y);
plot(Yt)
```



Note that the Signals & Systems Lab generates a plot by default when no output argument is provided in the call. An `image` plot is given by default, but `contour`, `mesh`, and `surf` plots are also possible.

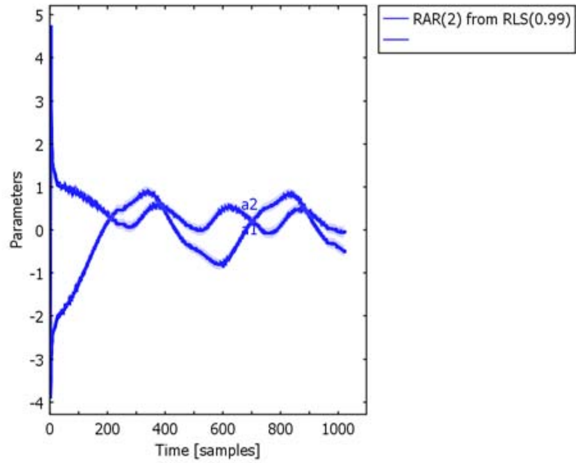
```
surf(Yt)
```



PARAMETRIC (MODEL-BASED) APPROACHES

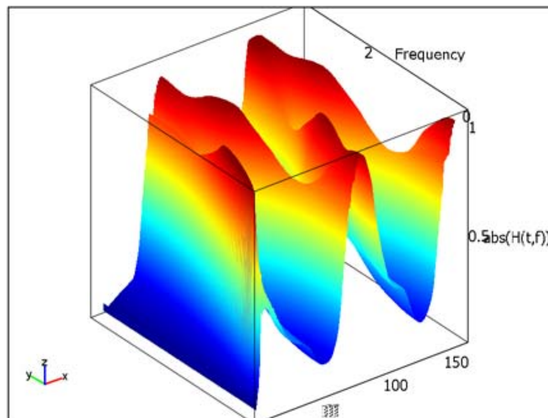
As an alternative to the short-time Fourier transform used in `sig2tfd`, you can estimate a time-varying model, which the following section describes, and convert this to the transform domain. Momentaneously, there is only one frequency, so an AR(2) model would do in the previous example.

```
m=estimate(rarx(2),y,'adg',0.99);
plot(m)
```



Conversion to a TFD object is done automatically by the TFD constructor (which calls the `rarx2tfd` method in the RARX object).

```
Ytm=tfd(m);
surf(Ytm)
```



Adaptive Filtering

Introduction

In the same way as TFD is the generalization of spectral analysis for nonstationary signals, adaptive filtering is the generalization of model estimation. The LTI object for time-varying systems is extended to a linear time-varying (LTV) object. It consists of a model structure and a time-varying parameter vector with its associated time-varying covariance matrix.

Basic Models

The common framework for the most well-known adaptive algorithms is provided by a time-varying linear regression,

$$y(t) = \varphi^T(t)\theta(t) + e(t).$$

You can express a time-varying AR(n) model as a time-varying linear regression in the following way:

$$\begin{aligned} y(t) + a_1(t)y(t-1) + \ldots + a_n(t)y(t-n) &= e(t), \\ \varphi(t) &= (-y(t-1) - \ldots - y(t-n))^T, \\ \theta(t) &= (a_1(t)\ldots a_n(t))^T. \end{aligned}$$

Similarly, a time-varying ARX model is given by

$$\begin{aligned} A(q,t)y(t) &= B(q,t)u(t) + e(t), \\ y(t) + a_1(t)y(t-1) + \ldots + a_n(t)y(t-n) &= b_1(t)u(t-1) + \ldots + b_m(t)u(t-m) + e(t). \end{aligned}$$

where the regression vector and parameter vector, respectively, are given by

$$\begin{aligned} \varphi(t) &= (-y(t-1) - \ldots - y(t-n), u(t-1)\ldots u(t-m))^T, \\ \theta(t) &= (a_1(t)\ldots a_n(t), b_1(t)\ldots b_m(t))^T. \end{aligned}$$

Other model structures as ARMA and ARMAX can be written as so-called pseudo-linear regressions

$$y(t) = \varphi^T(t, \theta(t))\theta(t) + e(t).$$

The difference is that the parameter vector is needed to compute the elements in the regressor, which you need to estimate the parameters. Nevertheless, this is a general and powerful approach to adaptive filtering. The Signals & Systems Lab supports ARX models, and the special cases of AR and FIR, but not ARMAX-type models.

The RARX Object

The output from the adaptive filters is a RARX object. The most important fields are the following:

- `th` contains the parameter vectors $\theta(t)$ stacked as rows in a matrix. The general convention for time-varying quantities is that time is always the first dimension.
- `P` is a three-dimensional matrix that contains the covariance matrices $P(t)$. Time is again the first dimension, so `P(t,:,:)` returns the covariance matrix at time t. Alternatively, when Monte Carlo realizations of the data are provided, the covariance matrix is replaced by `thMC` which is an (`MC,t,na+nb`) matrix.
- `lambda` is an estimate of time-varying noise variance.

Basic Algorithms

The `estimate` method supports five adaptive algorithms as described in the following sections.

LMS AND NLMS

The *least mean square* (LMS) algorithm minimizes

$$V(\theta) = E[y(t) - \varphi^T(t)\theta]^2$$

with a stochastic gradient method. This gives the following adaptive algorithm:

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)(y(t) - \varphi^T(t)\hat{\theta}(t-1)),$$

$$K_{\text{LMS}}(t) = \mu\varphi(t),$$

$$K_{\text{NLMS}}(t) = \mu\frac{\varphi(t)}{\alpha + |\varphi(t)|^2}.$$

The last equation gives the normalized version, NLMS, which is less sensitive to variations in the amplitudes in the regression vector.

WLS

Windowed least squares (WLS) applies a sliding window of size L to the data stream, and the least-squares (LS) estimate is computed at each time instant for all data inside the window, leading to the algorithm

$$\hat{\theta}(t) = \left[\sum_{t}^{k = t-L+1} \varphi(k)\varphi^T(k) \right]^{-1} \sum_{t}^{k = t-L+1} \varphi(k)y(k) .$$

The first factor multiplied by the measurement noise variance gives the covariance matrix of the estimate. A more efficient algorithm is used that updates the involved sums with new data and *downdates* (the reverse operation to updating, where measurement information is removed from an estimate) the same sums with the outgoing data, recursively.

RLS

Recursive least squares (RLS) is also based on the least-squares principle applying an exponential infinite data window rather than a finite one. Minimizing

$$V_t(\theta) = \sum_{k=1}^{t} \lambda^{t-k}(y(k) - \varphi^T(k)\theta)^2 .$$

gives the adaptive algorithm

$$\begin{aligned} \hat{\theta}(t) &= \hat{\theta}(t-1) + K(t)[y(t) - \varphi^T(t)\hat{\theta}(t-1)], \\ K(t) &= P(t)\varphi(t), \\ P(t) &= \left[P(t-1) - \frac{P(t-1)\varphi(t)\varphi^T(t)P(t-1)}{\lambda + \varphi^T(t)P(t-1)\varphi(t)} \right] / \lambda. \end{aligned}$$

The auxiliary quantity $P(t)$ can be identified as the covariance matrix.

KF

The *Kalman filter* (KF) is algorithmically closely related to RLS. It is based on a random walk model of the parameter vector, which gives a special case of a state-space model:

$$\begin{aligned} \theta(t+1) &= \theta(t) + w(t), \\ y(t) &= \varphi^T(t)\theta(t) + e(t). \end{aligned}$$

The Kalman filter applied to this model yields the algorithm:

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)[y(t) - \varphi^T(t)\hat{\theta}(t-1)],$$

$$K(t) = \frac{P(t-1)\varphi(t)}{R(t) + \varphi^T(t)P(t-1)\varphi(t)},$$

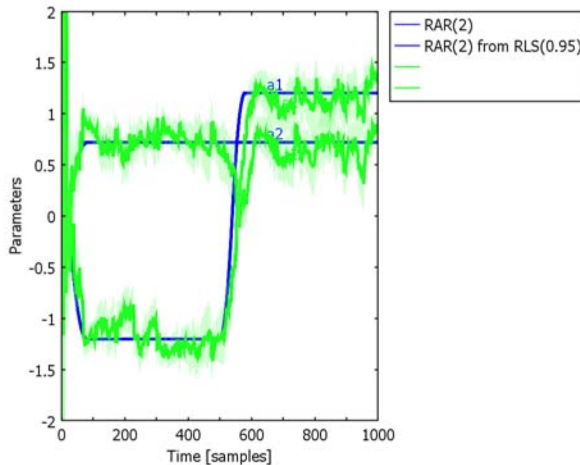
$$P(t) = P(t-1) - \frac{P(t-1)\varphi(t)\varphi^T(t)P(t-1)}{R(t) + \varphi^T(t)P(t-1)\varphi(t)} + Q(t).$$

Here, $Q(t)$ is the covariance matrix of the random walk, and this is the main design parameter to trade off tracking speed to noise suppression. A large $Q(t)$ gives fast tracking and poor variance properties.

Tutorial

This section describes adaptive filtering using a time-varying AR(2) model obtained from the `exrarx` function:

```
m0=exrarx('rar2',1000);
z=simulate(m0);
mhat=estimate(rarx(2),z,'adg',0.95)
Uncertain RAR(2) from RLS(0.95) estimated by RLS(0.95)
plot(m0,mhat,'Ylim',[-2 2])
```

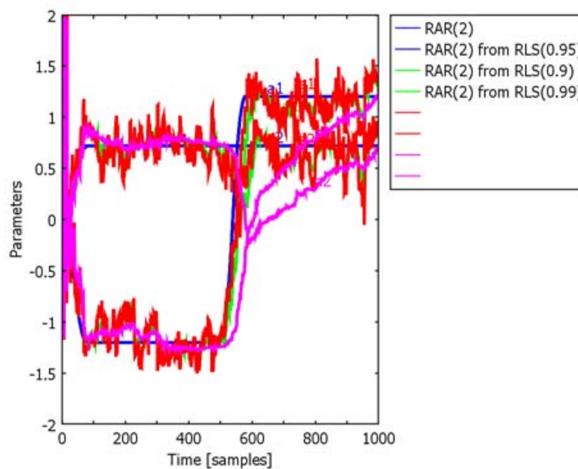


The plot shows a comparison of the estimated parameters, displayed using a shaded confidence interval, with the simulated parameters.

The RLS (default) algorithm with forgetting factor (adaptation gain) 0.95 is applied to the time-varying AR(2) model of length $N = 1000$. The true simulated system is compared to the estimated one using `plot`. By default, the plotting method adds a 90% confidence interval to estimated LTV objects.

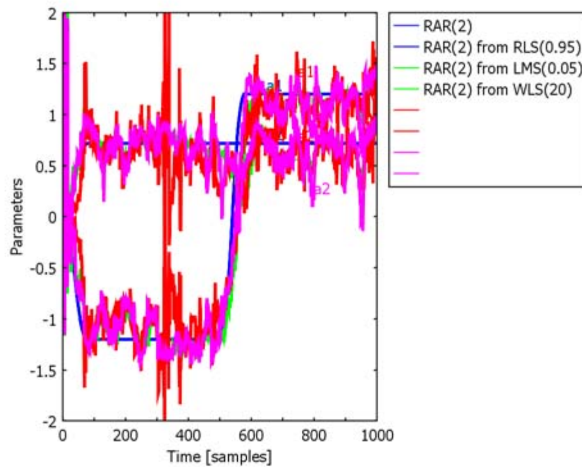
Tuning the tradeoff between estimation accuracy (small confidence interval) and tracking speed, the following lines of code evaluate different adaptation gains:

```
mhat1=estimate(rarx(2),z,'adg',0.9)
Uncertain RAR(2) from RLS(0.9) estimated by RLS(0.9)
mhat2=estimate(rarx(2),z,'adg',0.99)
Uncertain RAR(2) from RLS(0.99) estimated by RLS(0.99)
plot(m0,mhat,mhat1,mhat2,'conf',0,'Ylim',[-2 2])
```



Notice that one filter is tuned too fast (excessive estimation errors) and one too slow (time delay after changes). Other possible algorithms are LMS (or its normalized version) with step size μ and WLS (least squares over sliding window) with window size L. To get comparable algorithms, the rule of thumb is to relate $\mu = 1 - \lambda = 1/L$. The results are comparable.

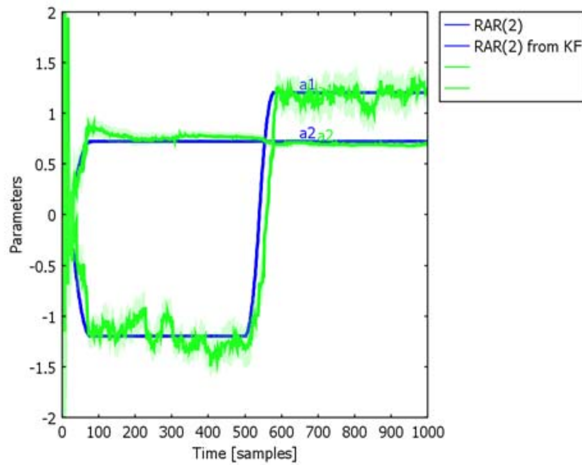
```
mhat3=estimate(rarx(2),z,'adg',0.05,'adm','lms');
mhat4=estimate(rarx(2),z,'adg',20,'adm','wls');
plot(m0,mhat,mhat3,mhat4,'conf',0,'Ylim',[-2 2])
```



The performances are comparable, but the LMS is close to the stability boundary, which a spike in the estimation error reveals.

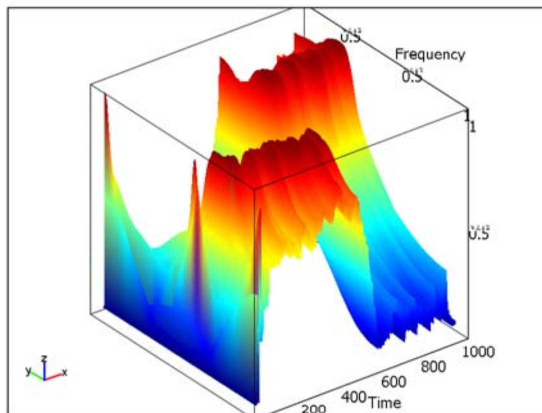
The Kalman filter adds another degree of freedom in that you can use different adaptation gains for different parameters. In this case, where the pole angle but not the radius changes, one parameter is constant. That is, you can set the adaptation gain for a_2 rather small.

```
mhat5=estimate(rarx(2),z,'adg',[1e-3 0;0 1e-6],'adm','kf');
plot(m0,mhat5,'conf',90,'Ylim',[-2 2])
```



Note that the AR part of ARX models can be transformed to a TFD. This is in parallel to model-based spectral analysis, where ARMA models provide intermediate signal representations when analyzing the spectral properties of a signal. The following line computes a parametric version of `sig2tfd`.

```
surf(tfd(mhat));
```



INDEX

A adaptive filtering 13, 267
aircraft pitch control 119
aliasing 264
analysis window 25
appending
  stochastic LTI objects 138
  systems 100
AR model 9
ARMA model 199
ARX models 196, 206
  converting 203
  for MIMO systems 201
  operations on 203
autonomous underwater vehicle dynamics 222
autoregressive model 9
AUV dynamics 222

B balanced realizations 105
  for model approximations 106
beta distribution 238
Blackman-Tukey's method 69, 73
Bode plots 110
  of amplitude or phase only 111
bouncing ball example 145, 219
Butterworth filter 41
Butterworth filters 62

C cancellations of poles and zeros 99
central limit theorem 245
Chebyshev filters of type I 62
chemical engineering examples 226
chirp signal 262
closed-loop system 12, 98, 124
complex conjugate of transfer functions 100
concatenating systems 101
confidence bounds 27
continuous-time signals 35, 50
control
  of aircraft pitch 119
controllability Gramian 80, 105
controllability matrix 105
controller design 122
converting
  ARX models 203
  LTI models 101
  signals 44
  state-space to transfer function 89
covariance functions 24, 79
covariance matrix 198
Cramer Rao lower bound 177
cumulative distribution functions 238

D data preprocessing 38
database of signals 44
DC motor model 103
decimation 39
designing filters 62
direct form IIR structure 60
discrete-time Fourier transform 7
discrete-time signals 33, 48
discrete-time systems 93
discretization 102
distributions
  beta 238
  defining new 249
  exponential 238
  F 238
  gamma 238
  Gaussian 238
  normal 238
  Student's t 238
  uniform 238

downdating 269
DTFT 7

E EEG signal 13, 58
EKF-SLAM 193
embedded Monte Carlo simulations 26
ensemble uncertainty 26
error function 239
estimated models 131
estimation 136
estimation procedure 66
exponential distribution 238
extended Kalman filter 152, 157

F fast Fourier transform 7
F-distribution 238
feedback
  for stochastic LTI objects 138
feedback control problem 130
feedback system 98
FFT 7
filter design 60
filter function 109
filtering 39, 60
  zero-phase 64
filters
  designing 62
FIR models 206
Fourier analysis 54
Fourier transform 22
Fourier Transform objects 22, 52
frequency analysis 5
frequency convention 23
frequency function 24
FT objects 22, 52
fusion 251

G gamma distribution 238
Gauss approximation formula 28
Gaussian distribution 238
genera 5
Gramians 105
gray-box identification 210

H Hamming window 41
harmonics 53
high-order FIR model 206
Hz frequency convention 23

I ideal filters 60
idempotent matrix 103
interpolation 39
inversion of square systems 99
investment model 127
isothermal plug flow reactor 226

K Kalman filters 152, 269
kernel function 25
kurtosis 239

L labels for plots 31
Laplace operator 10
Laplace transform 88
LaTeX
  documents 95
  formatting description using 119
leakage phenomenon 55
least mean square algorithm 268
least-squares estimations 136
least-squares method 197
linear regression 197
linear time-invariant models 9, 85
linear time-varying models 85
linear time-varying signals 24
localization applications 18
low-order ARX model 206
low-pass filtering 65
LTI models 9, 85
  converting between 101
  converting to other objects 108
LTI objects 23

LTI systems
connecting in parallel 96
connecting in series 97
feedback connection 98
viewing 110
LTV signals 24

M mapping 189
markers, user-defined for signals 31
matrix exponential 103
mean value 239
measurement noise 137
MIMO ARX models 201
MIMO systems 85, 91
concatenating systems to create 101
minimal realization 99
minimum-phase filter 66
model approximations 106
model estimation
ARMA 199
least squares 197
model truncations 105
model uncertainty 27
model validation 198
model-based signal analysis 13
moments 239, 242
Monte Carlo data 36
Monte Carlo principle 28
Monte Carlo samples 131
Monte Carlo simulations 26, 75, 82
multivariate distributions 19, 246
multivariate signals 33, 74

N Newton-Raphson method for NLS 211
NL models 209
NLS algorithm 209
noncausal filters 60, 66
noncausal Wiener filters 67
nonisothermal plug flow reactor 229
nonlinear least-squares algorithm 209
nonlinear transformation-based filters 171
nonuniformly sampled data 6
normal distribution 238
Nyquist curves 110, 111
Nyquist frequency 264

O observability Gramian 105
observability matrix 105
open-loop systems 12
output 208
output-error model 208

P parallel connection 96
of stochastic LTI objects 137
parameter estimation 209, 244
Parseval's formula 24
particle filter 152, 163
P-controller 123
PDF objects 238
PDFCLASS objects 36
PDFs 25
periodic functions 49
periodic signals, harmonics in 53
periodogram 69
pole-zero cancellations 99
pole-zero plots 110
positioning problem 18
preprocessing of data 38
prewindowing 39
probability density functions 25, 238
for defining uncertainty 131
process noise 137
pseudo-random binary sequence 49

R random number generation 239
random walk model 269
RARX objects 268
recursive least squares 269
resampling 39

Riccati equations 105
right inverse 100
robustness analysis 131
root locus 12
root locus plots 110, 112, 123
Rosenbrock's test example 214

S sampling, for simulations 109
scatter plot 134
sensor networks 181
series connection 97
 of stochastic LTI objects 138
short-time Fourier transform 14, 262
SIG objects 30
signal objects 30
signal preprocessing 5
signals
 converting 44
 database of 44
 low-frequency contents of 39
simulations 109
simultaneous mapping and localization 192
sinusoidal components 54
skewness 239
SLAM 192
smoothing 69
Spectool user interface 70
spectral analysis 8, 69
square wave, low-pass filtering of 65
state estimation 152
states
 removing for minimal realizations 107
state-space models 89, 119
 for discrete-time systems 93
 MIMO systems 91
 on innovation form 204
 printouts of 90
 stochastic 137
state-space realization 79
statistics 18
stem plots 22
step responses 12
stochastic LTI objects
 appending 138
 connecting in parallel 137
 connecting in series 138
 feedback connection of 138
stochastic signals 30, 36
stochastic state-space objects 137
stochastic systems 137
Student's t distribution 238
system analysis tools 105

T target tracking 186
TeX code, generating 94
TF models 206
TFTool user interface 61
time-frequency descriptions 13, 262
time-varying AR model 270
time-varying model 265
tracking 186
transfer functions
 complex conjugate of 100
transfer-function representation 10
transpose operation 100

U uncertain systems 131
uniform distribution 238
unscented Kalman filter 152

V validation of models 198
van der Pol equations 143
variance 239

W Welch method 69, 73
white noise 24
 filtered 69
Wiener filtering 67
windowed least-squares method 269

windowing 7
windows 39

Z zero-padding 55
zero-phase filtering 64
zero-pole plots 112