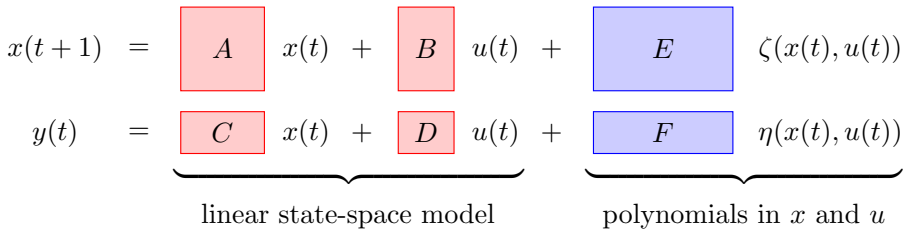
PNLSS 1.0

A polynomial nonlinear state-space toolbox for Matlab®

Documentation

written by Koen Tiels
Vrije Universiteit Brussel
Department ELEC



Many members of the nonlinear system identification team of the ELEC Department have contributed to this software.
The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC Grant Agreement n. 320378.

VRIJE UNIVERSITEIT BRUSSEL

erc

PNLSS 1.0

Documentation

http://homepages.vub.ac.be/~ktiels/pnlss.html

Koen Tiels
Vrije Universiteit Brussel
Department ELEC

September 12, 2016

Contents

1 License

Permission is granted to freely use the code in this package for non-commercial applications only. The code is provided “as is” without any warranty. The rest of this section presents the full license.

1.1 Full license

PNLSS 1.0
Software License Agreement

This software, PNLSS 1.0 (the "Program"), is being licensed.

Conditions (liability, etc.) are listed below.

GENERAL TERMS

LICENSE GRANT. Vrije Universiteit Brussel - dept. ELEC grants to Licensee a nonexclusive license to freely use the Program for non-commercial applications only.

Conditions

LIMITED WARRANTY/LIMITATION OF REMEDIES. EXCEPT AS EXPRESSLY PROVIDED BY THIS AGREEMENT (OR AS IMPLIED BY LAW WHERE THE LAW PROVIDES THAT THE PARTICULAR TERMS IMPLIED CANNOT BE EXCLUDED BY CONTRACT), ALL OTHER CONDITIONS, WARRANTIES, OR OTHER TERMS (INCLUDING ANY WITH REGARD TO INFRINGEMENT, MERCHANTABLE QUALITY, OR FITNESS FOR PURPOSE) ARE EXCLUDED. SOME STATES DO NOT ALLOW LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY LASTS, SO THE ABOVE LIMITATION MAY NOT APPLY TO LICENSEE. THIS WARRANTY GIVES LICENSEE SPECIFIC LEGAL RIGHTS AND LICENSEE MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE. LICENSEE ACCEPTS RESPONSIBILITY FOR ITS USE OF THE PROGRAM AND THE RESULTS OBTAINED THEREFROM.

LIMITATION OF LIABILITY. THE PROGRAM SHOULD NOT BE RELIED ON AS THE SOLE BASIS TO SOLVE A PROBLEM WHOSE INCORRECT SOLUTION COULD RESULT IN INJURY TO PERSON OR PROPERTY. IF A PROGRAM IS EMPLOYED IN SUCH A MANNER, IT IS AT THE LICENSEE’S OWN RISK AND VUB EXPLICITLY DISCLAIMS ALL LIABILITY FOR SUCH MISUSE TO THE EXTENT ALLOWED BY LAW.

VUB'S LIABILITY FOR DEATH OR PERSONAL INJURY RESULTING FROM NEGLIGENCE OR FOR ANY OTHER MATTER IN RELATION TO WHICH LIABILITY BY LAW CANNOT BE EXCLUDED OR LIMITED SHALL NOT BE EXCLUDED OR LIMITED. EXCEPT AS AFORESAID, VUB SHALL HAVE NO LIABILITY FOR ANY INDIRECT OR CONSEQUENTIAL LOSS (WHETHER FORESEEABLE OR OTHERWISE AND INCLUDING LOSS OF PROFITS, LOSS OF BUSINESS, LOSS OF OPPORTUNITY, AND LOSS OF USE OF ANY COMPUTER HARDWARE OR SOFTWARE). SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE EXCLUSION OR LIMITATION MAY NOT APPLY TO LICENSEE.

2 Getting started

2.1 What is PNLSS 1.0?

PNLSS 1.0 is a Matlab® toolbox to identify polynomial nonlinear state-space (PNLSS) models from measured data.

This documentation explains how to use the functions in the package by means of a tutorial in Section 3. The tutorial assumes that the reader is somewhat familiar with system identification and PNLSS modeling. Nevertheless, a brief explanation of PNLSS modeling is provided in Section 4. More comprehensive information on system identification can be found in e.g. Söderström & Stoica [1989]; Ljung [1999]; Pintelon & Schoukens [2012]. More information on PNLSS modeling in particular can be found in [Paduart, Lauwers, Swevers, Smolders, Schoukens & Pintelon, 2010]. The code in this toolbox is based on the code that Johan Paduart developed during his PhD [Paduart, 2008].

2.2 Installation

1. Unzip PNLSS_v1_0.zip in a folder of your choice. This folder will be referred to as the PNLSS folder in the rest of this documentation.
2. Add this folder to the Matlab search path using the `addpath` function.
3. Save the search path using the `savepath` function.

Alternatively, you can use `pathtool` to add the PNLSS folder and save the path.

2.3 Requirements

The software was tested on several 64-bit machines running Microsoft® Windows 7 Professional. One of these computers runs Matlab R2015a, while the others run Matlab R2014a.

The software is expected to run on some earlier versions of Matlab as well, however, the function `rms` may not be available. If so, just add a function `rms` in the PNLSS folder with the following content:

```
function rms_u = rms(u)
    rms_u = sqrt(mean(u.^2));
end
```

Matlab R2009b or higher is required because of the use of the tilde operator.

2.4 Package content

The software package consists of a number of Matlab functions (in m-files starting from a lowercase f), a number of scripts (in m-files starting from a lowercase s), a data set (dTutorialSISO.mat), the license (license_pnlss_v1_0.txt), and a contents file (Contents.m).

The contents file provides an overview of the functionality of the Matlab-files in the package. You can display the content file in the command window by browsing to the PNLSS folder and by running `help(pwd)` in the command window. This displays the content file with links to the help text of all the files. Most of the help texts contain one or more examples of how to use the particular function.

A function reference is available in Appendix A of this document.

3 Tutorial

Looking at a simple example is a good way to quickly learn the basics of a software toolbox. Therefore, let us start with a tutorial that explains the main work flow of estimating a PNLSS model from measured data with the toolbox. If you are not yet familiar with PNLSS modeling, feel free to have a look first at Section 4.

In this tutorial, you will learn how to

1. generate a multisine excitation signal,
2. compute the response of a PNLSS model to the excitation signal,
3. estimate a nonparametric linear model (i.e. the frequency response function (FRF)), the noise and nonlinear distortion levels,
4. estimate a parametric linear state-space model, and
5. estimate the parameters of the full PNLSS model.

The measured data are generated from an example PNLSS model. A multisine excitation signal is generated and the response of the PNLSS model to the multisine excitation is calculated. This response is then corrupted with noise. The goal is to estimate a PNLSS model that captures the input/output behavior of the original model. The PNLSS model is estimated from the input and noisy output data.

The code for this tutorial is available in the sTutorialSISO.m script. In the following sections, we explain this code step by step.

3.1 Load data-generating model

Let us start by loading the example PNLSS model that will be used to generate the data:

```
%% Load a PNLSS model (will be used to generate data for this
    example code)

load dTutorialSISO.mat model
true_model = model;
clear model % Will be estimated
```

The result is a structure `true_model` that is the description of a single input single output (SISO) second-order PNLSS model. This is indicated by the fields `m` (number of inputs), `p` (number of outputs) and `n` (model order).

The linear state-space matrices are given in the fields `A`, `B`, `C`, and `D`.

The nonlinear terms in both the state and output equation are all possible quadratic and cubic monomials (`nx = [2 3]`, `ny = [2 3]`) in the states and input. There are 16 such monomials, as indicated by the fields `n_nx` and `n_ny`. These monomials are represented by the matrices `xpowers` and `ypowers`. The corresponding monomial coefficients are in the matrices `E` and `F`. Each row in `xpowers` (and `ypowers`) corresponds to a particular monomial and each column corresponds to a particular state (first two columns) or input (last column). The values in the matrices `xpowers` and `ypowers` indicate to which power a particular state or input is raised in a particular monomial. For example, the first row in `xpowers` is `[2 0 0]` and indicates that the first monomial is $x_1^2 x_2^0 u^0 = x_1^2$.

These are all the fields that indicate the structure of the model and its parameterization. The other fields will become clear in the remainder of this tutorial. A description of all the fields in a PNLSS model can be found in the help text of the `fCreateNLSSmodel` function.

3.2 Generate multisine excitation

Toolbox function used in this section: `fMultisine`

In order to identify the model from data, we first need to generate an input signal with which we will excite the system. For the excitation signal, we choose a multisine.

A random-phase multisine signal $u(t)$ consists of a sum of harmonically related sine waves:

$$u(t) = \sum_{k=1}^{\lfloor N/2 \rfloor} A_k \sin\left(2\pi k \frac{f_s}{N} t + \phi_k\right),$$

where N is the number of samples, f_s is the sampling frequency, and A_k and ϕ_k are the amplitude and the phase of the sine wave with frequency $k\frac{f_s}{N}$. The phases are drawn from a uniform distribution in the interval from zero to 2π.

The code snippet below generates $R = 4$ phase realizations and $P = 3$ periods of an odd (only odd frequencies excited) random-phase multisine with $N = 1024$ samples. The multisine has a flat spectrum up to 90% of the Nyquist frequency. The multisine is scaled to an rms value of 0.05.

```
38 %% Generate multisine input (u)
39
40 RMSu = 0.05; % Root mean square value for the input signal
41 N = 1024;    % Number of samples
42 R = 4;       % Number of phase realizations (one for validation and
       one for performance testing)
43 P = 3;       % Number of periods
44 kind = 'Odd';           % 'Full','Odd','SpecialOdd', or 'RandomOdd
       ': kind of multisine
45 M = round(0.9*N/2);     % Last excited line
46 [u,lines,non_exc_odd,non_exc_even] = fMultisine(N, kind, M, R); %
       Multisine signal, excited and detection lines
47 u = u/rms(u(:,1))*RMSu; % Scale multisine to the correct rms level
48 u = repmat(u,[1 1 P]);  % N x R x P
49 u = permute(u,[1 3 2]); % N x P x R
```

3.3 Calculate response to excitation

Toolbox function used in this section: `fFilterNLSS`

Now that we have generated our excitation signals, we can apply them to the system, and measure its (noisy) response. In this case, our system is a PNLSS model. This allows us to illustrate how to compute the output of a given PNLSS model to a given input using the `fFilterNLSS` function:

```
51 %% Generate output (y)
52
53 % Calculate output of PNLSS system
54 true_model.T1 = [N 1+(0:P*N:(R-1)*P*N)]; % To generate steady state
       data
55 y = fFilterNLSS(true_model,u(:));    % Supposed to be the true
       system output (simulation)
```

Note that the `fFilterNLSS` function in line 55 takes the concatenated input vector `u(:)` as an argument. Hence, when filtering, there will be undesirable transient effects when transitioning from one phase realization to another. To make sure that we work with steady-state data, we add one extra period before the start of each phase realization. This is done with the transient parameter `T1`. The first element of `T1` indicates the number of extra samples to include before each transition. These extra samples will be removed in the post-processing of the data. In our case, this is one period, so N samples. The remaining elements

of `T1` are the starting indices of each phase realization in the concatenated vector `u(:)`.

Once the noise-free output is calculated, a small filtered Gaussian noise is added to it:

```
57 % Add colored noise to the output
58 y = reshape(y,[N P R]); % N x P x R
59 noise = 1e-3*std(y(:,end,end))*randn(size(y)); % Output noise
       signal
60 noise(1:end-1,:,:) = noise(1:end-1,:,:) + noise(2:end,:,:); % Do
       some filtering
61 y = y + noise; % Noise added to the output
```

3.4 Separate data in estimation, validation, and test sets

We will now separate the simulated data in three sets. The estimation data (first two realizations) will be used to estimate the model. The validation data (last period of the third realization) will be used to do model selection. The test data (last period of the fourth realization) will be used to test the performance of the estimated model.

```
63 %% Separate the data in estimation, validation, and test set
64
65 % Last realization, last period for performance testing
66 utest = u(:,end,R); utest = utest(:);
67 ytest = y(:,end,R); ytest = ytest(:);
68
69 % One but last realization, last period for validation and model
       selection
70 uval = u(:,end,R-1); uval = uval(:);
71 yval = y(:,end,R-1); yval = yval(:);
72
73 % All other realizations for estimation
74 R = R-2;
75 u = u(:,:,1:R);
76 y = y(:,:,1:R);
```

3.5 Estimate nonparametric linear model

Toolbox functions used in this section: `fCovarY` and `fCovarFrf`

We are now ready to start the estimation process. The first step is the estimation of a nonparametric linear model (a frequency response function (FRF)). For a periodic input, this can be done with the `fCovarFrf` function. It is an implementation of the robust method Pintelon & Schoukens [2012] that not only

provides the FRF `G`, but also the total (= noise + nonlinear) distortion level `covGML` and the noise distortion level `covGn` on this FRF.

The noise covariance matrix on the output spectrum is also calculated (with the function `fCovarY`) as it will be used as a weighting for the nonlinear optimization in Section 3.7.

```
78 %% Estimate nonparametric linear model (BLA)
79
80 u = permute(u,[1,4,3,2]); % N x m x R x P
81 y = permute(y,[1,4,3,2]); % N x p x R x P
82 covY = fCovarY(y); % Noise covariance (frequency domain)
83
84 U = fft(u); U = U(lines,:,:,:); % Input spectrum at excited lines
85 Y = fft(y); Y = Y(lines,:,:,:); % Output spectrum at excited lines
86
87 % Estimate best linear approximation, total distortion, and noise
       distortion
88 [G,covGML,covGn] = fCovarFrf(U,Y); % G: FRF; covGML: noise + NL;
       covGn: noise (all only on excited lines)
```

3.6 Estimate linear state-space model

Toolbox function used in this section: `fLoopSubSpace`

On top of the nonparametric linear model, a parametric model will be estimated. In particular, a linear state-space model will be estimated using a frequency domain subspace method McKelvey, Akçay & Ljung [1996]; Pintelon [2002]. The subspace method provides a good initial estimate of the linear model parameters `A`, `B`, `C`, and `D`, but to improve on those estimates, a Levenberg-Marquardt optimization is done on these parameters.

The subspace method and the Levenberg-Marquardt optimization are combined in the `fLoopSubSpace` function. The total distortion level `covGML` that was calculated in the previous section is used as a frequency weighting in the optimization. In this case, only one model order is selected (`na = 2`), but the code from line 100 onwards indicates how to proceed if you want to scan over a range of model orders. In particular, the model that performs best on the validation data in terms of rms output error is selected. The variable `maxr` is the maximum value of a subspace dimensioning parameter. Note that only model orders that are strictly smaller than `maxr` are scanned. The last argument in the `fLoopSubSpace` function indicates that at most 100 iterations of the Levenberg-Marquardt algorithm will be performed. The estimated state-space matrices of model order `n` are collected in the cell `models{n}`.

```
90 %% Estimate linear state-space model (frequency domain subspace)
91
92 % Choose model order
```

```
na   = 2;  % Model order
maxr = 10; % Subspace dimensioning parameter
freq = (lines-1)/N; % Excited frequencies (normalized)

% covGML = repmat(eye(1),[1 1 length(lines)]); % Uncomment for
    uniform weighting (= no weighting)
models = fLoopSubSpace(freq,G,covGML,na,maxr,100); % All estimated
    subspace models

% Extract linear state-space matrices from best model on validation
     data
Nval = length(uval); % Number of samples in validation data
fs = 1; % Sampling frequency
tval = (0:Nval-1)/fs; % Time vector validation data
min_err = Inf; % Initialize minimum error
min_na = NaN; % Initialize model order of best model
for n = na % Loop over model order(s)
    model = models{n}; % Select subspace model of the correct order
    A = model{1}; B = model{2}; C = model{3}; D = model{4}; %
        Extract state-space matrices
    [A,B,C] = dbalreal(A,B,C); % Compute balanced realization
    yval_hat = lsim(ss(A,B,C,D,1/fs),uval,tval); % Modeled output
    err = yval - yval_hat; % Error signal
    err = sqrt(mean(err(end-N+1:end).^2)); % Rms value of the last
        period of the error signal
    if err < min_err % If the model is the best so far
        min_na = n; % Update model order of the best model
        min_err = err; % Update minimum error
    end
end
model = models{min_na}; % Select the best model
[A,B,C,D] = model{:}; % Extract all the matrices of the best model
[A,B,C] = dbalreal(A,B,C); % Balanced realization
```

3.7 Estimate PNLSS model

Toolbox functions used in this section: `fSelectActive`, `fCreateNLSSmodel`, `fFilterNLSS`, `fSqrtInverse`, and `fLMnlssWeighted`

Next, the full nonlinear model is estimated. A Levenberg-Marquardt optimization is done on all model parameters starting with the linear state-space model as initialization.

First, the estimation data is averaged out over the periods:

```
%% Estimate PNLSS model

```

```
% Estimation data
u = mean(u,4); % Average over periods
y = mean(y,4); % Average over periods (be careful that the data are
     truly steady state)
m = 1; % Number of inputs
p = 1; % Number of outputs
u = u(:); % Concatenate the data: N*P*R x m
y = y(:); % Concatenate the data: N*P*R x p
```

Like this, the effect of the output noise is averaged out, and a more compact data set is obtained. Just as in Section 3.3, the data is concatenated, which will again lead to undesirable transient effects. By appropriately setting the transient parameters `T1` (for periodic data, as is the case here) and `T2` (for non-periodic data), we can again make sure that the modeled output is computed in steady-state. This steady-state modeled output is used during the optimization to evaluate the cost function. Note that due to the averaging over the periods, we essentially have only one (averaged) period, hence the difference in the parameter `T1` between lines 134 and 54.

```
% Transient settings
NTrans = N; % Add one period before the start of each realization
T1 = [NTrans 1+(0:N:(R-1)*N)]; % Number of transient samples and
    starting indices of each realization
T2 = 0; % No non-periodic transient handling
```

The nonlinear degrees in the state and output equation are chosen to be `nx = [2 3]`, `ny = [2 3]`, in correspondence to the quadratic and cubic monomials in the true system. All the coefficients in the matrices `E` and `F` corresponding to these monomials are set free for optimization:

```
% Nonlinear terms
nx = [2 3]; % Nonlinear degrees in state update equation
ny = [2 3]; % Nonlinear degrees in output equation
whichtermsx = 'full'; % Consider all monomials in the state update
    equation
whichtermsy = 'full'; % Consider all monomials in the output
    equation
```

The variables `whichtermsx` and `whichtermsy` are later on used in the function `fSelectActive` to apply these settings in the PNLSS model:

```
% Set which monomials will be optimized
model.xactive = fSelectActive(whichtermsx,n,m,n,nx); % Select
    active monomials in state equation
model.yactive = fSelectActive(whichtermsy,n,m,p,ny); % Select
    active monomials in output equation
```

Next, the number of Levenberg-Marquardt iterations and the starting value for the Levenberg-Marquardt damping factor λ are set:

```
143 % Settings Levenberg–Marquardt optimization
144 MaxCount = 100; % Number of Levenberg–Marquardt optimizations
145 lambda = 100; % Starting value Levenberg–Marquardt damping factor
```

If the optimization does not converge with the default value for λ, it is recommended to start with a large value for λ. The Levenberg-Marquardt algorithm then tends more to a gradient descent method, which is often more robust to an initial value far from the optimum than a Gauss-Newton method ($\lambda = 0$).

Next, the model order is set equal to the order of the best linear model, and this linear model is put in a PNLSS model form using the `fCreateNLSSmodel` function.

```
147 % Choose model order
148 n = min_na;
149
150 % Initial linear model in PNLSS form
151 model = fCreateNLSSmodel(A,B,C,D,nx,ny,T1,T2); % Initialize PNLSS
    model with linear state–space matrices
```

All the coefficients in the matrices `E` and `F` in this model are then set free for optimization in lines 153 till 155 as explained earlier.

The stability of a PNLSS model is input dependent due to the nonlinear state dependent monomials in the state equation. Occasionally, a PNLSS model that is stable on the estimation data turns out to be unstable on the validation or test data. To make sure that the model is stable on a particular data set, the input of this data set can be passed to the optimization routine. In each iteration, the output of the updated PNLSS model is calculated for this input `model.uval`. If the maximum absolute value of the output exceeds a specified bound `model.max_out`, then the updated PNLSS model is not accepted, and the optimization continues as if the iteration was unsuccessful.

```
157 % Protect for unstable models
158 model.uval = uval; % Optionally, a validation input signal can be
    passed for which the output of the estimated model should
    remain bounded
159 model.max_out = 1000*max(abs(yval)); % Bound on the output of the
    estimated model (if this bound is violated, the parameter
    update is considered as unsuccessful)
```

Now that the PNLSS model is still the linear model, we can easily calculate the linear model error on the estimation and test data with the `fFilterNLSS` function. Note that the transient parameter `T1` for the test data changes w.r.t. that for the estimation data. Since the test data has only one phase realization, adding one extra period in the test data would be achieved with `T1 = [NTrans 1]`, but the `1` can be removed as it will be added by default if `T1` has only one element.

```
% Compute linear model error
y_lin = fFilterNLSS(model,u); % Output of the initial linear model
    on the estimation data
modellintest = model; modellintest.T1 = NTrans; % Change transient
    parameter for linear model on test data
ytest_lin = fFilterNLSS(modellintest,utest); % Output of the
    initial linear model on the test data
err_lin = ytest - ytest_lin; % Output error initial linear model on
     test data
```

Next, the weighting in the cost function is set. Depending on the size of the weighting matrix `W`, frequency domain weighting (if `W` is a $p \times p \times \frac{N}{2}$ array), time domain weighting (if `W` is a $N \times p$ array), or no weighting (if `W = []`) is applied. We advice the user to start with no weighting due to the possible presence of dominant model errors. With the uniform weighting (= no weighting), these model errors are assumed to be uniformly spread over the frequencies. In this tutorial, the system and the model have the same structure (second-order SISO PNLSS model with quadratic and cubic monomials in the state and the output equation), so that the model should be able to predict the output of the system up to the filtered output noise. The estimated noise distortion on the output is used as a frequency weighting in this case.

```
% Set weighting
for kk = size(covY,3):-1:1
    W(:,:,kk) = fSqrtInverse(covY(:,:,kk)); % Frequency weighting
end;
%     W = []; % Uncomment for uniform (= no) weighting
```

Finally, the Levenberg-Marquardt optimization on all model parameters is done using the `fLMnlssWeighted` function.

```
% Levenberg-Marquardt optimization
[model, y_mod, models] = fLMnlssWeighted(u,y,model,MaxCount,W,
    lambda); % The actual optimisation of the PNLSS model
```

The variable `model` is the best model on the estimation data and `y_mod` is its modeled output. The models after each successful iteration are stored in `models`.

To get an idea of the quality of the estimated model, the model error of the linear and nonlinear model are compared on the estimation data. The nonlinear model should be able to reach the noise level, while the linear model can only reach the total distortion level.

```
% Time-domain plot on the estimation data
figure; plot([y y-y_lin y-y_mod]) % Estimation results: output,
    linear error and PNLSS error
xlabel('Time index')
```

```
ylabel('Output (errors)')
legend('output','linear error','PNLSS error')
title('Estimation results')
disp(' ')
disp(['rms(y-y_mod) = ' num2str(rms(y-y_mod)) ' (= Output error of the best PNLSS model on the estimation data)'])
disp(['rms(noise(:))/sqrt(P) = ' num2str(rms(noise(:))/sqrt(P)) ' (= Noise level)'])
```

Having access to the models after each successful iteration allows us to select the best model on the validation data. This is recommended to avoid overfitting. The plot shows the output errors of each of the models on the validation set. The selected model is marked in red. This plot shows how the model error evolves from that of the linear model to that of the final model and indicates whether or not the optimization has converged.

```
% Search best model over the optimisation path on a fresh set of data
valerrs = [];
for i = 1:length(models)
    models(i).T1 = NTrans; % Only one realization in the validation data
    yval_mod = fFilterNLSS(models(i),uval);  % Output model i on validation data
    valerr = yval - yval_mod; % Output error model i on validation data
    valerrs = [valerrs; rms(valerr)]; % Collect output errors of all models in a vector
end
figure;
plot(db(valerrs));
xlabel('Successful iteration number')
ylabel('Validation error [dB]')
title('Selection of the best model on a separate data set')
[min_valerr,i] = min(valerrs); % Select the best model on the validation data to avoid overfitting
hold on
plot(i,db(min_valerr),'r.','Markersize',10)
model = models(i);
```

3.8 Result on test data

Toolbox function used in this section: `fFilterNLSS`

Finally, the model error on the test data set is calculated. This time, the results are presented in a frequency domain plot.

```
%% Result on test data

% Compute output error on test data
valerr = ytest - fFilterNLSS(model,utest);

% Frequency-domain plot on the test data
figure;
fs = 1; % Just a normalised discrete-time simulation
freq = (0:N-1)/N*fs; % Normalized frequency vector
plottime = [ytest err_lin valerr]; % Test output and output errors
    linear and PNLSS model (in time domain)
plotfreq = fft(plottime); % Test output and output errors linear
    and PNLSS model (in frequency domain)
plot(freq(1:end/2),db(plotfreq(1:end/2,:)),'.')
xlabel('Frequency (normalised)')
ylabel('Output (errors) [dB]')
legend('Output','Linear error','PNLSS error')
title('Test results')
```

4 PNLSS modeling

This section provides a brief overview to PNLSS modeling and at the same time introduces the notation used in this document.

4.1 A PNLSS model

A polynomial nonlinear state-space model [Paduart et al., 2010] is a natural extension of a discrete-time linear state-space model. The state and output equation of the model are given by

$$\boldsymbol{x}(t+1) = \boldsymbol{A}\boldsymbol{x}(t) + \boldsymbol{B}\boldsymbol{u}(t) + \boldsymbol{E}\boldsymbol{\zeta}(\boldsymbol{x}(t), \boldsymbol{u}(t))$$

and

$$\boldsymbol{y}(t) = \boldsymbol{C}\boldsymbol{x}(t) + \boldsymbol{D}\boldsymbol{u}(t) + \boldsymbol{F}\boldsymbol{\eta}(\boldsymbol{x}(t), \boldsymbol{u}(t)),$$

respectively. Here, $\boldsymbol{u}(t) \in \mathrm{R}^m$ is the input vector at time t, $\boldsymbol{y}(t) \in \mathrm{R}^p$ is the corresponding output vector, and $\boldsymbol{x}(t) \in \mathrm{R}^n$ is the state vector. Matrix $\boldsymbol{A} \in \mathrm{R}^{n\times n}$ is the state matrix, $\boldsymbol{B} \in \mathrm{R}^{n\times m}$ is the input matrix, $\boldsymbol{C} \in \mathrm{R}^{p\times n}$ is the output matrix, and $\boldsymbol{D} \in \mathrm{R}^{p\times m}$ is the feed-through matrix. The multivariate functions $\boldsymbol{\zeta} : \mathrm{R}^{n+m} \to \mathrm{R}^{n_\zeta}$ and $\boldsymbol{\eta} : \mathrm{R}^{n+m} \to \mathrm{R}^{n_\eta}$ are monomials with degree larger than one. The matrices $\boldsymbol{E} \in \mathrm{R}^{n\times n_\zeta}$ and $\boldsymbol{F} \in \mathrm{R}^{p\times n_\eta}$ contain the corresponding monomial coefficients.

The parameters in this model are the elements of the matrices $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, $\boldsymbol{D}$, $\boldsymbol{E}$, and $\boldsymbol{F}$, as well as the initial conditions $\boldsymbol{x}_0 = \boldsymbol{x}(0)$ and $\boldsymbol{u}_0 = \boldsymbol{u}(0)$:

$$\boldsymbol{\theta}^T = \begin{bmatrix} \mathrm{vec}(\boldsymbol{A})^T & \mathrm{vec}(\boldsymbol{B})^T & \mathrm{vec}(\boldsymbol{C})^T & \mathrm{vec}(\boldsymbol{D})^T & \mathrm{vec}(\boldsymbol{E})^T & \mathrm{vec}(\boldsymbol{F})^T & \boldsymbol{x}_0^T & \boldsymbol{u}_0^T \end{bmatrix}.$$

4.2 Identifying a PNLSS model

The goal is to estimate a set of model parameters from measured input/output data such that the modeled output $\boldsymbol{y}(t)$ approximates the measured output $\boldsymbol{y}_m(t)$ as well as possible in (a weighted) mean-square sense. The cost function is either formulated in the time domain:

$$K_{\mathrm{time}}(\boldsymbol{\theta}) = \sum_{t=1}^{N} \left\| \boldsymbol{W}_{\mathrm{time}}(t) \star (\boldsymbol{y}(t, \boldsymbol{\theta}) - \boldsymbol{y}_m(t)) \right\|^2 \tag{1}$$

or in the frequency domain:

$$K_{\mathrm{freq}}(\boldsymbol{\theta}) = \sum_{k=1}^{N_F} \boldsymbol{\epsilon}^H(k, \boldsymbol{\theta}) \boldsymbol{W}_{\mathrm{freq}}(k) \boldsymbol{\epsilon}(k, \boldsymbol{\theta}), \tag{2}$$

where $\boldsymbol{W}_{\mathrm{time}} \in \mathrm{R}^{p \times N}$ is the weighting matrix in the time domain, $\star$ denotes the Hadamard product (i.e. the element-wise matrix product),

$$\boldsymbol{\epsilon}(k, \boldsymbol{\theta}) = \boldsymbol{Y}(k, \boldsymbol{\theta}) - \boldsymbol{Y}(k) \tag{3}$$

is the spectrum of the output error at frequency line k, N_F is the number of excited frequencies in the positive half of the frequency band, and $\boldsymbol{W}_{\mathrm{freq}}(k) \in \mathrm{C}^{p \times p}$ is a user-chosen frequency domain weighting matrix. Typically, $\boldsymbol{W}_{\mathrm{freq}}(k)$ is the inverse of the covariance of the output spectrum (if no model errors are present), but it can also be used to select the frequency band of interest. Note that the time-domain formulation offers less flexibility in case of multiple outputs, as it does not take into account a potential correlation between the errors of different outputs.

Note that transient effects should be taken into account when minimizing the cost function. Often, zero initial conditions (i.e. $\boldsymbol{x}(0) = \boldsymbol{0}$) are assumed, which may be wrong. Therefore, there is a mismatch between the modeled and the measured output. This mismatch decays to zero, but is preferably not neglected. Since the transient error decays to zero, the transient points can simply be discarded, however, some information can be lost by doing so. When working with periodic data, discarding the transient points does not lead to a loss of information. Alternatively, the initial state $\boldsymbol{x}_0$ (and initial input $\boldsymbol{u}_0$) can be estimated by leaving them free for optimization in the parameter vector $\boldsymbol{\theta}$.

Note moreover that the problem is nonlinear in the parameters. The model parameters will therefore be estimated using local nonlinear optimization, which requires good initial estimates. Here, these initial estimates are obtained from the best linear approximation (BLA) Pintelon & Schoukens [2012], which is the

linear model whose output approximates the system output best in mean-square sense. The BLA is first estimated nonparametrically (as a frequency response matrix (FRM)) and then parametrically using a frequency domain subspace method McKelvey et al. [1996]; Pintelon [2002]. The nonlinear optimization makes use of a Levenberg-Marquardt algorithm.

5 History and future work

5.1 History

The PNLSS software development was initiated at the department in the mid 2000's [Paduart, 2008]. The code was developed in Matlab with the time-critical parts (filtering, calculation of Jacobians, optimization, etc.) in so-called .mex-files. Thanks to the underlying C-implementation, these Matlab executable files ran faster than if a pure Matlab implementation would have been used.

Over the years, the code was every now and then transferred from one researcher to another, and many people at the department successfully used and further developed the PNLSS software. Unfortunately, the code stopped working on the computers that received an update to Matlab R2015a.

After many efforts to debug the code and oftentimes running into so-called Heisenbugs, it was decided to implement everything in Matlab and further document the code. The result is PNLSS 1.0. Although the code no longer crashes at seemingly random times, the full Matlab implementation makes the code slower than before. Moreover, not all of the original functionality is yet implemented in PNLSS 1.0.

5.2 Future work

PNLSS 2.0, the successor of PNLSS 1.0, will have an object-oriented implementation in Matlab. This will allow for an easier inclusion of external modules, like the Matlab Optimization Toolbox, which in turn can speed up the code. This modular structure will also allow the user to include his/her favorite (external) module for a specific task (initialization, cost function specification, Jacobian calculation, optimization).

Currently, only polynomial nonlinearities are implemented in the toolbox. The functionality of the toolbox will be extended by also allowing for non-polynomial nonlinearities [Marconato, Sjöberg, Suykens & Schoukens, 2014].

A third improvement will be the ability to specify structure in the state-space matrices a priori, or to retrieve physical information a posteriori.

Appendix

A Function reference

This appendix gives an overview of the functions in the toolbox. The functions are listed in the order they appear in the Contents.m file. As mentioned in Section 2.4, the content file can be displayed in Matlab by browsing to the PNLSS folder and by running `help(pwd)`.

Eventually, the goal of the functions and the scripts in the toolbox is to estimate a PNLSS model from data and to simulate the model on (other) data. These functions and scripts are grouped in six categories:

1. Signal generation and transient handling

   Estimating a PNLSS model starts from (measured) data. To collect some informative data, you first need to properly excite the system under test.
   Since the PNLSS model is a dynamic model, its response depends on the (complete) history of the inputs provided to it. This history is captured in the initial state. If this initial state is unknown, i.e. if there is a mismatch between the actual initial state and the assumed initial state (typically zero initial conditions are assumed), a transient error occurs.

2. Nonparametric frequency response estimation

   A PNLSS model is nonlinear in its parameters. Estimating the parameters in a PNLSS model thus requires initial estimates, which are here provided by a linear state-space model. This state-space model is first estimated nonparametrically.

3. Parametric subspace identification

   A frequency domain subspace method is implemented to estimate a parametric linear model on top of the previously determined nonparametric frequency response function/matrix. The parameters obtained from the subspace method serve as initial values for the parameters of the PNLSS model.

4. Nonlinear optimization

   A nonlinear optimization on the model parameters is performed using a Levenberg-Marquardt method. Also the initial conditions (initial state and input) can be estimated in this way.

5. Model construction and simulation

   Before the nonlinear optimization scheme can be performed, the linear model should be put in a PNLSS format that is recognizable for

the optimization functions.
Once the PNLSS model is available, it can be used to simulate data.

6. Utility

   The toolbox also provides some functions that are useful, but that don't belong to one particular category.

More detailed information on the functions in each category and a brief explanation of how these functions are connected is provided in the corresponding section.

A.1 Signal generation and transient handling

Functions in this category:

Function	Section
`fComputeIndicesTransient`	A.1.1
`fComputeIndicesTransientRemoval`	A.1.2
`fComputeIndicesTransientRemovalArb`	A.1.3
`fMultisine`	A.1.4

The function `fMultisine` generates a random-phase multisine signal [Pintelon & Schoukens, 2012] with a flat power spectrum:

$$u(t) = \sum_{k=1}^{M} A_k \sin(2\pi k f_0 t + \phi_k)$$

where an amplitude $A_k = 1$ if frequency kf_0 is excited and $A_k = 0$ if that frequency is not excited. The phases ϕ_k are independent and uniformly ditributed between 0 and 2π. Several phase realizations of this periodic signal can be generated. Applying a periodic signal has the advantage that nonlinear and noise distortions can be separated (see `fCovarFrf` in Section A.2.1).

Computing the output of a PNLSS model (filtering, see `fFilterNLSS` in Section A.5.4) involves a recursive calculation where the state is updated starting from the previous state and input. This computation thus requires an initial state and input $x(0)$ and $u(0)$. Typically these are assumed to be zero, but in case this is not in correspondence to the actual initial conditions, a transient error occurs. The transient error typically decays to zero in some finite time. It is important to take this transient error into account when estimating the model, to make sure that the model does not try to fit the transient error. There are several ways to deal with this transient error:

- When working with arbitrary (i.e. non-periodic or periodic) data, the first few samples (transient samples) can simply be discarded after filtering. This option makes use of the decaying behavior of the transient error, but some information is lost (i.e. the transient samples are not used to estimate the model). The function `fComputeIndicesTransientRemovalArb` can be used to remove samples from a signal.

- When working with periodic excitations, (a part of) an extra period (or periods) can be added before the actual excitation signal to make sure that when the complete data set is simulated, the part corresponding to the actual excitation is in steady state. Adding the extra data can be done with the function `fComputeIndicesTransient`. The extra data can be removed after the simulation with `fComputeIndicesTransientRemoval`. With this option, no information is lost.

- A third option is to estimate the initial state and input, i.e. add $x(0)$ and $u(0)$ to the parameter vector. This is implemented in `fLMnlssWeighted_x0u0` (see Section A.4.2).

The functions `fComputeIndicesTransient`, `fComputeIndicesTransientRemoval`, and `fComputeIndicesTransientRemovalArb` are typically not directly accessed as a user. Rather, these functions are used in the functions `fFilterNLSS` and `fLMnlssWeighted`, where they are accessed through the PNLSS model structure fields `T1` (for periodic data) and `T2` (for arbitrary data).

A.1.1 `fComputeIndicesTransient`

Computes indices for transient handling for periodic signals before filtering.

Usage:

```
indices = fComputeIndicesTransient(T1,N)
```

Description:

`fComputeIndicesTransient` computes the indices to be used with a vector `u` of length `N` that contains (several realizations of) a periodic signal, such that `u(indices)` has `T1(1)` transient samples prepended to each realization. The starting samples of each realization can be specified in `T1(2:end)`. Like this, steady-state data can be obtained from a PNLSS model by using `u(indices)` as an input signal to a PNLSS model (see `fFilterNLSS`) and removing the transient samples afterwards (see `fComputeIndicesTransientRemoval`).

Output parameters:

`indices` indices of a vector `u` that contains (several realizations of) a periodic signal, such that `u(indices)` has a number of transient samples added before each realization

Input parameters:

`T1` vector that indicates how the transient is handled. The first element `T1(1)` is the number of transient samples that should be prepended to each realization. The other elements `T1(2:end)` indicate the starting sample of each realization in the signal. If `T1` has only one element, `T1(2)` is put to one.

`N` length of the signal containing all realizations

Example:

```
Npp = 1000; % Number of points per period
R = 2; % Number of phase realizations
T = 100; % Number of transient samples
T1 = [T 1:Npp:(R-1)*Npp+1]; % Transient handling vector
N = R*Npp; % Total number of samples
indices = fComputeIndicesTransient(T1,N);
% => indices = [901:1000 1:1000 1901:2000 1001:2000]
%            = [transient samples realization 1, ...
%               realization 1, ...
%               transient samples realization 2, ...
%               realization 2]
```

A.1.2 `fComputeIndicesTransientRemoval`

Computes indices for transient handling for periodic signals after filtering.

Usage:

```
indices = fComputeIndicesTransientRemoval(T1,N,p)
```

Description:

Let `u` be a vector of length `N` containing (several realizations of) a periodic signal. Let `uTot` be a vector containing the signal(s) in `u` with `T1(1)` transient points prepended to each realization (see `fComputeIndicesTransient`). The starting samples of each realization can be specified in `T1(2:end)`. Let `yTot` be a vector/matrix containing the `p` outputs of a PNLSS model after applying the input `uTot`. Then `fComputeIndicesTransientRemoval` computes the indices to be used with the vectorized form of `yTot` such that the transient samples are removed from `yTot`, i.e. `y = yTot(indices)` contains the steady-state output(s) stacked on top of each other.

Output parameters:

`indices` If `uTot` is a vector containing (several realizations of) a periodic signal to which `T1(1)` transient points were added before each realization, and if `yTot` is the corresponding output vector (or matrix if more than one output), then `indices` is such that the transient points are removed from `y = yTot(indices)`.
If `p > 1`, then `indices` is a column vector and `y = yTot(indices)` is a column vector with the steady state outputs stacked on top of each other.

Input parameters:

`T1` vector that indicates how the transient is handled. The first element `T1(1)` is the number of transient samples that were prepended to each realization. The other elements `T1(2:end)` indicate the starting sample of each realization in the input signal. If `T1` has only one element, `T1(2)` is put to one.
`N` length of the input signal containing all realizations
`p` number of outputs

Example:

```
Npp = 1000; % Number of points per period
R = 2; % Number of phase realizations
T = 100; % Number of transient samples
T1 = [T 1:Npp:(R-1)*Npp+1]; % Transient handling vector
```

```
N = R*Npp; % Total number of samples
indices_tot = fComputeIndicesTransient(T1,N);
% => indices_tot = [901:1000 1:1000 1901:2000 1001:2000]
%                = [transient samples realization 1, ...
%                   realization 1, ...
%                   transient samples realization 2, ...
%                   realization 2]
p = 1; % One output
indices_removal = fComputeIndicesTransientRemoval(T1,N,p);
% => indices_removal = [101:1100 1201:2200].'
% => indices_tot(indices_removal) = 1:2000
%                                 = [realization 1, realization 2]

p = 2; % More than one output
indices_removal = fComputeIndicesTransientRemoval(T1,N,p);
% => indices_removal = [101:1100 1201:2200 2301:3300 3401:4400].'
% Let u be a vector containing [input realization 1;
%                               input realization 2],
% then uTot = u(indices_tot) is a vector containing
%             [transient samples realization 1;
%              input realization 1;
%              transient samples realization 2;
%              input realization 2]
% Let y1 be a vector containing the first output and y2 be a vector
% containing the second output when applying uTot as an input to a
% PNLSS model, and let yTot = [y1 y2] be a 2200 x 2 matrix with y1
% and y2 in its first and second column, respectively.
% Note that y1 = yTot(1:2200).' and y2 = yTot(2201:4400).' (see
% also ind2sub and sub2ind)
% Then yTot(indices_removal) = [y1(101:1100);
%                               y1(1201:2200);
%                               y2(101:1100);
%                               y2(1201:2200)]
%                            = [output 1 corresponding to input
    realization 1;
%                               output 1 corresponding to input
    realization 2;
%                               output 2 corresponding to input
    realization 1;
%                               output 2 corresponding to input
    realization 2]
```

A.1.3 `fComputeIndicesTransientRemovalArb`

Remove transients from arbitrary data.

Usage:

```
[indices, NT] = fComputeIndicesTransientRemovalArb(T2,N,p)
```

Description:

Computes the indices to be used with a `N` × `p` matrix containing `p` output signals of length `N`, such that `y(indices)` contains the transient-free output(s) of length `NT` stacked on top of each other (if more than one output). The transient samples to be removed are specified in `T2` (`T2 = 1:T2` if `T2` is scalar).

Output parameters:

`indices` vector of indices, such that `y(indices)` contains the output(s) without transients. If more than one output (`p` > 1), then `y(indices)` stacks the transient-free outputs on top of each other.

`NT` length of the signal without transients

Input parameters:

`T2` scalar indicating how many samples from the start are removed or vector indicating which samples are removed

`N` length of the total signal

`p` number of outputs

Examples:

```
% One output, T2 scalar
N = 1000; % Total number of samples
T2 = 200; % First 200 samples should be removed after filtering
p = 1; % One output
[indices, NT] = fComputeIndicesTransientRemovalArb(T2,N,p);
% => indices = (201:1000).'; % Indices of the transient-free output
     (in uint32 format in version 1.0)
% => NT = 800; % Number of samples in the transient-free output
```

```
% Two outputs, T2 scalar
N = 1000; % Total number of samples
T2 = 200; % First 200 samples should be removed after filtering
p = 2; % Two outputs
[indices, NT] = fComputeIndicesTransientRemovalArb(T2,N,p);
% => indices = ([201:1000 1201:2000]).'; % Indices of the transient
    -free outputs (in uint32 format in version 1.0)
```

```
% => NT = 800; % Number of samples in each transient-free output
% If y = [y1 y2] is a 1000 x 2 matrix with the two outputs y1 and
    y2,
% then y(indices) = [y1(201:1000);
%                    y2(201:1000)]
% is a vector with the transient-free outputs stacked on top of
% each other
```

```
% One output, T2 is a vector
N1 = 1000; % Number of samples in a first data set
N2 = 500; % Number of samples in a second data set
N = N1 + N2; % Total number of samples
T2_1 = 1:200; % Transient samples in first data set
T2_2 = 1:100; % Transient samples in second data set
T2 = [T2_1 (N1+T2_2)]; % Transient samples
p = 1; % One output
[indices, NT] = fComputeIndicesTransientRemovalArb(T2,N,p);
% => indices = ([201:1000 1101:1500]).'; (in uint32 format in
    version 1.0)
% => NT = 1200;
```

A.1.4 `fMultisine`

Generate a random-phase multisine signal with a flat spectrum.

Usage:

```
[u,lines,non_exc_odd,non_exc_even] = fMultisine(N,kind,M,R,Nblock)
[u,lines,non_exc_odd,non_exc_even] = fMultisine(N,lines,M,R,Nblock)
[u,lines,non_exc] = fMultisine(N,kind,M,R,Nblock)
[u,lines,non_exc] = fMultisine(N,lines,M,R,Nblock)
u = fMultisine()
```

Description:

`fMultisine` generates `R` realizations of a random-phase multisine signal with `N` samples and excitation up to line `M` (line 1, i.e. DC, not included). The excited lines in this band can be specified either as a vector with the excited lines or as a string indicating the type of multisine (e.g. all lines excited, only odd lines excited, etc. (see info with input parameter `kindOrLines`)).

Output parameters:

`u`	`N` × `R` matrix with multisine signal(s)
`lines`	Excited lines
`non_exc_odd_or_all`	Non-excited odd lines (if there are four output arguments) or all non-excited lines (if there are three output arguments)
`non_exc_even`	Non-excited even lines

Input parameters:

`N`	Number of samples (optional, default = 1024)
`kindOrLines`	String indicating the type of multisine: `'full'`: all lines excited `'odd'`: only odd lines excited `'special odd'`: odd frequencies `8*k+1` and `8*k+3` (⇒ lines `8*k+2` and `8*k+4`) excited for `k = 0, 1, ...` `'random odd'`: odd lines excited except for one random detection line in each group of `Nblock` consecutive odd lines or a vector with the excited lines (optional, default = `'full'`)
`M`	Last excited line (optional, default = 90% of Nyquist)
`R`	Number of realizations (optional, default = 1)
`Nblock`	Block length for random odd multisine (optional, default = 4)

Examples:

```
% Generate two realizations of a full multisine with 1000 samples
% and excitation up to one third of the Nyquist frequency
N = 1000; % One thousand samples
kind = 'full'; % Full multisine
M = floor(N/6); % Excitation up to one sixth of the sample
    frequency
R = 2; % Two phase realizations
u = fMultisine(N,kind,M,R); % Multsine signals
% Check spectra
figure
subplot(2,2,1)
plot(db(fft(u(:,1))),'+')
xlabel('Frequency line')
ylabel('Amplitude (dB)')
title({'Phase realization 1'})
subplot(2,2,2)
plot(db(fft(u(:,2))),'+')
xlabel('Frequency line')
ylabel('Amplitude (dB)')
title({'Phase realization 2'})
subplot(2,2,3)
plot(angle(fft(u(:,1))),'+')
xlabel('Frequency line')
ylabel('Phase (rad)')
title({'Phase realization 1'})
subplot(2,2,4)
plot(angle(fft(u(:,2))),'+')
xlabel('Frequency line')
ylabel('Phase (rad)')
title({'Phase realization 2'})
% The two realizations have the same amplitude spectrum, but
% different phase realizations (uniformly distributed between -pi
% and pi)
```

```
% Generate a random odd multisine where the excited odd lines are
% split in groups of three consecutive lines and where one line is
% randomly chosen in each group to be a detection line (i.e.
% without excitation)
N = 1000; % One thousand samples
kind = 'random odd'; % Random odd multisine
M = floor(N/6); % Excitation up to one sixth of the sample
    frequency
R = 1; % One phase realization
```

```
Nblock = 3; % One out of three consecutive odd lines is randomly
    selected to be a detection line
[u1,lines] = fMultisine(N,kind,M,R,Nblock); % Multisine signal and
    excited lines
% Generate another phase realization of this multisine with the
% same excited lines and detection lines
u2 = fMultisine(N,lines,[],1); % One realization of a multisine
    with the same excited lines as u1
% Change the coloring and rms level of the generated multisine
u = fMultisine(); % Default multisine
[b,a] = cheby1(2,10,2*0.1); % Filter coefficients
U = fft(u); % Multisine spectrum
U_colored = freqz(b,a,2*pi*(0:length(u)-1).'/length(u)).*U; %
    Filtered spectrum
u_colored = real(ifft(U_colored)); % Colored multisine signal
u_scaled = 2*u_colored/std(u_colored); % Scaled signal to rms value
     2 (u_colored is zero-mean)
```

A.2 Nonparametric frequency response estimation

Functions in this category:

Function	Section
`fCovarFrf`	A.2.1
`fCovarY`	A.2.2

A PNLSS model is nonlinear in its parameters. Estimating the parameters in a PNLSS model thus requires initial estimates.

The function `fCovarFrf` can be used to estimate a linear model nonparamtrically (i.e. in the form of a frequency response function/matrix). Moreover, when working with periodic data, the total (noise + nonlinear) and noise distortion levels are estimated with this function. Comparing the total and noise distortion levels indicate the gain that can potentially be made when going from a linear to a nonlinear model. Also, the total distortion level can be used as a frequency weighting when a parametric model is estimated on top of the nonparametric one (see `fLoopSubSpace` in Section A.3.6). The parametric model can then be used as an initialization for the PNLSS model.

The function `fCovarY` can be used to estimate the covariance matrix on the output spectrum, which can be used as a frequency weighting when estimating the PNLSS model (see `fLMnlssWeighted` and `fLMnlssWeighted_x0u0` in Sections A.4.1 and A.4.2).

A.2.1 `fCovarFrf`

Computes frequency response matrix and noise and total covariance matrix from input/output spectra.

Usage:

```
[G,covGML,covGn] = fCovarFrf(U,Y)
```

Description:

Estimates the frequency response matrix, and the corresponding noise and total covariance matrices from the spectra of periodic input/output data.
`covGn = 0` if only one period is specified.
`covGML = 0` if only one experiment block of `m` experiments is specified, i.e. there should be at least two times as many experiments as inputs to be able to estimate the total covariance.

Output parameters:

`G`	`p` × `m` × `F` Frequency Response Matrix (FRM)
`covGML`	`p*m` × `p*m` × `F` total covariance (= stochastic nonlinear contributions + noise)
`covGn`	`p*m` × `p*m` × `F` noise covariance

Input parameters:

`U`	`F` × `m` × `exp` × `P` input spectra at `F` frequency lines, with `m` inputs, `exp` experiments (typically phase realizations of a multisine, `exp` is preferably an integer multiple of `m`), and `P` periods
`Y`	`F` × `p` × `exp` × `P` output spectra with `p` outputs

Example:

```
% SISO example (Hammerstein system)
f_NL = @(x) x + 0.2*x.^2 + 0.1*x.^3; % Nonlinear function
[b,a] = cheby1(2,10,2*0.1); % Filter coefficients
N = 1000; % Number of samples per period
Lines = 2:133; % Excited frequency lines;
R = 3; % Number of phase realizations
PTrans = 1; % Number of transient periods
P = 2; % Number of periods
u = fMultisine(N,Lines,[],R); % Input signal: three phase
    realizations of a random-phase multisine
u = u/mean(std(u)); % Scale input signal
u = repmat(u,[PTrans+P,1]); % One transient period + two periods
x = f_NL(u); % Intermediate signal
```

```
y0 = filter(b,a,x); % Noise-free output signal
y = y0 + 0.01*mean(std(y0))*randn(size(y0)); % Noisy output signal
u(1:PTrans*N,:) = []; % Remove transient period(s)
x(1:PTrans*N,:) = []; % Remove transient period(s)
y(1:PTrans*N,:) = []; % Remove transient period(s)
u = reshape(u,[N,P,R]); % Reshape input signal
y = reshape(y,[N,P,R]); % Reshape output signal
U = fft(u); % Input spectrum
Y = fft(y); % Output spectrum
U = U(Lines,:,:); % Select only excited frequency lines
Y = Y(Lines,:,:); % Select only excited frequency lines
U = permute(U,[1,4,3,2]); % F x m x R x P
Y = permute(Y,[1,4,3,2]); % F x p x R x P
[G,covGML,covGn] = fCovarFrf(U,Y); % Compute FRF, total and noise
    distortion
figure
plot(Lines,db([G(:) covGML(:) covGn(:)]))
xlabel('Frequency line')
ylabel('Amplitude (dB)')
legend('FRF','Total distortion','Noise distortion')
```

A.2.2 `fCovarY`

Compute covariance matrix output spectra due to output noise from time domain output signals.

Usage:

```
covY = fCovarY(y)
```

Description:

Calculate frequency domain covariance matrix of the output spectrum starting from time domain output signals.

Output parameters:

`covY` `p` × `p` × `NFD` covariance matrix of the output(s). Calculates variations along the periods and averages over the realizations.

Input parameters:

`y` `N` × `p` × `R` × `P` output signal(s), where `N` is the number of samples, `p` is the number of outputs, `R` is the number of realizations, and `P` is the number of periods

Example:

```
y0 = randn(1000,1); % Noise-free signal
y0 = repmat(y0,3,1); % Three periods
y0 = reshape(y0,[1000,1,1,3]); % N x p x R x P
[b,a] = cheby1(2,10,2*0.1); % Coefficients noise filter
e = randn(size(y0)); % White noise
v = filter(b,a,e); % Colored noise
y = y0 + v; % Noisy signal
covY = fCovarY(y); % Noise covariance
figure
plot(db(freqz(b,a,2*pi*(1:500)/1000)*sqrt(1000)/sqrt(3)),'b')
hold on
plot(db(squeeze(sqrt(covY(:,:,1:500)))),'r')
xlabel('Frequency line')
ylabel('Amplitude (dB)')
title('Noise variance on the averaged signal (average over 3
    periods)')
legend('True','Estimated')
```

A.3 Parametric subspace identification

Functions in this category:

Function	Section
`fFreqDomSubSpace`	A.3.1
`fFreqDomSubSpaceCT`	A.3.2
`fIsUnstable`	A.3.3
`fJacobFreqSS`	A.3.4
`fLevMarqFreqSSz`	A.3.5
`fLoopSubSpace`	A.3.6
`fss2frf`	A.3.7
`fss2frfCT`	A.3.8
`fStabilize`	A.3.9
`fWeightJacobSubSpace`	A.3.10

Starting from frequency response function/matrix (FRF/FRM) data (obtained for example with the functions in the previous section), the frequency domain subspace method in McKelvey et al. [1996]; Pintelon [2002] can estimate a parametric linear state-space model. This subspace method can provide a continuous-time model (as implemented with the `fFreqDomSubSpaceCT` function) or a discrete-time model (as implemented with the `fFreqDomSubSpace` function).

The discrete-time model can then serve as an initialization for the PNLSS model, provided it is a stable model. The stability of the model can be checked with `fIsUnstable` (works also in continuous-time). If the model is unstable, an attempt can be made to stabilize the model with the `fStabilize` function.

The subspace method provides a good initial estimate for the linear model parameters. To improve on those estimates, a Levenberg-Marquardt optimization can be done with the function `fLevMarqFreqSSz` (only discrete-time). The Jacobians of the unweighted least-squares cost function, needed in the Levenberg-Marquardt optimization, are computed by the `fJacobFreqSS` function. If a weighted cost function is optimized, the weighting is added to the unweighted Jacobians by the `fWeightJacobSubSpace` function.

The subspace method and the Levenberg-Marquardt optimization on the linear model parameters are combined in the `fLoopSubSpace`. Moreover, several model orders and several values of a subspace dimensioning parameter can be scanned with this function.

The functions `fss2frfCT` and `fss2frf` perform the reverse operation of the functions `fFreqDomSubSpaceCT` and `fFreqDomSubSpace`, respectively. Starting from a linear state-space model in continuous time (`fss2frfCT`) or discrete time (`fss2frf`), they compute the frequency response function/matrix.

A.3.1 `fFreqDomSubSpace`

Estimate state-space model from Frequency Response Function (or Matrix).

Usage:

```
[A,B,C,D,unstable] = fFreqDomSubSpace(H,covarH,freq,n,r)
[A,B,C,D] = fFreqDomSubSpace(H,covarH,freq,n,r)
```

Description:

`fFreqDomSubSpace` estimates linear state-space models from samples of the frequency response function (or frequency response matrix). The frequency-domain subspace method in McKelvey et al. [1996] is applied with the frequency weighting in Pintelon [2002], i.e. weighting with the sampled covariance matrix.

Output parameters:

`A`	`n` $\times$ `n` state matrix
`B`	`n` $\times$ `m` input matrix
`C`	`p` $\times$ `n` output matrix
`D`	`p` $\times$ `m` feed-through matrix
`unstable`	boolean indicating whether or not the identified state-space model is unstable

Input parameters:

`H`	`p` $\times$ `m` $\times$ `F` Frequency Response Matrix (FRM)
`covarH`	`p*m` $\times$ `p*m` $\times$ `F` covariance tensor on `H` (0 if no weighting required)
`freq`	vector of normalized frequencies at which the FRM is given ($0 <$ freq < 0.5)
`n`	model order
`r`	number of block rows in the extended observability matrix (r $>$ n)

Example:

```
n = 2; % Model order
N = 1000; % Number of samples
fs = 1; % Sampling frequency
sys = drss(n); % Random state-space model
freq = (0:floor(N/2)-1)*fs/N; % Frequency vector
H = fss2frf(sys.A,sys.B,sys.C,sys.D,freq/fs); % FRF
covarH = 0; % No weighting
r = n+1; % Maximal number of block rows in the extended
    observability matrix
```

```
[A,B,C,D] = fFreqDomSubSpace(H,covarH,freq/fs,n,r);
sys_est = ss(A,B,C,D,1/fs); % Estimated state-space model
figure
bode(sys,'b-',sys_est,'r--') % Compare Bode plots of both models
legend('True','Estimated')
```

A.3.2 `fFreqDomSubSpaceCT`

Estimate state-space model from Frequency Response Function (or Matrix) (continuous-time).

Usage:

```
[A,B,C,D,unstable] = fFreqDomSubSpaceCT(H,covarH,s,n,r)
[A,B,C,D] = fFreqDomSubSpaceCT(H,covarH,s,n,r)
```

Description:

`fFreqDomSubSpaceCT` estimates linear state-space models from samples of the frequency response function (or frequency response matrix). The frequency-domain subspace method in McKelvey et al. [1996] is applied with z replaced by s.

Output parameters:

`A`	`n` × `n` state matrix
`B`	`n` × `m` input matrix
`C`	`p` × `n` output matrix
`D`	`p` × `m` feed-through matrix
`unstable`	boolean indicating whether or not the identified state-space model is unstable

Input parameters:

`H`	`p` × `m` × `F` Frequency Response Matrix (FRM)
`covarH`	`p*m` × `p*m` × `F` covariance tensor on `H` (0 if no weighting required)
`s`	vector `s = 1j*2*pi*freq`, where `freq` is a vector of frequencies (in Hz) at which the FRM is given
`n`	model order
`r`	number of block rows in the extended observability matrix (r > n)

Example:

```
n = 2; % Model order
N = 1000; % Number of samples
fs = 1; % Sampling frequency
sys = rss(n); % Random state-space model
freq = (0:floor(N/2)-1)*fs/N; % Frequency vector
s = 1j*2*pi*freq; % s vector
H = fss2frfCT(sys.A,sys.B,sys.C,sys.D,s); % FRF
covarH = 0; % No weighting
```

```
r = n+1; % Maximal number of block rows in the extended
    observability matrix
[A,B,C,D] = fFreqDomSubSpaceCT(H,covarH,s,n,r);
sys_est = ss(A,B,C,D); % Estimated state-space model
figure
bode(sys,'b-',sys_est,'r--') % Compare Bode plots of both models
legend('True','Estimated')
```

A.3.3 `fIsUnstable`

Determines if a linear state-space model is unstable.

Usage:

```
unstable = fIsUnstable(A,'s')
unstable = fIsUnstable(A,'z')
```

Description:

`unstable = fIsUnstable(A,'s')` determines if a continuous-time state-space model with state matrix `A` is unstable.
`unstable = fIsUnstable(A,'z')` determines if a discrete-time state-space model with state matrix `A` is unstable.

Output parameters:

`unstable` boolean indicating whether or not the state-space model is unstable

Input parameters:

`A`	`n` $\times$ `n` state matrix
`domain`	`'s'` for continuous-time state-space models
	`'z'` for discrete-time state-space models

A.3.4 fJacobFreqSS

Compute Jacobians of the unweighted errors w.r.t. elements A, B, and C matrices.

Usage:

```
[JA,JB,JC] = fJacobFreqSS(A,B,C,z)
```

Description:

Computes the Jacobians of the unweighted errors G_hat(f)− G(f) w.r.t. the elements in the A, B, and C state-space matrices. Let e(f)= G_hat(f)− G(f), where G_hat(f)= C∗(z(f)∗I − A)^(−1)∗B + D is the estimated and G(f) is the measured frequency response matrix (FRM), then
JA(:,:,sub2ind([n n],k,l),f) contains the partial derivative of e(f) w.r.t. A(k,l),
JB(:,:,sub2ind([n m],k,l),f) contains the partial derivative of e(f) w.r.t. B(k,l), and
JC(:,:,sub2ind([p n],k,l),f) contains the partial derivative of e(f) w.r.t. C(k,l).

Output parameters:

JA p × m × n² × F tensor where JA(:,:,sub2ind([n n],k,l),f) contains the partial derivative of the unweighted error e(f) at frequency f w.r.t. A(k,l)
JB p × m × n∗m × F tensor where JB(:,:,sub2ind([n m],k,l),f) contains the partial derivative of e(f) w.r.t. B(k,l)
JC p × m × p∗n × F tensor where JC(:,:,sub2ind([p n],k,l),f) contains the partial derivative of e(f) w.r.t. C(k,l)

Input parameters:

A n × n state matrix
B n × m input matrix
C p × n output matrix
z F × 1 vector z = exp(1j∗2∗pi∗freq(:)), where freq is a vector of normalized frequencies at which the Jacobians are computed
(0 < freq < 0.5)

A.3.5 fLevMarqFreqSSz

Optimize state-space matrices using Levenberg-Marquardt.

Usage:

```
[A,B,C,D] = fLevMarqFreqSSz(freq,G,covG,A0,B0,C0,D0,MaxCount)
```

Description:

Optimize state-space matrices `A`, `B`, `C`, and `D` using at most `MaxCount` Levenberg-Marquardt runs starting from initial values `A0`, `B0`, `C0`, and `D0`. The cost function that is optimized is the weighted sum of the mean-square differences between the provided frequency response matrix `G` at the normalized frequencies `freq` and the estimated frequency response matrix `G_hat = C*(z(freq)*I - A)^(-1)*B + D`. The cost function is weighted with the inverse of the covariance matrix of `G`, if this covariance matrix `covG` is provided (no weighting if `covG = 0`).

Output parameters:

`A`	`n` × `n` optimized state matrix
`B`	`n` × `m` optimized input matrix
`C`	`p` × `n` optimized output matrix
`D`	`p` × `m` optimized feed-through matrix

Input parameters:

`freq`	vector of normalized frequencies at which the FRM is given ($0 <$ `freq` < 0.5)
`G`	`p` × `m` × `F` array of FRF (or FRM) data
`covG`	`p*m` × `p*m` × `F` covariance tensor on `G` (0 if no weighting required)
`A0`	`n` × `n` initial state matrix
`B0`	`n` × `m` initial input matrix
`C0`	`p` × `n` initial output matrix
`D0`	`p` × `m` initial feed-through matrix
`MaxCount`	maximum number of Levenberg-Marquardt optimizations (maximum 1000, if `MaxCount = Inf`, then the algorithm stops if the cost functions of the last 10 successful steps differ by less than 0.1% or if 1000 iterations were performed)

A.3.6 fLoopSubSpace

Loop frequency-domain subspace method over multiple model orders and sizes of the extended observability matrix.

Usage:

```
[models,figHandle] = fLoopSubSpace(freq,G,covG,na,max_r,optimize,forcestability
,showfigs,fs)
models = fLoopSubSpace(freq,G,covG,na,max_r,optimize)
```

Description:

`fLoopSubSpace` estimates linear state-space models from samples of the frequency response function (or frequency response matrix). The frequency-domain subspace method in McKelvey et al. [1996] is applied with the frequency weighting in Pintelon [2002]. The subspace algorithm is looped over different model orders and different block rows of the extended observability matrix. If requested, a Levenberg-Marquardt algorithm is carried out to optimize the model parameters obtained with the subspace algorithm. Stability of the estimated models can be forced (to some extent). For each model order, the model that minimizes the weighted mean-square error

$$\texttt{e} = \frac{1}{\texttt{F}} \sum_{\texttt{f = 1}}^{\texttt{F}} \texttt{real(vec(G(f)- Gm(f))'*covGinv(f)*vec(G(f)- Gm(f)))}$$

is retained as the best model for that order. Here, `Gm(f)` is the FRF (or FRM) of the (optimized) subspace model.

Output parameters:

`models`	cell array where `models{n}` contains a 1×4 cell with the `A`, `B`, `C`, and `D` matrices of the best model of order `n` (see the Description for more information on 'best')
`figHandle`	figure handle of the last plotted figure

Input parameters:

`freq`	vector of frequencies at which the FRF (or FRM) is given (in Hz and in the interval `[0, fs/2]`)
`G`	`m` × `p` × `F` array of FRF (or FRM) data
`covG`	`m*p` × `m*p` × `F` covariance tensor on `G` (0 if no weighting required)
`na`	vector of model orders to scan
`max_r`	maximum number of block rows in the extended observability matrix (no models are estimated for orders `n` ≤ `max_r`)
`optimize`	number of Levenberg-Marquardt optimizations (0 if no optimization required, maximum 1000)
`forcestability`	boolean indicating whether or not a stable model is forced (optional, default = true)
`showfigs`	boolean indicating whether or not to show figures (optional, default = true)
`fs`	sampling frequency (in Hz) (optional, default = 1 Hz)

Example:

```
n = 2; % Model order
N = 1000; % Number of samples
fs = 1; % Sampling frequency
sys = drss(n); % Random state-space model
freq = (0:floor(N/2)-1)*fs/N; % Frequency vector
G = fss2frf(sys.A,sys.B,sys.C,sys.D,freq/fs); % FRF
covG = 0; % No weighting
max_r = n+3; % Maximal number of block rows in the extended
    observability matrix
optimize = 0; % No Levenberg-Marquardt optimization loops
models = fLoopSubSpace(freq,G,covG,n,max_r,optimize);
model = models{n}; % Select nth-order model
sys_est = ss(model{:},1/fs); % Estimated state-space model
figure
bode(sys,'b-',sys_est,'r--') % Compare Bode plots of both models
legend('True','Estimated')
```

A.3.7 `fss2frf`

Compute frequency response function from state-space parameters (discrete-time).

Usage:

```
GSS = fss2frf(A,B,C,D,freq)
```

Description:

`GSS = fss2frf(A,B,C,D,freq)` computes the frequency response function (FRF) or matrix (FRM) `GSS` at the normalized frequencies `freq` from the state-space matrices `A`, `B`, `C`, and `D`.
`GSS(f)= C*inv(exp(1j*2*pi*f)*I - A)*B + D`

Output parameters:

`GSS` `p` × `m` × `F` frequency response matrix

Input parameters:

`A` `n` × `n` state matrix
`B` `n` × `m` input matrix
`C` `p` × `n` output matrix
`D` `p` × `m` feed-through matrix
`freq` vector of normalized frequencies at which the FRM is computed ($0 <$ freq < 0.5)

A.3.8 `fss2frfCT`

Compute frequency response function from state-space parameters (continuous-time).

Usage:

```
GSS = fss2frfCT(A,B,C,D,s)
```

Description:

`GSS = fss2frfCT(A,B,C,D,s)` computes the frequency response function (FRF) or matrix (FRM) `GSS` at the frequencies `s/(2*pi*1j)` from the state-space matrices `A`, `B`, `C`, and `D`.
`GSS(f)= C*inv(1j*2*pi*f*I − A)*B + D`

Output parameters:

`GSS` `p` × `m` × `F` frequency response matrix

Input parameters:

`A` `n` × `n` state matrix
`B` `n` × `m` input matrix
`C` `p` × `n` output matrix
`D` `p` × `m` feed-through matrix
`s` vector `s = 1j*2*pi*freq`, where `freq` is a vector of frequencies (in Hz) at which the FRM is computed

A.3.9 `fStabilize`

Stabilize a linear state-space model.

Deprecated!

Usage:

```
[A,B,C,D] = fStabilize(A,B,C,D,'s',highest_s)
[A,B,C,D] = fStabilize(A,B,C,D,'z',[])
```

Description:

Stabilizes a state-space model by reflecting the unstable poles w.r.t. the stability border (i.e. in discrete-time, unstable poles are mirrored w.r.t the unit circle; in continuous-time, unstable poles are mirrored w.r.t. the imaginary axis). Like this, the amplitude characteristic of the unstable part remains the same, but the phase is changed. An all-pass section to compensate for this phase change is not added, as it increases the model order.
⇒ Take care when using this function that the phase of the stabilized FRF is not the same as the phase of the non-stabilized FRF.
After reflecting the unstable poles, the FRFs of the stable and the stabilized unstable part are summed, and a state-space model is estimated on the sum of these FRFs. Note that this new estimate is not guaranteed to be stable.

Output parameters:

`A`	`n` × `n` stabilized state matrix
`B`	`n` × `m` stabilized input matrix
`C`	`p` × `n` stabilized output matrix
`D`	`p` × `m` stabilized feed-through matrix

Input parameters:

`A`	`n` × `n` state matrix
`B`	`n` × `m` input matrix
`C`	`p` × `n` output matrix
`D`	`p` × `m` feed-through matrix
`domain`	`'s'` for continuous-time state-space models `'z'` for discrete-time state-space models
`highest_s`	largest `s`-value up to which the FRF of the stable and the stabilized unstable part are calculated before estimating a new state-space model on top of the sum of both

A.3.10 `fWeightJacobSubSpace`

Adds weighting to an unweighted Jacobian.

Usage:

```
out = fWeightJacobSubSpace(Jacob,C,mp,F,npar)
```

Description:

Computes the Jacobian of the weighted error `e_W(f)= W(:,:,f)*e(f)` w.r.t. the elements of a state-space matrix (`A`, `B`, `C`, or `D`), given the Jacobian `Jacob` of the unweighted error `e(f)= G_hat(f)− G(f)` w.r.t. the elements of the same state-space matrix, where `G_hat(f)= C*(z(f)*I − A)^(−1)*B + D`.

Remark:

Can be used more generally to add weighting to an unweighted Jacobian of a vector-valued function with `mp` outputs and `npar` inputs, and sampled in `F` operating points.

Output parameters:

`out` `m*p` × `F` × `npar` weighted Jacobian

Input parameters:

`Jacob` `m*p` × `F` × `npar` unweighted Jacobian
`W` `m*p` × `m*p` × `F` weighting matrix (e.g. square root of inverse of covariance matrix of `G`)
`mp` number of inputs times number of outputs
(optional, determined from `Jacob` if not provided)
`F` number of frequencies at which the Jacobian is computed
(optional, determined from Jacob if not provided)
`npar` number of parameters in the state-space matrix w.r.t. which the Jacobian is taken
(optional, determined from `Jacob` if not provided)

A.4 Nonlinear optimization

Functions and scripts in this category:

Function/script	Section
`fLMnlssWeighted`	A.4.1
`fLMnlssWeighted_x0u0`	A.4.2
`fComputeJF`	A.4.3
`fEdwdx`	A.4.4
`fEdwdu`	A.4.5
`fJNL`	A.4.6
`fJx0`	A.4.7
`fJu0`	A.4.8
`sJacobianAnalytical`	A.4.9
`sJacobianAnalytical_x0u0`	A.4.10

Starting from an initial model (for example the linear model provided by the `fLoopSubSpace` function (see Section A.3.6) and put in PNLSS format by the `fCreateNLSSmodel` function (see Section A.5.1)), a Levenberg-Marquardt optimization can be done on all model parameters using the `fLMnlssWeighted` function (or `fLMnlssWeighted_x0u0` if also the initial conditions are estimated). The necessary Jacobians are computed with the functions `fComputeJF` (Jacobian w.r.t. the matrix F), `fJNL` (Jacobians w.r.t. the matrices A, B, and E), `fJx0` and `fJu0` (Jacobians w.r.t. initial conditions $x(0)$ and $u(0)$). The computation of all the Jacobians is organized in a script `sJacobianAnalytical` (or `sJacobianAnalytical_x0u0` if also the initial conditions are estimated). The functions `fEdwdx` and `fEdwdu` are auxiliary functions for these scripts.

A.4.1 `fLMnlssWeighted`

Optimize PNLSS model using weighted Levenberg-Marquardt algorithm.

Usage:

```
[model,y_mod,models,Cost] = fLMnlssWeighted(u,y,model,MaxCount,W,lambda
,LambdaJump)
[model,y_mod,models,Cost] = fLMnlssWeighted(u,y,model,MaxCount,W)
```

Description:

`fLMnlssWeighted` performs a Levenberg-Marquardt optimization on the parameters of a PNLSS model (i.e. the elements of the matrices `A`, `B`, `C`, `D`, `E`, and `F`). The difference between the modeled and the measured output is minimized in a weighted least squares sense, either in the time or the frequency domain. A simple stabilization method can be applied by simulating a validation data set during estimation and checking whether or not the modeled output stays within prespecified bounds. If not, the Levenberg-Marquardt iteration acts as if the cost function increased.

Output parameters:

`model`	optimized model (= best on estimation data)
`y_mod`	output of the optimized model
`models`	collection of models (initial model + model after a successful iteration)
`Cost`	collection of the unweighted rms error at each iteration (`NaN` if iteration was not successful (i.e. when the weighted rms error increased))

Input parameters:

`u` — `N` × `m` input signal

`y` — `N` × `p` output signal

`model` — initial model (see `fCreateNLSSmodel`). Additionally, two optional fields can be added to the model to perform a simple stabilization method (both fields are needed to perform this method):

- `u_val` — `m` × `N_val` matrix with `N_val` samples of the validation input
  (optional, no default value)
- `max_out` — bound on the maximum absolute value of the simulated validation output
  (optional, no default value)

After each Levenberg-Marquardt iteration, the validation data is simulated (without taking into account the transient settings). If the simulated validation output does not respect the `max_out` bound, then the iteration is considered unsuccessful.

`MaxCount` — (maximum) number of iterations (there is not yet an early stopping criterion)

`W` — `p` × `p` × `NFD` weighting matrix if frequency-domain weighting (e.g. square root of covariance matrix of output noise spectrum), where `NFD` is the number of frequency bins in the positive half of the spectrum (of one period and one phase realization) of the input (e.g. `NFD = floor(Npp/2)`, where `Npp` is the number of samples in one period and one phase realization for a multisine excitation).
`N` × `p` weighting sequences if time-domain weighting.
`[]` if no weighting.
(optional, default is no weighting)

`lambda` — initial Levenberg-Marquardt parameter
(optional, default = 0, which corresponds to a Gauss-Newton algorithm). After a successful iteration, `lambda` is halved. After an unsuccessful iteration, `lambda` is multiplied with a factor $\sqrt{10}$, unless `lambda` was zero, in which case `lambda` is put equal to the dominant singular value of the Jacobian.

`LambdaJump` — each `LambdaJump` iterations, the Levenberg-Marquardt parameter is made smaller by a factor 10, so that the algorithm leans more towards a Gauss-Newton algorithm, which converges faster than a gradient-descent algorithm
(optional, default = 1001)

Example:

```
% Model input/output data of a Hammerstein system
N = 2e3; % Number of samples
u = randn(N,1); % Input signal
f_NL = @(x) x + 0.2*x.^2 + 0.1*x.^3; % Nonlinear function
```

```
[b,a] = cheby1(2,5,2*0.3); % Filter coefficients
x = f_NL(u); % Intermediate signal
y = filter(b,a,x); % Output signal
scale = u\x; % Scale factor
sys = ss(tf(scale*b,a,[])); % Initial linear model = scale factor
    times underlying dynamics
nx = [2 3]; % Quadratic and cubic terms in state equation
ny = [2 3]; % Quadratic and cubic terms in output equation
T1 = 0; % No periodic signal transient handling
T2 = 200; % Number of transient samples to discard
model = fCreateNLSSmodel(sys.a,sys.b,sys.c,sys.d,nx,ny,T1,T2); %
    Initial linear model
model.xactive = fSelectActive('inputsonly',2,1,2,nx); % A
    Hammerstein system only has nonlinear terms in the input
model.yactive = fSelectActive('inputsonly',2,1,1,nx); % A
    Hammerstein system only has nonlinear terms in the input
MaxCount = 50; % Maximum number of iterations
W = []; % No weighting
[modelOpt,yOpt] = fLMnlssWeighted(u,y,model,MaxCount,W); %
    Optimized model and modeled output
t = 0:N-1;
figure
        plot(t,y,'b')
        hold on
        plot(t,yOpt,'r')
        xlabel('Time')
        ylabel('Output')
        legend('True','Modeled')
```

A.4.2 `fLMnlssWeighted_x0u0`

Optimize PNLSS model and initial conditions using weighted Levenberg-Marquardt algorithm.

Usage:

```
[model,y_mod,models,Cost] = fLMnlssWeighted_x0u0(u,y,model,MaxCount,
W,lambda,LambdaJump)
[model,y_mod,models,Cost] = fLMnlssWeighted_x0u0(u,y,model,MaxCount,
W)
```

Description:

`fLMnlssWeighted_x0u0` performs a Levenberg-Marquardt optimization on the parameters of a PNLSS model (i.e. the elements of the matrices `A`, `B`, `C`, `D`, `E`, and `F`) and on the initial conditions (i.e. the initial state `x0` $= x(0)$ and the initial input `u0` $= u(0)$). The difference between the modeled and the measured output is minimized in a weighted least squares sense, either in the time or the frequency domain. A simple stabilization method can be applied by simulating a validation data set during estimation and checking whether or not the modeled output stays within prespecified bounds. If not, the Levenberg-Marquardt iteration acts as if the cost function increased.

Output parameters:

`model`	optimized model (= best on estimation data)
`y_mod`	output of the optimized model
`models`	collection of models (initial model + model after a successful iteration)
`Cost`	collection of the unweighted rms error at each iteration (`NaN` if iteration was not successful (i.e. when the weighted rms error increased))

Input parameters:

`u` `N` × `m` input signal

`y` `N` × `p` output signal

`model` initial model (see `fCreateNLSSmodel`) with extra fields

- `x0` initial value of the initial state
  (optional, default = `zeros(n,1)`)
- `u0` initial value of the initial input
  (optional, default = `zeros(m,1)`)
- `x0active` vector with the indices of the active elements (i.e. elements that will be optimized) in `x0`
  (optional, default = `[]`)
- `u0active` vector with the indices of the active elements (i.e. elements that will be optimized) in `u0`
  (optional, default = `[]`)

Additionally, two extra optional fields can be added to the model to perform a simple stabilization method (both fields are needed to perform this method):

- `u_val` `m` × `N_val` matrix with `N_val` samples of the validation input
  (optional, no default value)
- `max_out` bound on the maximum absolute value of the simulated validation output (optional, no default value)

After each Levenberg-Marquardt iteration, the validation data is simulated (without taking into account the transient settings). If the simulated validation output does not respect the `max_out` bound, then the iteration is considered unsuccessful.

`MaxCount` (maximum) number of iterations (there is not yet an early stopping criterion)

`W` `p` × `p` × `NFD` weighting matrix if frequency-domain weighting (e.g. square root of covariance matrix of output noise spectrum), where `NFD` is the number of frequency bins in the positive half of the spectrum (of one period and one phase realization) of the input (e.g. `NFD = floor(Npp/2)`, where `Npp` is the number of samples in one period and one phase realization for a multisine excitation).
`N` × `p` weighting sequences if time-domain weighting.
`[]` if no weighting.
(optional, default is no weighting)

`lambda` initial Levenberg-Marquardt parameter
(optional, default = 0, which corresponds to a Gauss-Newton algorithm). After a successful iteration, `lambda` is halved. After an unsuccessful iteration, `lambda` is multiplied with a factor $\sqrt{10}$, unless `lambda` was zero, in which case `lambda` is put equal to the dominant singular value of the Jacobian.

`LambdaJump` each `LambdaJump` iterations, the Levenberg-Marquardt parameter is made smaller by a factor 10, so that the algorithm leans more towards a Gauss-Newton algorithm, which converges faster than a gradient-descent algorithm
(optional, default = 1001)

Example:

```
% Model input/output data of a Hammerstein system with non-zero
    initial conditions
N = 200; % Number of samples
NTrans = 100; % Number of samples after zero initial conditions
u = randn(NTrans+N,1); % Input signal
f_NL = @(x) x + 0.2*x.^2 + 0.1*x.^3; % Nonlinear function
[b,a] = cheby1(2,5,2*0.3); % Filter coefficients
x = f_NL(u); % Intermediate signal
y = filter(b,a,x); % Output signal
u(1:NTrans) = []; % Remove first NTrans samples to obtain non-zero
    intial conditions
x(1:NTrans) = []; % Remove first NTrans samples to obtain non-zero
    intial conditions
y(1:NTrans) = []; % Remove first NTrans samples to obtain non-zero
    intial conditions
scale = u\x; % Scale factor
sys = ss(tf(scale*b,a,[])); % Initial linear model = scale factor
    times underlying dynamics
nx = [2 3]; % Quadratic and cubic terms in state equation
ny = [2 3]; % Quadratic and cubic terms in output equation
T1 = 0; % No periodic signal transient handling
T2 = []; % No transient samples to discard
model = fCreateNLSSmodel(sys.a,sys.b,sys.c,sys.d,nx,ny,T1,T2); %
    Initial linear model
model.xactive = fSelectActive('inputsonly',2,1,2,nx); % A
    Hammerstein system only has nonlinear terms in the input
model.yactive = fSelectActive('inputsonly',2,1,1,nx); % A
    Hammerstein system only has nonlinear terms in the input
MaxCount = 50; % Maximum number of iterations
W = []; % No weighting
[modelOpt,yOpt] = fLMnlssWeighted(u,y,model,MaxCount,W); %
    Optimized model and modeled output (without estimating initial
    conditions)
model_x0u0 = model; % Estimate initial conditions
model_x0u0.x0active = (1:model_x0u0.n).'; % Estimate initial
    conditions
model_x0u0.u0active = (1:model_x0u0.m).'; % Estimate initial
    conditions
[modelOpt_x0u0,yOpt_x0u0] = fLMnlssWeighted_x0u0(u,y,model_x0u0,
    MaxCount,W); % Optimized model and modeled output (initial
    conditions estimated);
t = 0:N-1;
figure
        plot(t,y,'b')
```

```
        hold on
        plot(t,y-yOpt,'r')
        plot(t,y-yOpt_x0u0,'g')
        xlabel('Time')
        ylabel('Output / output error')
        legend('True','Error PNLSS','Error PNLSS (initial
            conditions estimated)')
```

A.4.3 `fComputeJF`

Compute Jacobian of `F*eta` w.r.t. active terms in `eta`.

Usage:

```
JF = fComputeJF(p,yactive,n_ny,eta)
```

Description:

Computes the Jacobian `JF` of the active terms in `F*eta`, where `eta` is a `n_ny` $\times$ `N` matrix that contains `N` samples of `n_ny` known signals (typically nonlinear terms in states and inputs), and where `F` is a `p` $\times$ `n_ny` matrix containing the coefficients of these known signals for all `p` outputs. The indices of the active terms in `F` are indicated in `yactive`.

Output parameters:

`JF` `p*N` $\times$ `n_ny` Jacobian matrix with the derivatives of `F*eta` w.r.t. the active terms in `F`

Input parameters:

`p`	number of outputs
`yactive`	linear indices (see also `sub2ind` and `ind2sub`) of the active elements in `transpose(F)`
`n_ny`	number of terms in `eta`
`eta`	`n_ny` $\times$ `N` matrix with `N` samples of `n_ny` known signals

Example:

```
F = [1 2 3; 4 5 6]; % F matrix with coefficients
eta = [1:10:91; 2:10:92; 3:10:93]; % 10 samples of 3 known signals
p = size(F,1); % Number of outputs
yactive = [1:4 6]; % Fifth element in F not active (i.e. will not
    be optimized)
n_ny = size(eta,1); % Number of signals in eta
JF = fComputeJF(p,yactive,n_ny,eta); % Jacobian w.r.t. active
    elements in F
% => JF = [eta.'       zeros(10,2);
%          zeros(10,3) eta([1 3],:).'];
%       = [derivative of y1 w.r.t. [F(1,1) F(1,2) F(1,3) F(2,1) F
    (2,3)];
%          derivative of y2 w.r.t. [F(1,1) F(1,2) F(1,3) F(2,1) F
    (2,3)]];
```

A.4.4 `fEdwdx`

Multiply a matrix `E` with the derivative w.r.t. `x` of a polynomial `w(x,u)`.

Usage:

```
out = fEdwdx(contrib,pow,coeff,E,nx,n)
```

Description:

Multiplies a matrix `E` with the derivative of a polynomial `w(x,u)` w.r.t. the `n` elements in `x`. The samples of `x` and `u` are in a vector `contrib`. The derivative of `w(x,u)` w.r.t. `x` is given by the exponents in `x` and `u` (given in `pow`) and the corresponding coefficients (given in `coeff`). The maximum degree of a variable (an `x` or a `u`) in `w(x,u)` is given in `nx`.

Output parameters:

`out` `n_out` × `n` × `N` matrix that is the product of `E` and the derivative of the polynomial `w(x,u)` w.r.t. the elements in `x` at all samples.

Input parameters:

`contrib` `(n+m)` × `N` matrix containing `N` samples of the signals `x` and `u`

`pow` `n_nx` × `(n+m)` × `(n+m)` containing the exponents of the derivatives of `w(x,u)` w.r.t. `x` and `u`, i.e. `pow(i,j,k)` contains the exponent of `contrib j` in the derivative of the `i`th monomial w.r.t. `contrib k`.

`coeff` `n_nx` × `(n+m)` matrix containing the corresponding coefficients, i.e. `coeff(i,k)` contains the coefficient of the derivative of the `i`th monomial in `w(x,u)` w.r.t. `contrib k`.

`E` `n_out` × `n_nx` matrix

`nx` maximum degree of a variable (an `x` or a `u`) in `w(x,u)`

`n` number of `x` signals w.r.t. which derivatives are taken

Example:

```
% Consider w(x1,x2,u) = [x1^2    and E = [1 3 5
%                        x1*x2;           2 4 6]
%                        x2*u^2]
% then the derivatives of E*w w.r.t. x1 and x2 are given by
% E*[2*x1 0
%    1*x2 1*x1
%    0    1*u^2]
% and the derivative of w w.r.t. u is given by [0
%                                               0
%                                               2*x2*u]
E = [1 3 5; 2 4 6];
```

```
pow = zeros(3,3,3);
pow(:,:,1) = [1 0 0;
              0 1 0;
              0 0 0]; % Derivative w.r.t. x1 has terms 2*x1, 1*x2,
                  and 0
pow(:,:,2) = [0 0 0;
              1 0 0;
              0 0 2]; % Derivative w.r.t. x2 has terms 0, 1*x1, and
                   1*u^2
pow(:,:,3) = [0 0 0;
              0 0 0;
              0 1 1]; % Derivative w.r.t. u has terms 0, 0, and 2*
                  x2*u
coeff = [2 0 0;
         1 1 0;
         0 1 2];
nx = 2; % Maximum second degree factor in monomials of w (x1^2 in
    first monomial, u^2 in third monomial)
n = 2; % Two signals x
contrib = randn(3,10); % Ten random samples of signals x1, x2, and
    u
out = fEdwdx(contrib,pow,coeff,E,nx,n);
% => out(:,:,t) = E*[2*contrib(1,t) 0
%                    1*contrib(2,t) 1*contrib(1,t)
%                    0              1*contrib(3,t)^2]
```

A.4.5 fEdwdu

Multiply a matrix E with the derivative w.r.t. u of a polynomial w(x,u).

Usage:

```
out = fEdwdu(contrib,pow,coeff,E,nx,n)
```

Description:

Multiplies a matrix E with the derivative of a polynomial w(x,u) w.r.t. the m elements in u. The samples of x and u are in a vector contrib. The derivative of w(x,u) w.r.t. u is given by the exponents in x and u (given in pow) and the corresponding coefficients (given in coeff). The maximum degree of a variable (an x or a u) in w(x,u) is given in nx.

Output parameters:

out	n_out × m × N matrix that is the product of E and the derivative of the polynomial w(x,u) w.r.t. the elements in u at all samples.

Input parameters:

contrib	(n+m) × N matrix containing N samples of the signals x and u
pow	n_nx × (n+m) × (n+m) containing the exponents of the derivatives of w(x,u) w.r.t. x and u, i.e. pow(i,j,k) contains the exponent of contrib j in the derivative of the ith monomial w.r.t. contrib k.
coeff	n_nx × (n+m) matrix containing the corresponding coefficients, i.e. coeff(i,k) contains the coefficient of the derivative of the ith monomial in w(x,u) w.r.t. contrib k.
E	n_out × n_nx matrix
nx	maximum degree of a variable (an x or a u) in w(x,u)
m	number of u signals w.r.t. which derivatives are taken

Example:

```
% Consider w(x1,x2,u) = [x1^2    and E = [1 3 5
%                        x1*x2;           2 4 6]
%                        x2*u^2]
% then the derivatives of E*w w.r.t. x1 and x2 are given by
% E*[2*x1 0
%    1*x2 1*x1
%    0    1*u^2]
% and the derivative of w w.r.t. u is given by [0
%                                               0
%                                               2*x2*u]
E = [1 3 5; 2 4 6];
```

```
pow = zeros(3,3,3);
pow(:,:,1) = [1 0 0;
              0 1 0;
              0 0 0]; % Derivative w.r.t. x1 has terms 2*x1, 1*x2,
                  and 0
pow(:,:,2) = [0 0 0;
              1 0 0;
              0 0 2]; % Derivative w.r.t. x2 has terms 0, 1*x1, and
                   1*u^2
pow(:,:,3) = [0 0 0;
              0 0 0;
              0 1 1]; % Derivative w.r.t. u has terms 0, 0, and 2*
                  x2*u
coeff = [2 0 0;
         1 1 0;
         0 1 2];
nx = 2; % Maximum second degree factor in monomials of w (x1^2 in
    first monomial, u^2 in third monomial)
m = 1; % One signal u
contrib = randn(3,10); % Ten random samples of signals x1, x2, and
    u
out = fEdwdu(contrib,pow,coeff,E,nx,m);
% => out(:,:,t) = E*[0
%                    0
%                    2*contrib(2,t)*contrib(3,t)]
```

A.4.6 fJNL

Compute Jacobian w.r.t. A, B, and E by filtering an alternative state-space model.

Usage:

```
out = fJNL(input,Edwdx,C,Fdwdx,active)
```

Description:

Computing the Jacobian of the output y(t) of a nonlinear state-space model

$$\begin{aligned} \texttt{x(t+1)} &= \texttt{A x(t)} + \texttt{B u(t)} + \texttt{E zeta(x(t),u(t))} \\ \texttt{y(t)} &= \texttt{C x(t)} + \texttt{D u(t)} + \texttt{F eta(x(t),u(t))} \end{aligned}$$

w.r.t. the elements in the A, B, and E matrices can be performed by filtering an alternative nonlinear state-space model.
Let JA(i,j)(t) be the partial derivative of y(t) w.r.t. A(i,j) and let xA(i,j)(t) be the partial derivative of x(t) w.r.t. A(i,j). Similarly, consider JB(i,j)(t), JE(i,j)(t), xB(i,j)(t), and xE(i,j)(t) to be the partial derivative of y(t) and x(t) w.r.t. B(i,j) and E(i,j).
Then the Jacobians JA, JB, and JE can be computed by filtering three state-space models Paduart [2008]:

$$\begin{aligned} \texttt{xA(i,j)(t+1)} &= \texttt{A xA(i,j)(t)} + \texttt{I(i,j)x(t)} + \texttt{E}\frac{\partial \texttt{zeta(t)}}{\partial \texttt{x(t)}}\texttt{xA(i,j)(t)} \\ \texttt{JA(i,j)(t)} &= \texttt{C xA(i,j)(t)} + \texttt{F}\frac{\partial \texttt{eta(t)}}{\partial \texttt{x(t)}}\texttt{xA(i,j)(t)} \end{aligned}$$

$$\begin{aligned} \texttt{xB(i,j)(t+1)} &= \texttt{A xB(i,j)(t)} + \texttt{I(i,j)u(t)} + \texttt{E}\frac{\partial \texttt{zeta(t)}}{\partial \texttt{x(t)}}\texttt{xB(i,j)(t)} \\ \texttt{JB(i,j)(t)} &= \texttt{C xB(i,j)(t)} + \texttt{F}\frac{\partial \texttt{eta(t)}}{\partial \texttt{x(t)}}\texttt{xB(i,j)(t)} \end{aligned}$$

$$\begin{aligned} \texttt{xE(i,j)(t+1)} &= \texttt{A xE(i,j)(t)} + \texttt{I(i,j)zeta(t)} + \texttt{E}\frac{\partial \texttt{zeta(t)}}{\partial \texttt{x(t)}}\texttt{xE(i,j)(t)} \\ \texttt{JE(i,j)(t)} &= \texttt{C xE(i,j)(t)} + \texttt{F}\frac{\partial \texttt{eta(t)}}{\partial \texttt{x(t)}}\texttt{xE(i,j)(t)} \end{aligned}$$

where I(i,j) is a matrix of the appropriate dimensions with all zeros, except in position (i,j), where I(i,j)= 1 (see also fOne).
fJNL computes the Jacobian w.r.t. either A, B, or E, depending on the provided input (x(t), u(t), or zeta(t)).

Output parameters:

`out` `p`×`N`×`nactive` Jacobian matrix with the partial derivatives of `y(t)` w.r.t. the active elements in `A`, `B`, or `E` (depending on the provided `input`), where `p` is the number of outputs, `N` is the number of samples, and `nactive` is the number of active elements in `A`, `B`, or `E`.

Input parameters:

`input` `N` × `npar` matrix with the input samples in the alternative state-space model (`input` = states `x(t)` and `npar` = `n` if `JA` is calculated, `input` = input `u(t)` and `npar` = `m` if `JB` is calculated, and `input` = `zeta(t)` and `npar` = `n_nx` if `JE` is calculated).

`Edwdx` `n` × `n` × `N` matrix with `N` samples of $\mathtt{A} + \mathtt{E}\frac{\partial \mathtt{zeta(t)}}{\partial \mathtt{x(t)}}$

`C` `p` × `n` matrix `C`

`Fdwdx` `p` × `n` × `N` matrix with `N` samples of $\mathtt{F}\frac{\partial \mathtt{eta(t)}}{\partial \mathtt{x(t)}}$

`active` `nactive` × `1` vector with the linear indices (see also `sub2ind` and `ind2sub`) in `A.'`, `B.'`, or `E.'` indicating the active elements of `A`, `B`, or `E` (active elements = elements on which optimization will be done, see also `fSelectActive`).

A.4.7 fJx0

Compute Jacobian w.r.t. x0 by filtering an alternative state-space model.

Usage:

```
out = fJx0(A_Edwdx_0,Edwdx,C,Fdwdx,active)
```

Description:

Computing the Jacobian of the output y(t) of a nonlinear state-space model

$$\begin{aligned} \texttt{x(t+1)} &= \texttt{A x(t)} + \texttt{B u(t)} + \texttt{E zeta(x(t),u(t))} \\ \texttt{y(t)} &= \texttt{C x(t)} + \texttt{D u(t)} + \texttt{F eta(x(t),u(t))} \end{aligned}$$

w.r.t. the elements in the initial state x0 can be performed by filtering an alternative nonlinear state-space model.
Let Jx0(i)(t) be the partial derivative of y(t) w.r.t. x0(i) and let xx0(i)(t) be the partial derivative of x(t) w.r.t. x0(i).
Then the Jacobian Jx0 can be computed by filtering an alternative state-space model

$$\begin{aligned} \texttt{xx0(i)(t+1)} &= \texttt{A xx0(i)(t)} + \texttt{E}\frac{\partial \texttt{zeta(t)}}{\partial \texttt{x(t)}}\texttt{xx0(i)(t)} \\ \texttt{Jx0(i)(t)} &= \texttt{C xx0(i)(t)} + \texttt{F}\frac{\partial \texttt{eta(t)}}{\partial \texttt{x(t)}}\texttt{xx0(i)(t)} \end{aligned}$$

with initial state xx0(i)(0)= I(i,1)

where I(i,1) is a n × 1 vector with all zeros, except in position (i,1), where I(i,1)= 1 (see also fOne).

Output parameters:

out p × N × nactive Jacobian matrix with the partial derivatives of y(t) w.r.t. the active elements in x0, where p is the number of outputs, N is the number of samples, and nactive is the number of active elements in x0.

Input parameters:

A_Edwdx_0 n × n matrix $\texttt{A} + \texttt{E}\frac{\partial \texttt{zeta(0)}}{\partial \texttt{x0}}$

Edwdx n × n × N matrix with N samples of $\texttt{A} + \texttt{E}\frac{\partial \texttt{zeta(t))}}{\partial \texttt{x(t)}}$

C p × n matrix C

Fdwdx p × n × N matrix with N samples of $\texttt{F}\frac{\partial \texttt{eta(t)}}{\partial \texttt{x(t)}}$

active nactive × 1 vector with the indices in x0 indicating the active elements of x0 (active elements = elements on which optimization will be done).

A.4.8 fJu0

Compute Jacobian w.r.t. u0 by filtering an alternative state-space model.

Usage:

```
out = fJu0(B_EdwxIdu0,Edwdx,C,Fdwdx,active)
```

Description:

Computing the Jacobian of the output y(t) of a nonlinear state-space model

$$\begin{aligned} \texttt{x(t+1)} &= \texttt{A x(t)} + \texttt{B u(t)} + \texttt{E zeta(x(t),u(t))} \\ \texttt{y(t)} &= \texttt{C x(t)} + \texttt{D u(t)} + \texttt{F eta(x(t),u(t))} \end{aligned}$$

w.r.t. the elements in the initial input u0 can be performed by filtering an alternative nonlinear state-space model.
Let Ju0(i)(t) be the partial derivative of y(t) w.r.t. u0(i) and let xu0(i)(t) be the partial derivative of x(t) w.r.t. u0(i).
Then the Jacobian Ju0 can be computed by filtering an alternative state-space model

$$\begin{aligned} \texttt{xu0(i)(t+1)} &= \texttt{A xu0(i)(t)} + (\texttt{B I(i,1)} + \texttt{E}\frac{\partial \texttt{zeta(0)}}{\partial \texttt{u0(i)}})\texttt{delta(t,0)} \\ &\quad + \texttt{E}\frac{\partial \texttt{zeta(t)}}{\partial \texttt{x(t)}}\texttt{xu0(i)(t)} \\ \texttt{Ju0(i)(t)} &= \texttt{C xu0(i)(t)} \\ &\quad + \texttt{F}\frac{\partial \texttt{eta(t)}}{\partial \texttt{x(t)}}\texttt{xu0(i)(t)} \end{aligned}$$

where I(i,1) is a m × 1 vector with all zeros, except in position (i,1), where I(i,1)= 1 (see also fOne). The Kronecker delta delta(t,0) is one if t=0 and zero otherwise.

Output parameters:

out p × N × nactive Jacobian matrix with the partial derivatives of y(t) w.r.t. the active elements in u0, where p is the number of outputs, N is the number of samples, and nactive is the number of active elements in u0.

Input parameters:

B_EdwxIdu0	n × m matrix $\mathtt{B} + \mathtt{E}\frac{\partial \mathtt{zeta(0)}}{\partial \mathtt{u0}}$
Edwdx	n × n × N matrix with N samples of $\mathtt{A} + \mathtt{E}\frac{\partial \mathtt{zeta(t)}}{\partial \mathtt{x(t)}}$
C	p × n matrix C
Fdwdx	p × n × N matrix with N samples of $\mathtt{F}\frac{\partial \mathtt{eta(t)}}{\partial \mathtt{x(t)}}$
active	nactive × 1 vector with the indices in u0 indicating the active elements of u0 (active elements = elements on which optimization will be done).

A.4.9 `sJacobianAnalytical`

Script to compute the Jacobians in a nonlinear state-space model.

Description:

`sJacobianAnalytical` is a script that computes the steady-state Jacobians of a nonlinear state-space model

$$\begin{aligned} \texttt{x(t+1)} &= \texttt{A x(t)} + \texttt{B u(t)} + \texttt{E zeta(x(t),u(t))} \\ \texttt{y(t)} &= \texttt{C x(t)} + \texttt{D u(t)} + \texttt{F eta(x(t),u(t))} \end{aligned}$$

i.e. the partial derivatives of the modeled output w.r.t. the active elements in the `A`, `B`, `E`, `F`, `D`, and `C` matrices.
This script is called in `fLMnlssWeighted`.

A.4.10 `sJacobianAnalytical_x0u0`

Script to compute the Jacobians in a nonlinear state-space model.

Description:

`sJacobianAnalytical_x0u0` is a script that computes the steady-state Jacobians of a nonlinear state-space model

$$\begin{aligned} \texttt{x(t+1)} &= \texttt{A x(t)} + \texttt{B u(t)} + \texttt{E zeta(x(t),u(t))} \\ \texttt{y(t)} &= \texttt{C x(t)} + \texttt{D u(t)} + \texttt{F eta(x(t),u(t))} \end{aligned}$$

i.e. the partial derivatives of the modeled output w.r.t. the active elements in the `A`, `B`, `E`, `F`, `D`, and `C` matrices, and the initial states and inputs `x0` and `u0`. This script is called in `fLMnlssWeighted_x0u0`.

A.5 Model construction and simulation

Functions in this category:

Function	Section
`fCreateNLSSmodel`	A.5.1
`fSelectActive`	A.5.2
`fSScheckDims`	A.5.3
`fFilterNLSS`	A.5.4
`fFilterspeedNL`	A.5.5

Before the initialization with the linear subspace model can be optimized, this model should be put in PNLSS format. This can be done with the function `fCreateNLSSmodel`. The dimensions of the provided A, B, C, and D natrices are checked for consistency with the function `fSScheckDims`. By default, `fCreateNLSSmodel` creates a PNLSS model where the matrices E and F are intialized to zero matrices of the appropriate dimensions, and where all of their elements are set free for optimization. If you only want a subset of these parameters to be free for optimization (e.g. only nonlinear terms in the states but not in the inputs), you can use the function !fSelectActive!.

Once a PNLSS model is constructed, it can be simulated using the function `fFilterNLSS`. The function `fFilterspeedNL` is an auxiliary function for the function `fFilterNLSS`.

A.5.1 `fCreateNLSSmodel`

Create polynomial nonlinear state-space model from initial linear state-space model.

Usage:

```
model = fCreateNLSSmodel(A,B,C,D,nx,ny,T1,T2,sat)
```

Description:

Create a polynomial nonlinear state-space model from a linear initialization with state-space matrices `A`, `B`, `C`, and `D`. The state equation is extended with a multivariate polynomial in the states and the inputs. The nonlinear degree(s) of this polynomial is/are specified in `nx`. Similarly, the output equation is extended with a multivariate polynomial, where the degrees are specified in `ny`. The transient handling is reflected in `T1` (for periodic data) and `T2` (for aperiodic data). A saturation nonlinearity instead of a polynomial one is obsoleted; the optional parameter `sat` should be zero if specified.

Output parameters:

`model` structure containing the parameters and relevant data of the polynomial nonlinear state-space model. This structure has the following fields:

- `A` `n` × `n` state matrix
- `B` `n` × `m` input matrix
- `C` `p` × `n` output matrix
- `D` `p` × `m` feed-through matrix
- `lin.A` `n` × `n` state matrix of the linear initialization
- `lin.B` `n` × `m` input matrix of the linear initialization
- `lin.C` `p` × `n` output matrix of the linear initialization
- `lin.D` `p` × `m` feed-through matrix of the linear initialization
- `nx` vector with nonlinear degrees in state update
- `ny` vector with nonlinear degrees in output equation
- `n` number of states
- `m` number of inputs
- `p` number of outputs
- `xpowers` `n_nx` × `(n+m)` matrix containing the exponents of each of the `n_nx` monomials in the state update (see also `fCombinations`)
- `n_nx` number of monomials in the state update
- `E` `n` × `n_nx` matrix with polynomial coefficients in the state update
- `xactive` linear indices of the active elements in the transpose of the `E` matrix (active elements = elements on which optimization will be done). By default, all elements in the `E` matrix are set as active. See `fSelectActive` to change this property.
- `ypowers` `n_ny` × `(n+m)` matrix containing the exponents of each of the `n_ny` monomials in the output equation (see also `fCombinations`)
- `n_ny` number of monomials in the output equation
- `F` `p` × `n_ny` matrix with polynomial coefficients in the output equation
- `yactive` linear indices of the active elements in the transpose of the `F` matrix (active elements = elements on which optimization will be done). By default, all elements in the `F` matrix are set as active. See `fSelectActive` to change this property.
- `T1` vector that indicates how the transient is handled for periodic signals (see also the Input parameters)
- `T2` scalar indicating how many samples from the start are removed or vector indicating which samples are removed (see also `fComputeIndicesTransientRemovalArb`)
- `sat` obsolete, zero flag
- `satCoeff` obsolete, `n` × `1` vector of ones

Input parameters:

`A`	`n` × `n` state matrix
`B`	`n` × `m` input matrix
`C`	`p` × `n` output matrix
`D`	`p` × `m` feed-through matrix
`nx`	vector with nonlinear degrees in state equation
`ny`	vector with nonlinear degrees in output equation
`T1`	vector that indicates how the transient is handled for periodic signals. The first element `T1(1)` is the number of transient samples that should be prepended to each input realization. The other elements `T1(2:end)` indicate the starting sample of each realization in the signal. If `T1` has only one element, `T1(2)` is put to one (see also `fComputeIndicesTransient`).
`T2`	scalar indicating how many samples from the start are removed or vector indicating which samples are removed (see also `fComputeIndicesTransientRemovalArb`)
`sat`	obsolete, `sat` should be put equal to zero (optional, default is zero)

Example:

```
n = 3; % Number of states
m = 1; % Number of inputs
p = 1; % Number of outputs
sys = drss(n,p,m); % Random linear state-space model
nx = [2 3]; % Quadratic and cubic terms in state equation
ny = [2 3]; % Quadratic and cubic terms in output equation
T1 = 0; % No transient handling
T2 = []; % No transient handling
sat = 0; % Obsolete parameter sat = 0
model = fCreateNLSSmodel(sys.a,sys.b,sys.c,sys.d,nx,ny,T1,T2,sat);
    % Linear state-space model
N = 1e3; % Number of samples
u = randn(N,1); % Input signal
y = fFilterNLSS(model,u); % Modeled output signal
t = 0:N-1; % Time vector
y_lsim = lsim(sys,u,t); % Alternative way to calculate output of
    linear state-space model
figure
        plot(t,y_lsim,'b')
        hold on
        plot(t,y,'r')
        xlabel('Time')
        ylabel('Output')
        legend('lsim','PNLSS')
```

A.5.2 fSelectActive

Select active elements in E or F matrix.

Usage:

```
active = fSelectActive(structure,n,m,q,nx)
```

Description:

Select the active elements (i.e. those on which optimization will be done) in the E or F matrix. In particular, the linear indices (see also sub2ind and ind2sub) of the active elements in the transpose of the E or F matrix are calculated.

Output parameters:

`active` linear indices of the active elements in the transpose of the E or F matrix

Input parameters:

`structure` string indicating which elements in the E or F matrix are active. The possibilities are `'diagonal'`, `'inputsonly'`, `'statesonly'`, `'nocrossprod'`, `'affine'`, `'affinefull'`, `'full'`, `'empty'`, `'nolastinput'`, or `num2str(row_E)`. Below is an explanation of each of these structures:

- `'diagonal'` active elements in row j of the E matrix are those corresponding to pure nonlinear terms in state j (only for state equation)
- `'inputsonly'` only terms in inputs
- `'statesonly'` only terms in states
- `'nocrossprod'` no cross-terms
- `'affine'` only terms that are linear in one state
- `'affinefull'` only terms that are linear in one state or constant in the states
- `'full'` all terms
- `'empty'` no terms
- `'nolastinput'` no terms in last input (implemented since version 1.1)
- `num2str(row_E)` only row `row_E` in E matrix is active (only for state equation)

`n` number of states

`m` number of inputs

`q` number of rows in corresponding E/F matrix
`q = n` if E matrix is considered,
`q = p` if F matrix is considered

`nx` degrees of nonlinearity in E/F matrix

Example:

```
n = 2; % Number of states
m = 1; % Number of inputs
p = 1; % Number of outputs
nx = 2; % Degree(s) of nonlinearity
terms = fCombinations(n+m,nx); % Powers of all possible terms in n+
    m inputs of degree(s) nx
% => terms = [2 0 0;
%             1 1 0;
%             1 0 1;
%             0 2 0;
%             0 1 1;
%             0 0 2];
% There are six quadratic terms in the two states x1 and x2, and
% the input u, namely x1^2, x1*x2, x1*u, x2^2, x2*u, and u^2.
% The matrix E is a 2 x 6 matrix that contains the polynomial
% coefficients in each of these 6 terms for both state updates.
% The active elements will be calculated as linear indices in the
% transpose of E, hence E can be represented as
% E = [e1 e2 e3 e4  e5  e6;
%      e7 e8 e9 e10 e11 e12];
% The matrix F is a 1 x 6 matrix that contains the polynomial
% coefficients in each of the 6 terms for the output equation.
% The matrix F can be represented as
% F = [f1 f2 f3 f4 f5 f6];

% Diagonal structure
activeE = fSelectActive('diagonal',n,m,n,nx);
% => activeE = [1 10].';
% Only e1 and e10 are active. This corresponds to a term x1^2 in
% the first state equation and a term x2^2 in the second state
% equation.

% Inputs only structure
activeE = fSelectActive('inputsonly',n,m,n,nx);
% => activeE = [6 12].';
% Only e6 and e12 are active. This corresponds to a term u^2 in
% both state equations. In all other terms, at least one of the
% states (possibly raised to a certain power) is a factor.
activeF = fSelectActive('inputsonly',n,m,p,nx);
% => activeF = 6;
% Only f6 is active. This corresponds to a term u^2 in the output
% equation.

% States only structure
```

```
activeE = fSelectActive('statesonly',n,m,n,nx);
% => activeE = [1 2 4 7 8 10].';
% Only e1, e2, e4, e7, e8, and e10 are active. This corresponds to
% terms x1^2, x1*x2, and x2^2 in both state equations. In all other
% terms, the input (possibly raised to a certain power) is a
% factor.

% No cross products structure
activeE = fSelectActive('nocrossprod',n,m,n,nx);
% => activeE = [1 4 6 7 10 12].';
% Only e1, e4, e6, e7, e10, and e12 are active. This corresponds to
% terms x1^2, x2^2, and u^2 in both state equations. All other
% terms are crossterms where more than one variable is present as a
% factor.

% State affine structure
activeE = fSelectActive('affine',n,m,n,nx);
% => activeE = [3 5 9 11].';
% Only e3, e5, e9, and e11 are active. This corresponds to
% terms x1*u and x2*u in both state equations, since in these terms
% only one state appears, and it appears linearly.

% Full state affine structure
activeE = fSelectActive('affinefull',n,m,n,nx);
% => activeE = [3 5 6 9 11 12].';
% Only e3, e5, e6, e9, e11, and e12 are active. This corresponds to
% terms x1*u, x2*u and u^2 in both state equations, since in these
% terms at most one state appears, and if it appears, it appears
% linearly.

% Full structure
activeE = fSelectActive('full',n,m,n,nx);
% => activeE = (1:12).';
% All elements in the E matrix are active.

% Empty structure
activeE = fSelectActive('empty',n,m,n,nx);
% => activeE = [];
% None of the elements in the E matrix are active.

% One row in E matrix structure
row_E = 2; % Select which row in E is active
activeE = fSelectActive('row_E',n,m,n,nx);
% => activeE = [7 8 9 10 11 12].';
% Only the elements in the second row of E are active

```

```
% No terms in last input structure
% This is useful in a polynomial nonlinear state-space (PNLSS)
% model when considering the initial state as a parameter. The
% state at time one can be estimated by adding an extra input
% u_art(t) that is equal to one at time zero and zero elsewhere.
% Like this, an extended PNLSS model is estimated, where the last
% column in its B matrix corresponds to the state at time one in
% the original PNLSS model. To ensure that the optimization is only
% carried out on the parameters of the original PNLSS model, only
% the corresponding coefficients in the E/F matrix should be
% selected as active.
terms_extended = fCombinations(n+m+1,nx); % Powers of all possible
    terms with one extra input
% => terms_extended = [2 0 0 0;
%                      1 1 0 0;
%                      1 0 1 0;
%                      1 0 0 1;
%                      0 2 0 0;
%                      0 1 1 0;
%                      0 1 0 1;
%                      0 0 2 0;
%                      0 0 1 1;
%                      0 0 0 2];
% The nonlinear terms in the extra input should not be considered
% for optimization.
activeE_extended = fSelectActive('nolastinput',n,m+1,n,nx);
% => activeE_extended = [1 2 3 5 6 8 11 12 13 15 16 18].';
% Only the terms where the last input is raised to a power zero are
% active. This corresponds to the case where all terms in the
% original PNLSS model are active.
% The example below illustrates how to combine a certain structure
% in the original model (e.g. 'nocrossprod') with the estimation of
% the initial state.
activeE_extended = fSelectActive('nolastinput',n,m+1,n,nx);
activeE_extended = activeE_extended(fSelectActive('nocrossprod',n,m
    ,n,nx));
% => activeE_extended = [1 5 8 11 15 18].';
% This corresponds to the terms x1^2, x2^2, and u1^2 in both rows
% of the E_extended matrix, and thus to all terms in the original
% model, except for the crossterms.
% Note that an alternative approach is to include the initial state
% in the parameter vector (see also fLMnlssWeighted_x0u0).
```

A.5.3 `fSScheckDims`

Check consistency of state-space dimensions.

Usage:

```
[n,m,p] = fSScheckDims(A,B,C,D)
```

Description:

Returns the number of states, inputs, and outputs of a linear state-space model and checks if the sizes of the state-space matrices are consistent. Produces an error message if they are not consistent.

Output parameters:

`n` model order
`m` number of inputs
`p` number of outputs

Input parameters:

`A` $\mathtt{n} \times \mathtt{n}$ state matrix
`B` $\mathtt{n} \times \mathtt{m}$ input matrix
`C` $\mathtt{p} \times \mathtt{n}$ output matrix
`D` $\mathtt{p} \times \mathtt{m}$ feed-through matrix

A.5.4 `fFilterNLSS`

Calculate the output and the states of a nonlinear state-space model with transient handling.

Usage:

```
[y,states] = fFilterNLSS(model,u,x0,u0)%implemented since version 1.1
[y,states] = fFilterNLSS(model,u,x0)
[y,states] = fFilterNLSS(model,u)
y = fFilterNLSS(model,u)
```

Description:

`y = fFilterNLSS(model,u)` calculates the output `y` of a nonlinear state-space model by applying an input `u` and starting from a zero initial state `x0` and a zero initial input `u0`.
`y = fFilterNLSS(model,u,x0)` starts from an initial state `x0`.
`y = fFilterNLSS(model,u,x0,u0)` starts from an initial input `u0` (implemented since version 1.1).
`[y,states] = fFilterNLSS(model,u,x0, u0)` also passes the states at each time step.

Output parameters:

`y`	`N` $\times$ `p` matrix with `N` samples of the `p` outputs
`states`	`N` $\times$ `n` matrix with `N` samples of the `n` states

Input parameters:

`model`	PNLSS model (see also `fCreateNLSSmodel`)
`u`	`m` $\times$ `N` matrix with `N` samples of the `m` inputs If `u` is an `N` $\times$ `m` matrix, then `u` is transposed.
`x0`	`n` $\times$ `1` vector with the initial states (optional, default is zero initial states)
`u0`	`m` $\times$ `1` vector with the initial inputs (optional, default is zero initial inputs, implemented since version 1.1)

Example:

```
n = 3; % Number of states
m = 1; % Number of inputs
p = 1; % Number of outputs
sys = drss(n,p,m); % Random linear state-space model
nx = [2 3]; % Quadratic and cubic terms in state equation
ny = [2 3]; % Quadratic and cubic terms in output equation
T1 = 0; % No transient handling
```

```
T2 = []; % No transient handling
sat = 0; % Obsolete parameter sat = 0
model = fCreateNLSSmodel(sys.a,sys.b,sys.c,sys.d,nx,ny,T1,T2,sat);
    % Linear state-space model
N = 1e3; % Number of samples
u = randn(N,1); % Input signal
y = fFilterNLSS(model,u); % Modeled output signal
t = 0:N-1; % Time vector
y_lsim = lsim(sys,u,t); % Alternative way to calculate output of
    linear state-space model
figure
        plot(t,y_lsim,'b')
        hold on
        plot(t,y,'r')
        xlabel('Time')
        ylabel('Output')
        legend('lsim','PNLSS')
```

A.5.5 fFilterspeedNL

Calculate the output and the states of a nonlinear state-space model without transient handling.

Usage:

```
[y,states] = fFilterspeedNL(A,B,C,D,E,F,xpowers,ypowers,max_nx,max_ny
,u,x0,u0)%implemented since version 1.1
[y,states] = fFilterspeedNL(A,B,C,D,E,F,xpowers,ypowers,max_nx,max_ny
,u,x0)
```

Description:

Calculate the output and the states of a nonlinear state-space model

$$\texttt{x(t+1)} = \texttt{A x(t)} + \texttt{B u(t)} + \texttt{E zeta(x(t),u(t))}$$
$$\texttt{y(t)} = \texttt{C x(t)} + \texttt{D u(t)} + \texttt{F eta(x(t),u(t))}$$

where `zeta` and `eta` are polynomials whose exponents are given in `xpowers` and `ypowers`, respectively. The maximum degree in one variable (a state or an input) in `zeta` or `eta` is given in `max_nx` and `max_ny`, respectively. The initial state is given in `x0`.
This is a low-level function that is used in `fFilterNLSS`.

Output parameters:

`y`	`p` × `N` matrix with `N` samples of the `p` outputs
`states`	`n` × `N` matrix with `N` samples of the `n` states

Input parameters:

`A`	`n` × `n` state matrix
`B`	`n` × `m` input matrix
`C`	`p` × `n` output matrix
`D`	`p` × `m` feed-through matrix
`E`	`n` × `nzeta` matrix with polynomial coefficients in the state equation
`F`	`p` × `neta` matrix with polynomial coefficients in the output equation
`xpowers`	`nzeta` × `(n+m)` matrix with the exponents in the states and the inputs for each of the `nzeta` terms
`ypowers`	`neta` × `(n+m)` matrix with the exponents in the states and the inputs for each of the `neta` terms
`max_nx`	maximum degree in one variable in `zeta`
`max_ny`	maximum degree in one variable in `eta`
`u`	`m` × `N` matrix with `N` samples of the `m` inputs
`x0`	`n` × `1` vector with the initial state
`u0`	`m` × `1` vector with the initial input (optional, default is zero initial inputs, implemented since version 1.1)

Example:

```
% A PNLSS model with only input nonlinearities has an equivalent
% linear state-space model structure with an extended input vector.
% Output of a PNLSS model with only input nonlinearities
sys = drss(2,1,1); % Second-order SISO discrete-time state-space
    model
set(sys,'Ts',1); % Set unit sampling time
[A,B,C,D] = ssdata(sys); % Linear state-space matrices
xpowers = [0 0 2;
           0 0 3]; % Quadratic and cubic input term in state update
ypowers = [0 0 2;
           0 0 3]; % Quadratic and cubic input term in output
               equation
E = [1 2;
     3 4]; % Monomial coefficients in state update
F = [1 2]; % Monomial coefficients in output equation
max_nx = 3; % Maximum degree of a monomial in state update
max_ny = 3; % Maximum degree of a monomial in output equation
u = randn(1,1000); % Input signal
x0 = [1; 2]; % Start from non-zero initial state
y = fFilterspeedNL(A,B,C,D,E,F,xpowers,ypowers,max_nx,max_ny,u,x0);
     % Compute output
% Output corresponding linear state-space model with extended input
     vector
u_ext = [u(:) u(:).^2 u(:).^3]; % Extended input vector
B_ext = [B E]; % Extended input matrix
D_ext = [D F]; % Extended feed-through matrix
t = 0:999; % Time vector
y_lsim = lsim(ss(A,B_ext,C,D_ext,1),u_ext,t,x0); % Output extended
    linear state-space model
% Compare the PNLSS and lsim approach to calculate the output
figure
        plot(t,y_lsim,'b')
        hold on
        plot(t,y,'r')
        xlabel('Time')
        ylabel('Output')
        legend('lsim','PNLSS')
```

A.6 Utility

Functions in this category:

Function	Section
`fHerm`	A.6.1
`fMetricPrefix`	A.6.2
`fNormalizeColumns`	A.6.3
`fOne`	A.6.4
`fPlotFrfMIMO`	A.6.5
`fReIm`	A.6.6
`fSqrtInverse`	A.6.7
`fVec`	A.6.8
`fCombinations`	A.6.9
`fTermNL`	A.6.10

This section collects the functions that are used throughout the PNLSS identification process, but that don't belong to one particular step in this process.

The function `fHerm` computes the average of a square matrix and its Hermitian transpose, which is useful when estimating the noise covariance matrix in `fCovarFrf` (see Section A.2.1).

The function `fMetricPrefix` returns an appropriate metric prefix and a corresponding scaling factor to display large (or small) quantities, e.g. 10000 Hz can also be displayed as 10 kHz.

The function `fNormalizeColumns` normalizes the columns of a matrix to have unit rms value. This can be useful to improve the condition number of a Jacobian matrix.

The function `fOne` constructs a matrix with one one, and zeros elsewhere. This is useful when computing the Jacobians of a nonlinear state-space model.

The function `fPlotFrfMIMO` makes amplitude versus frequency plots of the elements of a frequency response matrix . This is useful to display the results of the subspace method.

The function `fReIm` stacks the real and imaginary part of a matrix on top of each other. This can be useful to force real parameters when estimating them using a linear least-squares method, while the regressor matrix is complex.

The function `fSqrtInverse` computes a matrix, which when multiplied with itself, produces the inverse of a specified matrix. This is useful when computing a frequency domain weighting matrix starting from the noise covariance matrix.

The function `fVec` vectorizes a matrix or tensor by placing all its columns on top of each other.

The function `fCombinations` lists the exponents of all nonlinear terms of a certain degree (or certain degrees) in a multivariate polynomial. This is useful when constructing a PNLSS model.

The function `fTermNL` computes the outputs of a multivariate polynomial starting from the inputs and the list of exponents.

A.6.1 `fHerm`

Average of square matrix and its Hermitian transpose.

Usage:

```
B = fHerm(A)
```

Description:

Computes the average of a square matrix `A` and its complex conjugate transpose `A'`.

Output parameters:

`B` average of `A` and `A'`

Input parameters:

`A` square matrix

A.6.2 fMetricPrefix

Returns an appropriate metric prefix and a corresponding scaling factor.

Usage:

```
[label,scale] = fMetricPrefix(in)
```

Description:

Returns a metric prefix label and the appropriate scaling factor with which `in` has to be multiplied in order to use the prefix. The supported prefixes are:

Prefix	Label	Scale	Range
	`''`	1	$\texttt{in} \leq 10^{-16.5}$
femto	`'f'`	10^{15}	$10^{-16.5} < \texttt{in} \leq 10^{-13.5}$
pico	`'p'`	10^{12}	$10^{-13.5} < \texttt{in} \leq 10^{-10.5}$
nano	`'n'`	10^{9}	$10^{-10.5} < \texttt{in} \leq 10^{-7.5}$
micro	`'\mu'`	10^{6}	$10^{-7.5} < \texttt{in} \leq 10^{-4.5}$
milli	`'m'`	10^{3}	$10^{-4.5} < \texttt{in} \leq 10^{-1.5}$
	`''`	1	$10^{-1.5} < \texttt{in} < 10^{1.5}$
kilo	`'k'`	10^{-3}	$10^{1.5} \leq \texttt{in} < 10^{4.5}$
mega	`'M'`	10^{-6}	$10^{4.5} \leq \texttt{in} < 10^{7.5}$
giga	`'G'`	10^{-9}	$10^{7.5} \leq \texttt{in} < 10^{10.5}$
tera	`'T'`	10^{-12}	$10^{10.5} \leq \texttt{in} < 10^{13.5}$
	`''`	1	$10^{13.5} \leq \texttt{in}$

Output parameters:

`label` prefix label
`scale` scaling factor

Input parameters:

`in` number

Example:

```
% 10000 Hz can also be displayed as 10 kHz
in = 10000;
[label,scale] = fMetricPrefix(in);
disp([num2str(in*scale) ' ' label 'Hz'])
```

A.6.3 fNormalizeColumns

Normalizes the columns of a matrix with their rms value.

Usage:

```
[Jn,scaling] = fNormalizeColumns(J)
```

Description:

Normalizes the columns of a matrix (e.g. a Jacobian) to have unit rms value. Zero columns are unchanged, and get assigned unit scaling.

Output parameters:

`Jn`	matrix with normalized columns
`scaling`	rms values of the columns of `J`

Input parameters:

`J`	matrix with unnormalized columns

Example:

```
% If J is a regression matrix, its columns can be scaled to
% possibly obtain a better conditioned least-squares problem. The
% scaling factors can then be used to compute the unnormalized
% parameter values.
N = 100; % Number of data samples
J = [9e14*randn(N,1) randn(N,1)]; % Badly conditioned regression
    matrix
theta = [2; 3]; % Parameter vector
y = J*theta; % Output vector
[Jn,scaling] = fNormalizeColumns(J); % Normalize columns J
theta_n = (Jn\y)./scaling(:); % Estimated parameter vector from
    normalized regressors
theta_u = J\y; % Estimated parameter vector from unnormalized
    regressors
```

A.6.4 fOne

Constructs a matrix with only one one, and zeros elsewhere.

Usage:

```
out = fOne(p,m,i)
```

Description:

Constructs a `p` × `m` matrix with a one in position `i`, and zeros elsewhere.
This is useful in constructing the Jacobians of the error `e(f)= G_hat(f)− G(f)` w.r.t. the elements of a state-space matrix (`A`, `B`, `C`, or `D`, but typically `D`, since the Jacobian w.r.t. the elements of `D` is a zero matrix where one element is one), where `G_hat(f)= C*(z(f)*I − A)^(−1)*B + D`.

Output parameters:

`out` `p` × `m` matrix with a one in position `i` (`out(i)= 1`), and zeros elsewhere

Input parameters:

`p` number of rows in `out`
`m` number of columns in `out`
`i` position at which `out` has a one (`out(i)= 1`, or `out(k,l)= 1`, where `i = sub2ind([p m],k,l)`)

Example:

```
1 out = fOne(2,3,3); % A 2 x 3 zero matrix with a one at position 3
2 % => out = [0 1 0;
3 %           0 0 0];
```

A.6.5 `fPlotFrfMIMO`

Make amplitude versus frequency plots of the elements of a frequency response matrix.

Usage:

```
fPlotFrfMIMO(G,freq)
fPlotFrfMIMO(G,freq,LineSpec)
fPlotFrfMIMO(G,freq,LineSpec,Name,Value,...)
```

Description:

`fPlotFrfMIMO` makes an amplitude versus frequency plot for each of the components of the frequency response matrix `G`.

Remark

`fPlotFrfMIMO` does not make a call to `figure`, so `figure` should be called before `fPlotFrfMIMO` if you want to make sure that the plot is made on a new figure. `fPlotFrfMIMO` makes a call to `hold on`, so that consecutive calls of `fPlotFrfMIMO` makes amplitude versus frequency plots of frequency response matrices (with the same number of inputs and outputs) on top of each other.

Input parameters:

`G`	`p` $\times$ `m` $\times$ `F` frequency response matrix (FRM)
`freq`	vector of frequencies at which the FRM is given (in Hz)
`LineSpec`	sets the line style, marker symbol, and color. (optional, default = `'b'`)
`Name`, `Value` pair arguments	see `help plot` for a list

A.6.6 `fReIm`

Stacks the real and imaginary part of a matrix on top of each other.

Usage:

```
B = fReIm(A)
```

Description:

Stacks the real and imaginary part of matrix `A` on top of each other, i.e. `B = [real(A); imag(A)]`.
To stack the real and imaginary part next to each other, i.e. `[real(A)imag(A)]`, use `B = fReIm(A.').'`

Output parameters:

`B` real matrix with real and imaginary part of `A` stacked on top of each other

Input parameters:

`A` complex matrix

A.6.7 `fSqrtInverse`

Computes `B`, such that `B*B = inv(A)`.

Usage:

```
B = fSqrtInverse(A)
```

Description:

`B = fSqrtInverse(A)` computes the inverse of the matrix square root of a square positive definite matrix `A`.

Output parameters:

`B` inverse of matrix square root of `B`

Input parameters:

`A` positive definite matrix

Example:

```
V = orth(randn(3,3)); % Random eigenvectors
D = diag(1+abs(randn(3,1))); % Random positive eigenvalues
A = V*D*V.'; % Positive definite random square matrix
B = fSqrtInverse(A);
disp('B*B'), B*B
disp('inv(A)'), inv(A)
```

A.6.8 `fVec`

Vectorization of a matrix or tensor.

Usage:

```
out = fVec(in)
```

Description:

Stacks all the columns of a matrix/tensor on top of each other.

Output parameters:

`out` `numel(in)` $\times$ `1` vector with all columns of `in` stacked on top of each other

Input parameters:

`in` matrix or tensor

A.6.9 `fCombinations`

Lists all nonlinear terms in a multivariate polynomial.

Usage:

```
out = fCombinations(n,degrees)
```

Description:

Lists the exponents of all possible monomials in a multivariate polynomial with `n` inputs. Only the nonlinear degrees in `degrees` are considered.

Output parameters:

`out` `ncomb` × `n` matrix of exponents

Input parameters:

`n`	number of inputs
`degrees`	vector with the degrees of nonlinearity

Example:

```
% A polynomial with all possible quadratic and cubic terms in the
% variables x and y contains the monomials x*x, x*y, y*y, x*x*x,
% x*x*y, x*y*y, and y*y*y.
out = fCombinations(2,[2 3])
% => out = [2 0;        -> x^2 * y^0 = x*x
%           1 1;        -> x^1 * y^1 = x*y
%           0 2;        -> x^0 * y^2 = y*y
%           3 0;        -> x^3 * y^0 = x*x*x
%           2 1;        -> x^2 * y^1 = x*x*y
%           1 2;        -> x^1 * y^2 = x*y*y
%           0 3]        -> x^0 * y^3 = y*y*y
% Element (i,j) of out indicates the power to which variable j is
% raised in monomial i. For example, out(5,:) = [2 1], which means
% that the fifth monomial is equal to x^2*y^1 = x*x*y.
```

A.6.10 `fTermNL`

Construct polynomial terms.

Usage:

```
out = fTermNL(contrib,pow,max_degree)
```

Description:

`out = fTermNL(contrib,pow,max_degree)` computes polynomial terms, where `contrib` contains the input signals to the polynomial and `pow` contains the exponents of each term in each of the inputs. The maximum degree of an individual input is given in `max_degree`.

Output parameters:

`out`	`nterms` × `N` matrix with `N` samples of each term

Input parameters:

`contrib`	`(n+m)` × `N` matrix with `N` samples of the input signals to the polynomial. Typically, these are the `n` states and the `m` inputs of the nonlinear state-space model.
`pow`	`nterms` × `(n+m)` matrix with the exponents of each term in each of the inputs to the polynomial.
`max_degree`	maximum degree in an individual input of the polynomial

Example:

```
n = 2; % Number of states
m = 1; % Number of inputs
N = 1000; % Number of samples
x = randn(n,N); % States
u = randn(m,N); % Input
contrib = [x; u]; % States and input combined
pow = [2 0 0;
       1 1 0;
       1 0 1;
       0 2 0;
       0 1 1;
       0 0 2]; % All possible quadratic terms in states and input:
           x1^2, x1*x2, x1*u, x2^2, x2*u, u^2
max_degree = max(max(pow)); % Maximum degree in an individual state
     or input
out = fTermNL(contrib,pow,max_degree);
% => out = [x(1,:).^2;
```

```
%           x(1,:).*x(2,:);
%           x(1,:).*u;
%           x(2,:).^2;
%           x(2,:).*u;
%           u.^2];
```

References

Ljung, L. (1999). *System identification: Theory for the User*. (2nd ed.). Upper Saddle River, NJ: Prentice-Hall.

Marconato, A., Sjöberg, J., Suykens, A., Johan, & Schoukens, J. (2014). Improved initialization for nonlinear state-space modeling. *IEEE Transactions on Instrumentation and Measurement*, *63*, 972–980.

McKelvey, T., Akçay, H., & Ljung, L. (1996). Subspace-based multivariable system identification from frequency response data. *IEEE Transactions on Automatic Control*, *41*, 960–979.

Paduart, J. (2008). *Identification of nonlinear systems using Polynomial Nonlinear State Space models*. Ph.D. thesis Vrije Universiteit Brussel.

Paduart, J., Lauwers, L., Swevers, J., Smolders, K., Schoukens, J., & Pintelon, R. (2010). Identification of nonlinear systems using Polynomial Nonlinear State Space models. *Automatica*, *46*, 647–656.

Pintelon, R. (2002). Frequency-domain subspace system identification using non-parametric noise models. *Automatica*, *38*, 1295–1311.

Pintelon, R., & Schoukens, J. (2012). *System Identification: A Frequency Domain Approach*. (2nd ed.). Wiley-IEEE Press.

Söderström, T., & Stoica, P. (1989). *System Identification*. Prentice Hall International (UK) Ltd.