SAF Library Manual

Version 1.1

DIET Department
"Sapienza" University of Rome
via Eudossiana 18, 00184 Roma

SAF LIBRARY MANUAL

Abstract

This manual is a brief introduction and description of Matlab code implementing all files need to create, use and adapt architectures based on a novel class of nonlinear adaptive filters, called ***Spline Adaptive Filter*** *or simply* ***SAF****. Many other source codes for comparisons purpose have been provided.*

This manual also includes several demo scripts, implementing many of the experimental results that can be find in the reference literature. The source code is released under the BSD-3 license.

Authors hope that this library could be helpful for the many researchers working in the field of nonlinear signal processing, in order to provide a simple and fast way to implement SAF algorithms for comparisons with their own approaches.

Authors will be extremely gratefully to ones who wish to contribute to this library by suggesting improvements on the codes or pointing out possible errors.

Rome, October 3, 2016

Michele Scarpiniti
Danilo Comminiello
Raffaele Parisi
Aurelio Uncini

For information, please contact: michele.scarpiniti@uniroma1.it

The BSD 3-Clause License

Contents

CHAPTER 1

Introduction

The SAF library provides a MATLAB implementation of the basic functions of the novel nonlinear Spline Adaptive Filter (SAF), introduced in the following recent works [6, 7, 8, 9, 10]. This kind of adaptive filter derives from the general family of block-oriented nonlinear architectures.

In addition, some scripts implementing a number of demo extracted from the experimental results of [6, 7, 8, 9, 10], were provided. The aims of these demo files is to show how the library functions can be used in practical implementations.

All the files are written in MATLAB language.

The next two chapters are dedicated to the description of the library functions and the demo scripts, respectively. The rest of this chapter provides an introduction on block-oriented nonlinear architectures, a brief summary of the SAF architectures and a description of the general idea of the proposed software implementation.

1.1 Introduction on block-oriented nonlinear architectures

In nonlinear filtering one of the most used structures is the so-called *block-oriented* representation, in which linear time invariant (LTI) models are connected with memoryless nonlinear functions. The basic classes of block-oriented nonlinear systems are represented by the Wiener and Hammerstein models and all those architectures originated by the connection of these two classes according to different topologies (i.e., parallel, cascade, feedback etc.). More specifically, the Wiener model consists of a cascade of an LTI filter followed by a static nonlinear function and sometimes it is known as linear-nonlinear (LN) model. Conversely, the Hammerstein model consists of a cascade connection of a static nonlinear function followed by an LTI filter and is sometimes indicated as nonlinear-linear (NL) model.

However, in many practical cases, due to the generality of the model to be identified, the Wiener and Hammerstein architectures might not be the best solutions. For this reason cascade models, such as the linear-nonlinear-linear (LNL) or

the nonlinear-linear-nonlinear (NLN) models, were also introduced [4]. Cascade models, here called *sandwich models*, constitute a relatively simple but important class of nonlinear models that can approximate a wide class of nonlinear systems.

One of the most well-known functional models is based on the Volterra series [5]. Unfortunately, due to the large number of free parameters required, the Volterra adaptive filter (VAF) is generally employed only in situations of mild nonlinearity.

1.1.1 Notation

In this manual matrices are represented by boldface capital letters, i.e. $\mathbf{A} \in \mathbb{R}^{M\times N}$. All vectors are column vectors, denoted by boldface lowercase letters, like $\mathbf{w} \in \mathbb{R}^{M\times 1} = [w[0], w[1], \ldots, w[M-1]]^T$, where $w[i]$ denotes the i-th individual entry of $\mathbf{w}$. In recursive algorithm definition, a discrete-time subscript index n is added. For example, the weight vector, calculated according to some law, is written as $\mathbf{w}_{n+1} = \mathbf{w}_n + \Delta\mathbf{w}_n$. In the case of signal regression, vectors are indicated as $\mathbf{x}_n \in \mathbb{R}^{M\times 1} = [x[n], x[n-1], \ldots, x[n-M+1]]^T$. Moreover, regarding the interpolated nonlinearity, the LUT control points are denoted as $\mathbf{q}_n \in \mathbb{R}^{Q\times 1} = [q_1, q_2, \ldots, q_Q]^T$, where q_j is the j-th control point, while the i-th span is denoted by $\mathbf{q}_{i,n} \in \mathbb{R}^{4\times 1} = [q_i, q_{i+1}, q_{i+2}, q_{i+3}]^T$.

The Euclidean norm and the transpose of a vector are denoted by $\|\cdot\|$ and $(\cdot)^T$, respectively.

1.1.2 The Wiener system

The Wiener model consists of a cascade of an LTI filter followed by a static nonlinear function, as shown in Figure 1.1. The output of the Wiener system can be

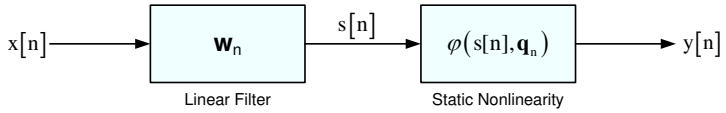

Fig. 1.1: Nonlinear Wiener model.

evaluated as follows:

$$\begin{aligned} y[n] &= \varphi\left(s[n]\right), \\ s[n] &= \mathbf{w}_n^T \mathbf{x}_n. \end{aligned} \tag{1.1}$$

1.1.3 The Hammerstein system

The Hammerstein model consists of a cascade of a static nonlinear function followed by an LTI filter, as shown in Figure 1.2. The output of the Hammerstein

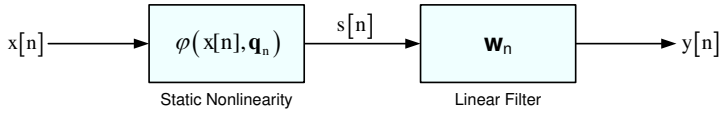

Fig. 1.2: Nonlinear Hammerstein model.

system can be evaluated as follows:

$$\begin{aligned} y[n] &= \mathbf{w}_n^T \mathbf{s}_n, \\ \mathbf{s}_n &= [s[n], s[n-1], \ldots, s[n-M+1]]^T, \\ s[n-k] &= \varphi\left(x[n-k]\right), \quad k = 0, 1, 2, \ldots, M-1. \end{aligned} \tag{1.2}$$

1.1.4 The Sandwich 1 system

The Sandwich 1 model consists of a cascade of a static nonlinear function, an LTI filter followed by a second static nonlinear function, as shown in Figure 1.3. The output of the Sandwich 1 system can be evaluated as follows:

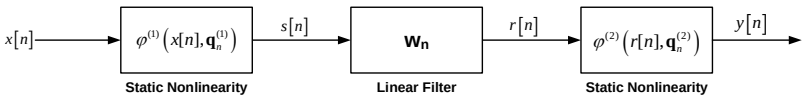

Fig. 1.3: Nonlinear Sandwich 1 model.

$$\begin{aligned} y[n] &= \varphi^{(2)}\left(r[n]\right), \\ r[n] &= \mathbf{w}_n^T \mathbf{s}_n, \\ \mathbf{s}_n &= [s[n], s[n-1], \ldots, s[n-M+1]]^T, \\ s[n-k] &= \varphi^{(1)}\left(x[n-k]\right), \quad k = 0, 1, 2, \ldots, M-1. \end{aligned} \tag{1.3}$$

1.1.5 The Sandwich 2 system

The Sandwich 2 model consists of a cascade of an LTI filter, a static nonlinear function, followed by a second LTI filter, as shown in Figure 1.4. The output of the

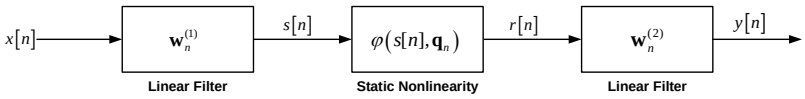

Fig. 1.4: Nonlinear Sandwich 2 model.

Sandwich 2 system can be evaluated as follows:

$$\begin{aligned} y[n] &= \mathbf{w}_n^{(2)T} \mathbf{r}_n, \\ \mathbf{r}_n &= [r[n], r[n-1], \ldots, r[n-M+1]]^T, \\ r[n-k] &= \varphi\left(s[n-k]\right), \quad k = 0, 1, 2, \ldots, M-1, \\ s[n-k] &= \mathbf{w}_n^{(1)T} \mathbf{x}_{n-k}, \quad k = 0, 1, 2, \ldots, M-1. \end{aligned} \tag{1.4}$$

1.2 Summary of SAF

Splines are smooth parametric curves defined by interpolation of properly defined control points, also called knots, collected in a lookup table (LUT). Let $y[n] = \varphi\left(s[n]\right)$ be some function to be estimated, the spline estimation provides

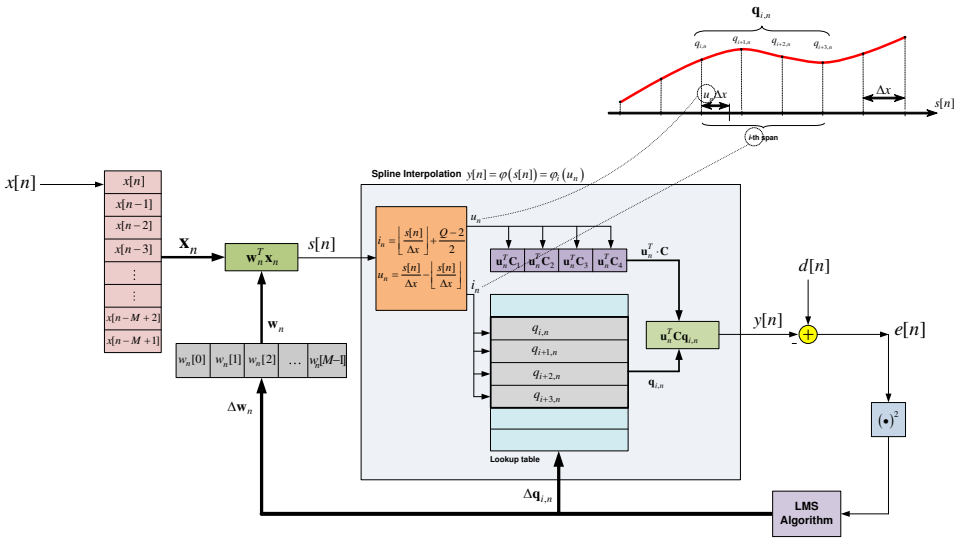

Fig. 1.5: Schematic structure of the Wiener SAF architecture adapted by the LMS approach. The vector $\mathbf{q}_{i,n}$ denotes the four LUT control points of the curve span addressed by the index i_n, while Δx is the uniform space between two consecutive knots. The output $y[n]$ is computed by considering the four control points $\mathbf{q}_{i,n}$, the spline basis matrix $\mathbf{C}$ and the span-abscissa u_n.

an approximation $\varphi_{i_n}(u_n)$ based on two parameters u_n and i_n directly depending on $s[n]$. In the general case, given Q equispaced control points, the spline curve results as a polynomial interpolation through $Q-1$ adjacent spans. In this specific application, we use cubic spline curves, so for each input occurrence $s[n]$ the spline employs four control points selected inside the LUT.

For simplicity, the rest of this section is referred to the particular case of a Wiener SAF. A similar reasoning can be made for the other SAF architectures.

With reference to Fig. 1.5, each spline span, controlled by four adjacent points of the LUT, is addressed by the so called span index i_n and, inside each span, a normalized local span-abscissa parameter $u_n \in [0,\ 1]$ is defined. These two latter parameters are evaluated as follows [6]

$$\begin{aligned} u_n &= \frac{s[n]}{\Delta x} - \left\lfloor \frac{s[n]}{\Delta x} \right\rfloor, \\ i_n &= \left\lfloor \frac{s[n]}{\Delta x} \right\rfloor + \frac{Q-1}{2}, \end{aligned} \tag{1.5}$$

where Δx is the uniform space between knots, $\lfloor \bullet \rfloor$ is the floor operator and Q is the total number of control points. For simplicity of notation, in the following we will use $i \equiv i_n$.

The schematic structure of the Wiener SAF is shown in Fig.1.5.

The complete expression of the spline interpolation is

$$y[n] = \varphi_i(u_n) = \mathbf{u}_n^{\mathrm{T}} \mathbf{C} \mathbf{q}_{i,n}, \tag{1.6}$$

where $\mathbf{u}_n \in \mathbb{R}^{4\times 1} = \begin{bmatrix} u_n^3 & u_n^2 & u_n & 1 \end{bmatrix}^{\mathrm{T}}$, $\mathbf{q}_{i,n} \in \mathbb{R}^{4\times 1} = \begin{bmatrix} q_i & q_{i+1} & q_{i+2} & q_{i+3} \end{bmatrix}^{\mathrm{T}}$

and $\mathbf{C} \in \mathbb{R}^{4\times4}$ is the spline basis matrix. Imposing different constraints to the approximation relationship, several spline basis with different properties can be evaluated in a similar manner. Examples of such a basis, suitable in signal processing applications, are Catmull-Rom (CR) spline and B-spline [6, 7]:

$$\mathbf{C}_{CR} = \frac{1}{2}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}, \tag{1.7}$$

$$\mathbf{C}_{B} = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}. \tag{1.8}$$

The CR-splines, initially developed for computer graphics purposes, represent a local interpolating scheme. CR-spline, in fact, specifies a curve that passes exactly through all of the control points, a feature which is not necessarily true for other spline methodologies, such as B-spline. Overall, the CR-spline results in a more local approximation with respect to the B-spline. However, both spline bases satisfy the *variation diminishing* property [1].

Denoting with $\mathbf{c}_k$ the k-th row of the $\mathbf{C}$ matrix, the expression of $\varphi_i(u_n)$ in (1.6) can be rewritten as

$$\begin{aligned} \varphi_i(u_n) &= u_n^3\mathbf{c}_1\mathbf{q}_{i,n} + u_n^2\mathbf{c}_2\mathbf{q}_{i,n} + u_n\mathbf{c}_3\mathbf{q}_{i,n} + \mathbf{c}_4\mathbf{q}_{i,n} \\ &\approx u_n\mathbf{c}_3\mathbf{q}_{i,n} + \mathbf{c}_4\mathbf{q}_{i,n}, \end{aligned} \tag{1.9}$$

where we have neglected the power terms with a degree greater or equal to two. This is due to the fact that the variable u_n is always inside the interval $[0, 1]$ and the same happens to the product $\mathbf{c}_k\mathbf{q}_{i,n}$ for the nonlinear functions normally used in this kind of applications.

Let us define the output error $e[n]$ as (see Figure 1.5)

$$e[n] = d[n] - y[n], \tag{1.10}$$

where $d[n]$ is the reference signal, i.e. the output signal of the system to be estimated including any possible noise. The on-line learning rule is derived by minimizing the cost function $\hat{J}(\mathbf{w}_n, \mathbf{q}_{i,n}) = E\{e^2[n]\}$. In order to derive the LMS algorithm, as usual, the expectation is approximated by the instantaneous error

$$J(\mathbf{w}_n, \mathbf{q}_{i,n}) = e^2[n]. \tag{1.11}$$

For the minimization of (1.11), we proceed by applying the *stochastic gradient* adaptation, which simply leads to the LMS iterative learning algorithms

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \mu_w[n]\frac{\varphi_i'(u_n)}{\Delta x}e[n]\mathbf{x}_n, \tag{1.12}$$

$$\mathbf{q}_{i,n+1} = \mathbf{q}_{i,n} + \mu_q[n]e[n]\mathbf{C}^{\mathrm{T}}\mathbf{u}_n, \tag{1.13}$$

where the parameters $\mu_w[n]$ and $\mu_q[n]$ represent the learning rates for the weights and for the control points respectively. Note that, in contrast to [6] where for simplicity the term $\mu_w[n]$ incorporates other constant values, in (1.12) we provide

the explicit dependence on the Δx parameter. In (1.12) the derivative of the i-th span $\varphi_i'(u_n)$ is evaluated as

$$\varphi_i'(u_n) = \dot{\mathbf{u}}_n^{\mathrm{T}} \mathbf{C} \mathbf{q}_{i,n}, \tag{1.14}$$

where $\dot{\mathbf{u}}_n \in \mathbb{R}^{4\times 1} = \begin{bmatrix} 3u_n^2 & 2u_n & 1 & 0 \end{bmatrix}^{\mathrm{T}}$. In addition, the following relation holds:

$$\varphi'\left(s[n]\right) = \frac{1}{\Delta x}\varphi_i'(u_n). \tag{1.15}$$

In a similar way it is possible to derive the second derivative of the span output:

$$\varphi_i''(u_n) = \ddot{\mathbf{u}}_n^{\mathrm{T}} \mathbf{C} \mathbf{q}_{i,n}, \tag{1.16}$$

where $\ddot{\mathbf{u}}_n \in \mathbb{R}^{4\times 1} = \begin{bmatrix} 6u_n & 2 & 0 & 0 \end{bmatrix}^{\mathrm{T}}$.

In the following, we list the adaptive algorithms for the different SAF models that have been implemented in the proposed MATLAB library.

1.2.1 Wiener SAF (WSAF)

By applying the stochastic gradient adaptation [6], it is easy to obtain the LMS iterative algorithm

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \mu_w[n]e[n]\varphi_i'(u_n)\mathbf{x}_n, \tag{1.17}$$

for the weights of the linear filter and

$$\mathbf{q}_{i,n+1} = \mathbf{q}_{i,n} + \mu_q[n]e[n]\mathbf{C}^T\mathbf{u}_n, \tag{1.18}$$

for the spline control points. The parameters $\mu_w[n]$ and $\mu_q[n]$, represent the learning rates for the weights and for the control points respectively and, for simplicity, incorporate the other constant values.

A summary of the proposed LMS algorithm for the nonlinear WSAF, in the case of constant learning rates, can be found in Algorithm 1.

Algorithm 1 – Summary of the WSAF-LMS algorithm

Initialize: $\mathbf{w}_{-1} = \delta[n]$, $\mathbf{q}_{-1}$

1: **for** $n = 0, 1, \ldots$ **do**
2: $\quad s[n] = \mathbf{w}_n^T\mathbf{x}_n$
3: $\quad u = s[n]/\Delta x - \lfloor s[n]/\Delta x \rfloor$
4: $\quad i = \lfloor s[n]/\Delta x \rfloor + (Q-1)/2$
5: $\quad y[n] = \mathbf{u}^T\mathbf{C}\mathbf{q}_{i,n}$
6: $\quad e[n] = d[n] - y[n]$
7: $\quad \mathbf{w}_{n+1} = \mathbf{w}_n + \mu_w e[n]\dot{\mathbf{u}}^T\mathbf{C}\mathbf{q}_{i,n}\mathbf{x}_n$
8: $\quad \mathbf{q}_{i,n+1} = \mathbf{q}_{i,n} + \mu_q e[n]\mathbf{C}^T\mathbf{u}$
9: **end for**

1.2.2 Hammerstein SAF (HSAF)

By applying the stochastic gradient adaptation as done in [7], it is possible to obtain the LMS iterative algorithm

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \mu_w[n]e[n]\mathbf{s}_n, \tag{1.19}$$

for the weights of the linear filter and

$$\mathbf{q}_{i,n+1} = \mathbf{q}_{i,n} + \mu_q[n]e[n]\mathbf{C}^T\mathbf{U}_{i,n}\mathbf{w}_n, \tag{1.20}$$

for the spline control points. In (1.20), $\mathbf{U}_{i,n} \in \mathbb{R}^{4\times M} = [\mathbf{u}_{i,n}, \mathbf{u}_{i,n-1}, \ldots, \mathbf{u}_{i,n-M+1}]$ is a matrix which collects M past vectors $\mathbf{u}_{i,n-k}$, where each vector assumes the value of $\mathbf{u}_{n-k}$ if evaluated in the same span i of the current sample input $x[n]$ or in an overlapped span, otherwise it is a zero vector. As in the Wiener case, the learning rates $\mu_w[n]$ and $\mu_q[n]$ incorporate the other constant values.

A summary of the proposed LMS algorithm for the nonlinear HSAF architecture, in the case of constant learning rates, can be found in Algorithm 2.

Algorithm 2 – Summary of the HSAF-LMS algorithm

Initialize: $\mathbf{w}_{-1} = \delta[n]$, $\mathbf{q}_{-1}$

for $n = 0, 1, \ldots$ **do**
- $u_n = x[n]/\Delta x - \lfloor x[n]/\Delta x \rfloor$
- $i = \lfloor x[n]/\Delta x \rfloor + (Q-1)/2$
- $s[n] = \mathbf{u}_n^T\mathbf{C}\mathbf{q}_{i,n}$
- $y[n] = \mathbf{w}_n^T\mathbf{s}_n$
- $e[n] = d[n] - y[n]$
- $\mathbf{w}_{n+1} = \mathbf{w}_n + \mu_w[n]e[n]\mathbf{s}_n$
- $\mathbf{U}_{i,n} = [\mathbf{u}_{i,n}, \mathbf{u}_{i,n-1}, \ldots, \mathbf{u}_{i,n-M+1}]$
- $\mathbf{q}_{i,n+1} = \mathbf{q}_{i,n} + \mu_q[n]e[n]\mathbf{C}^T\mathbf{U}_{i,n}\mathbf{w}_n$

end for

1.2.3 Sandwich 1 SAF (S1SAF)

By applying the stochastic gradient adaptation as done in [9], it is possible to obtain the LMS iterative algorithm

$$\mathbf{q}_{i,n+1}^{(2)} = \mathbf{q}_{i,n}^{(2)} + \mu_q^{(2)}[n]e[n]\mathbf{C}^T\mathbf{u}_n^{(2)}, \tag{1.21}$$

for the spline control points of the second nonlinearity.

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \mu_w[n]e[n]\varphi_i'(u_n^{(2)})\mathbf{s}_n, \tag{1.22}$$

for the weights of the linear filter and

$$\mathbf{q}_{i,n+1}^{(1)} = \mathbf{q}_{i,n}^{(1)} + \mu_q^{(1)}[n]e[n]\varphi_i'(u_n^{(2)})\mathbf{C}^T\mathbf{U}_{i,n}^{(1)}\mathbf{w}_n, \tag{1.23}$$

for the spline control points of the first nonlinearity. The parameters $\mu_w[n]$, $\mu_q^{(1)}[n]$ and $\mu_q^{(2)}[n]$ represent the learning rates for the weights and for the control points respectively and, for simplicity, incorporate the other constant values.

A summary of the proposed LMS algorithm for the nonlinear S1SAF architecture, in the case of using constant learning rates, can be found in Algorithm 3.

Algorithm 3 – Summary of the S1SAF-LMS algorithm

Initialize: $\mathbf{w}_{-1} = \alpha\boldsymbol{\delta}$, $\mathbf{q}_{-1}^{(1)}$, $\mathbf{q}_{-1}^{(2)}$

for $n = 0, 1, \ldots$ **do**

$u_n^{(1)} = x[n]/\Delta x^{(1)} - \lfloor x[n]/\Delta x^{(1)} \rfloor$

$i^{(1)} = \lfloor x[n]/\Delta x^{(1)} \rfloor + (Q^{(1)} - 1)/2$

$s[n] = \mathbf{u}_n^{(1)T}\mathbf{C}\mathbf{q}_{i,n}^{(1)}$

$r[n] = \mathbf{w}_n^T\mathbf{s}_n$

$u_n^{(2)} = r[n]/\Delta x^{(2)} - \lfloor r[n]/\Delta x^{(2)} \rfloor$

$i^{(2)} = \lfloor r[n]/\Delta x^{(2)} \rfloor + (Q^{(2)} - 1)/2$

$y[n] = \mathbf{u}_n^{(2)T}\mathbf{C}\mathbf{q}_{i,n}^{(2)}$

$e[n] = d[n] - y[n]$

$\mathbf{U}_{i,n}^{(1)} = \left[\mathbf{u}_{i,n}^{(1)}, \mathbf{u}_{i,n-1}^{(1)}, \ldots, \mathbf{u}_{i,n-M+1}^{(1)}\right]$

$\mathbf{q}_{i,n+1}^{(2)} = \mathbf{q}_{i,n}^{(2)} + \mu_q^{(2)} e[n]\mathbf{C}^T\mathbf{u}_n^{(2)}$

$\mathbf{w}_{n+1} = \mathbf{w}_n + \mu_w e[n]\varphi_i'(u_n^{(2)})\mathbf{s}_n$

$\mathbf{q}_{i,n+1}^{(1)} = \mathbf{q}_{i,n}^{(1)} + \mu_q^{(1)} e[n]\varphi_i'(u_n^{(2)})\mathbf{C}^T\mathbf{U}_{i,n}^{(1)}\mathbf{w}_n$

end for

1.2.4 Sandwich 2 SAF (S2SAF)

By applying the stochastic gradient adaptation as done in [9], it is possible to obtain the LMS iterative algorithm

$$\mathbf{w}_{n+1}^{(2)} = \mathbf{w}_n^{(2)} + \mu_w^{(2)}[n]e[n]\mathbf{r}_n, \tag{1.24}$$

for the weights of the second linear filter

$$\mathbf{q}_{i,n+1} = \mathbf{q}_{i,n} + \mu_q[n]e[n]\mathbf{C}^T\mathbf{U}_{i,n}\mathbf{w}_n^{(2)}. \tag{1.25}$$

for the spline control points of the nonlinearity, and

$$\mathbf{w}_{n+1}^{(1)} = \mathbf{w}_n^{(1)} + \mu_w^{(1)}[n]e[n]\varphi_i'(u_n)\mathbf{X}_n\mathbf{w}_n^{(2)}, \tag{1.26}$$

for the weights of the first linear filter. The parameters $\mu_w^{(1)}[n]$, $\mu_w^{(2)}[n]$ and $\mu_q[n]$ are the learning rates for the weights and for the control points respectively.

A summary of the proposed LMS algorithm for the nonlinear S2SAF architecture, in the case of constant learning rates, can be found in Algorithm 4.

1.3 The basic idea of the SAF Library

In a new software framework implementation, it is very important to focus on data structures and how this data are successfully processed.

Algorithm 4 – Summary of the S2SAF-LMS algorithm

Initialize: $\mathbf{w}_{-1}^{(1)} = \alpha\boldsymbol{\delta}$, $\mathbf{w}_{-1}^{(2)} = \alpha\boldsymbol{\delta}$, $\mathbf{q}_{-1}$

for $n = 0, 1, \ldots$ **do**

$s[n] = \mathbf{w}_n^{(1)T}\mathbf{x}_n$

$u_n = s[n]/\Delta x - \lfloor s[n]/\Delta x \rfloor$

$i = \lfloor s[n]/\Delta x \rfloor + (Q-1)/2$

$r[n] = \mathbf{u}_n^T\mathbf{C}\mathbf{q}_{i,n}$

$y[n] = \mathbf{w}_n^{(2)T}\mathbf{r}_n$

$e[n] = d[n] - y[n]$

$\mathbf{X}_n = [\mathbf{x}_n, \mathbf{x}_{n-1} \ldots, \mathbf{x}_{n-M_2+1}]$

$\mathbf{U}_{i,n} = [\mathbf{u}_{i,n}, \mathbf{u}_{i,n-1}, \ldots, \mathbf{u}_{i,n-M_2+1}]$

$\mathbf{w}_{n+1}^{(2)} = \mathbf{w}_n^{(2)} + \mu_w^{(2)} e[n]\mathbf{r}_n$

$\mathbf{q}_{i,n+1} = \mathbf{q}_{i,n} + \mu_q e[n]\mathbf{C}^T\mathbf{U}_{i,n}\mathbf{w}_n^{(2)}$

$\mathbf{w}_{n+1}^{(1)} = \mathbf{w}_n^{(1)} + \mu_w^{(1)} e[n]\varphi_i'(u_n)\mathbf{X}_n\mathbf{w}_n^{(2)}$

end for

The idea implemented in the SAF library is based on the *struct* data structure F, that collects all filter parameters:

```
F = struct('param1', p1, 'param2', p2, 'param3', p3, 'param4', p4, ...);
```

There is a MATLAB function dedicated to the creation of such a data structure and named as `create_` followed by a specific name. This function receives as input all the filter parameters (`p1`,`p2`,`p3`,`p4`,...) and returns the created structure `F` after an initialization of all dummy parameters:

```
F = create_myStruct(p1,p2,p3,p4);
```

The names given to different items (`param1`,`param2`,`param3`,`param4`,...) will depend on the type of chosen filter.

As an example, we report the function the creates an LMS SAF adaptive filter

```
function lms_af = create_SPL_lms_adaptive_filter_1(M, mu, mQ, delta, af)


% LMS SAF adaptive filter definition and initialization ------------------
w = zeros(M,1);     % Linear filter taps
w(floor(M/2)) = 1;  % Adaptive filter weights i.c.
dI = delta;         % Regularizing parameter
xd = zeros(M,1);    % Buffer of the desired signal
xw = zeros(M,1);    % Buffer of the filter status
%-------------------------------------------------------------------------
lms_af = struct('M',M, 'mu',mu,'mQ',mQ, 'dI',dI, 'w',w, 'xd',xd, ...
    'xw',xw, 'af',af);
%-------------------------------------------------------------------------
```

In the previous function, the input parameters are M (the length of the linear filter), μ (the learning rate of the linear filter), μ_q (the learning rate of the spline control

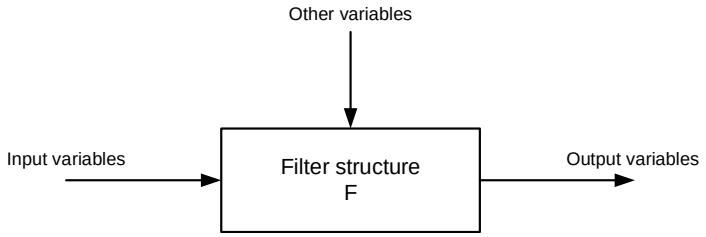

Fig. 1.6: Graphical idea of the proposed approach using a filter structure.

points) and `af`, that is a structure that collects all the parameters describing the spline nonlinearity. This last structure is created by the following

```
af = create_activation_function(afinit, aftype, DeltaX, Gain, Slope, M, ↩
    Pord);
```

in the same fashion of the general idea.

Then the `create_SPL_lms_adaptive_filter_1` function creates some items with the input variables and some internal variable, with suitable initializations, that can be used during the computation. In the considered example, the function defines the three additional vectors w (for the filter weights), xw (a buffer for implementing the input delay-line) and xd (a buffer for implementing the desired signal delay-line).

The just created data structure will be used as both input and output of the adaptive filter function (named `AF_`). In this way the AF function read all parameters from the F structure, update them if necessary, and the return the updated structure:

```
[F, output_param] = AF_myFunc(F, input_param, other_param);
```

The adaptive function `AF_myFucn()` reads the filter data structure F, the input parameters `input_param` and eventually other parameters `other_param`, and returns the updated data structure F and the output parameters `output_param`. Figure 1.6 shows a graphical idea of the proposed approach.

The same approach for the feed-forward function, named as `FW_` followed by an appropriate name:

```
[F, output_param] = FW_myFunc(F, input_param, other_param);
```

CHAPTER 2

SAF Library

This chapter is dedicated to the description of the SAF library functions, needed to implement all examples in the related papers [6, 7, 8, 9, 10]. All these function are implemented in MATLAB language. The chapter is divided into four sections.

2.1 Computing the spline nonlinearity and its derivative

This first section introduces the two MATLAB functions that implement the output of a spline function and its first derivative with respect its input sample.

2.1.1 ActFunc

This function evaluates the output of spline nonlinearity. More details can be found in [6].

The usage is as follows:

```
[x, af] = ActFunc(s, af);
```

where

- s is the nonlinearity input $s[n]$.
- af is the nonlinearity structure;
- x is the nonlinearity output $x[n]$.

The complete Matlab code is reported below.

```
function [x, af] = ActFunc(s, af)


af.s = s;    % Input of the nonlinearity
```

```
% Nonlinearity selection
switch (af.aftype)

  case -1  % Signed Sigmoidal
      x = (2*af.Gain/(1+exp(-s*af.Slope))-af.Gain);

  case  0  % Linear
      x = s*af.Slope; % Defaut value

  case  1  % Unsigned Sigmoidal
      x = af.Gain/(1+exp(-s*af.Slope));

  case  2  % Gaussian
      x = af.Gain*exp( -s^2/af.Slope );

  case  3  % Polynomial
      x = 0;
      for j=1:af.Pord,
          x = x + af.Q(j)*s.^j;  % Sum of monomials
          af.g(j) = s.^j;
      end

  case 20 % Quadratic spline
      np = af.lut_len;             % Number of control points
      Su = s/af.DeltaX + (np-1)/2; % First part of Eq. (7.b)
      uIndex = floor(Su);          % Span index i in Eq. (7.b)
      u = Su - uIndex;             % Local abscissa u in Eq. (7.a)
      if uIndex<1                  % The index must start from 1
          uIndex = 1;
      end
      if uIndex>(np-2),
          uIndex = np - 2;         % The index cannot exceed np - 2
      end
      af.g = [1 u u^2]*af.C;           % First part of Eq. (5): u^T C
      x = af.g*af.Q(uIndex : uIndex+2); % Eq. (5): u^T C q_i
      af.uIndex  = uIndex;             % For derivative computation
      af.uSpline = u;                  % For derivative computation

  otherwise  % Cubic spline
      np = af.lut_len;             % Number of control points
      Su = s/af.DeltaX + (np-1)/2; % First part of Eq. (7.b)
      uIndex = floor(Su);          % Span index i in Eq. (7.b)
      u = Su - uIndex;             % Local abscissa u in Eq. (7.a)
      if uIndex<1                  % The index must start from 1
          uIndex = 1;
      end
      if uIndex>(np-3),            % The index cannot exceed np - 3
          uIndex = np - 3;
      end
      af.g = [u^3 u^2 u 1]*af.C;       % First part of Eq. (5): u^T C
      x = af.g*af.Q(uIndex : uIndex+3); % Eq. (5): u^T C q_i
      af.uIndex  = uIndex;             % For derivative computation
      af.uSpline = u;                  % For derivative computation

end
```

2.1.2 dActFunc

This function evaluates the first derivative of a spline nonlinearity with respect its input. More details can be found in [6].

The usage is as follows:

```
dx = dActFunc(y, af);
```

where

- y is the nonlinearity input $y[n]$.
- af is the nonlinearity structure;
- dx is the derivative of the nonlinearity with respect its input.

The complete Matlab code is reported below.

```
function dx = dActFunc(y, af)


Sl = af.Slope;  % Slope
G  = af.Gain;   % Gain

% Nonlinearity selection
switch (af.aftype)

  case -1  % Signed Sigmoid
      dx = ( Sl/(2*G)*(G^2-y^2) );

  case  0  % Linear
      dx = af.Slope ;   % Default value

  case  1  % Unsigned Sigmoid
      dx = ( (Sl/G)*y*(G-y) );

  case  2   % Gaussian
      dx = (-2*G/Sl)*af.s*exp( -af.s^2 / Sl );

  case  3   % Polynomial
      dx = af.Q(1);
      for k=2:af.Pord,
          dx = dx + k*af.Q(k)*y.^(k-1);
      end

  case 20   % Quadratic spline
      [~,af] = ActFunc(y,af);
      uIndex  = af.uIndex;    % Local index i
      u  = af.uSpline;        % Local abscissa u
      g  = [0 1 2*u]*af.C;    % Dot u vector
      dx = g*af.Q( uIndex : uIndex+2 )/af.DeltaX;   % Eq. (6)

  otherwise   % Cubic spline
      [~,af] = ActFunc(y,af);
      uIndex  = af.uIndex;       % Local iindex i
      u  = af.uSpline;           % Local abscissa u
      g  = [3*u^2 2*u 1 0]*af.C; % Dot u vector
      dx = g*af.Q( uIndex : uIndex+3 )/af.DeltaX;   % Eq. (6)

end
```

2.2 Creating and initializing the filter structures

This second section is devoted to the description of the MATLAB functions that create the needed data structures for using them in SAF, as described in section 1.3.

2.2.1 create_activation_function

This function creates and initializes nonlinear function implemented by splines. More details can be found in [6, 7].

The usage is as follows:

```
AF = create_activation_function(afinit, aftype, DeltaX, Gain, Slope, M, ↩
    Pord);
```

where

- AF is the spline nonlinear structure;
- afinit is the type of initialization (linear, Gaussian, random,...). In particular:
  - -1: signed sigmoid;
  - 0: linear function;
  - 1: unsigned sigmoid;
  - 2: Gaussian;
  - 3: random.
- aftype is the type of spline basis (B, Catmull-Rom, Hermite,...). In particular:
  - -1: fixed signed sigmoid;
  - 0: fixed linear function;
  - 1: fixed unsigned sigmoid;
  - 2: fixed Gaussian;
  - 3: fexible polynomial function;
  - 4: fexible Catmull-Rom spline;
  - 5: fexible B-spline;
  - 6: flexible Bernstein spline;
  - 7: flexible parametric τ-spline;
  - 8: flexible Hermite spline;
  - 9: flexible Beziér spline;
  - 20: flexible quadratic B-spline.
- DeltaX is the space between knots Δ_x;
- Gain is the x-axis range limits;
- Slope is the initial slope;
- M is the length of the linear filter M;
- Pord is the polynomial order in case of polynomial nonlinearity P_{ord}.

The complete Matlab code is reported below.

```
function AF = create_activation_function(afinit, aftype, DeltaX, Gain, ↩
    Slope, M, Pord)


% Spline activation function definition and initialization ----------------
if nargin==0, help create_activation_function; return; end
if nargin < 7
    Pord = 3;
    if nargin<6
        M=1;
        if nargin<5
            Slope=2.16;
            if nargin<4
                Gain=1.1;
                if nargin<3
                    DeltaX=0.4;
                    if nargin<2
                        aftype=-1;
                        if nargin<1
                            afinit=-1;
                        end
                    end
                end
            end
        end
    end
end

%-------------------------------------------------------------------------
% Check fo nonlinearity type
if afinit == -1,
    Slope  = Slope*2*(1/Gain);
end

if afinit == 1,
    Slope  = Slope*4*(1/Gain);
end

if ((afinit<-1) || (afinit>3) )
    fprintf('Activation function error type\n');
    aftype = -1;
end

% LUT parameters ----------------------------------------------------------
Table_Length = TabFuncLen(DeltaX, Gain ,Slope, afinit );
lut_len      = Table_Length; % Length of LUT
s = 0.0;     % Linear combiner output
x = 0.0;     % Nonlinearity output
uIndex = 1;  % Span index
uSpline = 0; % Local abscissa

% Polynomial nonlinearity -------------------------------------------------
if (aftype == 3)
    Q = zeros(Pord,1);     % Polynomial nonlinearity
    Q(1) = 1;
    g = zeros(1,Pord);     % Dot u vector
    gM = zeros(Pord,M);    % U matrix (for Hammerstein filter) in [2]
else
    Q = zeros(lut_len,1);  % Look-Up Table nonlinearity
end
```

```
% If NOT spline ---------------------------------------------------------
if (aftype < 4)
    C = 0;
end;

% Catmul-Rom spline nonllinearity ----------------------------------------
if (aftype == 4)
C = 0.5*[-1  3 -3  1; ...
          2 -5  4 -1; ...
         -1  0  1  0; ...          % Page 775, top of column 1
          0  2  0  0];
end

% B spline nonllinearity -------------------------------------------------
if (aftype == 5)
    C = (1/6)*[-1  3 -3  1; ...
                3 -6  3  0; ...     % Page 774, bottom of column 2
               -3  0  3  0; ...
                1  4  1  0];
end

% Bernstein polynomial nonlinearity --------------------------------------
if (aftype == 6)
    C =       [-1  3 -3  1; ...
               3 -6  3  0; ...   % Bernstein polynomials
              -3  3  0  0; ...
               1  0  0  0];
end

% Parametric spline nonlinearity -----------------------------------------
if (aftype == 7)
     tau = 0.5;   % tau = 0.5 ==> CR-spline
     C =       [-tau   2-tau  tau-2   tau; ...
                2*tau tau-3  3-2*tau -tau; ...   % Parametric spline
               -tau   0      tau      0 ; ...
                0     1      0        0 ];
end

% Hermite spline nonlinearity --------------------------------------------
if (aftype == 8)
    C =       [ 2 -2  1  1; ...
              -3  3 -2 -1; ...   % Hermite polynomials
               0  0  1  0; ...
               1  0  0  0];
end

% Bezier spline nonllinearity --------------------------------------------
if (aftype == 9)
    C = (1/6)*[ 1  3 -3  1; ...
                3 -6  3  0; ...   % Bezier polynomials
               -3  3  0  0; ...
                1  0  0  0];
end

% Quadratic B-spline nonlinearity ----------------------------------------
if (aftype == 20)
     C =       [ 1  1  0; ...
                -2  2  0; ...
                 1 -2  1];
end
```

```
% Spline order ----------------------------------------------------------
P = length(C) - 1;

% Dummy arrays for learning algorithms. See Eqs. (5) and (6) -------------
if (aftype > 3)
    g = zeros(1,P+1);   % The dot u vector
    gM = zeros(M,P+1);  % The U matrix (for Hammerstein SAF) in [2]
end
indexes = ones(M,1);

% STRUCTURE DEFINITION ---------------------------------------------------
AF = struct('aftype', aftype, 'afinit',afinit, 's',s, 'x',x, 'Q',Q, ...
            'lut_len', lut_len, 'uIndex',uIndex, 'uSpline',uSpline, ...
            'Slope',Slope, 'Gain',Gain, 'DeltaX',DeltaX, ...
            'P', P, 'Pord', Pord, 'C',C, 'g',g, 'gM',gM, 'indexes',↩
                indexes);
% ----------------------------------------------------------------------

% For spline interpolation -----------------------------------------------
if (aftype>1 && aftype ~= 3)
    LutSlope = (Table_Length - 1)/2.0;  % New slope
    X = -LutSlope*DeltaX;
    for j=1 : Table_Length   % Table_Length
        AF.Q(j) = FUNC(X, Gain, Slope, afinit);
        X = X + DeltaX;
    end
end
% ----------------------------------------------------------------------


%=============== Function TabFuncLen() ==================================
function Table_Lenght = TabFuncLen(DX, G, S, ty)
    ii = 0;
    X  = 0;
    if (ty~=2 && ty~=3)
        F = 0.0;
        crtGain = G - 0.005*G; % Max atc func value ----------------------
        while (F<crtGain)
            F  = FUNC(X,G,S,ty);
            X  = X + DX;
            ii = ii + 1;
        end
    elseif (ty==3)
        ii = 11;
    else
        crtGain = 0.005*G;  % Gaussian
        F = G;
        while (F>crtGain )
             ii = ii + 1;
             F  = FUNC(X,G,S,ty);
             X  = X + DX;
        end
    end
    Table_Lenght = ii*2 + 1;   % always odd
% End function TabFuncLen() ============================================

% ================== Function FUNC() ====================================
function value = FUNC(X, G, S, ty)
% Nonlinear function implementation
    switch  ty
        case -1
            value = 2*G/(1 + exp(-X*S) ) - G;  % Signed Sigmoid
```

```
        case  0
            value = X*S;                % Linear
        case  1
            value = G/(1+exp(-X*S));    % Unsigned Sigmoidal
        case  2
            value = G*exp(-(X*X)/5.0);  % Gaussian
        case 3
            value = 2*G*(rand - 0.5);   % Random
        otherwise
            value = 0;
    end
% End function FUNC() ================================================
```

2.2.2 create_III_ord_Volterra_filter_1

This function creates and initializes the structure implementing a III-order Volterra AF architecture and adapted by the LMS adaptive algorithm. More details can be found in [5].

The usage is as follows:

```
Volterra_3rd_f = create_III_ord_Volterra_filter_1(M, K, mu, delta);
```

where

- Volterra_3rd_f is the Volterra adaptive filter structure;
- M is the number of filter coefficients M;
- K is the order of APA projection K;
- mu is the learning rate μ;
- delta is the regularization parameter δ.

The complete Matlab code is reported below.

```
function Volterra_3rd_f = create_III_ord_Volterra_filter_1(M, K, mu, ↩
    delta)


M2 = M*(M+1)/2;          % Copmuted II  degree Volterra filter length
M3 = M*(M+2)*(M+1)/6;    % Computed III degree Volterra filter length
MP = M + M2 + M3;        % Full Volterra filter length
dI  = delta*eye(K);      % Regularizing diagonal matrix
bl  = zeros(M+1,2);      % Upper block pointers address
xd  = zeros(K,1);        % Buffer for desired samples
xh1 = zeros(M,1);        % Buffer for I   degree Volterra filter status
xh2 = zeros(M2,1);       % Buffer for II  degree Volterra filter status
xh3 = zeros(M3,1);       % Buffer for III degree Volterra filter status
xh  = [xh1;  xh2; xh3];  % Volterra filter status
h   = zeros(MP,1);       % Volterra AF taps
X  = zeros(K,MP);        % Total covariance data matrix for Volterra APA
X1 = zeros(K,M);         % I   ord cov. data matrix for Volterra APA
X2 = zeros(K,M2);        % II  ord cov. data matrix for Volterra APA
X3 = zeros(K,M3);        % III ord cov. data matrix for Volterra APA
h1 = zeros(M,1);         % Buffer for I   degree Volterra filter
h2 = zeros(M2,1);        % Buffer for II  degree Volterra filter
h3 = zeros(M3,1);        % Buffer for III degree Volterra filter
```

```
% II order vector-block of pointers to the Kronecker product ------------
for k=1:M
    bl(k,1)  = k ;   % Upper block addres (1 : bl(k))
end;

% III order vector-block of pointers to the Kronecker product ------------
kk = 1;
for i = 1:M
    kk = kk + bl(M,1) - bl(i,1) +  1 ;
    bl(i+1,2) = kk - 1;  % Dummy pointer array for next order block ↩
        addressing
end
bl(1,2) = 0;

Circuit = 'III order full Volterra FIlter';   % Architecture name

%-------------------------------------------------------------------------
Volterra_3rd_f = struct('Circuit', Circuit, ...
    'M',M,'M2',M2,'M3',M3,'MP',MP, ...                    % Memory ↩
        lengths
    'xh1',xh1,'xh2',xh2,'xh3',xh3,'xh',xh, ...            % filters ↩
        status
    'bl',bl, ...                                          % pointers ↩
        matrix for the Kronecker product
    'K',K,'mu',mu,'dI',dI, ...                            % APA order, ↩
        learning rate regularizing matrix
    'h1',h1,'h2',h2,'h3',h3,'h', h, ...                   % array of ↩
        Volterra coeffs.
    'X', X, 'X1', X1, 'X2', X2, 'X3', X3, 'xd',xd);       % APA matricies↩
         and statusd status
```

2.2.3 create_LNLJenkins_lms_adaptive_filter_1

This function creates and initializes the structure implementing an adaptive linear-nonlinear-linear (LNL) sandwich model. More details can be found in [2]. It has been used for comparisons in [9].

The usage is as follows:

```
lms_af = create_LNLJenkins_lms_adaptive_filter_1(M1, M2, M3, mu1, mu2, ↩
    mu3, delta, af);
```

where

- lms_af is the adaptive filter structure;
- M1 is the length of the first part of LNL filter M_1;
- M2 is the length of the second part of LNL filter M_2;
- M3 is the length of the third part of LNL filter M_3;
- mu1 is the step-size of the first part of LNL filter μ_1;
- mu2 is the step-size of the second part of LNL filter μ_2;
- mu3 is the step-size of the third part of LNL filter μ_3;
- delta is the regularization parameter δ;

- af is the spline nonlinearity structure.

The complete Matlab code is reported below.

```
function lms_af = create_LNLJenkins_lms_adaptive_filter_1(M1, M2, M3, mu1↩
    , mu2, mu3, delta, af)


% LNL Jenkins adaptive filter definition and initialization ---------------
w1  = zeros(M1,1);       % Taps of the first part of LNL filter
w2  = zeros(M2,1);       % Taps of the first second of LNL filter
w3  = zeros(M3,1);       % Taps of the first third of LNL filter
w1(floor(M1/2)) = 1;     % Adaptive filter weights i.c.
w2(floor(M2/2)) = 1;     % Adaptive filter weights i.c.
dI  = delta;             % Regularizing diagonal matrix
xd  = zeros(M3,1);       % Buffer of the desired samples
xw1 = zeros(M1,1);       % Buffer of the filter 1 status
xw2 = zeros(M2,1);       % Buffer of the filter 2 status
xw3 = zeros(M3,1);       % Buffer of the filter 3 status
X   = zeros(M1,M3);      % Buffer matrix of the past inputs delay lines
Z   = zeros(M2,M3);      % Buffer matrix of the past nonlinear delay ↩
    lines
% -------------------------------------------------------------------------
lms_af = struct('M1',M1, 'M2',M2, 'M3',M3, 'mu1',mu1, 'mu2',mu2, 'mu3',↩
    mu3, ...
    'dI',dI, 'w1',w1, 'w2',w2, 'w3',w3, 'xd',xd, 'xw1',xw1, 'xw2',xw2, ↩
        ...
    'xw3',xw3, 'X',X, 'Z',Z, 'af',af);
```

2.2.4 create_Sandwich1SPL_lms_adaptive_filter_1

This function creates and initializes the structure implementing a S1SAF architecture and adapted by the LMS adaptive algorithm. More details can be found in [9].

The usage is as follows:

```
lms_af = create_Sandwich1SPL_lms_adaptive_filter_1(M, mu, mQ1, mQ2, delta↩
    , af1, af2);
```

where

- lms_af is the adaptive filter structure;
- M is the length of the linear filter M;
- mu is the step-size of the linear filter μ;
- mQ1 is the step-size of the first nonlinearity μ_{Q_1};
- mQ2 is the step-size of the second nonlinearity μ_{Q_2};
- delta is the regularization parameter δ;
- af1 is the first spline nonlinearity structure;
- af2 is the second spline nonlinearity structure.

The complete Matlab code is reported below.

```
function lms_af = create_Sandwich1SPL_lms_adaptive_filter_1(M, mu, mQ1, ↩
    mQ2, delta, af1, af2)


% LMS Sandwich 1 adaptive filter definition and initialization ----------
w = zeros(M,1);     % Linear filter taps
w(floor(M/2)) = 1;  % Adaptive filter weights I.C.
dI = delta;         % Regularizing parameter
xd = zeros(M,1);    % Buffer of the desired signal
xw = zeros(M,1);    % Buffer of the filter status
% ----------------------------------------------------------------------
lms_af = struct('M',M, 'mu',mu,'mQ1',mQ1, 'mQ2',mQ2, 'dI',dI, 'w',w, ...
    'xd',xd, 'xw',xw, 'af1',af1, 'af2',af2);
```

2.2.5 create_Sandwich2SPL_lms_adaptive_filter_1

This function creates and initializes the structure implementing a S2SAF architecture and adapted by the LMS adaptive algorithm. More details can be found in [9].

The usage is as follows:

```
lms_af = create_Sandwich2SPL_lms_adaptive_filter_1(M1, M2, mu1, mu2, mQ, ↩
    delta, af);
```

where

- lms_af is the adaptive filter structure;
- M1 is the length of the first linear filter M_1;
- M2 is the length of the second linear filter M_2;
- mu1 is the step-size of the first linear filter μ_1;
- mu2 is the step-size of the second linear filter μ_2;
- mQ is the step-size of the nonlinearity μ_Q;
- delta is the regularization parameter δ;
- af is the spline nonlinearity structure.

The complete Matlab code is reported below.

```
function lms_af = create_Sandwich2SPL_lms_adaptive_filter_1(M1, M2, mu1, ↩
    mu2, mQ, delta, af)


% LMS Sandwich 2 adaptive filter definition and initialization ----------
w1  = zeros(M1,1);    % Linear filter 1 taps
w2  = zeros(M2,1);    % Linear filter 2 taps
w1(floor(M1/2)) = 1;  % Adaptive filter weights i.c.
w2(floor(M2/2)) = 1;  % Adaptive filter weights i.c.
dI  = delta;          % Regularizing parameter
xd  = zeros(M2,1);    % Buffer of the desired signal
xw1 = zeros(M1,1);    % Buffer of the filter 1 status
xw2 = zeros(M2,1);    % Buffer of the filter 2 status
xs  = zeros(M2,1);    % Buffer of the nonlinearity
X   = zeros(M1,M2);   % Buffer matrix of the past inputs delay lines
```

```
rp  = zeros(M2,1);    % Buffer of the past nonlinearity derivatives
%-------------------------------------------------------------------------
lms_af = struct('M1',M1, 'M2',M2, 'mu1',mu1, 'mu2',mu2, 'mQ',mQ, ...
    'dI',dI, 'w1',w1, 'w2',w2, 'xd',xd, 'xw1',xw1, 'xw2',xw2, 'X',X, ...
    'rp',rp, 'xs',xs, 'af',af);
```

2.2.6 create_SPL_apa_adaptive_filter_1

This function creates and initializes the structure implementing a WSAF architecture and adapted by the APA adaptive algorithm. More details can be found in [6].

The usage is as follows:

```
apa_af = create_SPL_apa_adaptive_filter_1(M, K, mu, mQ, delta, af);
```

where

- apa_af is the adaptive filter structure;
- M is the length of the linear filter M;
- K is the order of the APA projection K;
- mu is the step-size of the linear filter μ;
- mQ is the step-size of the nonlinearity μ_Q;
- delta is the regularization parameter δ;
- af is the spline nonlinearity structure.

The complete Matlab code is reported below.

```
function apa_af = create_SPL_apa_adaptive_filter_1(M, K, mu, mQ, delta, ↩
    af)


% APA adaptive filter definition and initialization ---------------------
w = zeros(M,1);       % Linear filter taps
w(floor(M/2)+1) = 1;  % Adaptive filter weights i.c.
dI = delta*eye(K);    % Regularizing diagonal matrix
X  = zeros(K,M);      % Covariance data matrix for APA
xd = zeros(K,1);      % Buffer of the desired signal
xw = zeros(M,1);      % Buffer of the filter status
%-------------------------------------------------------------------------
apa_af = struct('M',M, 'K',K, 'mu',mu,'mQ',mQ, 'dI',dI, 'w',w, 'X',X, ...
    'xd',xd, 'xw',xw, 'af',af);
```

2.2.7 create_SPL_apa_iir_adaptive_filter_1

This function creates and initializes the structure implementing an IIR WSAF architecture and adapted by the APA adaptive algorithm. More details can be found in [8].

The usage is as follows:

```
apa_af = create_SPL_apa_iir_adaptive_filter_1(M, N, K, mu, mQ, delta, af)
```

where

- apa_af is the adaptive filter structure;
- M is the length of the MA part of the linear filter M;
- N is the length of the AR part of the linear filter N;
- K is the order of the APA projection K;
- mu is the step-size of the linear filter μ;
- mQ is the step-size of the nonlinearity μ_Q;
- delta is the regularization parameter δ;
- af is the spline nonlinearity structure.

The complete Matlab code is reported below.

```
apa_af = create_SPL_apa_iir_adaptive_filter_1(M, N, K, mu, mQ, delta, af)


% APA adaptive filter definition and initialization ------------------------
Mt = M + N;              % Total number of parametrs
b = zeros(M,1);          % MA filter taps
a = zeros(N,1);          % AR filter taps
w = zeros(Mt,1);         % Total AF taps (ARMA model)
beta  = zeros(M,N+1);    % Derivatives with respect MA coefficients
alpha = zeros(N,N+1);    % Derivatives with respect AR coefficients
dI = delta*eye(K);       % Regularizing diagonal matrix
X  = zeros(K,M);         % Covariance data matrix for APA
xd = zeros(K,1);         % Buffer of desired samples
xw = zeros(M,1);         % Buffer of filter status
sw = zeros(N,1);         % Buffer of IIR status
% -------------------------------------------------------------------------
apa_af = struct('M',M, 'N',N, 'K',K, 'mu',mu,'mQ',mQ, 'beta',beta, ...
    'alpha',alpha, 'dI',dI, 'b',b, 'a',a, 'w',w, 'X',X, 'xd',xd, ...
    'xw',xw, 'sw',sw, 'af',af);
```

2.2.8 create_SPL_lms_adaptive_filter_1

This function creates and initializes the structure implementing a WSAF architecture and adapted by the LMS adaptive algorithm. More details can be found in [6].

The usage is as follows:

```
lms_af = create_SPL_lms_adaptive_filter_1(M, mu, mQ, delta, af)
```

where

- lms_af is the adaptive filter structure;
- M is the length of the linear filter M;
- mu is the step-size of the linear filter μ;
- mQ is the step-size of the nonlinearity μ_Q;
- delta is the regularization parameter δ;

- af is the spline nonlinearity structure.

The complete Matlab code is reported below.

```
function lms_af = create_SPL_lms_adaptive_filter_1(M, mu, mQ, delta, af)


% LMS SAF adaptive filter definition and initialization ------------------
w = zeros(M,1);     % Linear filter taps
w(floor(M/2)) = 1;  % Adaptive filter weights i.c.
dI = delta;         % Regularizing parameter
xd = zeros(M,1);    % Buffer of the desired signal
xw = zeros(M,1);    % Buffer of the filter status
%-------------------------------------------------------------------------
lms_af = struct('M',M, 'mu',mu,'mQ',mQ, 'dI',dI, 'w',w, 'xd',xd, ...
    'xw',xw, 'af',af);
```

2.2.9 create_SPL_lms_iir_adaptive_filter_1

This function creates and initializes the structure implementing an IIR WSAF architecture and adapted by the LMS adaptive algorithm. More details can be found in [8].

The usage is as follows:

```
lms_af = create_SPL_lms_iir_adaptive_filter_1(M, N, mu, mQ, delta, af)
```

where

- lms_af is the adaptive filter structure;
- M is the length of the MA part of the linear filter M;
- N is the length of the AR part of the linear filter N;
- mu is the step-size of the linear filter μ;
- mQ is the step-size of the nonlinearity μ_Q;
- delta is the regularization parameter δ;
- af is the spline nonlinearity structure.

The complete Matlab code is reported below.

```
lms_af = create_SPL_lms_iir_adaptive_filter_1(M, N, mu, mQ, delta, af)


% LMS IIR SAF adaptive filter definition and initialization --------------
Mt    = M + N;          % Total number of parametrs
b     = zeros(M,1);     % MA taps
a     = zeros(N,1);     % AR taps
w     = zeros(Mt,1);    % Complete AF taps (ARMA model)
beta  = zeros(M,N+1);   % Derivatives with respect MA coefficients
alpha = zeros(N,N+1);   % Derivatives with respect AR coefficients
dI = delta;             % Regularizing parameter
xd = zeros(Mt,1);       % Buffer of the desired signal
xw = zeros(M,1);        % Buffer of the filter status
sw = zeros(N+1,1);      % Buffer of the IIR status
%-------------------------------------------------------------------------
```

```
lms_af = struct('M',M, 'N',N, 'mu',mu,'mQ',mQ, 'dI',dI, 'w',w, 'b',b, ...
  'a',a, 'beta',beta, 'alpha',alpha, 'xd',xd, 'xw',xw, 'sw',sw, 'af',af);
```

2.3 Computing the model feed-forward outputs

The third section of this chapter introduces the functions that are able to evaluate the feed-forward phase of the proposed adaptive architectures, i.e. the evaluation of an output sample given an input sample.

2.3.1 FW_HPOLY_F

This function evaluates the output of a FIR Hammerstein polynomial structure implemented by a polynomial adaptive filter (see Figure 1.2). More details can be found in [7].

The usage is as follows:

```
[F, y, s] = FW_HPOLY_F(F, x);
```

where

- F is the adaptive filter structure;
- x is the input signal sample $x[n]$;
- y is the output signal sample $y[n]$;
- s is the nonlinearity output $\mathbf{s}_n$.

The complete Matlab code is reported below.

```
function [F, y, s] = FW_HPOLY_F(F, x)


M = F.M;                        % Length of the linear filter

F.xw(2:M) = F.xw(1:M-1);        % Shift the input delay-line
F.xw(1) = x;                    % Load a new input into the delay-line

s = zeros(M,1);                 % Nonlinear output buffer
for i=1:M,
    [s(i),F.af] = ActFunc(F.xw(i),F.af);   % Evaluation of the ↩
        nonlinearity
end
y = s'*F.w;                     % Output sample
```

2.3.2 FW_HSPL_F

This function evaluates the output of a Hammerstein nonlinear structure implemented by a HSAF architecture (see Figure 1.2). More details can be found in [7].

The usage is as follows:

```
[F, y, s] = FW_HSPL_F(F, x);
```

where

- F is the adaptive filter structure;
- x is the input signal sample $x[n]$;
- y is the output signal sample $y[n]$;
- s is the linear combiner input array $\mathbf{s}_n$.

The complete Matlab code is reported below.

```
function [F, y, s] = FW_HSPL_F(F, x)


M = F.M;                        % Length of the linear filter

F.xw(2:M) = F.xw(1:M-1);        % Shift of the input delay-line
F.xw(1) = x;                    % Load a new input into the delay-line

s = zeros(M,1);                 % Linear combiner input array
for i=1:M,
    [s(i),F.af] = ActFunc(F.xw(i),F.af); % Evaluating the nonlinearity ↩
        output
end
y = s'*F.w;                     % Filter output
```

2.3.3 FW_III_ord_VF_F

This function evaluates the output of a III-order Volterra nonlinear structure. More details can be found in [5].

The usage is as follows:

```
[VF, y] = FW_III_ord_VF_F(VF, x);
```

where

- VF is the adaptive filter structure;
- x is the input signal sample $x[n]$;
- y is the output signal sample $y[n]$.

The complete Matlab code is reported below.

```
M = VF.M;    % Number of coefficients

% Regression delay-line update (I ord Volterra Kernel) ------------------
VF.xh1(2:M) = VF.xh1(1:M-1);       % Shift of the input delay-line
VF.xh1(1) = x;                     % Load a new input into the delay-line

% Kronecker product for II order Volterra kernel generation -------------
kk = 1;
MM = VF.bl(M,1);
for i = 1:M
    VF.xh2(kk:kk + MM - VF.bl(i,1) ) = VF.xh1(i)*VF.xh1(VF.bl(i,1):MM);
    kk = kk + MM - VF.bl(i,1) + 1;
end
```

```
% Kronecker product for III order Volterra kernel generation ----------
kk = 1;
MM = VF.bl(M+1,2);
for i = 1:M
    VF.xh3(kk:kk + MM - VF.bl(i,2) - 1) = VF.xh1(i)*VF.xh2(VF.bl(i,2)+1:↩
        MM);
    kk = kk + MM - VF.bl(i,2);
end

% Full buffer composition ----------------------------------------------
VF.xh = [VF.xh1; VF.xh2; VF.xh3];     % Full buffer composition

y = VF.h.'*VF.xh;          % (K,1) = (K,M) x (M,1) output array
```

2.3.4 FW_Sandwich1SPL_F

This function evaluates the output of a Sandwich 1 nonlinear structure implemented by a S1SAF architecture (see Figure 1.3). More details can be found in [9].

The usage is as follows:

```
[F, y, s, r] = FW_Sandwich1SPL_F(F, x);
```

where

- F is the adaptive filter structure;
- x is the input signal sample $x[n]$;
- y is the output signal sample $y[n]$;
- s is the first nonlinearity output $\mathbf{s}_n$;
- r is the linear filter output $r[n]$.

The complete Matlab code is reported below.

```
M = F.M;                        % Length of the linear filter

F.xw(2:M) = F.xw(1:M-1);        % Shift the input filter delay-line
F.xw(1) = x;                    % Load a new input into the delay-line

s = zeros(M,1);
for i=1:M,
    [s(i),F.af1] = ActFunc(F.xw(i),F.af1);  % Output of first ↩
        nonlinearity in Eq. (11)
end
r = s'*F.w;                     % Linear filter output in Eq. (12)
[y,F.af2] = ActFunc(r,F.af2);   % System output in Eq. (13)
```

2.3.5 FW_Sandwich2SPL_F

This function evaluates the output of a Sandwich 2 nonlinear structure implemented by a S2SAF architecture (see Figure 1.4). More details can be found in [9].

The usage is as follows:

```
[F, y, s, r] = FW_Sandwich2SPL_F(F, x);
```

where

- F is the adaptive filter structure;
- x is the input signal sample $x[n]$;
- y is the output signal sample $y[n]$;
- s is the first linear combiner output $s[n]$;
- r is the nonlinearity output $r[n]$.

The complete Matlab code is reported below.

```
M1 = F.M1;                          % Length of the first linear filter
M2 = F.M2;                          % Length of the second linear filter

F.xw1(2:M1) = F.xw1(1:M1-1);        % Shift the input filter 1 delay-line
F.xw1(1) = x;                       % Load a new input into the delay-line
s = F.xw1'*F.w1;                    % First linear filter output
[r,F.af] = ActFunc(s,F.af);         % Nonlinearity output
F.xw2(2:M2) = F.xw2(1:M2-1);        % Shift the input filter 2 delay-line
F.xw2(1) = r;                       % Load a new input into the delay-line
y = F.xw2'*F.w2;                    % System output
```

2.3.6 FW_WPOLY_F

This function evaluates the output of a FIR Wiener polynomial structure implemented by a polynomial adaptive filter (see Figure 1.1). More details can be found in [6].

The usage is as follows:

```
[F, y, s] = FW_WPOLY_F(F, x);
```

where

- F is the adaptive filter structure;
- x is the input signal sample $x[n]$;
- y is the output signal sample $y[n]$;
- s is the linear combiner output $s[n]$.

The complete Matlab code is reported below.

```
function [F, y, s] = FW_WPOLY_F(F, x)


M = F.M;                        % Length of the linear filter

F.xw(2:M) = F.xw(1:M-1);        % Shift the input delay-line
F.xw(1) = x;                    % Load a new input into the delay-line

s = F.xw.'*F.w;                 % Linear filter output
y = ActFunc(s,F.af);            % Nonlinear output
```

2.3.7 FW_WSPL_F

This function evaluates the output of a Wiener nonlinear structure implemented by a WSAF architecture (see Figure 1.1). More details can be found in [6].

The usage is as follows:

```
[F, y] = FW_WSPL_F(F, x);
```

where

- F is the adaptive filter structure;
- x is the input signal sample $x[n]$;
- y is the output signal sample $y[n]$.

The complete Matlab code is reported below.

```
function [F, y] = FW_WSPL_F(F, x)


M = F.M;                        % Length of the linear filter

F.xw(2:M) = F.xw(1:M-1);        % Shift the input delay-line
F.xw(1) = x;                    % Load a new input into the delay-line

[y, F.af] = ActFunc(F.xw'*F.w, F.af); % Output of the nonlinearity, Eq. ↩
    (8)
```

2.3.8 FW_WSPL_IIR_F

This function evaluates the output of a IIR Wiener nonlinear structure implemented by a IIR WSAF architecture (see Figure 1.1). More details can be found in [8].

The usage is as follows:

```
[F, y] = FW_WSPL_IIR_F(F, x);
```

where

- F is the adaptive filter structure;
- x is the input signal sample $x[n]$;
- y is the output signal sample $y[n]$.

The complete Matlab code is reported below.

```
function [F, y] = FW_WSPL_IIR_F(F, x)


M = F.M;                                % Length of the MA part
N = F.N;                                % Length of the AR part

F.xw(2 : M) = F.xw(1 : M-1);            % Shift the input delay-line
F.xw(1) = x;                            % Load a new input into the ↩
    delay-line
F.sw(2:N+1) = F.sw(1:N);                % Shift the internal delay-line
```

```
F.sw(1) = F.xw'*F.b + F.sw(2:N+1)'*F.a;  % Load a new input into the ↩
    internal delay-line with the output of an IIR filter
[y,F.af] = ActFunc(F.sw(1),F.af);        % Output of the IIR WSAF
```

2.4 Adapting the SAF architectures

This fourth and last section lists all the adaptive algorithm function, implementing the learning rules for the different type of SAF architectures, as shown in section 1.2.

2.4.1 AF_APA_HPOLY_F

This function implements the adaptation of an FIR Hammerstein polynomial structure by using the APA adaptive algorithm (see Figure 1.2). More details can be found in [7].

The usage is as follows:

```
[F, y, e] = AF_APA_HPOLY_F(F, x, d);
```

where

- F is the adaptive filter structure;
- x is the input signal sample $x[n]$;
- d is the desired signal sample $d[n]$;
- y is the output signal sample $y[n]$;
- e is the error signal sample $e[n]$.

The complete Matlab code is reported below.

```
function [F, y, e] = AF_APA_HPOLY_F(F, x, d)


M = F.M;                % Length of the linear filter
K = F.K;                % APA order

% Covariance Matrix Update ----------------------------------------------
F.xw(2 : M) = F.xw(1 : M-1);          % Shift the input delay-line
[F.xw(1), F.af] = ActFunc(x, F.af);   % Load a new input into the delay-↩
    line
for i = 1:K-1                         % Shift the covarince matrix X
    F.X(i,1:M) = F.X(i+1,1:M);
end
F.X(K,1:M) = F.xw(1:M);               % Fill the covariance matrix X

% Desired-output buffer for APA -----------------------------------------
F.xd(1:K-1) = F.xd(2:K);              % Shift desired delay-line
F.xd(K) = d ;                         % Load a new desired sample into the↩
     delay-line

yy = F.X*F.w;                         % (K,1) = (K,M) x (M,1) output array
ee = F.xd - yy;                       % (K,1) = (K,1) - (K,1) error array
e = ee(K);                            % A priori output error sample
y = yy(K);                            % Output sample
```

```
% Linear Filter ---------------------------------------------------------
F.w = F.w + F.mu*(F.X'/(F.dI + F.X*F.X'))*ee;  % Updating the filter taps
F.w(1) = 1;                                    % If first tap should be 1

% Polynomial Nonlinearity ------------------------------------------------
if (F.af.aftype == 3)         % 3 = Polynomial
    F.af.gM(:,2:M) = F.af.gM(:,1:M-1);         % Shift the U matrix ↩
        buffer
    F.af.gM(:,1) = F.af.g.';                   % Load a new vector in ↩
        matrix U
    u = F.af.gM*F.w;                           % Evaluating u vector

    e_av = F.mQ*e;
    F.af.Q = F.af.Q + e_av*u;                  % Updating the polynomial ↩
        coefficients
end
```

2.4.2 AF_APA_HSPL_F

This function implements the adaptation of a Hammerstein SAF (HSAF) structure by using the APA adaptive algorithm (see Figure 1.2). More details can be found in [7].

The usage is as follows:

```
[F, y, e] = AF_APA_HSPL_F(F, x, d);
```

where

- F is the adaptive filter structure;
- x is the input signal sample $x[n]$;
- d is the desired signal sample $d[n]$;
- y is the output signal sample $y[n]$;
- e is the error signal sample $e[n]$.

The complete Matlab code is reported below.

```
function [F, y, e] = AF_APA_HSPL_F(F, x, d)


M = F.M;            % Length of the linear filter
K = F.K;            % APA order

% Covariance Matrix Update -----------------------------------------------
F.xw(2 : M) = F.xw(1 : M-1);              % Shift the filter delay-line
[F.xw(1), F.af] = ActFunc(x, F.af);       % Load a new input into the ↩
    delay-line
for j=2:M,                                % Constructing the matrix U
    if F.af.uIndex == F.af.indexes(j)
        F.af.gM(j,:) = F.af.gM(j-1,:);
    else
        F.af.gM(j,:) = zeros(1,4);
    end
end
F.af.gM(1,:) = F.af.g;                    % Load a new vector in matrix U
```

```
F.af.indexes(2:M) = F.af.indexes(1:M-1);
F.af.indexes(1) = F.af.uIndex;
for i = 1:K-1                          % Shift the covarince matrix X
    F.X(i,1:M) = F.X(i+1,1:M);
end
F.X(K,1:M) = F.xw(1:M);                % Fill the covariance matrix X

% Desired-output buffer for APA ---------------------------------------
F.xd(1:K-1) = F.xd(2:K);               % Shift the delay-line
F.xd(K) = d ;                          % Load a new sample into the ↩
    delay-line
yy = F.X*F.w;                          % Nonlinear output array
ee = F.xd - yy;                        % Error array
e = ee(K);                             % A priori output error sample
y = yy(K);                             % Output sample

% APA weights and control points update -------------------------------
F.w = F.w + F.mu*(F.X'/(F.dI + F.X*F.X'))*ee;          % APA learning

if F.af.aftype > 1  % Kind act. f. -1 0 1 2 4 5 *
    e_av = F.mQ*e;
    ii = F.af.uIndex : F.af.uIndex + F.af.P;           % P = Spline order
    F.af.Q(ii) = F.af.Q(ii) + e_av*(F.w'*F.af.gM).';   % LMS for control ↩
        points
end
```

2.4.3 AF_APA_WSPL_F

This function implements the adaptation of a Wiener SAF (WSAF) structure by using the APA adaptive algorithm (see Figure 1.1). More details can be found in [6].

The usage is as follows:

```
[F, y, e] = AF_APA_WSPL_F(F, u, d);
```

where

- F is the adaptive filter structure;
- u is the input signal sample $u[n]$;
- d is the desired signal sample $d[n]$;
- y is the output signal sample $y[n]$;
- e is the error signal sample $e[n]$.

The complete Matlab code is reported below.

```
function [F, y, e] = AF_APA_WSPL_F(F, u, d)


M = F.M;          % Length of the linear filter
K = F.K;          % APA order
yy = zeros(K,1); % Nonlinearity output

% Covariance Matrix Update --------------------------------------------
F.xw(2 : M) = F.xw(1 : M-1);      % Shift of the input delay-line
F.xw(1) = u;                      % Load a new input into the delay-line
for i = 1:K-1                     % Shift the covarince matrix X
```

```
    F.X(i,1:M) = F.X(i+1,1:M);
end
F.X(K,1:M) = F.xw(1:M);          % Fill the covariance matrix X

% Desired-output buffer for APA ------------------------------------------
F.xd(1:K-1) = F.xd(2:K);         % Shift the desired delay-line
F.xd(K) = d ;                    % Load a new desired output into the ↩
    delay-line

s = F.X*F.w;      % (K,1) = (K,M) x (M,1) output array
for k=1:K
    [yy(k),F.af] = ActFunc(s(k),F.af);
end

ee = F.xd - yy;  % (K,1) = (K,1) - (K,1) error array
e = ee(K);       % A priori output error sample
y = yy(K);       % Output sample

% Learning ---------------------------------------------------------------
for k=1:K
   ee(k) = ee(k)*dActFunc(y,F.af);
end

% APA weights update -----------------------------------------------------
F.w = F.w + F.mu*(F.X'/(F.dI + F.X*F.X'))*ee;  % (M,1) = (M,1) + (M,K) x ↩
    (K,1) modified APA of Eq. (15)

% LMS LUT control points update
% ----------------------------------------------------------------------
if F.af.aftype > 1  % Kind act. f. -1 0 1 2 4 5 6 7 ... 20  */
    e_av = (F.mQ*e)/1; % e oppure e(K) ?
    ii = F.af.uIndex :  F.af.uIndex + F.af.P;  % P = Spline order
    F.af.Q(ii) = F.af.Q(ii) + e_av*F.af.g';    % LMS in Eq. (16)
end
```

2.4.4 AF_APA_WSPL_IIR_F

This function implements the adaptation of an IIR Wiener SAF (WSAF) structure by using the APA adaptive algorithm (see Figure 1.1). More details can be found in [8].

The usage is as follows:

```
[F, y, e] = AF_APA_WSPL_IIR_F(F, x, d);
```

where

- F is the adaptive filter structure;
- x is the input signal sample $x[n]$;
- d is the desired signal sample $d[n]$;
- y is the output signal sample $y[n]$;
- e is the error signal sample $e[n]$.

The complete Matlab code is reported below.

```
function [F, y, e] = AF_APA_WSPL_IIR_F(F, x, d)


M = F.M;   % Length of the MA part
N = F.N;   % Length of the AR part
K = F.K;   % APA order
yy = zeros(K,1);

% Covar Matrix Up Date ------------------------------------------------------
F.xw(2 : M) = F.xw(1 : M-1);     % Shift of the input delay-line
F.xw(1) = x;                     % Load a new input into the delay-line
for i = 1:K-1
    F.X(i,1:M) = F.X(i+1,1:M);   % Shift the covarince matrix X
end
F.X(K,1:M) = F.xw(1:M);          % Fill the covariance matrix X

% Desired-output buffer for APA ---------------------------------------------
F.xd(1:K-1) = F.xd(2:K);         % Shift of the input delay-line
F.xd(K) = d ;                    % Load a new desired output into the ↩
    delay-line

ss = F.X*F.w;                    % (K,1) = (K,M) x (M,1) output array
for k=1:K
    [yy(k),F.af] = ActFunc(ss(k),F.af);
end

ee = F.xd - yy;     % (K,1) = (K,1) - (K,1) error array
e = ee(K);          % A priori output error sample
y = yy(K);          % Output sample

% Learning ------------------------------------------------------------------
for k=1:K  %  err*phi'(s)
    ee(k) = ee(k)*dActFunc(y,F.af);
end

% APA weights updating ------------------------------------------------------
F.w = F.w + F.mu*(F.X'/(F.dI + F.X*F.X'))*ee;  % (M,1) = (M,1) + (M,K) x ↩
    (K,1)

% LMS LUT control points updating -------------------------------------------
if F.af.aftype > 1      % Kind act. f. -1 0 1 2 4 5 6 7 ... 20  */
    e_av = (F.mQ*e)/1;
    ii = F.af.uIndex :  F.af.uIndex + F.af.P;   % P = Spline order
    F.af.Q(ii) = F.af.Q(ii) + e_av*F.af.g';
end
```

2.4.5 AF_III_ord_VF_APA_F

This function implements the adaptation of a III-order Volterra AF structure by using the APA adaptive algorithm. More details can be found in [5].

The usage is as follows:

```
[VF, y, e] = AF_III_ord_VF_APA_F(VF, x, d);
```

where

- VF is the Volterra adaptive filter structure;
- x is the input signal sample $x[n]$;

- d is the desired signal sample $d[n]$;
- y is the output signal sample $y[n]$;
- e is the error signal sample $e[n]$.

The complete Matlab code is reported below.

```
function [VF, y, e] = AF_III_ord_VF_APA_F(VF, x, d)


K = VF.K;    % APA order
M = VF.M;    % Number of coefficients

% Regression delay-line update (I ord Volterra Kernel) ------------------
VF.xh1(2 : M) = VF.xh1(1 : M-1);   % Shift the input delay-line
VF.xh1(1) = x;                     % Load a new input into the delay-line

% Kronecker product for II order Volterra kernel generation -------------
kk = 1;
MM = VF.bl(M,1);
for i = 1:M
    VF.xh2(kk : kk + MM - VF.bl(i,1) ) = VF.xh1(i)*VF.xh1( VF.bl(i,1):MM );
    kk = kk + MM - VF.bl(i,1) +  1  ;
end

% Kronecker product for III order Volterra kernel generation ------------
kk = 1;
MM = VF.bl(M+1,2);
for i = 1:M
    VF.xh3(kk : kk + MM - VF.bl(i,2) - 1 ) = VF.xh1(i)*VF.xh2( VF.bl(i,2)+1:MM );
    kk = kk + MM - VF.bl(i,2);
end

% Full buffer composition -----------------------------------------------
VF.xh = [VF.xh1; VF.xh2; VF.xh3 ];

% Update X covariance matrix for APA agorithm ---------------------------
M = VF.MP;       % Now M is the Volterra entire buffer length
for i = 1:K-1    % Shift the covarince matrix X
    VF.X(i,1:M) = VF.X(i+1,1:M);
end
VF.X(K,1:M) = VF.xh(1:M);  % Fill the covariance matrix X

% Desired-output buffer for APA -----------------------------------------
VF.xd(1:K-1) = VF.xd(2:K);  % Shift the input delay-line
VF.xd(K) = d ;              % Load a new desired output into the delay-line
yy = VF.X*VF.h;             % (K,1) = (K,M) x (M,1) output array
ee = VF.xd - yy;            % (K,1) = (K,1) - (K,1) error array
VF.h = VF.h + VF.mu*(VF.X'/(VF.dI + VF.X*VF.X'))*ee;  % (M,1) = (M,1) + (M,K) x (K,1)
y = yy(VF.K);               % Output sample
e = ee(VF.K);               % Error sample
```

2.4.6 AF_LMS_HPOLY_F

This function implements the adaptation of an FIR Hammerstein polynomial structure by using the LMS adaptive algorithm (see Figure 1.2). More details can be

found in [7].

The usage is as follows:

```
[F, y, e] = AF_LMS_HPOLY_F(F, x, d);
```

where

- F is the adaptive filter structure;
- x is the input signal sample $x[n]$;
- d is the desired signal sample $d[n]$;
- y is the output signal sample $y[n]$;
- e is the error signal sample $e[n]$.

The complete Matlab code is reported below.

```
function [F, y, e] = AF_LMS_HPOLY_F(F, x, d)


M=F.M;                                  % Length of the linear filter

F.xw(2 : M) = F.xw(1 : M-1);            % Shift the input delay-line
[F.xw(1), F.af] = ActFunc(x, F.af);     % Load a new input into the delay-↩
    line

y = F.xw.'*F.w;                         % (1,1) = (1,M) x (M,1) output ↩
    array
e = d - y;                              % Error evaluation

% Linear Filter ---------------------------------------------------------
F.w = F.w + F.mu*F.xw.*e;               % Updating the filter taps
%F.w(1) = 1;                            % If first tap should be 1

% Polynomial Nonlinearity -----------------------------------------------
if F.af.aftype == 3                     % 3 = Polynomial
    F.af.gM(:,2:M) = F.af.gM(:,1:M-1); % Shift the U matrix buffer
    F.af.gM(:,1) = F.af.g.';            % Load a new vector in matrix U
    u = F.af.gM*F.w;                    % Evaluating u

    e_av = F.mQ*e;
    F.af.Q = F.af.Q + e_av*u;           % Updating the polynomial ↩
        coefficients
end
```

2.4.7 AF_LMS_HSPL_F

This function implements the adaptation of a Hammerstein SAF (HSAF) structure by using the LMS adaptive algorithm (see Figure 1.2). More details can be found in [7].

The usage is as follows:

```
[F, y, e] = AF_LMS_HSPL_F(F, x, d);
```

where

- F is the adaptive filter structure;

- x is the input signal sample $x[n]$;
- d is the desired signal sample $d[n]$;
- y is the output signal sample $y[n]$;
- e is the error signal sample $e[n]$.

The complete Matlab code is reported below.

```
function [F, y, e] = AF_LMS_HSPL_F(F, x, d)


M = F.M;                                % Length of the linear filter
F.xw(2 : M) = F.xw(1 : M-1);            % Shift the input delay-line
[F.xw(1), F.af] = ActFunc(x, F.af);     % Load a new input into the delay-↩
    line
for j=2:M,                              % Constructing the matrix U
    if F.af.uIndex == F.af.indexes(j)
        F.af.gM(j,:) = F.af.gM(j-1,:);
    else
        F.af.gM(j,:) = zeros(1,4);
    end
end
F.af.gM(1,:) = F.af.g;                  % Load a new vector in matrix U
F.af.indexes(2:M) = F.af.indexes(1:M-1);
F.af.indexes(1) = F.af.uIndex;

y = F.xw.'*F.w;                         % Filter output
e = d - y;                              % Error estimation


% LMS weights and control points update ----------------------------------
F.w = F.w + F.mu*F.xw.*e;               % LMS in Eq. (10)

if F.af.aftype > 1  % Kind act. f. -1 0 1 2 4 5 *
    e_av = F.mQ*e;
    ii = F.af.uIndex : F.af.uIndex + F.af.P;        % P = Spline order
    F.af.Q(ii) = F.af.Q(ii) + e_av*(F.w'*F.af.gM).'; % LMS in Eq. (11)
end
```

2.4.8 AF_LMS_LNLJenkins_F

This function implements the adaptive linear-nonlinear-linear (LNL) sandwich model by using the LMS adaptive algorithm (see Figure 1.4). More details can be found in [2].

The usage is as follows:

```
[F, y, e] = AF_LMS_LNLJenkins_F(F, x, d);
```

where

- F is the adaptive filter structure;
- x is the input signal sample $x[n]$;
- d is the desired signal sample $d[n]$;
- y is the output signal sample $y[n]$;

- e is the error signal sample $e[n]$.

The complete Matlab code is reported below.

```
[F, y, e] = AF_LMS_LNLJenkins_F(F, x, d)


M1 = F.M1;                          % Length of the first filter
M2 = F.M2;                          % Length of the second filter
M3 = F.M3;                          % Length of the thirs filter

F.xw1(2 : M1) = F.xw1(1 : M1-1);    % Shift the filter 1 delay-line
F.xw1(1) = x;                       % Load a new input in the delay-line

F.xw2(2 : M2) = F.xw2(1 : M2-1);    % Shift the filter 2 delay-line
F.xw2(1) = F.xw1'*F.w1;             % Load a new sample in the delay-line
z = zeros(F.af.Pord,1);
for j=1:F.af.Pord,
    z(j) = F.xw2(1).^j;             % Evaluate the polynomial nonlinearity
end

F.xw3(2 : M3) = F.xw3(1 : M3-1);    % Shift the filter 3 delay-line
F.xw3(1) = z'*F.w2;                 % Load a new sample in the delay-line

y = F.xw3'*F.w3;                    % Output evaluation
e = d - y;                          % Error evaluation

zp = zeros(F.af.Pord,1);
for j=2:F.af.Pord,
    zp(j) = j*F.xw2(1).^(j-1);      % Evaluate the polynomial nonlinearity
end
zp(1) = 1;
zq = zp'*F.w2;
F.X(:,2:M3) = F.X(:,1:M3-1);
F.X(:,1) = zq.*F.xw1;
F.Z(:,2:M3) = F.Z(:,1:M3-1);
F.Z(:,1) = z;

% LMS weights update ------------------------------------------------------
F.w1 = F.w1 + F.mu1*conj(e)*F.X*F.w3;   % Updatade first filter
F.w2 = F.w2 + F.mu2*conj(e)*F.Z*F.w3;   % Updatade second filter
F.w3 = F.w3 + F.mu3*conj(e)*F.xw3;      % Updatade third filter

% NLMS weights update -----------------------------------------------------
% F.w1 = F.w1 + F.mu1*conj(e)*F.X*F.w3/(F.dI + F.w3'*F.X'*F.X*F.w3);
% F.w2 = F.w2 + F.mu2*conj(e)*F.Z*F.w3/(F.dI + F.w3'*F.Z'*F.Z*F.w3);
% F.w3 = F.w3 + F.mu3*conj(e)*F.xw3/(F.dI + F.xw3'*F.xw3);
```

2.4.9 AF_LMS_MATHEWS_POLY_F

This function implements the adaptive linear-nonlinear-linear (LNL) sandwich model by using the LMS adaptive algorithm (see Figure 1.4). More details can be found in [3].

The usage is as follows:

```
[F, y, e] = AF_LMS_MATHEWS_POLY_F(F, x, d);
```

where

- F is the adaptive filter structure;
- x is the input signal sample $x[n]$;
- d is the desired signal sample $d[n]$;
- y is the output signal sample $y[n]$;
- e is the error signal sample $e[n]$.

The complete Matlab code is reported below.

```
[F, y, e] = AF_LMS_MATHEWS_POLY_F(F, x, d)


M = F.M;                                % Length of the linear filter
P = F.af.Pord;                          % Polynomial order

theta = [F.w(2:M).' F.af.Q.'].';
Lambda = diag(F.mu*ones(1,M+P-1));

F.xw(2 : M) = F.xw(1 : M-1);            % Shift the filter delay-line
[F.xw(1), F.af] = ActFunc(x, F.af);     % Load a new input into the delay-↩
    line
F.af.gM(:,2:M) = F.af.gM(:,1:M-1);      % Shift the filter delay-line
F.af.gM(:,1) = F.af.g.';                % Load a new sample into the delay-↩
    line

H = [F.xw(2:M).' F.af.g].';               % Construct the matrix H
Psi = [F.xw(2:M).' (F.af.gM*F.w).'].';  % Construct the matrix Psi

y = H.'*theta;                          % Output evaluation
e = d - y;                              % Error evaluation

% LMS weights and Polynomial coefficients update ------------------------
theta = theta + Lambda*Psi*inv(F.dI + H.'*Lambda*Psi)*e;  % Update all ↩
    parameters

F.w(2:M) = theta(1:M-1);                % Linear filter weights
F.w(1) = 1;                             % First tap is always 1
F.af.Q = theta(M:M+P-1);                % Polynomials coefficients
```

2.4.10 AF_LMS_Sandwich1SPL_F

This function implements the adaptation of a Sandwich 1 SAF (S1SAF) structure by using the LMS adaptive algorithm (see Figure 1.3). More details can be found in [9].

The usage is as follows:

```
[F, y, e] = AF_LMS_Sandwich1SPL_F(F, x, d);
```

where

- F is the adaptive filter structure;
- x is the input signal sample $x[n]$;
- d is the desired signal sample $d[n]$;
- y is the output signal sample $y[n]$;

- e is the error signal sample $e[n]$.

The complete Matlab code is reported below.

```
[F, y, e] = AF_LMS_Sandwich1SPL_F(F, x, d)


M = F.M;          % Length of the linear filter

F.xw(2 : M) = F.xw(1 : M-1);           % Shift the input delay-line
[F.xw(1), F.af1] = ActFunc(x, F.af1);  % Load a new input into the delay-↩
    line

for j=2:M,          % Construct the U matrix
    if F.af1.uIndex == F.af1.indexes(j)
        F.af1.gM(j,:) = F.af1.gM(j-1,:);
    else
        F.af1.gM(j,:) = zeros(1,4);
    end
end
F.af1.gM(1,:) = F.af1.g;
F.af1.indexes(2:M) = F.af1.indexes(1:M-1);
F.af1.indexes(1) = F.af1.uIndex;

r = F.xw.'*F.w;                % Linear filter output array
[y,F.af2] = ActFunc(r,F.af2);  % Output array
e = d - y;                     % Error array
ee = e*dActFunc(r,F.af2);      % Error multiplied the nonlinearity ↩
    derivative

% LMS weights and control points update ----------------------------------
F.w = F.w + F.mu*conj(ee)*F.xw;   % cpx LMS in Eq. (30)

if F.af1.aftype > 1  % Kind act. f. -1 0 1 2 4 5 *
    e_av = F.mQ1*ee;
    ii = F.af1.uIndex : F.af1.uIndex + F.af1.P;  % P = Spline order

    F.af1.Q(ii) = F.af1.Q(ii) + e_av*(F.w'*F.af1.gM).';  % LMS in Eq. ↩
        (31)
end

if F.af2.aftype > 1  % Kind act. f. -1 0 1 2 4 5 *
    e_av = F.mQ2*e;
    ii = F.af2.uIndex : F.af2.uIndex + F.af2.P;  % P = Spline order

    F.af2.Q(ii) = F.af2.Q(ii) + e_av*F.af2.g.';   % LMS in Eq. (32)
end
```

2.4.11 AF_LMS_Sandwich2SPL_F

This function implements the adaptation of a Sandwich 2 SAF (S2SAF) structure by using the LMS adaptive algorithm (see Figure 1.4). More details can be found in [9].

The usage is as follows:

```
[F, y, e] = AF_LMS_Sandwich2SPL_F(F, x, d);
```

where

- F is the adaptive filter structure;
- x is the input signal sample $x[n]$;
- d is the desired signal sample $d[n]$;
- y is the output signal sample $y[n]$;
- e is the error signal sample $e[n]$.

The complete Matlab code is reported below.

```
[F, y, e] = AF_LMS_Sandwich2SPL_F(F, x, d)


M1 = F.M1;                          % Length of the first linear filter
M2 = F.M2;                          % Length of the seconf linear filter

F.xw1(2 : M1) = F.xw1(1 : M1-1);  % Shift the input filter 1 delay-line
F.xw1(1) = x;                       % Load a new input into the delay-line
F.xs(2 : M2)  = F.xs(1 : M2-1);   % Shift the nonlinearity delay-line
F.xs(1) = F.xw1'*F.w1;              % Load a new input into the delay-line
F.xw2(2 : M2) = F.xw2(1 : M2-1);  % Shift the filter 2 delay line
[F.xw2(1),F.af] = ActFunc(F.xs(1),F.af);  % Load a new input into the ↩
    delay-line
F.X(:,2:M2) = F.X(:,1:M2-1);       % Shift the data matrix delay-line
F.X(:,1) = F.xw1;                   % Load a new input into the matrix ↩
    delay-line
y = F.xw2'*F.w2;                    % Output array
e = d - y;                          % Error array
ee = e*dActFunc(F.xs(1),F.af);     % Error multiplied the nonlinearity ↩
    derivative

for j=2:M2,   % Update the U matrix
    if (F.af.uIndex >= F.af.indexes(j)-F.af.P) && (F.af.uIndex <= F.af.↩
        indexes(j)+F.af.P)
        F.af.gM(j,:) = F.af.gM(j-1,:);
    else
        F.af.gM(j,:) = zeros(1,4);
    end
end
F.af.gM(1,:) = F.af.g;
F.af.indexes(2:M2) = F.af.indexes(1:M2-1);
F.af.indexes(1) = F.af.uIndex;

% LMS weights and LUT control points update ------------------------------
F.w1 = F.w1 + F.mu1*conj(ee)*F.X*F.w2;  % cpx LMS in Eq. (32)
F.w2 = F.w2 + F.mu2*conj(e)*F.xw2;      % cpx LMS in Eq. (30)

if F.af.aftype > 1  % Kind act. f. -1 0 1 2 4 5 *
    e_av = F.mQ*e;
    ii = F.af.uIndex : F.af.uIndex + F.af.P;  % P = Spline order

    F.af.Q(ii) = F.af.Q(ii) + e_av*F.af.gM.'*F.w2;   % LMS in Eq. (31)
end
```

2.4.12 AF_LMS_WPOLY_F

This function implements the adaptation of an FIR Wiener polynomial (WPOLY) structure by using the LMS adaptive algorithm (see Figure 1.1). More details can be found in [6].

The usage is as follows:

```
[F, y, e] = AF_LMS_WPOLY_F(F, u, d);
```

where

- F is the adaptive filter structure;
- u is the input signal sample $u[n]$;
- d is the desired signal sample $d[n]$;
- y is the output signal sample $y[n]$;
- e is the error signal sample $e[n]$.

The complete Matlab code is reported below.

```
[F, y, e] = AF_LMS_WPOLY_F(F, u, d)


if nargin==0, help AF_LMS; return; end

M = F.M;                                        % Length of the linear ↩
    filter

% Linear Filter ---------------------------------------------------------
F.xw(2:M)         = F.xw(1:M-1);                % Shift the input delay-line
F.xw(1)           = u;                          % Load a new input into the ↩
    delay-line
s                 = F.w'*F.xw;                  % Linear combiner output
[y, F.af]         = ActFunc(s,F.af);            % Nonlinear output
e                 = d - y;                      % Error evaluation
ee                = e*dActFunc(s, F.af);        % Error multiplied by the ↩
    derivative
F.w               = F.w + F.mu*conj(ee)*F.xw; % Updating the filter taps

% Polynomial Nonlinearity -----------------------------------------------
if F.af.aftype == 3    % 3 = Polynomial
    e_av = F.mQ*e;
    F.af.Q = F.af.Q + e_av*F.af.g';           % Updating the polynomial ↩
        coefficients
end
```

2.4.13 AF_LMS_WPOLY_IIR_F

This function implements the adaptation of an IIR Wiener polynomial (WPOLY) structure by using the LMS adaptive algorithm (see Figure 1.1). More details can be found in [8].

The usage is as follows:

```
[F, y, e] = AF_LMS_WPOLY_IIR_F(F, u, d);
```

where

- F is the adaptive filter structure;
- u is the input signal sample $u[n]$;

- d is the desired signal sample $d[n]$;
- y is the output signal sample $y[n]$;
- e is the error signal sample $e[n]$.

The complete Matlab code is reported below.

```
[F, y, e] = AF_LMS_WPOLY_IIR_F(F, u, d)


if nargin==0, help AF_LMS; return; end

M = F.M;  % Length of the MA part
N = F.N;  % Length of the AR part

% Linear Filter ---------------------------------------------------------
F.xw(2:M)         = F.xw(1:M-1);                          % Shift the input ↩
    delay-line
F.xw(1)           = u;                                    % Load a new input↩
     into the delay-line
F.sw(2:N+1)       = F.sw(1:N);                            % Shift the ↩
    internal delay-line
F.sw(1)           = F.b'*F.xw + F.a'*F.sw(2:N+1);         % Load a new input↩
     into the internal delay-line with the output of an IIR filter
[y, F.af]         = ActFunc(F.sw(1),F.af);                % Output of the ↩
    IIR Polynomial AF
e                 = d - y;                                % Error evaluation
ee                = e*dActFunc(y, F.af);                  % Error multiplied↩
     by the nonlinear derivative
F.w               = [F.b.'  F.a.'].';                     % ARMA parameters
F.beta(:,2:N+1)   = F.beta(:,1:N);                        % Shift the beta ↩
    delay-line
F.beta(:,1)       = F.xw + F.beta(:,2:N+1)*F.a;           % Updating the ↩
    beta delay-line
F.alpha(:,2:N+1) = F.alpha(:,1:N);                        % Shift the alpha ↩
    delay-line
F.alpha(:,1)      = F.sw(2:N+1) + F.alpha(:,2:N+1)*F.a; % Updating the ↩
    alpha delay-line
eta               = [F.beta(:,1).'  F.alpha(:,1).']; % Constructing the↩
     eta vector
F.w               = F.w + F.mu*conj(ee)*eta;              % Updating all the↩
     parameters
F.b               = F.w(1:M);                             % Updating the MA ↩
    parameters
F.a               = F.w(M+1:M+N);                         % Updating the AR ↩
    parameters

% Polynomial Nonlinearity -----------------------------------------------
if F.af.aftype == 3      % 3 = Polynomial
    e_av = F.mQ*e;
    F.af.Q = F.af.Q + e_av*F.af.g';   % Coefficients update
end
```

2.4.14 AF_LMS_WSPL_F

This function implements the adaptation of a Wiener SAF (WSAF) structure by using the LMS adaptive algorithm (see Figure 1.1). More details can be found in [6].

The usage is as follows:

```
[F, y, e] = AF_LMS_WSPL_F(F, u, d);
```

where

- F is the adaptive filter structure;
- u is the input signal sample $u[n]$;
- d is the desired signal sample $d[n]$;
- y is the output signal sample $y[n]$;
- e is the error signal sample $e[n]$.

The complete Matlab code is reported below.

```
[F, y, e] = AF_LMS_WSPL_F(F, u, d)


M         = F.M;                % Length of the linear filter
F.xw(2:M) = F.xw(1:M-1);        % Shift of the filter delay-line
F.xw(1)   = u;                  % Load a new input into the delay-line
s         = F.w'*F.xw;          % Otput of the linear filter
[y,F.af]  = ActFunc(s,F.af);    % Output of the nonlinearity
e         = d - y;              % Error evaluation
ee        = e*dActFunc(s,F.af); % Error multiplied by the derivative

% LMS weights and control points update ----------------------------------
F.w       = F.w + F.mu*conj(ee)*F.xw;   % cpx LMS in Eq. (15)

if F.af.aftype > 1   % Kind act. f. -1 0 1 2 3 4 5 6 7 ... 20  */
    e_av = F.mQ*e;
    ii = F.af.uIndex : F.af.uIndex + F.af.P;   % P = Spline order
    F.af.Q(ii) = F.af.Q(ii) + e_av*F.af.g';    % LMS in Eq. (16)
end
```

2.4.15 AF_LMS_WSPL_IIR_F

This function implements the adaptation of an IIR Wiener SAF (WSAF) structure by using the LMS adaptive algorithm (see Figure 1.1). More details can be found in [8].

The usage is as follows:

```
[F, y, e] = AF_LMS_WSPL_IIR_F(F, u, d);
```

where

- F is the adaptive filter structure;
- u is the input signal sample $u[n]$;
- d is the desired signal sample $d[n]$;
- y is the output signal sample $y[n]$;
- e is the error signal sample $e[n]$.

The complete Matlab code is reported below.

```
[F, y, e] = AF_LMS_WSPL_IIR_F(F, u, d)


if nargin==0, help AF_LMS; return; end

M = F.M;     % Length of the MA part
N = F.N;     % Length of the AR part

% Linear Filter ---------------------------------------------------------
F.xw(2:M)         = F.xw(1:M-1);                    % Shift the input delay-line
F.xw(1)           = u;                              % Load a new input into the delay-line
F.sw(2:N+1)       = F.sw(1:N);                      % Shift the internal delay-line
F.sw(1)           = F.b'*F.xw + F.a'*F.sw(2:N+1);   % Load a new input into the internal delay-line with the output of an IIR filter
[y, F.af]         = ActFunc(F.sw(1),F.af);          % Output of the IIR WSAF
e                 = d - y;                          % Error evaluation
ee                = e*dActFunc(y, F.af);            % Error multiplied by the nonlinear derivative
F.w               = [F.b.'  F.a.'].';               % ARMA parameters
F.beta(:,2:N+1)   = F.beta(:,1:N);                  % Shift the beta delay-line
F.beta(:,1)       = F.xw + F.beta(:,2:N+1)*F.a;     % Updating the beta delay-line in Eq. (14)
F.alpha(:,2:N+1)  = F.alpha(:,1:N);                 % Shift the alpha delay-line
F.alpha(:,1)      = F.sw(2:N+1) + F.alpha(:,2:N+1)*F.a; % Updating the alpha delay-line in Eq. (15)
eta               = [F.beta(:,1).'  F.alpha(:,1).']; % Constructing the eta vector in Eq. (16)
F.w               = F.w + F.mu*conj(ee)*eta;        % Updating all the parameters as in Eq. (17)
F.b               = F.w(1:M);                       % Updating the MA parameters
F.a               = F.w(M+1:M+N);                   % Updating the AR parameters

% Spline Nonlinearity ---------------------------------------------------
if F.af.aftype > 1    % Kind act. f. -1 0 1 2 3 4 5 6 7 ... 20  */
    e_av = F.mQ*e;
    ii = F.af.uIndex : F.af.uIndex + F.af.P;      % P = Spline order
    for  m = 1:1
        F.af.Q(ii) = F.af.Q(ii) + e_av*F.af.g';   % LMS in Eq. (19)
    end
end
```

CHAPTER 3

Demo Scripts

In this chapter, some demo files will be presented. The aim of these scripts is to provide a simple example of how the library files can be used. Moreover, these scripts are a good start point for developing own algorithms based on SAF, simply changing, for example, the model definition, the number of filters, type of nonlinearities, and so on.

These demo examples are extracted from the experimental results that can be found in [6, 7, 8, 9, 10].

In the following sections, the scripts will be described in alphabetically order.

3.1 HSAF_demo

This demo file implements a convergence test of a Hammerstein spline adaptive filter (HSAF). The idea is shown in Figure 3.1 It is the first experiment of [7].

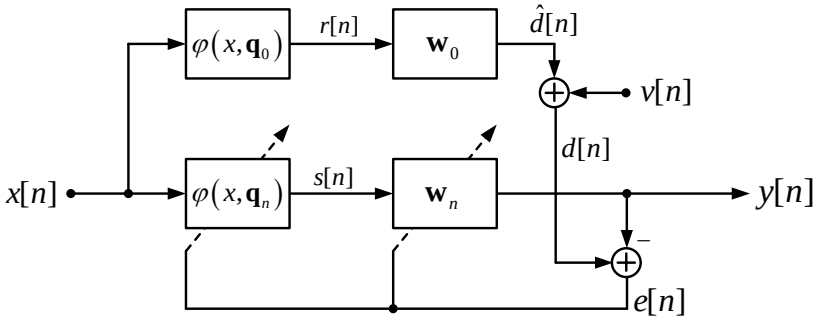

Fig. 3.1: Example of model used for the identification of an unknown Hammerstein system.

The experiment consists in the identification of an unknown Hammerstein system composed by a linear component $\mathbf{w}_0 = [0.6, -0.4, 0.25, -0.15, 0.1, -0.05, 0.001]^T$

and a nonlinear memoryless target function implemented by a 23-points length LUT $\mathbf{q}_0$ and interpolated by a uniform third degree spline with an interval sampling $\Delta x = 0.2$ defined as

$$\mathbf{q}_0 = \{-2.2, -2.0, -1.8, \ldots, -1.0, -0.8, -0.91, -0.40, -0.20, \\ 0.05, 0.0, -0.40, 0.58, 1.0, 1.0, 1.2, 1.4, \ldots, 2.2\}.$$

The input signal $x[n]$ consists in 30000 samples of the signal generated by the following relationship

$$x[n] = ax[n-1] + \sqrt{1-a^2}\xi[n],$$

where $\xi[n]$ is a zero mean white Gaussian noise with unitary variance and $0 \leq a < 1$ is a parameter that determines the level of correlation between adjacent samples. Experiments were conducted with a set to 0.1 and 0.95. In addition it is considered an additive white noise $v[n]$ such that the signal to noise ratio is $SNR = 60$ dB. The learning rates are set to $\mu_w = \mu_q = 0.1$. The filter weights are initialized using $\alpha = 0.1$.

The complete MATLAB code is reported below.

```
clear all;
close all;
disp('HSAF_demo');
% ------------------------------------------------------------------------

%%  Parameters setting

% Input parameters ------------------------------------------------------
Lx = 30000;                 % Length of input signal
nRun = 10;                  % Number of runs
out_noise_level_dB = 60;  % SNR
out_noise_level = 10^(-out_noise_level_dB/20);  % Noise level

x  = zeros(Lx,1);           % x (Nx1) input signal array definition

% Colored signal generation ---------------------------------------------
a = 0.1;
b = sqrt(1-a^2);
%x = filter( b, [1 -a], randn( size(x))) ;  % H(z) = b/(1+a*z^-1)
disp('..... ');

% Adaptive filter definition --------------------------------------------
M =  7;             % Length of linear filter
mu0 = 0.1;          % Learning rate for linear filter
mQ0 = 0.1;          % Learning rate for control points
if Lx < 30000,      % Batch for evaluating MSE
        B = 100;
else
        B = 4000;
end

% Spline activation function definition and initialization --------------
afinit = 0;     % Init act. func. -1 0 ... (ONLY -1, bip.sig. or  0 =↩
    linear )
aftype = 4;     % Kind act. f. -1 0 1 2 4 5; (4 = CR-spline, 5 = B-spline ↩
    )
Slope  = 1;     % Slope
DeltaX = 0.2;  % Delta X
```

```
x_range = 2;   % Range limit

% Creating the nonlinearity ---------------------------------------------
af0 = create_activation_function(afinit, aftype, DeltaX, x_range, Slope, ↩
    M);  % Model
af1 = create_activation_function(afinit, aftype, DeltaX, x_range, Slope, ↩
    M);  % SAF


%% Initialization

% --- Target Definition ------------------------------------------------
TH1 = create_SPL_lms_adaptive_filter_1(M,mu0,mQ0,1e-2,af0); % Target h ↩
    Model LMS

% TARGET: Nonlinear memoryless function implemented by Spline ↩
    interpolated LUT
Q0 = [      -2.20
            -2.00
            -1.80
            -1.60
            -1.40
            -1.20
            -1.00
            -0.80
            -0.91
            -0.40
            -0.20
             0.05
             0.0
            -0.40
             0.58
             1.00
             1.00
             1.20
             1.40
             1.60
             1.80
             2.00
             2.20
];
TH1.af.Q = Q0;
QL = length(Q0);   % Number of control points

% Linear filter ---------------------------------------------------------
TH1.w  = [0.6  -0.4  0.25  -0.15  0.1  -0.05  0.001]';  % MA system to be↩
     identified

% --- SAF definition ---------------------------------------------------
H1 = create_SPL_lms_adaptive_filter_1(M,mu0,mQ0,1e-2,af1); % HSAF LMS

% Initialize ------------------------------------------------------------
N = Lx + M + 1;   % Total samples
for i = Lx+1:N
    x(i)=0;
end

dn = zeros(N,1);          % Noise desired output array
d  = zeros(N,1);          % Desired signal array
y  = zeros(N,1);          % Output array
e  = zeros(Lx,1);         % Error array
em = zeros(Lx,1);         % Mean square error
```

```
wm   = zeros(M,1);        % Mean value of w
varW = zeros(M,1);        % Variance value of w
qm   = zeros( QL, 1 );    % Mean value Spline coeff
varQ = zeros( QL, 1 );    % Variance value Spline coeff


%% Main loop ------------------------------------------------------------
disp('Algorithm start ... ');
t = clock;

for n = 0 : nRun-1
    fprintf('Test nr. %d/%d\n', n+1, nRun);
    x = filter( b, [1 -a], randn( size(x)) ) ;   % H(z) = b/(1+a*z^-1)
    dn = out_noise_level * randn( size(x) ) ;    % Noise

    % SAF I.C. ------------------------------------------------------------
    H1.w (:) = 0;
    H1.w (1) = 0.1;

    % Set Activation Func I.C. --------------------------------------------
    H1.af.Q = af0.Q;

    % HSAF Evaluation -----------------------------------------------------
    for k = 1 : Lx
        % Computing the desired output ------------------------------------
        [TH1, d(k), snk] = FW_HSPL_F(TH1, x(k));     % Hammerstein model

        % Updating HSAF ---------------------------------------------------
        [H1, y(k), e(k)] = AF_LMS_HSPL_F(H1, x(k), d(k) + dn(k) );  % SAF↩
             LMS ( Eqs. (10) and (11) )
    end

    em = em + (e.^2);   % Squared error

    % SAF run-time mean and variance estimation ---------------------------
    wm   = (1/(n+1))*H1.w + (n/(n+1))*wm;
    varW = varW  + (n/(n+1))*((TH1.w - wm).^2);

    qm   = (1/(n+1))*H1.af.Q + (n/(n+1))*qm;
    varQ = varQ     + (n/(n+1))*((TH1.af.Q - qm).^2);

end
em  = em/nRun;  % MSE
H1.af.Q = qm;

%-------------------------------------------------------------------------
% Average MSE evaluation
mse = mean(em(end-B-M-1:end-M-1));  % Average MSE
%-------------------------------------------------------------------------
fprintf('\n');


%% Results

%--------------------------------------------------------------------------
% Print table of means and variances
%--------------------------------------------------------------------------
fprintf('\n');
fprintf('Number of iterations = %d\n',nRun);
fprintf('Learning rates:  muW = %5.3f   muQ = %5.3f\n', mu0, mQ0);
fprintf('a = %4.2f  b = %4.2f\n',a, b );
fprintf('Number of filter weights = %d\n', M);
```

```
fprintf('Number of control points = %d\n', QL);
fprintf('AF type = %d\n',aftype);
fprintf('DeltaX  = %4.2f\n',DeltaX);
fprintf('SNR_dB  = %4.2f dB\n',out_noise_level_dB);
fprintf('Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse,10*log10(mse)↩
    );
fprintf('\n');
fprintf('Mean and Variance Tables ----------------------------------------\n');
for i=1:QL
    fprintf('i=%2d  q0 =%5.2f    qm =%9.6f    varQ = %10.3e \n', i,  TH1.↩
        af.Q(i), qm(i), varQ(i) );
end
fprintf('\n');
fprintf('-----------------------------------------------------------------\n');
for i=1:M
    fprintf('i=%d  w0 =%5.2f    wm =%9.6f    varW = %10.3e \n', i, TH1.w(i↩
        ), wm(i), varW(i) );
end
fprintf('-----------------------------------------------------------------\n');


% -------------------------------------------------------------------------
% Plotting figures
% -------------------------------------------------------------------------

% Plot Spline functions ---------------------------------------------------
yLIM = 1.5;
xLIM = 3.0;
figure1 = figure('PaperSize',[20.98 29.68]);
box('on');
hold on;
hold('all');
ylim([-yLIM yLIM]);
xlim([-xLIM xLIM]);
grid on;
KK = 500;
yy1 = zeros(1,KK);
yy2 = zeros(1,KK);
xa1 = zeros(1,KK);
dx = 2*xLIM/KK;
xx = -xLIM;
for k = 1:KK
    yy1(k) = ActFunc(xx, TH1.af);   % Model
    yy2(k) = ActFunc(xx, H1.af);    % Adapted
    xa1(k) = xx;
    xx = xx + dx;
end
xlabel('Input {\itx}[{\itn}] ','FontSize', 12, 'FontWeight', 'demi');
ylabel('SAF output {\ity}[{\itn}]','FontSize', 12, 'FontWeight', 'demi');
title('Profile of model and adapted nonlinearity \varphi(x[n])','FontSize↩
    ', 12, 'FontWeight', 'demi');
plot(xa1,yy1,'-.r','LineWidth',2);
plot(xa1,yy2,'k','LineWidth',2);
legend('Target','Adapted','Location','SouthEast');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');

% Filter coefficients -----------------------------------------------------
figure2 = figure('PaperSize',[20.98 29.68]);
hold on;
plot(TH1.w,'LineWidth',2);
plot(H1.w,'m','LineWidth',2);
xlabel('time {\itn}','FontSize', 12, 'FontWeight', 'demi');
```

```
ylabel('Linear combiner coefficients','FontSize', 12, 'FontWeight', 'demi↩
    ');
legend('Model','Adapted');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');

% MSE dB ----------------------------------------------------------------
figure3 = figure('PaperSize',[10 15]);
box('on');hold on;hold('all');
ylim([-out_noise_level_dB-5 10]);
grid on;
edb = 10*log10( em );
[bb,aa] = butter(2, 0.02 );
plot( filter(bb,aa,edb ),'Color',[1 0 0],'LineWidth',2);
noiseLevel(1: length(edb)-1 ) = -out_noise_level_dB;
plot( noiseLevel,'--','Color',[0 0 1],'LineWidth',2 );
title('Hammerstein SAF convergence test','FontSize', 12, 'FontWeight', '↩
    demi');
xlabel('Samples','FontSize', 12, 'FontWeight', 'demi');
ylabel('MSE [dB]   10log({\itJ}({\bfw},{\bfQ}))','FontSize', 12, '↩
    FontWeight', 'demi');
legend('MSE', 'NoiseLevel' );
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);
% -----------------------------------------------------------------------

fprintf('END HSAF_demo ------------------------------------------------\n');
```

3.2 HSAF_DX_compare_demo

This experiment consists in the identification of a nonlinear dynamic system composed by two blocks (see Figure 3.1). The first block is the following nonlinearity

$$s[n] = \frac{1}{8} + \lfloor 2x[n] - 1 \rfloor ,$$

where $\lfloor \bullet \rfloor$ is the floor operator and the second block is the FIR filter with taps

$$\mathbf{h} = \begin{bmatrix} 1 & 0.75 & 0.5 & 0.25 & 0 & -0.25 \end{bmatrix}^T .$$

The learning rates are set to $\mu_w = 0.01$ and $\mu_q = 0.02$ and B-spline basis is used using $M = 15$. We compare the proposed HSAF architecture with different values of Δx chosen in the set

$$\Delta x = \{0.1,\ 0.2,\ 0.3,\ 0.5\} .$$

A comparison of the averaged MSE over 100 trials is also performed.

The complete MATLAB code is reported below.

```
clear all;
close all;
disp('HSAF_DX_compare_demo');
% -----------------------------------------------------------------------

%%  Parameters setting

% Input parameters ------------------------------------------------------
Lx = 50000;                     % Length of the input signal
nRun = 10;                      % Number of runs
```

```
out_noise_level_dB = 60;       % SNR
out_noise_level = 10^(-out_noise_level_dB/20);   % Noise level

x = randn(Lx,1);               % x (Nx1) input signal array definition

% Colored signal generation ------------------------------------------
a = 0.0;
b = sqrt(1-a^2);
%x = filter( b, [1 -a], randn( size(x))) ;  % H(z) = b/(1+a*z^-1)
disp('..... ');

% Adaptive filter definition -----------------------------------------
M  =  15;              % Length of the linear filter
K  =   1;              % Affine Projection order; K=1 => NLMS
mu1 = 0.01 ;           % Learning rate of the linear filter
mQ1 = 0.02;            % Learning rate for ctrl points
delta = 0.1;           % APA regularization parameters
if Lx < 30000,         % Batch for evaluating MSE
        B = 100;
else
        B = 4000;
end

% Spline activation function definition and initialization -----------
afinit  = 0;       % Init act. func. -1 0 ... (ONLY -1, bip.sig. or  0 =↩
    linear )
aftype  = 5;       % Kind act. f. -1 0 1 2 4 5; (4 = CR-spline, 5 = B-↩
    spline )
Slope   = 1;       % Slope
x_range = 4;       % Range limit
% Definition of several values of Delta X parameter ------------------
DeltaX1 = 0.1;
DeltaX2 = 0.2;
DeltaX3 = 0.3;
DeltaX4 = 0.5;

% Creating the nonlinearity ------------------------------------------
af1 = create_activation_function(afinit, aftype, DeltaX1, x_range, Slope,↩
     M);   % Delta X 1
af2 = create_activation_function(afinit, aftype, DeltaX2, x_range, Slope,↩
     M);   % Delta X 2
af3 = create_activation_function(afinit, aftype, DeltaX3, x_range, Slope,↩
     M);   % Delta X 3
af4 = create_activation_function(afinit, aftype, DeltaX4, x_range, Slope,↩
     M);   % Delta X 4


%% Initialization

% --- SAF definition ---------------------------------------------------
F1 = create_SPL_apa_adaptive_filter_1(M,K,mu1,mQ1,delta,af1);   % Delta X↩
     1
F2 = create_SPL_apa_adaptive_filter_1(M,K,mu1,mQ1,delta,af2);   % Delta X↩
     2
F3 = create_SPL_apa_adaptive_filter_1(M,K,mu1,mQ1,delta,af3);   % Delta X↩
     3
F4 = create_SPL_apa_adaptive_filter_1(M,K,mu1,mQ1,delta,af4);   % Delta X↩
     4



% Initialize ---------------------------------------------------------
```

```
N = Lx + M + 1;   % Total samples
for i = Lx+1:N
    x(i)=0;
end

dn = zeros(N,1);        % Noise output array
d  = zeros(N,1);        % Desired signal array
y1  = zeros(N,1);       % Output array
y2  = zeros(N,1);       % Output array
y3  = zeros(N,1);       % Output array
y4  = zeros(N,1);       % Output array
e1  = zeros(Lx,1);      % Error array
e2  = zeros(Lx,1);      % Error array
e3  = zeros(Lx,1);      % Error array
e4  = zeros(Lx,1);      % Error array
em1 = zeros(Lx,1);      % Mean square error
em2 = zeros(Lx,1);      % Mean square error
em3 = zeros(Lx,1);      % Mean square error
em4 = zeros(Lx,1);      % Mean square error


%% Main loop ------------------------------------------------------------
disp('Algorithm start ... ');
t = clock;

for n = 0 : nRun-1
    fprintf('Test nr. %d/%d\n', n+1, nRun);

    % Nonlinear system signal generation ---------------------------------
    x = randn(Lx,1);          % x (Nx1) input signal array definition
    if a>0,
        x = filter( b, [1 -a], x);              % Colored input
    end
    z = 1/8 + fix((8*x - 4)/4);                 % Nonlinearity
    h = zeros(1,6);
    for k=0:5
        h(k+1) = 1 - k/4;
    end
    d = 0.5*filter(h, 1, z);                    % Desired signal
    dn = out_noise_level*randn( size(x) );      % Noisy desired signal

    % SAF I.C. -----------------------------------------------------------
    F1.w(:) = 0;
    F1.w(1) = 1;  % Set filter i.c.
    F2.w(:) = 0;
    F2.w(1) = 1;  % Set filter i.c.
    F3.w(:) = 0;
    F3.w(1) = 1;  % Set filter i.c.
    F4.w(:) = 0;
    F4.w(1) = 1;  % Set filter i.c.

    % Set Activation Func I.C. -------------------------------------------
    F1.af.Q = af1.Q;
    F2.af.Q = af2.Q;
    F3.af.Q = af3.Q;
    F4.af.Q = af4.Q;

    % HSAF Evaluation ----------------------------------------------------
    for k = 1 : Lx
        % Delta X 1 ------------------------------------------------------
        [F1, y1(k), e1(k)] = AF_APA_HSPL_F(F1, x(k), d(k) + dn(k) );
```

```
        % Delta X 2 ------------------------------------------------------------
        [F2, y2(k), e2(k)] = AF_APA_HSPL_F(F2, x(k), d(k) + dn(k) );

        % Delta X 3 ------------------------------------------------------------
        [F3, y3(k), e3(k)] = AF_APA_HSPL_F(F3, x(k), d(k) + dn(k) );

        % Delta X 4 ------------------------------------------------------------
        [F4, y4(k), e4(k)] = AF_APA_HSPL_F(F4, x(k), d(k) + dn(k) );
    end

    % RUN time averaging -------------------------------------------------------
    em1 = em1 + e1.^2;    % Squared error
    em2 = em2 + e2.^2;    % Squared error
    em3 = em3 + e3.^2;    % Squared error
    em4 = em4 + e4.^2;    % Squared error

end

em1 = em1/nRun;    % MSE
em2 = em2/nRun;    % MSE
em3 = em3/nRun;    % MSE
em4 = em4/nRun;    % MSE

%-------------------------------------------------------------------------------
% Average MSE evaluation
mse1 = mean(em1(end-B-M-1:end-M-1));  % Average MSE
mse2 = mean(em2(end-B-M-1:end-M-1));  % Average MSE
mse3 = mean(em3(end-B-M-1:end-M-1));  % Average MSE
mse4 = mean(em4(end-B-M-1:end-M-1));  % Average MSE
%-------------------------------------------------------------------------------
fprintf('\n');


%% Results

% -----------------------------------------------------------------------------
% Print table
% -----------------------------------------------------------------------------

fprintf('\n');
fprintf('Mean Square Errors ↩
    ----------------------------------------------\n');
fprintf('HSAF with Delta X = %3.2f Steady-state MSE = %5.7f, equal to ↩
    %5.3f dB\n',DeltaX1,mse1,10*log10(mse1));
fprintf('HSAF with Delta X = %3.2f Steady-state MSE = %5.7f, equal to ↩
    %5.3f dB\n',DeltaX2,mse2,10*log10(mse2));
fprintf('HSAF with Delta X = %3.2f Steady-state MSE = %5.7f, equal to ↩
    %5.3f dB\n',DeltaX3,mse3,10*log10(mse3));
fprintf('HSAF with Delta X = %3.2f Steady-state MSE = %5.7f, equal to ↩
    %5.3f dB\n',DeltaX4,mse4,10*log10(mse4));
fprintf('↩
    ----------------------------------------------------------------------\n')↩
    ;


% -----------------------------------------------------------------------------
% Plotting figures
% -----------------------------------------------------------------------------

% Plot Spline functions ----------------------------------------------------
yLIM = 1.5;
xLIM = 3.0;
```

```
figure1 = figure('PaperSize',[20.98 29.68]);
box('on');
hold on;
hold('all');
grid on;
ylim([-yLIM yLIM]);
xlim([-xLIM xLIM]);
KK = 500;
yy1 = zeros(1,KK);
yy2 = zeros(1,KK);
yy3 = zeros(1,KK);
yy4 = zeros(1,KK);
xa1 = zeros(1,KK);
dx = 2*xLIM/KK;
xx = -xLIM;
for k = 1:KK
    yy1(k) = ActFunc(xx, F1.af);   % Delta X 1: Blue
    yy2(k) = ActFunc(xx, F2.af);   % Delta X 2: Black
    yy3(k) = ActFunc(xx, F3.af);   % Delta X 3: Green
    yy4(k) = ActFunc(xx, F4.af);   % Delta X 4: Magenta
    xa1(k) = xx;
    xx = xx + dx;
end
title('Profile of nonlinearities after learning','FontSize', 12, '↩
    FontWeight', 'demi');
xlabel('Nonlinearity input {\itx}[{\itn}] ','FontSize', 12, 'FontWeight',↩
     'demi');
ylabel('Nonlinearity output {\its}[{\itn}]','FontSize', 12, 'FontWeight',↩
     'demi');
plot(xa1,yy1,'b','LineWidth',2);   % Delta X 1: Blue
plot(xa1,yy2,'k','LineWidth',2);   % Delta X 2: Black
plot(xa1,yy3,'g','LineWidth',2);   % Delta X 3: Green
plot(xa1,yy4,'m','LineWidth',2);   % Delta X 4: Magenta
legend('Delta X 1','Delta X 2','Delta X 3','Delta X 4','Location','↩
    SouthEast');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);


% MSE dB ----------------------------------------------------------------
figure2 = figure('PaperSize',[20.98 29.68]);
box('on');
hold on;
grid on;
ylim([-out_noise_level_dB-5 0]);
[bb,aa] = butter(2, 0.005);
edb1 = 10*log10( em1 );
edb2 = 10*log10( em2 );
edb3 = 10*log10( em3 );
edb4 = 10*log10( em4 );
plot(filter(bb,aa,edb1),'b','LineWidth',2);       % Delta X 1: Blue
plot(filter(bb,aa,edb2),'k','LineWidth',2);       % Delta X 2: Black
plot(filter(bb,aa,edb3),'g','LineWidth',2);       % Delta X 3: Green
plot(filter(bb,aa,edb4),'m','LineWidth',2);       % Delta X 4: Magenta
title('Comparisons of MSE by using different \Delta_x values','FontSize',↩
     12, 'FontWeight', 'demi');
xlabel('Samples','FontSize', 12, 'FontWeight', 'demi');
ylabel('MSE [dB]   10log({\itJ}({\bfw},{\bfQ}))','FontSize', 12, '↩
    FontWeight', 'demi');
legend('Delta X 1','Delta X 2','Delta X 3','Delta X 4');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
```

3.3 HSAF_Polynomial_compare_demo

The script implements the fourth experiment of [7]. This experiment consists in the identification of a nonlinear dynamic system composed by two blocks (see Figure 3.1). The first block is the following nonlinearity

$$s[n] = \frac{1}{8} + \lfloor 2x[n] - 1 \rfloor ,$$

where $\lfloor \bullet \rfloor$ is the floor operator and the second block is the FIR filter with taps

$$\mathbf{h} = \begin{bmatrix} 1 & 0.75 & 0.5 & 0.25 & 0 & -0.25 \end{bmatrix}^T .$$

The learning rates are set to $\mu_w = 0.01$ and $\mu_q = 0.02$ and B-spline basis with $\Delta x = 0.2$ is used. We compare the proposed HSAF architecture with a polynomial Hammerstein architecture with the learning rate set to $\mu_p = 10^{-3}$, the approach proposed in [3] with $\mu = 10^{-5}$ implementing both a third order polynomial nonlinearity and a 3-rd order Volterra adaptive filter. For all architecture we use $M = 15$ parameters and a total of $5 \cdot 10^4$ samples are used. A comparison of the averaged MSE over 100 trials is also performed.

The complete MATLAB code is reported below.

```
clear all;
close all;
disp('HSAF_Polynomial_Compare_demo');
% ------------------------------------------------------------------------

%% Parameters setting

% Input parameters ------------------------------------------------------
Lx = 50000;                     % Length of the input signal
nRun = 10;                      % Number of runs
out_noise_level_dB = Inf;       % SNR
out_noise_level = 10^(-out_noise_level_dB/20);   % Noise level

x = zeros(Lx,1);                % x (Nx1) input signal array definition

% Colored signal generation ---------------------------------------------
a = 0.0;
b = sqrt(1-a^2);
% x = filter(b, [1 -a], randn( size(x)));   % H(z) = b/(1+a*z^-1)
disp('..... ');

% Adaptive filter definition --------------------------------------------
M  = 6;             % Length of the linear part of HSAF
MV = 15;            % Length of the Volterra filter
K  =  1;            % Affine Projection order; K=1 => NLMS
Kv =  1;            % Volterra Affine Projection order; K=1 => NLMS
delta = 1e-2;       % APA regularization parameters
mu  = 0.01;         % Learning rate for the linear filter of HSAF
mQ  = 0.02;         % Learning rate for ctrl points;
mu2 = 0.001;        % Learning rate for the linear part of Polynomial ↩
    filter
mQ2 = 0.001;        % Learning rate for the nonlinear part
mu3 = 0.00001;      % Learning rate for the linear filter of Mathews ↩
    system
mQ3 = 0.00001;      % Learning rate for the nonlinear filter
muV = 0.01;         % Learning rate for the Volterra filter
```

```
if Lx < 30000,     % Batch for evaluating MSE
        B = 100;
else
        B = 4000;
end

% Spline activation function definition and initialization ----------
afinit  = 0;        % Kind of init act. f. -1 0 1 2 3 */
aftype  = 5;        % Kind act. f. -1 0 1 2 3 4 5
Slope   = 1;        % Slope
DeltaX  = 0.2;      % Delta X
x_range = 2;        % Range limit
aftype2 = 3;        % Kind act. f. -1 0 1 2 3 4 5
Pord    = 3;        % Polynomial Order


%% Initialization

% Creating the nonlinearity ------------------------------------------

% Targets ------------------------------------------------------------
[af01]  = create_activation_function(afinit, aftype, DeltaX, x_range, ↩
    Slope, M);         % HSAF
[af02]  = create_activation_function(afinit, aftype2, DeltaX, x_range, ↩
    Slope, M, Pord);  % Polynomial
[af03]  = create_activation_function(afinit, aftype2, DeltaX, x_range, ↩
    Slope, M, Pord);  % Mathews

% Models -------------------------------------------------------------
[af1]   = create_activation_function(afinit, aftype, DeltaX, x_range, ↩
    Slope, M);         % HSAF
[af2]   = create_activation_function(afinit, aftype2, DeltaX, x_range, ↩
    Slope, M, Pord);  % Polynomial
[af3]   = create_activation_function(afinit, aftype2, DeltaX, x_range, ↩
    Slope, M, Pord);  % Mathews

% --- Models definition -------------------------------------------------
apa_af1 = create_SPL_apa_adaptive_filter_1(M, K, mu, mQ, delta, af1);   %↩
     HSAF APA
%apa_af1 = create_SPL_lms_adaptive_filter_1(M, mu, mQ, delta, af1);      ↩
    % HSAF LMS
apa_af2 = create_SPL_apa_adaptive_filter_1(M, K, mu2, mQ2, delta, af2); %↩
     Polynomial APA
%apa_af2 = create_SPL_lms_adaptive_filter_1(M, mu2, mQ2, delta, af2);    ↩
    % Polynomial LMS
apa_af3 = create_SPL_lms_adaptive_filter_1(M, mu2, mQ2, delta, af3);    %↩
     Mathews LMS
%VF    = create_II_ord_Volterra_filter_1(MV, Kv, muV, delta);            ↩
    % II Order Volterra
VF    = create_III_ord_Volterra_filter_1(MV, Kv, muV, delta);           %↩
     III Order Volterra


% Initialize ---------------------------------------------------------
N = Lx + M + K;
for i = Lx+1:N
    x(i)=0;
end

dn = zeros(N,1);        % Noise output array
d  = zeros(N,1);        % Desired signal array
y1  = zeros(N,1);       % Output array
```

```
y2  = zeros(N,1);       % Output array
y3  = zeros(N,1);       % Output array
y4  = zeros(N,1);       % Output array
e1  = zeros(Lx,1);      % Error array
e2  = zeros(Lx,1);      % Error array
e3  = zeros(Lx,1);      % Error array
e4  = zeros(Lx,1);      % Error array
em1 = zeros(Lx,1);      % Mean square error
em2 = zeros(Lx,1);      % Mean square error
em3 = zeros(Lx,1);      % Mean square error
em4 = zeros(Lx,1);      % Mean square error


%% Main loop ------------------------------------------------------------------
disp('HSAF_Polynomial_compare algorithm start ... ');
t = clock;

for n = 0:nRun-1
    fprintf('Test nr. %d/%d\n', n+1, nRun);

    % Nonlinear system signal generation -----------------------------------
    x = randn(Lx,1);           % x (Nx1) input signal array definition
    if a>0,
        x = filter( b, [1 -a], x);              % Colored input
    end
    z = 1/8 + fix((8*x - 4)/4);                 % Nonlinearity
    h = zeros(1,6);
    for k=0:5
        h(k+1) = 1 - k/4;
    end
    d = 0.5*filter(h, 1, z);                    % Desired signal
    dn = out_noise_level*randn( size(x) );      % Noisy desired signal

    % Models I.C. ----------------------------------------------------------
    apa_af1.w(:) = 0;
    apa_af1.w(1) = 1;
    apa_af2.w(:) = 0;
    apa_af2.w(1) = 1;
    apa_af3.w(:) = 0;
    apa_af3.w(1) = 1;
    %VF    = create_II_ord_Volterra_filter_1(MV, Kv, muV, delta);   % II ↩
        Order Volterra
    VF    = create_III_ord_Volterra_filter_1(MV, Kv, muV, delta);  % III ↩
        Order Volterra

    % Nonlinearities I.C. --------------------------------------------------
    af1.Q  = af01.Q;
    af2.Q  = af02.Q;
    af3.Q  = af03.Q;

   % Models Evaluation -----------------------------------------------------
    for k = 1 : Lx

        % Updating HSAF ----------------------------------------------------
        [apa_af1, y1(k), e1(k)] = AF_APA_HSPL_F(apa_af1, x(k), d(k) + dn(↩
            k) );          % HSAF
        %[apa_af1, y1(k), e1(k)] = AF_LMS_HSPL_F(apa_af1, x(k), d(k) + dn↩
            (k) );          % HSAF

        % Updating HPOLY ---------------------------------------------------
        [apa_af2, y2(k), e2(k)] = AF_APA_HPOLY_F(apa_af2, x(k), d(k) + dn↩
            (k) );         % HPOLY
```

```
        %[apa_af2, y2(k), e2(k)] = AF_LMS_HPOLY_F(apa_af2, x(k), d(k) + ↩
            dn(k) );          % HPOLY

        % Updating Mathews ----------------------------------------------
        [apa_af3, y3(k), e3(k)] = AF_LMS_MATHEWS_POLY_F(apa_af3, x(k), d(↩
            k) + dn(k) );  % MATHEWS

        % Updating VAF --------------------------------------------------
        %[VF, y4(k), e4(k)] =  AF_II_ord_VF_APA_F( VF, x(k), d(k) + dn(k)↩
             );              % II Order Volterra
        [VF, y4(k), e4(k)] =  AF_III_ord_VF_APA_F( VF, x(k), d(k) + dn(k)↩
             );             % III Order Volterra
    end

    em1  = em1  + (e1.^2);   % Squared error
    em2  = em2  + (e2.^2);   % Squared error
    em3  = em3  + (e3.^2);   % Squared error
    em4  = em4  + (e4.^2);   % Squared error

end

em1 = em1/nRun;       % MSE
em2 = em2/nRun;       % MSE
em3 = em3/nRun;       % MSE
em4 = em4/nRun;       % MSE

%--------------------------------------------------------------------------
% Average MSE evaluation
mse1 = mean(em1(end-B-M-1:end-M-1));  % Average MSE
mse2 = mean(em2(end-B-M-1:end-M-1));  % Average MSE
mse3 = mean(em3(end-B-M-1:end-M-1));  % Average MSE
mse4 = mean(em4(end-B-M-1:end-M-1));  % Average MSE
%--------------------------------------------------------------------------
fprintf('\n');


%% Results

%---------------------------------------------------------------------------
% Print table
%---------------------------------------------------------------------------

fprintf('\n');
fprintf('Mean Square Errors ↩
    ----------------------------------------------\n');
fprintf('      HSAF Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse1↩
    ,10*log10(mse1));
fprintf('Polynomial Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse2↩
    ,10*log10(mse2));
fprintf('   Mathews Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse3↩
    ,10*log10(mse3));
fprintf('       VAF Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse4↩
    ,10*log10(mse4));
fprintf('↩
    ----------------------------------------------------------------------\n')↩
    ;


%---------------------------------------------------------------------------
% Plotting figures
%---------------------------------------------------------------------------
```

```
% Plot Spline functions ------------------------------------------------------
yLIM = 1.5;
xLIM = 3.0;
figure1 = figure('PaperSize',[20.98 29.68]);
box('on');
hold on;
hold('all');
grid on;
ylim([-yLIM yLIM]);
xlim([-xLIM xLIM]);
KK = 500;
yy1 = zeros(1,KK);
yy2 = zeros(1,KK);
yy3 = zeros(1,KK);
yy4 = zeros(1,KK);
yy5 = zeros(1,KK);
xa1 = zeros(1,KK);
dx = 2*xLIM/KK;
xx = -xLIM;
for k = 1:KK
    yy1(k) = ActFunc(xx, apa_af1.af);  % Spline
    yy2(k) = ActFunc(xx, apa_af2.af);  % Polynomial
    yy3(k) = ActFunc(xx, apa_af3.af);  % Mathews
    [VF,yy4(k)] = FW_III_ord_VF_F(VF, xx); % Volterra
    %yy5(k) = real(xx.^0.33333333);     % Target 33
    %yy5(k) = nthroot(xx,3);            % Target 33
    yy5(k) = 1/8 + fix((8*xx - 4)/4);  % Target 34
    %yy5(k) = xx - 0.3*xx.^2 + 0.2*xx.^3; % Target 103
    xa1(k) = xx;
    xx = xx + dx;
end
title('Profile of nonlinearities after learning','FontSize', 12, '↩
    FontWeight', 'demi');
xlabel('Nonlinearity input {\itx}[{\itn}] ','FontSize', 12, 'FontWeight',↩
     'demi');
ylabel('Nonlinearity output {\its}[{\itn}]','FontSize', 12, 'FontWeight',↩
     'demi');
plot(xa1,yy1,'k','LineWidth',2);     % HSAF
plot(xa1,yy2,'g','LineWidth',2);     % Polynomial
plot(xa1,yy3,'r','LineWidth',2);     % Mathews
plot(xa1,yy4,'c','LineWidth',2);     % Volterra
plot(xa1,yy5,'-.b','LineWidth',2);   % Target
legend('HSAF','Polynomial','Mathews','Volterra','Location','SouthEast');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);

% MSE dB ----------------------------------------------------------------------
figure2 = figure('PaperSize',[20.98 29.68]);
box('on');
hold on;
grid on;
ylim([-out_noise_level_dB-5 0]);
[bb,aa] = butter(2, 0.005);
edb1 = 10*log10( em1 );
edb2 = 10*log10( em2 );
edb3 = 10*log10( em3 );
edb4 = 10*log10( em4 );
plot(filter(bb,aa,edb1),'Color',[0 0 0],'LineWidth',2);        % HSAF
plot(filter(bb,aa,edb2),'Color',[0 1 0],'LineWidth',2);        % Poly
plot(filter(bb,aa,edb3),'Color',[1 0 0],'LineWidth',2);        % Mathews
plot(filter(bb,aa,edb4),'Color',[0.2 0.9 0.8],'LineWidth',2); % Volterra
title('Comparisons of MSE','FontSize', 12, 'FontWeight', 'demi');
```

```
xlabel('Samples','FontSize', 12, 'FontWeight', 'demi');
ylabel('MSE [dB]    10log({\itJ}({\bfw},{\bfQ}))','FontSize', 12, '↩
    FontWeight', 'demi');
legend('HSAF','Polynomial','Mathews','3-rd Order Volterra');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
```

3.4 S1SAF_compare_demo

This script implemets the third experiment in [9]. It consists in the identification of a nonlinear Sandwich 1 dynamic system. The first and last blocks are the following two static nonlinear functions:

$$s[n] = \frac{x[n]}{\sqrt{0.1 + 0.9x^2[n]}},$$

and

$$y[n] = r[n] - 0.5r^2[n] + 0.2r^3[n],$$

while the second block is the following FIR filter

$$H(z) = 1 + 0.5z - 0.25z^{-1} + 0.05z^{-2} + 0.001z^{-3}.$$

The input signal $x[n]$ consists of 50,000 samples of the signal generated by the following relationship

$$x[n] = ax[n-1] + \sqrt{1-a^2}\xi[n],$$

where $\xi[n]$ is a zero mean white Gaussian noise with unitary variance and $a = 0.1$. An $\mathrm{SNR} = 30$ dB is also considered. The learning rates are set to $\mu_w = \mu_q^{(1)} = \mu_q^{(2)} = 0.05$. The filter weights are initialized using $\alpha = 0.1$ and the length is set to $M = 15$.

The proposed methods were compared with a standard full 3-rd order Volterra filter [5] (with $M_v = 15$), the LNL approach proposed in [2] (with $M_1 = M_2 = 5$), where the nonlinearity is implemented as a 3-rd order polynomial, the approach proposed by Jeraj and Mathews in [3] (with $M_a = 1$, $M_b = 5$ and $N = 3$), the WSAF [6] (with $M = 5$) and HSAF [7] (with $M = 5$) algorithms and the S2SAF architecture, with a suitable set of parameters.

The complete MATLAB code is reported below.

```
clear all;
close all;
disp('S1SAF_compare_demo');
% ------------------------------------------------------------------------

%% Parameters setting

% Input parameters ------------------------------------------------------
Lx = 50000;                 % Length of input signal
nRun = 10;                  % Number of runs
out_noise_level_dB = Inf;   % SNR
out_noise_level = 10^(-out_noise_level_dB/20);  % Noise level

x = randn(Lx,1);            % x (Nx1) input signal array definition
```

```
% Colored signal generation ------------------------------------------------
a = 0;
b = sqrt(1-a^2);
% x = filter( b, [1 -a], randn( size(x))) ; % H(z) = b/(1+a*z^-1)
disp('..... ');

% Adaptive filter definition ------------------------------------------------
M  = 15;            % Length of S1SAF
MV = 15;            % Length of Volterra filter
K =  1;             % Affine Projection order; K=1 => NLMS
M1 = 5;             % Length of the first linear part of S2SAF
M2 = 5;             % Length of the second linear part of S2SAF
Kv = 1;             % Affine Projection order; K=1 => NLMS
Pord = 3;           % Polynomial order
delta = 1e-2;       % APA regularization parameters
mu  = 0.005;        % Learning rate linear filter S1SAF
mQ1 = 0.005;        % Learning rate for first nonlinearity S1SAF
mQ2 = 0.005;        % Learning rate for second nonlinearity S1SAF
muV = 0.1;          % Learning rate for Volterra filter
muM = 0.00001;      % Learning rate for Mathews linear part
mQM = 0.00001;      % Learning rate for Mathews nonlinear part
muJ1 = 0.0001;      % Learning rate for LNL Jenkins first part
muJ2 = 0.0001;      % Learning rate for LNL Jenkins second part
muJ3 = 0.001;       % Learning rate for LNL Jenkins third part
mu1S = 0.005;       % Learning rate for the first linear part of S2SAF
mu2S = 0.009;       % Learning rate for the second linear part of S2SAF
mQS  = 0.04;        % Learning rate for the nonlinearity of S2SAF
if Lx < 30000,      % Batch for evaluating MSE
        B = 100;
else
        B = 4000;
end

% Spline activation function definition and initialization ------------------
% NL1
afinit = 0;     % Init act. func. -1 0 ... (ONLY -1, bip.sig. or 0 = ↩
    linear)
aftype = 4;     % Kind act. f. -1 0 1 2 3 4 5
Slope  = 1;     % Slope
DeltaX = 0.2;   % Delta X
x_range = 2;    % Range limit

% NL2
afinit2 = 0;    % Init act. func. -1 0 ... (ONLY -1, bip.sig. or 0 = ↩
    linear)
aftype2 = 4;    % Kind act. f. -1 0 1 2 3 4 5
Slope2  = 1;    % Slope
DeltaX2 = 0.2;  % Delta X
x_range2 = 2;   % Range limit

aftype3 = 3;    % Kind act. f. -1 0 1 2 3 4 5


%--- Nonlinearity definition ------------------------------------------------
af01 = create_activation_function(afinit, aftype, DeltaX, x_range, Slope,↩
     M);        % First S1SAF nonlinearity
af02 = create_activation_function(afinit2, aftype2, DeltaX2, x_range2, ↩
    Slope2, M);   % Second S1SAF nonlinearity
af03 = create_activation_function(afinit, aftype3, DeltaX, x_range, Slope↩
    , M, Pord); % Polynomial Mathews nonlinearity
af04 = create_activation_function(afinit, aftype, DeltaX, x_range, Slope,↩
     M1);       % S2SAF nonlinearity
```

```
% ---- Models definition ------------------------------------------------
H1 = create_Sandwich1SPL_lms_adaptive_filter_1(M,mu,mQ1,mQ2,delta,af01,...
    af02);      % S1SAF
H2 = create_SPL_lms_adaptive_filter_1(M,mu,mQ2,delta,af02); ...
                            % WSAF
H3 = create_SPL_lms_adaptive_filter_1(M,mu,mQ1,delta,af01); ...
                            % HSAF
H4 = create_SPL_lms_adaptive_filter_1(M,muM,mQM,delta,af03); ...
                           % Polynomial Mathews
H5 = create_LNLJenkins_lms_adaptive_filter_1(M,Pord,M,muJ1,muJ2,muJ3,...
    delta,af03);  % Hedge et al.
VF = create_III_ord_Volterra_filter_1(MV, Kv, muV, delta); ...
                             % III Order Volterra
A1 = create_Sandwich2SPL_lms_adaptive_filter_1(M1,M2,mu1S,mu2S,mQS,delta,...
    af04);    % S2SAF

% ---- SAF definition ---------------------------------------------------

% Initialize -------------------------------------------------------------
N = Lx + M + K;
for i = Lx+1:N
    x(i) = 0;
end

dn  = zeros(N,1);        % Noise output array
d   = zeros(N,1);        % Desired signal array
y1  = zeros(N,1);        % Output array
y2  = zeros(N,1);        % Output array
y3  = zeros(N,1);        % Output array
y4  = zeros(N,1);        % Output array
y5  = zeros(N,1);        % Output array
y6  = zeros(N,1);        % Output array
y7  = zeros(N,1);        % Output array
e1  = zeros(Lx,1);       % Error array
e2  = zeros(Lx,1);       % Error array
e3  = zeros(Lx,1);       % Error array
e4  = zeros(Lx,1);       % Error array
e5  = zeros(Lx,1);       % Error array
e6  = zeros(Lx,1);       % Error array
e7  = zeros(Lx,1);       % Error array
em1 = zeros(Lx,1);       % Mean square error
em2 = zeros(Lx,1);       % Mean square error
em3 = zeros(Lx,1);       % Mean square error
em4 = zeros(Lx,1);       % Mean square error
em5 = zeros(Lx,1);       % Mean square error
em6 = zeros(Lx,1);       % Mean square error
em7 = zeros(Lx,1);       % Mean square error


%% Main loop -------------------------------------------------------------
disp('S1SAF_compare_demo start ... ');
t = clock;

for n = 0 : nRun-1
    fprintf('Test nr. %d/%d\n', n+1, nRun);

    % Nonlinear system signal generation ---------------------------------
    x = randn(Lx,1);          % x (Nx1) input signal array definition
    if a>0,
        x = filter( b, [1 -a], x);              % Colored input
    end
```

```
x = x/max(abs(x));                                  % Normalization
y = x./(sqrt(0.1 + 0.9*x.^2));                      % First nonlinearity
z = filter([1 0.5 -0.25 0.05 0.001], 1, y);         % Linear filter
d = z - 0.5*z.^2 + 0.2*z.^3;                        % Second nonlinearity
dn = out_noise_level*randn( size(x) );              % Noisy desired signal

% SAF I.C. ------------------------------------------------------------
H1.w (:)  = 0;
H1.w  (1) = 1;  % Set filter i.c.
H2.w (:)  = 0;
H2.w  (1) = 1;  % Set filter i.c.
H3.w (:)  = 0;
H3.w  (1) = 1;  % Set filter i.c.
H4.w (:)  = 0;
H4.w  (1) = 1;  % Set filter i.c.
H5.w1(:)  = 0;
H5.w1 (1) = 1;  % Set filter i.c.
H5.w2(:)  = 0;
H5.w2 (1) = 1;  % Set filter i.c.
H5.w3(:)  = 0;
H5.w3 (1) = 1;  % Set filter i.c.

VF = create_III_ord_Volterra_filter_1(MV, Kv, muV, delta);  % III ↩
    Order Volterra
A1.w1 (:) = 0;
A1.w1 (1) = 1;  % Set filter i.c.
A1.w2 (:) = 0;
A1.w2 (1) = 1;  % Set filter i.c.

% Set Activation Func I.C. --------------------------------------------
H1.af1.Q = af01.Q;
H1.af2.Q = af02.Q;
H2.af.Q = af02.Q;
H3.af.Q = af01.Q;
H4.af.Q = af03.Q;
H5.af.Q = af03.Q;
A1.af.Q = af04.Q;

% Models evaluations --------------------------------------------------
for k = 1 : Lx
    % Updating S1SAF ------------------------------------------------
    [H1, y1(k), e1(k)] = AF_LMS_Sandwich1SPL_F(H1, x(k), d(k) + dn(k)↩
        );  % S1SAF LMS

    % Updating VAF --------------------------------------------------
    [VF, y2(k), e2(k)] =  AF_III_ord_VF_APA_F( VF, x(k), d(k) + dn(k)↩
        );  % III Order Volterra APA

    % Updating WSAF -------------------------------------------------
    [H2, y3(k), e3(k)] = AF_LMS_WSPL_F(H2, x(k), d(k) + dn(k) ); ↩
                % WSAF LMS

    % Updating HSAF -------------------------------------------------
    [H3, y4(k), e4(k)] = AF_LMS_HSPL_F(H3, x(k), d(k) + dn(k) ); ↩
                % HSAF LMS

    % Updating Polynomial Mathews -----------------------------------
    [H4, y5(k), e5(k)] = AF_LMS_MATHEWS_POLY_F(H4, x(k), d(k) + dn(k)↩
        );  % MATHEWS

    % Updating LNL Jenkins ------------------------------------------
    [H5, y6(k), e6(k)] = AF_LMS_LNLJenkins_F(H5, x(k), d(k) + dn(k) )↩
```

```
            ;    % LNL Jenkins

        % Updating S2SAF ------------------------------------------------------
        [A1, y7(k), e7(k)] = AF_LMS_Sandwich2SPL_F(A1, x(k), d(k) + dn(k)↩
             ); % S2SAF LMS
    end

    % Squared Error ------------------------------------------------------------
    em1  = em1 + (e1.^2);    % Squared Error
    em2  = em2 + (e2.^2);    % Squared Error
    em3  = em3 + (e3.^2);    % Squared Error
    em4  = em4 + (e4.^2);    % Squared Error
    em5  = em5 + (e5.^2);    % Squared Error
    em6  = em6 + (e6.^2);    % Squared Error
    em7  = em7 + (e7.^2);    % Squared Error

end

% MSE ----------------------------------------------------------------------
em1  = em1/nRun;   % MSE
em2  = em2/nRun;   % MSE
em3  = em3/nRun;   % MSE
em4  = em4/nRun;   % MSE
em5  = em5/nRun;   % MSE
em6  = em6/nRun;   % MSE
em7  = em7/nRun;   % MSE

%--------------------------------------------------------------------------
% Average MSE evaluations
mse1 = mean(em1(end-B-M-1:end-M-1));  % Average MSE
mse2 = mean(em2(end-B-M-1:end-M-1));  % Average MSE
mse3 = mean(em3(end-B-M-1:end-M-1));  % Average MSE
mse4 = mean(em4(end-B-M-1:end-M-1));  % Average MSE
mse5 = mean(em5(end-B-M-1:end-M-1));  % Average MSE
mse6 = mean(em6(end-B-M-1:end-M-1));  % Average MSE
mse7 = mean(em7(end-B-M-1:end-M-1));  % Average MSE
%--------------------------------------------------------------------------
fprintf('\n');

%% Results

% ------------------------------------------------------------------------
% Print table
% ------------------------------------------------------------------------

fprintf('\n');
fprintf('Mean Square Errors ↩
    ------------------------------------------\n');
fprintf('  S1SAF Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse1,10*↩
    log10(mse1));
fprintf('    VAF Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse2,10*↩
    log10(mse2));
fprintf('   WSAF Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse3,10*↩
    log10(mse3));
fprintf('   HSAF Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse4,10*↩
    log10(mse4));
fprintf('Mathews Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse5,10*↩
    log10(mse5));
fprintf('Jenkins Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse6,10*↩
    log10(mse6));
fprintf('  S2SAF Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse7,10*↩
    log10(mse7));
```

```
fprintf('↩
    ------------------------------------------------------------------------\n')↩
    ;


% ------------------------------------------------------------------------
% Plotting figures
% ------------------------------------------------------------------------

% Plot nonlinear functions ------------------------------------------------
yLIM = 1.5;
xLIM = 3.0;

% Without target
figure1_1 = figure('PaperSize',[10 15]);
box('on');
hold on;
hold('all');
ylim([-yLIM yLIM]);
xlim([-xLIM xLIM]);
grid on;
KK = 500;
yy1 = zeros(1,KK);
yy2 = zeros(1,KK);
yy3 = zeros(1,KK);
yy4 = zeros(1,KK);
yy5 = zeros(1,KK);
yy6 = zeros(1,KK);
yy7 = zeros(1,KK);
yy8 = zeros(1,KK);
xa1 = zeros(1,KK);
dx = 2*xLIM/KK;
xx = -xLIM;
for k = 1:KK
    yy1(k) = ActFunc(xx, H1.af1);          % First nonlinearity of S1SAF
    yy2(k) = ActFunc(xx, H1.af2);          % Second nonlinearity of S1SAF
    yy3(k) = ActFunc(xx, H2.af);           % WSAF
    yy4(k) = ActFunc(xx, H3.af);           % HSAF
    [VF,yy5(k)] = FW_III_ord_VF_F(VF, xx); % Volterra
    yy6(k) = ActFunc(xx, H4.af);           % Mathews
    xa1(k) = xx;
    xx = xx + dx;
    yy7(k) = xx./(sqrt(0.1 + 0.9*xx.^2));  % First target nonlinearity
    yy8(k) = xx - 0.5*xx.^3 + 0.02*xx.^5;  % Second target nonlinearity
end
title('Profile of nonlinearities after learning','FontSize', 12, '↩
    FontWeight', 'demi');
xlabel('Nonlinearity input {\its}[{\itn}] ','FontSize', 12, 'FontWeight',↩
     'demi');
ylabel('Nonlinearity output {\ity}[{\itn}]','FontSize', 12, 'FontWeight',↩
     'demi');
plot(xa1,yy1,'-k','LineWidth',2);  % adapted S1
plot(xa1,yy2,'-.r','LineWidth',2); % adapted S1
plot(xa1,yy3,'-g','LineWidth',2);  % adapted WSAF
plot(xa1,yy4,'-c','LineWidth',2);  % adapted HSAF
plot(xa1,yy5,'-m','LineWidth',2);  % adapted Volterra
plot(xa1,yy6,'-y','LineWidth',2);  % Mathews
legend('Adapted 1', 'Adapted 2', 'WSAF', 'HSAF', 'Volterra', 'Mathews');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);

% With target
```

```
figure1_2 = figure('PaperSize',[10 15]);
box('on');
hold on;
grin on;
ylim([-yLIM yLIM]);
xlim([-xLIM xLIM]);
plot(xa1,yy1,'-k','LineWidth',2);          % First nonlinearity of S1SAF
plot(xa1,yy2,'-b','LineWidth',2);          % Second nonlinearity of S1SAF
plot(xa1,yy7,'-.r','LineWidth',2);         % First target nonlinearity
plot(xa1,yy8,'-.g','LineWidth',2);         % Second target nonlinearity
legend('Adapted 1', 'Adapted 2', 'Target 1', 'Target 2');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);

% MSE dB ------------------------------------------------------------------
figure2 = figure('PaperSize',[10 15]);
box('on');
hold on;
grid on;
ylim([-out_noise_level_dB-5 0]);
[bb,aa] = butter(3, 0.01 );
edb1 = 10*log10( em1 );
edb2 = 10*log10( em2 );
edb3 = 10*log10( em3 );
edb4 = 10*log10( em4 );
edb5 = 10*log10( em5 );
edb6 = 10*log10( em6 );
edb7 = 10*log10( em7 );
plot(filter(bb,aa,edb1),'k-','LineWidth',2);       % S1SAF
plot(filter(bb,aa,edb2),'b-.','LineWidth',2);      % Volterra
plot(filter(bb,aa,edb3),'r--','LineWidth',2);      % WSAF
plot(filter(bb,aa,edb4),'m:','LineWidth',2);       % HSAF
plot(filter(bb,aa,edb5),'c-.','LineWidth',2);      % Mathews
plot(filter(bb,aa,edb6),'g--','LineWidth',2);      % Jenkins
plot(filter(bb,aa,edb7),'y-','LineWidth',2);       % S2SAF
if out_noise_level_dB ~= Inf
    noiseLevel(1: length(edb1)-1 ) = -out_noise_level_dB;  % Noise level
    plot( noiseLevel,'--','Color',[0 0 1],'LineWidth',2 );
end
title('Comparisons of Sandwich SAF architectures','FontSize', 12, '↩
    FontWeight', 'demi');
xlabel('Samples','FontSize', 12, 'FontWeight', 'demi');
ylabel('MSE [dB]   10log({\itJ}({\bfw},{\bfQ}))','FontSize', 12, '↩
    FontWeight', 'demi');
if out_noise_level_dB ~= Inf
    legend('S1SAF', 'Volterra', 'WSAF', 'HSAF', 'Jeraj & Mathews', 'Hegde↩
        et al.', 'S2SAF', 'NoiseLevel' );
else
    legend('S1SAF', 'Volterra', 'WSAF', 'HSAF', 'Jeraj & Mathews', 'Hegde↩
        et al.', 'S2SAF' );
end
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);

% Filter coefficients -----------------------------------------------------
figure3 = figure('PaperSize',[20.98 29.68]);
hold on;
grid on;
title('Adapted linear filters','FontSize', 12, 'FontWeight', 'demi');
plot(H1.w,'-b','LineWidth',2.5);       % S1SAF
plot(H2.w ,'-r','LineWidth',2.5);      % WSAF
plot(H3.w ,'-g','LineWidth',2.5);      % HSAF
```

```
plot(H4.w ,'-c','LineWidth',2.5);       % Mathews
xlabel('samples {\itn}','FontSize', 12, 'FontWeight', 'demi');
ylabel('Linear combiner coefficients','FontSize', 12, 'FontWeight', 'demi↩
    ');
legend('S1SAF', 'WSAF', 'HSAF', 'Mathews');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);
```

3.5 S1SAF_demo

The script implements the first experiment of [9]. This experiment consists in the identification of an unknown Sandwich 1 system composed by a linear component $\mathbf{w}_0 = [0.6, -0.4, 0.25, -0.15, 0.1, -0.05, 0.001]^T$ and two nonlinear memoryless target functions. These functions are implemented by two 23-point length LUTs $\mathbf{q}_0^{(1)}$ and $\mathbf{q}_0^{(2)}$, interpolated by a uniform third degree spline with an interval sampling $\Delta x^{(1)} = \Delta x^{(2)} = 0.2$ and defined as

$$\mathbf{q}_0^{(1)} = \{-2.2, -2.0, -1.8, \ldots, -1.0, -0.8, -0.91, -0.40, \\ -0.20, 0.05, 0.0, -0.40, 0.58, 1.0, 1.0, 1.2, 1.4, \ldots, 2.2\}.$$

and

$$\mathbf{q}_0^{(2)} = \{-2.2, -2.0, -1.8, \ldots, -1.0, -0.8, -0.60, -0.10, \\ -0.20, 0.0, 0.02, 0.40, 0.60, 0.8, 1.35, 1.2, 1.4, \ldots, 2.2\}.$$

The input signal $x[n]$ consists of 50,000 samples of the signal generated by the following relationship

$$x[n] = ax[n-1] + \sqrt{1-a^2}\xi[n],$$

where $\xi[n]$ is a zero mean white Gaussian noise with unitary variance and $0 \leq a < 1$ is a parameter that determines the level of correlation between adjacent samples. Experiments are conducted with a set to 0.1 and 0.95. In addition it is considered an additive white noise $v[n]$ such that the signal to noise ratio (SNR) is 30 dB.

The learning rates are set to $\mu_w = \mu_q^{(1)} = \mu_q^{(2)} = 0.05$. The linear filter is initialized as $\mathbf{w}_{-1} = \alpha\boldsymbol{\delta}$, with $\boldsymbol{\delta} \in \mathbb{R}^{M\times 1} = [1,\ 0,\ \ldots,\ 0]^T$ and using $\alpha = 0.1$. The filter length is set to $M = 7$.

The complete MATLAB code is reported below.

```
clear all;
close all;
disp('S1SAF_demo');
% ------------------------------------------------------------------------

%% Parameters setting

% Input parameters ------------------------------------------------------
Lx = 30000;                 % Length of input signal
nRun = 10;                  % Number of runs
out_noise_level_dB = 30;    % SNR
out_noise_level = 10^(-out_noise_level_dB/20);  % Noise level

x = zeros(Lx,1);            % x (Nx1) input signal array definition
```

```
% Colored signal generation ---------------------------------------------
a = 0.0;
b = sqrt(1-a^2);
% x = filter( b, [1 -a], randn( size(x))) ; % H(z) = b/(1+a*z^-1)
disp('..... ');

% Adaptive filter definition ---------------------------------------------
M =  7;             % Length of the linear filter
K =  1;             % Affine Projection order; K=1 => NLMS
mu0 = 0.05;         % Learning rate for linear filter
mQ1 = 0.05;         % Learning rate for ctrl points
mQ2 = 0.05;         % Learning rate for ctrl points
if Lx < 30000,      % Batch for evaluating MSE
        B = 100;
else
        B = 4000;
end

% Spline activation function definition and initialization ---------------
% NL 1
afinit1 = 0;      % Init act. func. -1 0 ... (ONLY -1, bip.sig. or 0 = ↩
    linear)
aftype1 = 4;      % Kind act. f. -1 0 1 2 4 5
Slope1  = 1;      % Slope
DeltaX1 = 0.2;    % Delta X
x_range1 = 2;     % Range limit

% NL 2
afinit2 = 0;      % Init act. func. -1 0 ... (ONLY -1, bip.sig. or 0 = ↩
    linear)
aftype2 = 4;      % Kind act. f. -1 0 1 2 4 5
Slope2 = 1;       % Slope
DeltaX2 = 0.2;    % Delta X
x_range2 = 2;     % Range limit


% Creating the nonlinearity ----------------------------------------------
af01 = create_activation_function(afinit1, aftype1, DeltaX1, x_range1, ↩
    Slope1, M); % NL 1
af02 = create_activation_function(afinit2, aftype2, DeltaX2, x_range2, ↩
    Slope2, M); % NL 2


%% Initialization

% --- Model definition ------------------------------------------------------
%H1 = create_Sandwich1SPL_apa_adaptive_filter_1(M,K,mu0,mQ1,mQ2,1e-2,af01↩
    ,af02); % Linear Filter Model APA
H1 = create_Sandwich1SPL_lms_adaptive_filter_1(M,mu0,mQ1,mQ2,1e-2,af01,↩
    af02); % Linear Filter Model LMS

% --- Target Definition -----------------------------------------------------
%TH1 = create_Sandwich1SPL_apa_adaptive_filter_1(M,K,mu0,mQ1,mQ2,1e-2,↩
    af01,af02); % Target h Model APA
TH1 = create_Sandwich1SPL_lms_adaptive_filter_1(M,mu0,mQ1,mQ2,1e-2,af01,↩
    af02); % Target h Model LMS

% TARGET: Nonlinear memoryless function implemented by HW-Spline ↩
    interpolated LUT
% NL 1
Q01 = [     -2.20
            -2.00
```

```
            -1.80
            -1.60
            -1.40
            -1.20
            -1.00
            -0.80
            -0.91
            -0.40
            -0.20
             0.05
             0.00
            -0.40
             0.58
             1.00
             1.00
             1.20
             1.40
             1.60
             1.80
             2.00
             2.20
];

% NL 2
Q02 = [     -2.20
            -2.00
            -1.80
            -1.60
            -1.40
            -1.20
            -1.00
            -0.80
            -0.60
            -0.10
            -0.20
             0.00
             0.02
             0.40
             0.60
             0.80
             1.35
             1.20
             1.40
             1.60
             1.80
             2.00
             2.20
];

TH1.af1.Q = Q01;
TH1.af2.Q = Q02;
QL1 = length(Q01);  % Number of control points
QL2 = length(Q02);  % Number of control points

% Linear filter ---------------------------------------------------------
TH1.w  = [0.6  -0.4  0.25  -0.15  0.1  -0.05  0.001]';  % MA system to be↩
     identified

%--- SAF definition ---------------------------------------------------------

% Initialize ------------------------------------------------------------
N = Lx + M + K;      % Total samples
```

```
for i = Lx+1:N
    x(i) = 0;
end

dn    = zeros(N,1);        % Noise output array
d     = zeros(N,1);        % Desired signal array
y     = zeros(N,1);        % Output array
e     = zeros(Lx,1);       % Error array
em    = zeros(Lx,1);       % Mean square error
wm    = zeros( M,1 );      % Mean value of w
varW  = zeros( M,1 );      % Variance value of w
qm1   = zeros( QL1, 1 );   % Mean value Spline 1 coeff
varQ1 = zeros( QL1, 1 );   % Variance value Spline 1 coeff
qm2   = zeros( QL2, 1 );   % Mean value Spline 2 coeff
varQ2 = zeros( QL2, 1 );   % Variance value Spline 2 coeff


%% Main loop ------------------------------------------------------------
disp('S1SAF algorithm start ... ');
t = clock;

for n = 0 : nRun-1
    fprintf('Test nr. %d/%d\n', n+1, nRun);
    x = filter( b, [1 -a], randn( size(x)) ) ; % H(z) = b/(1+a*z^-1)
    dn = out_noise_level * randn( size(x) ) ;

    % SAF I.C. ---------------------------------------------------------
    H1.w (:) = 0;
    H1.w (1) = 0.1;    % Set filter I.C.

    % Set Activation Func I.C. -----------------------------------------
    H1.af1.Q = af01.Q;
    H1.af2.Q = af02.Q;

    % S1SAF Evaluation -------------------------------------------------
    for k = 1 : Lx
        % Computing the desired output ---------------------------------
        [TH1, d(k), s, r] = FW_Sandwich1SPL_F(TH1, x(k));   % Sandwich 1 ↩
            model

        % Updating S1SAF -----------------------------------------------
        %[H1, y(k), e(k)] = AF_APA_Sandwich1SPL_F(H1, x1(k), d(k) + dn(k)↩
            ); % SandwichSAF APA
        [H1, y(k), e(k)] = AF_LMS_Sandwich1SPL_F(H1, x(k), d(k) + dn(k) )↩
            ;   % SandwichSAF LMS
    end

    em  = em  + (e.^2);     % Squared Error

    % SAF run-time mean and variance estimation ------------------------
    wm   = (1/(n+1))*H1.w + (n/(n+1))*wm;
    varW = varW + (n/(n+1))*((TH1.w - wm).^2);

    qm1   = (1/(n+1))*H1.af1.Q + (n/(n+1))*qm1;
    varQ1 = varQ1 + (n/(n+1))*((TH1.af1.Q - qm1).^2);

    qm2   = (1/(n+1))*H1.af2.Q + (n/(n+1))*qm2;
    varQ2 = varQ2 + (n/(n+1))*((TH1.af2.Q - qm2).^2);

end

em = em/nRun;  % MSE
```

```
H1.af1.Q = qm1;
H1.af2.Q = qm2;

%-------------------------------------------------------------------------
% Average MSE evaluation
mse = mean(em(end-B-M-1:end-M-1));  % Average MSE
%-------------------------------------------------------------------------
fprintf('\n');


%% Results

% -------------------------------------------------------------------------
% Print table
% -------------------------------------------------------------------------
fprintf('\n');
fprintf('Number of iterations = %d\n',nRun);
fprintf('Learning rates:  muW = %5.3f   muQ1 = %5.3f   muQ2\n', mu0, mQ1,...
    mQ2);
fprintf('a = %4.2f  b = %4.2f\n',a, b );
fprintf('Number of filter weights = %d\n', M);
fprintf('Number of control points = %d and %d\n', QL1, QL2);
fprintf('AF type 1 = %d   AF type 2 = %d\n',aftype1, aftype2);
fprintf('DeltaX1 = %4.2f    DeltaX2 = %4.2f\n',DeltaX1, DeltaX2);
fprintf('SNR_dB  = %4.2f dB\n',out_noise_level_dB);
fprintf('Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse,10*log10(mse)...
    );
fprintf('\n');
fprintf('Mean and Variance Tables ...
    ------------------------------------\n');
for i=1:QL1
    fprintf('i=%2d  q0 =%5.2f    qm =%9.6f    varQ = %10.3e \n', i,  Q01(i...
        ), qm1(i),  varQ1(i) );
end
fprintf('\n');
for i=1:QL2
    fprintf('i=%2d  q0 =%5.2f    qm =%9.6f    varQ = %10.3e \n', i,  Q02(i...
        ), qm2(i),  varQ2(i) );
end
fprintf('\n');
for i=1:M
    fprintf('i=%d  w0 =%5.2f    wm =%9.6f    varW = %10.3e \n', i, TH1.w(i...
        ), wm(i),  varW(i)  );
end
fprintf('...
    ------------------------------------------------------------------------\n')...
    ;


%-------------------------------------------------------------------------
% Plotting figures
%-------------------------------------------------------------------------

% Plot Spline functions -------------------------------------------------
yLIM = 1.5;
xLIM = 3.0;
figure1 = figure('PaperSize',[10 15]);
box('on');
hold on;
hold('all');
ylim([-yLIM yLIM]);
xlim([-xLIM xLIM]);
```

```
grid on;
KK = 500;
yy1 = zeros(1,KK);
yy2 = zeros(1,KK);
yy3 = zeros(1,KK);
yy4 = zeros(1,KK);
xa1 = zeros(1,KK);
dx = 2*xLIM/KK;
xx = -xLIM;
for k = 1:KK
    yy1(k) = ActFunc(xx, TH1.af1);
    yy2(k) = ActFunc(xx, TH1.af2);
    yy3(k) = ActFunc(xx, H1.af1);
    yy4(k) = ActFunc(xx, H1.af2);
    xa1(k) = xx;
    xx = xx + dx;
end
title('Profile of spline nonlinearity after learning','FontSize', 12, '↩
    FontWeight', 'demi');
xlabel('Linear combiner output {\its}[{\itn}] ','FontSize', 12, '↩
    FontWeight', 'demi');
ylabel('SAF output {\ity}[{\itn}]','FontSize', 12, 'FontWeight', 'demi');
plot(xa1,yy1,'-.k','LineWidth',2);  % reference 1
plot(xa1,yy2,'-.b','LineWidth',2);  % reference 2
plot(xa1,yy3,'r','LineWidth',2);    % adapted 1
plot(xa1,yy4,'m','LineWidth',2);    % adapted 2
legend('Target 1','Target 2','SPL 1','SPL 2');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);

% Filter coefficients ------------------------------------------------------
figure2 = figure('PaperSize',[20.98 29.68]);
hold on;
grid on;
plot(TH1.w,'-k','LineWidth',2.5);
plot(H1.w,'--r','LineWidth',2.5);
title('Comparison of model and adapted linear filter','FontSize', 12, '↩
    FontWeight', 'demi');
xlabel('samples {\itn}','FontSize', 12, 'FontWeight', 'demi');
ylabel('Linear combiner coefficients','FontSize', 12, 'FontWeight', 'demi↩
    ');
legend('Model','Adapted');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);

% MSE dB -------------------------------------------------------------------
figure3 = figure('PaperSize',[10 15]);
box('on');
hold on;
grid on;
ylim([-out_noise_level_dB-5 0]);
[bb,aa] = butter(3, 0.01 );
edb1 = 10*log10( em );
plot( filter(bb,aa,edb1 ),'Color',[1 0 0],'LineWidth',2);
noiseLevel(1: length(edb1)-1 ) = -out_noise_level_dB;
plot( noiseLevel,'--','Color',[0 0 1],'LineWidth',2 );
title('Sandwich SAF convergence test','FontSize', 12, 'FontWeight', 'demi↩
    ');
xlabel('Samples','FontSize', 12, 'FontWeight', 'demi');
ylabel('MSE [dB]   10log({\itJ}({\bfw},{\bfQ}))','FontSize', 12, '↩
    FontWeight', 'demi');
legend('MSE', 'NoiseLevel' );
```

```
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);
```

3.6 S2SAF_compare_demo

This script implements the fourth experiment in [9]. It consists in the identification of a nonlinear Sandwich 2 dynamic system. The first and last blocks are two 4-th order IIR filters, Butterworth and Chebychev respectively, with transfer functions

$$H_B(z) = \frac{\left(0.2851 + 0.5704z^{-1} + 0.2851z^{-2}\right)}{\left(1 - 0.1024z^{-1} + 0.4475z^{-2}\right)} \cdot \frac{\left(0.2851 + 0.5701z^{-1} + 0.2851z^{-2}\right)}{\left(1 - 0.0736z^{-1} + 0.0408z^{-2}\right)},$$

and

$$H_C(z) = \frac{\left(0.2025 + 0.2880z^{-1} + 0.2025z^{-2}\right)}{\left(1 - 1.01z^{-1} + 0.5861z^{-2}\right)} \cdot \frac{\left(0.2025 + 0.0034z^{-1} + 0.2025z^{-2}\right)}{\left(1 - 0.6591z^{-1} + 0.1498z^{-2}\right)},$$

while the second block is the following nonlinearity

$$y[n] = \frac{2x[n]}{1 + |x[n]|^2}.$$

This system is similar to radio frequency amplifiers used in satellite communications (High Power Amplifier), in which the linear filters model the dispersive transmission paths, while the nonlinearity models the amplifier saturation. The input signal $x[n]$ consists of 50,000 samples of the signal generated by the following relationship

$$x[n] = ax[n-1] + \sqrt{1-a^2}\xi[n],$$

where $\xi[n]$ is a zero mean white Gaussian noise with unitary variance and $a = 0.1$. The learning rates are set to $\mu_w^{(1)} = 0.005$, $\mu_w^{(2)} = 0.009$ and $\mu_q = 0.04$. The filter weights are initialized using $\alpha = 0.1$ and the lengths are set to $M_1 = M_2 = 5$.

The proposed methods were compared with a standard full 3-rd order Volterra filter [5] (with $M_v = 15$), the LNL approach proposed in [2] (with $M_1 = M_2 = 5$), where the nonlinearity is implemented as a 3-rd order polynomial, the approach proposed by Jeraj and Mathews in [3] (with $M_a = 1$, $M_b = 5$ and $N = 3$), the WSAF [6] (with $M = 5$) and HSAF [7] (with $M = 5$) algorithms and the S1SAF architecture, with a suitable set of parameters.

The complete MATLAB code is reported below.

```
clear all;
close all;
disp('S2SAF_compare_demo');
% ------------------------------------------------------------------------

%% Parameters setting

% Input parameters ------------------------------------------------------
Lx = 30000;                 % Length of the input signal
nRun = 10;                  % Number of runs
out_noise_level_dB = Inf;   % SNR
```

```
out_noise_level = 10^(-out_noise_level_dB/20);   % Noise level

x = randn(Lx,1);           % x (Nx1) input signal array definition

% Colored signal generation ------------------------------------------------
a = 0.1;
b = sqrt(1-a^2);
% x = filter( b, [1 -a], randn( size(x))); % H(z) = b/(1+a*z^-1)
disp('..... ');


% Adaptive filter definition -----------------------------------------------
M1 = 5;            % Length of the first linear filter
M2 = 5;            % Length of the second linear filter
MV = 5;            % Length of the Volterra filter
M3 = 5;            % Length of the S1SAF linear filter
K =  1;            % Affine Projection order; K=1 => NLMS
Kv = 1;            % Volterra Affine Projection order; K=1 => NLMS
Pord = 3;          % Polynomial order
delta = 1e-2;      % APA regularization parameters
mu1 = 0.005;       % Learning rate for the first linear filter
mu2 = 0.009;       % Learning rate for the second linear filter
mQ0 = 0.04;        % Learning rate for the spline ctrl points
muV = 0.1;         % Learning rate for the Volterra filter
muM = 0.00001;     % Learning rate for the linear part of Mathews ↩
    architecture
mQM = 0.00001;     % Learning rate for the nonlinear part of Mathews ↩
    architecture
muJ1 = 0.0001;     % Learning rate for the first part of LNL Jenkins ↩
    architecture
muJ2 = 0.0001;     % Learning rate for the second part of LNL Jenkins ↩
    architecture
muJ3 = 0.001;      % Learning rate for the third part of LNL Jenkins ↩
    architecture
muS  = 0.05;       % Learning rate for the linear part of S1SAF
mQ1S = 0.05;       % Learning rate for the first nonlinearity of S1SAF
mQ2S = 0.005;      % Learning rate for the second nonlinearity of S1SAF
if Lx < 30000,     % Batch for evaluating MSE
        B = 100;
else
        B = 4000;
end

% Spline activation function definition and initialization -----------------
afinit = 0;     % Init act. func. -1 0 ... (ONLY -1, bip.sig. or 0 = ↩
    linear)
aftype = 4;     % Kind act. f. -1 0 1 2 3 4 5
aftype2 = 3;    % Kind act. f. -1 0 1 2 3 4 5
Slope  = 1;     % Slope
DeltaX = 0.2;   % Delta X
x_range = 2;    % Range limit

% --- Nonlinearity definition ------------------------------------------------
af01 = create_activation_function(afinit, aftype, DeltaX, x_range, Slope,↩
     M2);         % S2SAF
af02 = create_activation_function(afinit, aftype2, DeltaX, x_range, Slope↩
    , M2, Pord);  % Polynomial Mathews
af03 = create_activation_function(afinit, aftype, DeltaX, x_range, Slope,↩
     M3);         % First S1SAF nonlinearity
af04 = create_activation_function(afinit, aftype, DeltaX, x_range, Slope,↩
     M3);         % Second S1SAF nonlinearity
```

```
%--- Models definition ---------------------------------------------------
H1 = create_Sandwich2SPL_lms_adaptive_filter_1(M1,M2,mu1,mu2,mQ0,delta,...
    af01);          % S2SAF
H2 = create_SPL_lms_adaptive_filter_1(M1,mu1,mQ0,delta,af01); ...
                                % WSAF
H3 = create_SPL_lms_adaptive_filter_1(M2,mu2,mQ0,delta,af01); ...
                                % HSAF
H4 = create_SPL_lms_adaptive_filter_1(M2,muM,mQM,delta,af02); ...
                                % Polynomial Mathews
H5 = create_LNLJenkins_lms_adaptive_filter_1(M1,Pord,M2,muJ1,muJ2,muJ3,...
    delta,af02);    % Hedge et al.
VF = create_III_ord_Volterra_filter_1(MV, Kv, muV, delta); ...
                                  % III Order Volterra
A1 = create_Sandwich1SPL_lms_adaptive_filter_1(M3,muS,mQ1S,mQ2S,delta,...
    af03,af04);      % S1SAF

%--- SAF definition ------------------------------------------------------

% Initialize ---------------------------------------------------------------
N = Lx + M1 + K;
for i = Lx+1:N
    x(i)=0;
end

dn  = zeros(N,1);        % Noise output array
d   = zeros(N,1);        % Desired signal array
y1  = zeros(N,1);        % Output array
y2  = zeros(N,1);        % Output array
y3  = zeros(N,1);        % Output array
y4  = zeros(N,1);        % Output array
y5  = zeros(N,1);        % Output array
y6  = zeros(N,1);        % Output array
y7  = zeros(N,1);        % Output array
e1  = zeros(Lx,1);       % Error array
e2  = zeros(Lx,1);       % Error array
e3  = zeros(Lx,1);       % Error array
e4  = zeros(Lx,1);       % Error array
e5  = zeros(Lx,1);       % Error array
e6  = zeros(Lx,1);       % Error array
e7  = zeros(Lx,1);       % Error array
em1 = zeros(Lx,1);       % Mean square error
em2 = zeros(Lx,1);       % Mean square error
em3 = zeros(Lx,1);       % Mean square error
em4 = zeros(Lx,1);       % Mean square error
em5 = zeros(Lx,1);       % Mean square error
em6 = zeros(Lx,1);       % Mean square error
em7 = zeros(Lx,1);       % Mean square error


%% Main loop ---------------------------------------------------------------
disp('S2SAF_compare_demo start ... ');
t = clock;

for n = 0 : nRun-1
    fprintf('Test nr. %d/%d\n', n+1, nRun);

    % Nonlinear system signal generation ---------------------------------
    % 1998_Sicuranza-Mathews HPA Model -----------------------------------
    x = randn(Lx,1);           % x (Nx1) input signal array definition
    if a>0,
        x = filter(b, [1 -a], x) ;    % H(z) = a/1-b*z^-1
    end
```

```
y = filter(conv([0.2851 0.5704 0.2851], [0.2851 0.5701 0.2851]), ...
           conv([1      -0.1024 0.4475], [1      -0.0736 0.0408]), x); ↩
               % First linear filter

z = y./(1+abs(y.^2));       % Nonlinearity

d = filter(conv([0.2025 0.288 0.2025],[0.2025 0.0034 0.2025]), ...
           conv([1      -1.01  0.5861],[1      -0.6591 0.1498]), z);  %↩
               Second linear filter

dn = out_noise_level*randn( size(x) );    % Noisy desired signal

% SAF I.C. ------------------------------------------------------------
H1.w1 (:) = 0;
H1.w1 (1) = 1;  % Set filter i.c.
H1.w2 (:) = 0;
H1.w2 (1) = 1;  % Set filter i.c.
H2.w (:)  = 0;
H2.w  (1) = 1;  % Set filter i.c.
H3.w (:)  = 0;
H3.w  (1) = 1;  % Set filter i.c.
H4.w (:)  = 0;
H4.w  (1) = 1;  % Set filter i.c.
H5.w1(:)  = 0;
H5.w1 (1) = 1;  % Set filter i.c.
H5.w2(:)  = 0;
H5.w2 (1) = 1;  % Set filter i.c.
H5.w3(:)  = 0;
H5.w3 (1) = 1;  % Set filter i.c.

VF = create_III_ord_Volterra_filter_1(MV, Kv, muV, delta);  % III ↩
    Order Volterra
A1.w (:) = 0;
A1.w (1) = 1;

% Set Activation Func I.C. --------------------------------------------
H1.af.Q = af01.Q;
H2.af.Q = af01.Q;
H3.af.Q = af01.Q;
H4.af.Q = af02.Q;
H5.af.Q = af02.Q;
A1.af1.Q = af03.Q;
A1.af2.Q = af04.Q;

% Set Activation Func I.C. --------------------------------------------
for k = 1 : Lx
    % Updating S2SAF ------------------------------------------------
    [H1, y1(k), e1(k)] = AF_LMS_Sandwich2SPL_F(H1, x(k), d(k) + dn(k)↩
         );   % S2SAF LMS

    % Updating VAF --------------------------------------------------
    [VF, y2(k), e2(k)] = AF_III_ord_VF_APA_F( VF, x(k), d(k) + dn(k) ↩
        );    % III Order Volterra

    % Updating WSAF -------------------------------------------------
    [H2, y3(k), e3(k)] = AF_LMS_WSPL_F(H2, x(k), d(k) + dn(k) ); ↩
                % WSAF LMS

    % Updating HSAF -------------------------------------------------
    [H3, y4(k), e4(k)] = AF_LMS_HSPL_F(H3, x(k), d(k) + dn(k) ); ↩
                % HSAF LMS
```

```
        % Updating Polynomial Mathews ------------------------------------
        [H4, y5(k), e5(k)] = AF_LMS_MATHEWS_POLY_F(H4, x(k), d(k) + dn(k)↩
             );   % Polyomial Mathews

        % Updating LNL Jenkins --------------------------------------------
        [H5, y6(k), e6(k)] = AF_LMS_LNLJenkins_F(H5, x(k), d(k) + dn(k) )↩
            ;     % LNL Jenkins

        % Updating S1SAF --------------------------------------------------
        [A1, y7(k), e7(k)] = AF_LMS_Sandwich1SPL_F(A1, x(k), d(k) + dn(k)↩
             );   % S1SAF LMS
    end

    % Squared Error -----------------------------------------------------
    em1  = em1 + (e1.^2);
    em2  = em2 + (e2.^2);
    em3  = em3 + (e3.^2);
    em4  = em4 + (e4.^2);
    em5  = em5 + (e5.^2);
    em6  = em6 + (e6.^2);
    em7  = em7 + (e7.^2);

end

% MSE ---------------------------------------------------------------------
em1  = em1/nRun;
em2  = em2/nRun;
em3  = em3/nRun;
em4  = em4/nRun;
em5  = em5/nRun;
em6  = em6/nRun;
em7  = em7/nRun;

%-------------------------------------------------------------------------
% Average MSE evaluations
mse1 = mean(em1(end-B-M1-1:end-M1-1));  % Average MSE
mse2 = mean(em2(end-B-M1-1:end-M1-1));  % Average MSE
mse3 = mean(em3(end-B-M1-1:end-M1-1));  % Average MSE
mse4 = mean(em4(end-B-M1-1:end-M1-1));  % Average MSE
mse5 = mean(em5(end-B-M1-1:end-M1-1));  % Average MSE
mse6 = mean(em6(end-B-M1-1:end-M1-1));  % Average MSE
mse7 = mean(em7(end-B-M1-1:end-M1-1));  % Average MSE
%-------------------------------------------------------------------------
fprintf('\n');


%% Results

% ------------------------------------------------------------------------
% Print table
% ------------------------------------------------------------------------

fprintf('\n');
fprintf('Mean Square Errors ↩
    ------------------------------------------\n');
fprintf('  S2SAF Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse1,10*↩
    log10(mse1));
fprintf('    VAF Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse2,10*↩
    log10(mse2));
fprintf('   WSAF Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse3,10*↩
    log10(mse3));
```

```
fprintf('   HSAF Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse4,10*↩
    log10(mse4));
fprintf('Mathews Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse5,10*↩
    log10(mse5));
fprintf('Jenkins Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse6,10*↩
    log10(mse6));
fprintf('  S1SAF Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse7,10*↩
    log10(mse7));
fprintf('↩
    ----------------------------------------------------------------------\n')↩
    ;


%-------------------------------------------------------------------------
% Plotting figures
%-------------------------------------------------------------------------

yLIM = 1.5;
xLIM = 3.0;

% Without target
figure1_1 = figure('PaperSize',[10 15]);
box('on');
hold on;
hold('all');
ylim([-yLIM yLIM]);
xlim([-xLIM xLIM]);
grid on;
KK = 500;
yy1 = zeros(1,KK);
yy2 = zeros(1,KK);
yy3 = zeros(1,KK);
yy4 = zeros(1,KK);
yy5 = zeros(1,KK);
yy6 = zeros(1,KK);
yy7 = zeros(1,KK);
yy8 = zeros(1,KK);
xa1 = zeros(1,KK);
dx = 2*xLIM/KK;
xx = -xLIM;
for k = 1:KK
    yy1(k) = ActFunc(xx, H1.af);           % S2SAF
    yy2(k) = xx - 0.5*xx.^3 + 0.02*xx.^5;  % Target
    yy3(k) = ActFunc(xx, H2.af);           % WSAF
    yy4(k) = ActFunc(xx, H3.af);           % HSAF
    [VF,yy5(k)] = FW_III_ord_VF_F(VF, xx); % Volterra
    yy6(k) = ActFunc(xx, H4.af);           % Mathews
    yy7(k) = 2*xx./(1 + abs(xx.^2));       % Target
    xa1(k) = xx;
    xx = xx + dx;
end
title('Profile of spline nonlinearity after learning','FontSize', 12, '↩
    FontWeight', 'demi');
xlabel('Linear combiner output {\its}[{\itn}] ','FontSize', 12, '↩
    FontWeight', 'demi');
ylabel('SAF output {\ity}[{\itn}]','FontSize', 12, 'FontWeight', 'demi');
plot(xa1,yy1,'-k','LineWidth',2);  % S2SAF
plot(xa1,yy2,'-.r','LineWidth',2); % Target
plot(xa1,yy3,'-g','LineWidth',2);  % WSAF
plot(xa1,yy4,'-c','LineWidth',2);  % HSAF
plot(xa1,yy5,'-m','LineWidth',2);  % Volterra
plot(xa1,yy6,'-y','LineWidth',2);  % Mathews
```

```
legend('S2SAF', 'Target', 'WSAF', 'HSAF', 'Volterra', 'Mathews');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);

% With target
figure1_2 = figure('PaperSize',[10 15]);
box('on');
hold on;
grid on;
ylim([-yLIM yLIM]);
xlim([-xLIM xLIM]);
plot(xa1,yy1,'-k','LineWidth',2);  % S2SAF
plot(xa1,yy7,'-.r','LineWidth',2); % Target
legend('S2SAF', 'Target');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);

% MSE dB ----------------------------------------------------------------
figure2 = figure('PaperSize',[10 15]);
box('on');
hold on;
grid on;
ylim([-out_noise_level_dB-5 0]);
[bb,aa] = butter(3, 0.01 );
edb1 = 10*log10( em1 );
edb2 = 10*log10( em2 );
edb3 = 10*log10( em3 );
edb4 = 10*log10( em4 );
edb5 = 10*log10( em5 );
edb6 = 10*log10( em6 );
edb7 = 10*log10( em7 );
plot(filter(bb,aa,edb1),'k-','LineWidth',2);   % S2SAF
plot(filter(bb,aa,edb2),'b-.','LineWidth',2);  % Volterra
plot(filter(bb,aa,edb3),'r--','LineWidth',2);  % WSAF
plot(filter(bb,aa,edb4),'m:','LineWidth',2);   % HSAF
plot(filter(bb,aa,edb5),'c-.','LineWidth',2);  % Mathews
plot(filter(bb,aa,edb6),'g--','LineWidth',2);  % Jenkins
plot(filter(bb,aa,edb7),'y-','LineWidth',2);   % S1SAF
if out_noise_level_dB ~= Inf
    noiseLevel(1: length(edb1)-1 ) = -out_noise_level_dB;  % Noise level
    plot( noiseLevel,'--','Color',[0 0 1],'LineWidth',2 );
end
title('Comparisons of Sandwich SAF architectures','FontSize', 12, '↩
    FontWeight', 'demi');
xlabel('Samples','FontSize', 12, 'FontWeight', 'demi');
ylabel('MSE [dB]   10log({\itJ}({\bfw},{\bfQ}))','FontSize', 12, '↩
    FontWeight', 'demi');
if out_noise_level_dB ~= Inf
    legend('S2SAF', 'Volterra', 'WSAF', 'HSAF', 'Jeraj & Mathews', 'Hegde↩
        et al.', 'S1SAF', 'NoiseLevel' );
else
    legend('S2SAF', 'Volterra', 'WSAF', 'HSAF', 'Jeraj & Mathews', 'Hegde↩
        et al.', 'S1SAF' );
end
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);


% Filter coefficients ---------------------------------------------------
figure3 = figure('PaperSize',[20.98 29.68]);
hold on;
grid on;
```

```
title('Adapted linear filters','FontSize', 12, 'FontWeight', 'demi');
plot(H1.w1,'-b','LineWidth',2.5);      % First linear filter of S2SAF
plot(H1.w2,'-k','LineWidth',2.5);      % Second linear filter of S2SAF
plot(H2.w ,'-r','LineWidth',2.5);      % WSAF
plot(H3.w ,'-g','LineWidth',2.5);      % HSAF
plot(H4.w ,'-c','LineWidth',2.5);      % Mathews
xlabel('samples {\itn}','FontSize', 12, 'FontWeight', 'demi');
ylabel('Linear combiner coefficients','FontSize', 12, 'FontWeight', 'demi↩
    ');
legend('S2SAF w1', 'S2SAF w2', 'WSAF', 'HSAF', 'Mathews');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);
```

3.7 S2SAF_demo

The script implements the second experiment of [9]. This experiment consists in the identification of an unknown Sandwich 2 system composed by two linear components $\mathbf{w}_0^{(1)} = [0.6, -0.4, 0.25, -0.15, 0.1, -0.05, 0.001]^T$ and $\mathbf{w}_0^{(2)} = [1, 0.5, -0.25, 0.15, 0.25, -0.10, 0.05]^T$, and a nonlinear memoryless target function implemented by a 23-point length LUT $\mathbf{q}_0$ interpolated by a uniform third degree spline with an interval sampling $\Delta x = 0.2$. $\mathbf{q}_0$ is defined as

$$\mathbf{q}_0 = \{-2.2, -2.0, -1.8, \ldots, -1.0, -0.8, -0.91, -0.40, \\ -0.20, 0.05, 0.0, -0.40, 0.58, 1.0, 1.0, 1.2, 1.4, \ldots, 2.2\}.$$

The input signal $x[n]$ consists of 50,000 samples of the signal generated by the following relationship

$$x[n] = ax[n-1] + \sqrt{1-a^2}\xi[n],$$

where $\xi[n]$ is a zero mean white Gaussian noise with unitary variance and $0 \leq a < 1$ is a parameter that determines the level of correlation between adjacent samples. Also in this case experiments are conducted with $a = 0.1$ and $a = 0.95$. Again it is considered an additive white noise $v[n]$ such that SNR $= 30$ dB. The learning rates are set to $\mu_w^{(1)} = 0.005$, $\mu_w^{(2)} = 0.009$ and $\mu_q = 0.04$. The filter weights are initialized using $\alpha = 0.1$. The filter lengths are set to $M_1 = M_2 = 7$.

The complete MATLAB code is reported below.

```
clear all;
close all;
disp('S2SAF_demo');
% ------------------------------------------------------------------------

%%  Parameters setting

% Input parameters ------------------------------------------------------
Lx = 30000;                 % Length of the input signal
nRun = 10;                  % Number of runs
out_noise_level_dB = 30;  % SNR
out_noise_level = 10^(-out_noise_level_dB/20);  % Noise level

x = zeros(Lx,1);   % x (Nx1) input signal array definition

% Colored signal generation ---------------------------------------------
```

```
a = 0.0;
b = sqrt(1-a^2);
% x = filter( b, [1 -a], randn( size(x))) ; % H(z) = b/(1+a*z^-1)
disp('..... ');


% Adaptive filter definition ---------------------------------------------
M1 = 7;             % Length of the first linear filter
M2 = 7;             % Length of the second linear filter
mu1 = 0.005;        % Learning rate of the first linear filter
mu2 = 0.009;        % Learning rate of the second linear filter
mQ0 = 0.004;        % Learning rate for ctrl points
if Lx < 30000,      % Batch for evaluating MSE
        B = 100;
else
        B = 4000;
end


% Spline activation function definition and initialization ---------------
afinit = 0;     % Init act. func. -1 0 ... (ONLY -1, bip.sig. or 0 = ↩
    linear)
aftype = 4;     % Kind act. f. -1 0 1 2 4 5
Slope  = 1;     % Slope
DeltaX = 0.2;   % Delta X
x_range = 2;    % Limit range


% Creating the nonlinearity ----------------------------------------------
af01 = create_activation_function(afinit, aftype, DeltaX, x_range, Slope,↩
     M2); % i.c.

%% Initialization

% --- Model definition ------------------------------------------------------
H1 = create_Sandwich2SPL_lms_adaptive_filter_1(M1,M2,mu1,mu2,mQ0,1e-2,↩
    af01); % Linear Filter Model LMS

% --- Target Definition -----------------------------------------------------
TH1 = create_Sandwich2SPL_lms_adaptive_filter_1(M1,M2,mu1,mu2,mQ0,1e-2,↩
    af01); % Target h Model LMS

% TARGET: Nonlinear memoryless function implemented by HW-Spline ↩
    interpolated LUT
Q0  = [     -2.20
            -2.00
            -1.80
            -1.60
            -1.40
            -1.20
            -1.00
            -0.80
            -0.91
            -0.40
            -0.20
             0.05
             0.00
            -0.40
             0.58
             1.00
             1.00
             1.20
```

```
            1.40
            1.60
            1.80
            2.00
            2.20
];

TH1.af.Q = Q0;
QL = length(Q0);  % Number of control points

% Linear filters ---------------------------------------------------------
TH1.w1  = [0.6   -0.4   0.25   -0.15   0.1   -0.05   0.001]';  % MA system to ↩
    be identified
%TH1.w1  = [1     0     0       0      0     0      0     ]';  % Ideal impulse
TH1.w2  = [1     0.5  -0.25    0.15   0.25  -0.10   0.050]';  % MA system to ↩
    be identified
%TH1.w2  = [1     0     0       0      0     0      0     ]';  % Ideal impulse

% ---- SAF definition ---------------------------------------------------

% Initialize -------------------------------------------------------------
N = Lx + M1 + 1;    % Total number of samples
for i = Lx+1:N
    x(i)=0;
end

d     = zeros(N,1);      % Desired signal array
dn    = zeros(N,1);      % Noise output array
y     = zeros(N,1);      % Output array
e     = zeros(Lx,1);     % Error array
em    = zeros(Lx,1);     % Mean square error array
wm1   = zeros(M1,1);     % Mean value of w
varW1 = zeros(M1,1);     % Variance value of w
wm2   = zeros(M2,1);     % Mean value of w
varW2 = zeros(M2,1);     % Variance value of w
qm    = zeros(QL,1);     % Mean value Spline coeff
varQ  = zeros(QL,1);     % Variance value Spline coeff


%% Main loop -------------------------------------------------------------
disp('S2SAF algorithm start ... ');
t = clock;

for n = 0 : nRun-1
    fprintf('Test nr. %d/%d\n', n+1, nRun);
    x = filter( b, [1 -a], randn( size(x))) ; % H(z) = b/(1+a*z^-1)
    dn = out_noise_level * randn( size(x) ) ;

    % SAF I.C. ---------------------------------------------------------
    H1.w1 (:) = 0;
    H1.w1 (1) = 1;   % Set filter i.c.
    H1.w2 (:) = 0;
    H1.w2 (1) = 1;   % Set filter i.c.

    % Set Activation Func I.C. -----------------------------------------
    H1.af.Q = af01.Q;

    % S2SAF Evaluation -------------------------------------------------
    for k = 1 : Lx
        % Computing the desired output ---------------------------------
        [TH1, d(k)] = FW_Sandwich2SPL_F(TH1, x(k));      % Sandwich 2 ↩
            model
```

```
        % Updating S2SAF ------------------------------------------------------
        [H1, y(k), e(k)] = AF_LMS_Sandwich2SPL_F(H1, x(k), d(k) + dn(k));↩
               % Sandwich2SAF LMS
    end

    em  = em  + (e.^2);  % Squared error


    % SAF run-time mean and variance estimation ----------------------------
    wm1   = (1/(n+1))*H1.w1 + (n/(n+1))*wm1;
    varW1 = varW1 + (n/(n+1))*((TH1.w1 - wm1).^2);

    wm2   = (1/(n+1))*H1.w2 + (n/(n+1))*wm2;
    varW2 = varW2 + (n/(n+1))*((TH1.w2 - wm2).^2);

    qm   = (1/(n+1))*H1.af.Q + (n/(n+1))*qm;
    varQ = varQ + (n/(n+1))*((TH1.af.Q - qm).^2);

end

em  = em/nRun;  % MSE
H1.af.Q = qm;

%-------------------------------------------------------------------------
% Average MSE evaluation
mse = mean(em(end-B-M1-1:end-M1-1));  % Average MSE
%-------------------------------------------------------------------------
fprintf('\n');

%% Results

%-------------------------------------------------------------------------
% Print table
%-------------------------------------------------------------------------
fprintf('\n');
fprintf('Number of iterations = %d\n',nRun);
fprintf('Learning rates:  muW1 = %5.3f   muw2 = %5.3f   muQ\n', mu1, mu2,↩
     mQ0);
fprintf('a = %4.2f  b = %4.2f\n',a, b );
fprintf('Number of filter weights M1 = %d   M2 = %d\n', M1, M2);
fprintf('Number of control points = %d\n', QL);
fprintf('AF type = %d\n',aftype);
fprintf('DeltaX = %4.2f\n',DeltaX);
fprintf('SNR_dB  = %4.2f dB\n',out_noise_level_dB);
fprintf('Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse,10*log10(mse)↩
    );
fprintf('\n');
fprintf('Mean and Variance Tables ↩
    ------------------------------------------\n');
for i=1:QL
    fprintf('i=%2d  q0 =%5.2f    qm =%9.6f    varQ = %10.3e \n', i,  Q0(i)↩
        , qm(i),  varQ(i) );
end
fprintf('\n');
for i=1:M1
    fprintf('i=%d  w01 =%5.2f    wm1 =%9.6f    varW1 = %10.3e \n', i, TH1.↩
        w1(i), wm1(i),  varW1(i)  );
end
fprintf('\n');
for i=1:M2
    fprintf('i=%d  w02 =%5.2f    wm2 =%9.6f    varW2 = %10.3e \n', i, TH1.↩
```

```
        w2(i), wm2(i),  varW2(i)  );
end
fprintf('END Sandwich 2 SAF test ↩
    ----------\n');
%------------------------------------------------------------------------

%------------------------------------------------------------------------
% Plotting figures
%------------------------------------------------------------------------
yLIM = 1.5;
xLIM = 3.0;
figure1 = figure('PaperSize',[10 15]);
box('on');
hold on;
hold('all');
ylim([-yLIM yLIM]);
xlim([-xLIM xLIM]);
grid on;
KK = 500;
yy1 = zeros(1,KK);
yy2 = zeros(1,KK);
xa1 = zeros(1,KK);
dx = 2*xLIM/KK;
xx = -xLIM;
for k = 1:KK
    yy1(k) = ActFunc(xx, TH1.af);
    yy2(k) = ActFunc(xx, H1.af);
    xa1(k) = xx;
    xx = xx + dx;
end
title('Profile of spline nonlinearity after learning','FontSize', 12, '↩
    FontWeight', 'demi');
xlabel('Linear combiner output {\its}[{\itn}] ','FontSize', 12, '↩
    FontWeight', 'demi');
ylabel('SAF output {\ity}[{\itn}]','FontSize', 12, 'FontWeight', 'demi');
plot(xa1,yy1,'-.k','LineWidth',2);  % reference
plot(xa1,yy2,'r','LineWidth',2);    % adapted
legend('Target','Adapted');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);

% Filter coefficients ---------------------------------------------------
figure2 = figure('PaperSize',[20.98 29.68]);
title('Comparison of model and adapted linear filter','FontSize', 12, '↩
    FontWeight', 'demi');
hold on;
grid on;
plot(TH1.w1,'-k','LineWidth',2.5);
plot(TH1.w2,'-r','LineWidth',2.5);
plot(H1.w1,'--b','LineWidth',2.5);
plot(H1.w2,'--m','LineWidth',2.5);
xlabel('samples {\itn}','FontSize', 12, 'FontWeight', 'demi');
ylabel('Linear combiner coefficients','FontSize', 12, 'FontWeight', 'demi↩
    ');
legend('Model 1','Model 2','Adapted 1','Adapted 2');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);

% MSE dB ----------------------------------------------------------------
figure3 = figure('PaperSize',[10 15]);
box('on');
hold on;
```

```
grid on;
ylim([-out_noise_level_dB-5 0]);
[bb,aa] = butter(3, 0.01 );
edb1 = 10*log10( em );
plot( filter(bb,aa,edb1 ),'Color',[1 0 0],'LineWidth',2);
noiseLevel(1: length(edb1)-1 ) = -out_noise_level_dB;
plot( noiseLevel,'--','Color',[0 0 1],'LineWidth',2 );
title('Sandwich SAF convergence test','FontSize', 12, 'FontWeight', 'demi↩
    ');
xlabel('Samples','FontSize', 12, 'FontWeight', 'demi');
ylabel('MSE [dB]   10log({\itJ}({\bfw},{\bfQ}))','FontSize', 12, '↩
    FontWeight', 'demi');
legend('MSE', 'NoiseLevel' );
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);
```

3.8 WPOLY_demo

This script implemets an adaptive Wiener polynomial filter [6, 5] (see Figure 3.2). The experiment consists in the identification of an unknown Wiener system composed by a linear component

$$\mathbf{w}_0 = [1, -0.4, 0.25, -0.15, 0.1, -0.05, 0.001]^T,$$

and a nonlinear memoryless target function is the following 3-rd order polynomial

$$y[n] = s^3[n] - 0.3s^2[n] + 0.2s[n].$$

The input signal $x[n]$ consists in 50,000 samples of the signal generated by the following relationship

$$x[n] = ax[n-1] + \sqrt{1-a^2}\xi[n],$$

where $\xi[n]$ is a zero mean white Gaussian noise with unitary variance and $0 \leq a < 1$ is a parameter that determines the level of correlation between adjacent samples. Experiments were conducted with $a = 0.1$. In addition it is considered an additive white noise $v[n]$ with a signal to noise ratio $SNR = 30$ dB. The learning rates are set to $\mu_w = \mu_q = 0.0005$.

The complete MATLAB code is reported below.

```
clear all;
close all;
disp('WPOLY_demo');
% ------------------------------------------------------------------------

%% Parameters setting

% Input parameters ------------------------------------------------------
Lx = 50000;                     % Length of the input signal
nRun = 10;                      % Number of runs
out_noise_level_dB = 30;        % SNR
out_noise_level = 10^(-out_noise_level_dB/20);   % Noise level

x = zeros(Lx,1);                % x (Nx1) input signal array definition
```

```
% Colored signal generation ------------------------------------------
a = 0.0;
b = sqrt(1-a^2);
% x = filter(b, [1 -a], randn( size(x)));   % H(z) = b/(1+a*z^-1)
disp('..... ');

% Adaptive filter definition -----------------------------------------
M = 7;              % Length of the linear filter
K = 1;              % Affine Projection order; K=1 => NLMS
Pord = 3;           % Polynomial order
mu0 = 0.0005;       % Learning rate LMS/APA;
mQ0 = 0.0005;       % Learning rate for ctrl points;
if Lx < 30000,      % Batch for evaluating MSE
        B = 100;
else
        B = 4000;
end

% Spline activation function definition and initialization -----------
afinit = 0;         % Init act. func. -1 0 ... (ONLY -1, bip.sig. or  0 =↩
    linear )
aftype = 3;         % Kind act. f. -1 0 1 2 3 4 5
Slope  = 1;         % Slope
DeltaX = 0.2;       % Delta X
x_range = 2;        % Range limit


%% Initialization

% Creating the nonlinearity ------------------------------------------
af0 = create_activation_function(afinit, aftype, DeltaX, x_range, Slope, ↩
    M, Pord);   % Target
af1 = create_activation_function(afinit, aftype, DeltaX, x_range, Slope, ↩
    M, Pord);   % Model

% --- Model definition ------------------------------------------------
%H1 = create_SPL_apa_adaptive_filter_1(M,K,mu0,mQ0,1e-2,af1);    % Model ↩
    APA
H1 = create_SPL_lms_adaptive_filter_1(M,mu0,mQ0,1e-2,af1);      % Model ↩
    LMS

% --- Target Definition -----------------------------------------------
%TH1 = create_SPL_apa_adaptive_filter_1(M,K,mu0,mQ0,1e-2,af0);    % Target↩
     Model APA
TH1 = create_SPL_lms_adaptive_filter_1(M,mu0,mQ0,1e-2,af0);      % Target ↩
    Model LMS

% --- Target Polynomial -----------------------------------------------
Q0 = [ 1   -0.3   0.2 ].';   %3-th order
%Q0 = [ 1.4   0.7   1.2   0.9   1.2 ].';   %5-th order
%Q0 = [ 0.6984   0.1045   -0.0264   -0.0168   -0.0011   0.0008 ].';  %6-th ↩
    order
%Q0 = [ 0.0008   -0.0011   -0.0168   -0.0264    0.1045    0.6984   -0.1968 ].';↩
     %7-th order
%Q0 = [ -0.1968   0.6984   0.1045   -0.0264   -0.0168   -0.0011   0.0008 ].';  ↩
    %7-th order
%Q0 = [ 1.2861   0.4493   -1.5836   -1.2253    1.4548   1.1157   -0.6093  ↩
    -0.4541   0.1188   0.0871   -0.0089   -0.0065].';  % 12-th order
%Q0 = [1.7998   1.0069   -6.6350   -3.7890   15.9373   5.4953   -19.0208  ↩
    -3.9872 12.1806   1.5391   -4.2507   -0.3003   0.7606   0.0232  ↩
    -0.0546].'; % 15-th order
TH1.af.Q = Q0;
```

```
QL = length(Q0);

%--- Target linear part ------------------------------------------------------
TH1.w  = [1  -0.4  0.25  -0.15  0.1  -0.05  0.001]';  % MA system to be ↩
    identified

% Initialize ---------------------------------------------------------------
N = Lx + M + K;
for i = Lx+1:N
    x(i)=0;
end

dn = zeros(N,1);        % Noise output array
d  = zeros(N,1);        % Desired signal array
y  = zeros(N,1);        % Output array
e  = zeros(N,1);        % Error array
em = zeros(N,1);        % Mean square error
wm   = zeros(M,1);      % Mean value of w
varW = zeros(M,1);      % Variance value of w
qm   = zeros( QL, 1 );  % Mean value Spline coeff
varQ = zeros( QL, 1 );  % Variance value Spline coeff


%% Main loop ---------------------------------------------------------------
disp('WPOLY algorithm start ... ');
t = clock;

for n = 0 : nRun-1
    fprintf('Test nr. %d/%d\n', n+1, nRun);
    x = filter( b, [1 -a], randn( size(x)));  % H(z) = b/(1+a*z^-1)
    dn = out_noise_level*randn( size(x) );    % Noisy desired signal

    % WPOLY I.C. ----------------------------------------------------------
    H1.w (:) = 0;
    H1.w (1) = 1;    % Set filter i.c.

    % Nonlinearity I.C. ---------------------------------------------------
    H1.af = create_activation_function(afinit, aftype,  DeltaX, x_range, ↩
        Slope, M, Pord);

    % WPOLY Evaluation ----------------------------------------------------
    for k = 1 : Lx
        % Computing the desired output ------------------------------------
        [TH1, d(k), snk] = FW_WPOLY_F(TH1, x(k));       % Polynomial ↩
            Wiener model

        % Updating WPOLY --------------------------------------------------
        %[H1, y(k), e(k)] = AF_APA_WPOLY_F(H1, x(k), d(k) + dn(k) );  % ↩
            WPOLY APA
        [H1, y(k), e(k)] = AF_LMS_WPOLY_F(H1, x(k), d(k) + dn(k) );   % ↩
            WPOLY LMS
    end

    em  = em  + (e.^2);   % Squared error

    % SAF run-time mean and variance estimation ---------------------------
    wm   = (1/(n+1))*H1.w + (n/(n+1))*wm;
    varW = varW + (n/(n+1))*((H1.w - wm).^2);

    qm   = (1/(n+1))*H1.af.Q + (n/(n+1))*qm;
    varQ = varQ + (n/(n+1))*((H1.af.Q - qm).^2);
```

```
end

em = em/nRun;       % MSE
H1.af.Q = qm;

%-------------------------------------------------------------------------
% Average MSE evaluation
mse = mean(em(end-B-M-1:end-M-1));  % Average MSE
%-------------------------------------------------------------------------
fprintf('\n');


%% Results

% -----------------------------------------------------------------------
% Print table of means and variances
% -----------------------------------------------------------------------
fprintf('\n');
fprintf('Number of iterations = %d\n',nRun);
fprintf('Learning rates:  muW = %5.3f   muQ = %5.3f\n', mu0, mQ0);
fprintf('a = %4.2f  b = %4.2f\n',a, b );
fprintf('Number of filter weights = %d\n', M);
fprintf('Number of control points = %d\n', QL);
fprintf('AF type = %d\n',aftype);
fprintf('DeltaX  = %4.2f\n',DeltaX);
fprintf('SNR_dB  = %4.2f dB\n',out_noise_level_dB);
fprintf('Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse,10*log10(mse)↩
    );
fprintf('\n');
fprintf('Mean and Variance Tables ↩
    ----------------------------------------\n');
for i=1:QL
    fprintf('i=%2d  q0 =%5.2f    qm =%9.6f    varQ = %10.3e \n', i,  H1.af↩
        .Q(i), qm(i), varQ(i) );
end
fprintf('\n');
fprintf('↩
    ------------------------------------------------------------------------\n')↩
    ;
for i=1:M
    fprintf('i=%d  w0 =%5.2f    wm =%9.6f    varW = %10.3e \n', i, H1.w(i)↩
        , wm(i), varW(i) );
end
fprintf('↩
    ------------------------------------------------------------------------\n')↩
    ;


% -----------------------------------------------------------------------
% Plotting figures
% -----------------------------------------------------------------------

% Plot Spline functions -------------------------------------------------
yLIM = 1.5;
xLIM = 3.0;
figure1 = figure('PaperSize',[20.98 29.68]);
box('on');
hold on;
hold('all');
ylim([-yLIM yLIM]);
xlim([-xLIM xLIM]);
grid on;
```

```
KK = 500;
yy1 = zeros(1,KK);
yy2 = zeros(1,KK);
xa1 = zeros(1,KK);
dx = 2*xLIM/KK;
xx = -xLIM;
for k = 1:KK
    yy1(k) = ActFunc(xx, TH1.af);   % Target
    yy2(k) = ActFunc(xx, H1.af);    % WPOLY
    xa1(k) = xx;
    xx = xx + dx;
end
title('Profile of polynomial nonlinearity after learning');
xlabel('Nonlinearity input {\itx}[{\itn}] ');
ylabel('Nonlinearity output {\its}[{\itn}]');
plot(xa1,yy1,'-.k','LineWidth',2);  % Target
plot(xa1,yy2,'r','LineWidth',2);    % WPOLY
legend('Target','WPLOY');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);

% MSE dB ------------------------------------------------------------------
figure2 = figure('PaperSize',[10 15]);
box('on');
hold on;
grid on;
ylim([-out_noise_level_dB-5 0]);
[bb,aa] = butter(3, 0.02 );
edb = 10*log10( em );
plot( filter(bb,aa,edb ),'Color',[1 0 0],'LineWidth',2);   % MSE
noiseLevel(1: length(edb)-1 ) = -out_noise_level_dB;
plot( noiseLevel,'--','Color',[0 0 1],'LineWidth',2 );     % Noise level
title('Wiener Polynomial convergence test');
xlabel('Samples');
ylabel('MSE [dB]   10log({\itJ}({\bfw},{\bfQ}))');
legend('MSE', 'NoiseLevel' );
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);

% Filter coefficients -----------------------------------------------------
figure3 = figure('PaperSize',[20.98 29.68]);
hold on;
grid on;
title('Comparison of model and adapted linear filter');
plot(TH1.w,'--r','LineWidth',2.5);   % Target
plot(H1.w,'-k','LineWidth',2.5);     % WPLOY
xlabel('samples {\itn}');
ylabel('Linear combiner coefficients');
legend('Target','WPOLY');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);
```

3.9 WSAF_demo

This script implements the first experiment in [6]. The idea is shown in Figure 3.2 The experiment consists in the identification of an unknown Wiener system composed by a linear component

$$\mathbf{w}_0 = [0.6, -0.4, 0.25, -0.15, 0.1]^T,$$

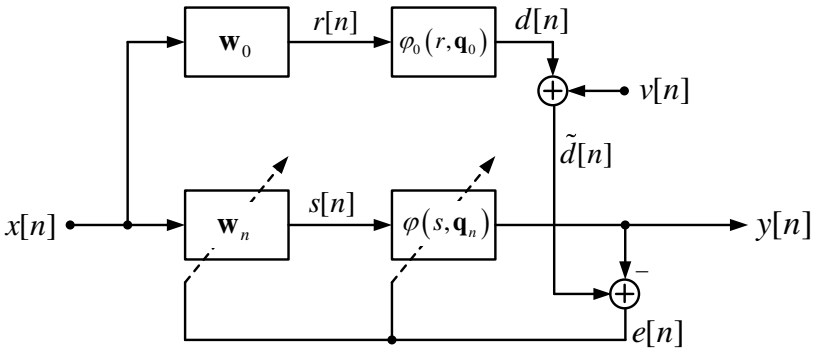

Fig. 3.2: Example of model used for the identification of an unknown Wiener system.

and a nonlinear memoryless target function implemented by a 21 points length LUT $\mathbf{q}_0$, interpolated by a uniform third degree spline with an interval sampling $\Delta x = 0.2$ defined as

$$\mathbf{q}_0 = \{-2, -1.8, \ldots, -1.0, -0.8, -0.91, 0.42, -0.01, \\ -0.1, 0.1, -0.15, 0.58, 1.2, 1.0, 1.2, \ldots, 2.0\}.$$

The input signal $x[n]$ consists in 50,000 samples of the signal generated by the following relationship

$$x[n] = ax[n-1] + \sqrt{1-a^2}\xi[n],$$

where $\xi[n]$ is a zero mean white Gaussian noise with unitary variance and $0 \leq a < 1$ is a parameter that determines the level of correlation between adjacent samples. Experiments were conducted with a set into the interval $[0,\ 0.99]$. In addition it is considered an additive white noise $v[n]$ with a signal to noise ratio $SNR = 30$ dB. The learning rates are set to $\mu_w = \mu_q = 0.008$, while $\delta_w = 0.01$.

The complete MATLAB code is reported below.

```
clear all;
close all;
disp('WSAF_demo');
%-------------------------------------------------------------------------

%% Parameters setting

% Input parameters ------------------------------------------------------
Lx = 50000;                 % Length of input signal
nRun = 10;                  % Number of runs
out_noise_level_dB = 30;    % SNR
out_noise_level = 10^(-out_noise_level_dB/20);  % Noise level

x = zeros(Lx,1);            % x (Nx1) input signal array definition

% Colored signal generation ---------------------------------------------
a = 0.1;
b = sqrt(1-a^2);
%x = filter( b, [1 -a], randn( size(x))) ;  % H(z) = b/(1+a*z^-1)
disp('...... ');
```

```
% Adaptive filter definition ---------------------------------------------
M =  5;            % Filter length
K =  1;            % fir APA (K=1 => NLMS)
mu0  = 0.008;      % Learning rate for linear filter
mQ0  = 0.008;      % Learning rate for control points
if Lx < 30000,     % Batch for evaluating MSE
        B = 100;
else
        B = 4000;
end

% Spline activation function definition and initialization ----------------
afinit = 0;    % Init act. func. -1 0 ... (ONLY -1, bip.sig. or  0 =linear )
aftype = 4;    % Kind act. f. -1 0 1 2 4 5; (4 = CR-spline, 5 = B-spline )
Slope  = 1;    % Slope
DeltaX = 0.2;  % Delta X
x_range = 2;   % Range limit

% Creating the nonlinearity -----------------------------------------------
af0 = create_activation_function(afinit, aftype, DeltaX, x_range, Slope, M);  % Model
af1 = create_activation_function(afinit, aftype, DeltaX, x_range, Slope, M);  % SAF

% Printing the control points ---------------------------------------------
fprintf('af.lut_len = %d \n', af0.lut_len );
for j=1 : af0.lut_len
    fprintf('%16.2f \n', af0.Q(j) );
end


%% Initialization

% --- Model definition ------------------------------------------------------
%apa_af0 =  create_SPL_apa_adaptive_filter_1(M,K,mu0,mQ0,1e-2,af0); % Model APA
apa_af0 =  create_SPL_lms_adaptive_filter_1(M,mu0,mQ0,1e-2,af0); % Model LMS

% TARGET: Nonlinear memoryless function implemented by Spline interpolated LUT
apa_af0.af.Q = [  % With E{\varphi'(u)/\varphi'(0) > 0}
            -2.20
            -2.00
            -1.80
            -1.60
            -1.40
            -1.20
            -1.00
            -0.80
            -0.91
            -0.40
             0.20
            -0.05
             0.00
            -0.15
             0.58
             1.00
             1.00
```

```
            1.20
            1.40
            1.60
            1.80
            2.00
            2.20
];
QL = length(apa_af0.af.Q);  % Number of control points

% Linear filter ---------------------------------------------------------
apa_af0.w = [0.6   -0.4   0.25   -0.15   0.1]';  % MA system to be identified

% --- SAF definition ----------------------------------------------------
%apa_af1 = create_SPL_apa_adaptive_filter_1(M ,K, mu0, mQ0, 1e-2, af1);  ↩
    % Model APA
apa_af1 = create_SPL_lms_adaptive_filter_1(M, mu0, mQ0, 1e-2, af1); % ↩
    Model LMS

% Initialize ------------------------------------------------------------
N = Lx + M + K;     % Total samples
for i = Lx+1:N
    x(i) = 0;
end

dn = zeros(N,1);         % Noise output array
d  = zeros(N,1);         % Desired signal array
y  = zeros(N,1);         % Output array
e  = zeros(N,1);         % Error array
em = zeros(N,1);         % Mean square error
wm   = zeros(M,1);       % Mean value of w
varW = zeros(M,1);       % Variance value of w
qm   = zeros( QL, 1 );   % Mean value Spline coeff
varQ = zeros( QL, 1 );   % Variance value Spline coeff


%% Main loop ------------------------------------------------------------
disp('WSAF algorithm start ... ');
t = clock;

for n = 0 : nRun-1
    fprintf('Test nr. %d/%d\n', n+1, nRun);
    x = filter( b, [1 -a], randn( size(x)) ) ;   % H(z) = b/(1+a*z^-1)
    dn = out_noise_level * randn( size(x) ) ;   % Noise

    % SAF I.C. ----------------------------------------------------------
    apa_af1.w = [1 0 0 0 0]';  % Set filter i.c.

    % Set Nonlinear Func I.C. -------------------------------------------
    apa_af1.af = create_activation_function(afinit, aftype, DeltaX, ↩
        x_range, Slope, M);

    % WSAF Evaluation ---------------------------------------------------
    for k = 1 : Lx
        % Computing the desired output ----------------------------------
        [apa_af0,d(k)] = FW_WSPL_F(apa_af0, x(k));   % Model

        % Updating WSAF -------------------------------------------------
        %[apa_af1,y(k),e(k)] = AF_APA_WSPL_F(apa_af1, x(k), d(k) + dn(k))↩
            ;  % SAF APA ( Eqs. (5) and (6) )
        [apa_af1,y(k),e(k)] = AF_LMS_WSPL_F(apa_af1, x(k), d(k) + dn(k));↩
              % SAF LMS ( Eqs. (5) and (6) )
    end
```

```
    em = em + (e.^2);  % Squared Error

    % SAF run-time mean and variance estimation ------------------------
    wm   = (1/(n+1))*apa_af1.w + (n/(n+1))*wm;
    varW = varW + (n/(n+1))*((apa_af1.w - wm).^2);

    qm   = (1/(n+1))*apa_af1.af.Q + (n/(n+1))*qm;
    varQ = varQ + (n/(n+1))*((apa_af1.af.Q - qm).^2);

end
em  = em/nRun;  % MSE
apa_af1.af.Q = qm;

%-------------------------------------------------------------------------
% Average MSE evaluation
mse = mean(em(end-B-M-1:end-M-1));  % Average MSE
%-------------------------------------------------------------------------
fprintf('\n');


%% Results

% -----------------------------------------------------------------------
% Print table of means and variances
% -----------------------------------------------------------------------
fprintf('\n');
fprintf('Number of iterations = %d\n',nRun);
fprintf('Learning rates:  muW = %5.3f   muQ = %5.3f\n', mu0, mQ0);
fprintf('a = %4.2f  b = %4.2f\n',a, b );
fprintf('Number of filter weights = %d\n', M);
fprintf('Number of control points = %d\n', QL);
fprintf('AF type = %d\n',aftype);
fprintf('DeltaX  = %4.2f\n',DeltaX);
fprintf('SNR_dB  = %4.2f dB\n',out_noise_level_dB);
fprintf('Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse,10*log10(mse)↩
    );
fprintf('\n');
fprintf('Mean and Variance Tables ↩
    ----------------------------------------\n');
for i=1:QL
    fprintf('i=%2d  q0 =%5.2f    qm =%9.6f    varQ = %10.3e \n', i,  ↩
        apa_af1.af.Q(i), qm(i), varQ(i) );
end
fprintf('\n');
fprintf('↩
    ------------------------------------------------------------------\n')↩
    ;
for i=1:M
    fprintf('i=%d  w0 =%5.2f    wm =%9.6f    varW = %10.3e \n', i, apa_af1↩
        .w(i), wm(i), varW(i) );
end
fprintf('↩
    ------------------------------------------------------------------\n')↩
    ;

% -----------------------------------------------------------------------
% Plotting figures
% -----------------------------------------------------------------------

% Plot Spline functions ---------------------------------------------------
yLIM = 1.5;
```

```
xLIM = 3.0;
figure1 = figure('PaperSize',[20.98 29.68]);
box('on');
hold on;
hold('all');
ylim([-yLIM yLIM]);
xlim([-xLIM xLIM]);
grid on;
KK = 500;
yy1 = zeros(1,KK);
yy2 = zeros(1,KK);
xa1 = zeros(1,KK);
dx = 2*xLIM/KK;
xx = -xLIM;
for k = 1:KK
    yy1(k) = ActFunc(xx, apa_af0.af);   % Model
    yy2(k) = ActFunc(xx, apa_af1.af);   % Adapted
    xa1(k)=xx;
    xx = xx + dx;
end
xlabel('Linear combiner output {\its}[{\itn}] ','FontSize', 12, '↩
    FontWeight', 'demi');
ylabel('SAF output {\ity}[{\itn}]','FontSize', 12, 'FontWeight', 'demi');
title('Profile of model and adapted nonlinearity \varphi(s[n])','FontSize↩
    ', 12, 'FontWeight', 'demi');
plot(xa1,yy1,'-.r','LineWidth',2);
plot(xa1,yy2,'k','LineWidth',2);
legend('Target','Adapted','Location','SouthEast');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');

% Filter coefficients ------------------------------------------------------
figure2 = figure('PaperSize',[20.98 29.68]);
hold on;
plot(apa_af0.w,'LineWidth',2);
plot(apa_af1.w,'m','LineWidth',2);
xlabel('time {\itn}','FontSize', 12, 'FontWeight', 'demi');
ylabel('Linear combiner coefficients','FontSize', 12, 'FontWeight', 'demi↩
    ');
legend('Model','Adapted');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');

% MSE dB -------------------------------------------------------------------
figure3 = figure('PaperSize',[10 15]);
box('on');hold on;hold('all');
ylim([-out_noise_level_dB-5 10]);
grid on;
edb = 10*log10( em );
[bb,aa] = butter(2, 0.02 );
plot( filter(bb,aa,edb ),'Color',[1 0 0],'LineWidth',2);
noiseLevel(1: length(edb)-1 ) = -out_noise_level_dB;
plot( noiseLevel,'--','Color',[0 0 1],'LineWidth',2 );
title('Wiener SAF convergence test','FontSize', 12, 'FontWeight', 'demi')↩
    ;
xlabel('Samples','FontSize', 12, 'FontWeight', 'demi');
ylabel('MSE [dB]   10log({\itJ}({\bfw},{\bfQ}))','FontSize', 12, '↩
    FontWeight', 'demi');
legend('MSE', 'NoiseLevel' );
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);
```

3.10 WSAF_DX_compare_demo

This script implements the fourth experiment in [6] and compare a 3-rd order Volterra filter [5] with the WSAF approach using different Δx values. In this test we want to prove the robustness of the proposed approach with respect to the Δx parameter, chosen in the set

$$\Delta x = \{0.15, 0.3, 0.45, 0.6\}\,.$$

The system to identify is the Back and Tsoi NARMA model and consists in a cascade of the following 3-rd order IIR filter

$$H(z) = \frac{0.0154 + 0.0462z^{-1} + 0.0462z^{-2} + 0.0154z^{-3}}{1 - 1.99z^{-1} + 1.572z^{-2} - 0.4583z^{-3}},$$

and the following nonlinearity

$$y[n] = \sin(s[n]).$$

The input signal $x[n]$ consists of 50,000 samples of the signal generated by the following relationship

$$x[n] = ax[n-1] + \sqrt{1-a^2}\xi[n],$$

where $\xi[n]$ is a zero mean white Gaussian noise with unitary variance and $a = 0.95$. The learning rates are set to $\mu_w = \mu_q = 0.02$, $\delta_w = 0.01$ and the filter length is $M = 5$ taps, for both the SAF and the 3-rd order Volterra filter.

The complete MATLAB code is reported below.

```
clear all;
close all;
disp('WSAF_DX_compare_demo');
%-------------------------------------------------------------------------

%% Parameters setting

% Input parameters ------------------------------------------------------
Lx = 50000;                     % Length of the input signal
nRun = 10;                      % Number of runs
out_noise_level_dB = 60;        % SNR
out_noise_level = 10^(-out_noise_level_dB/20);   % Noise level

x = randn(Lx,1);                % x (Nx1) input signal array definition

% Colored signal generation ---------------------------------------------
a = 0.0;
b = sqrt(1-a^2);
%x = filter( b, [1 -a], randn( size(x))) ;  % H(z) = b/(1+a*z^-1)
disp('..... ');

% Adaptive filter definition --------------------------------------------
M  =  10;               % Length of the linear filter
K  =   4;               % Affine Projection order; K=1 => NLMS
mu1 = 0.02;              % Learning rate of the linear filter
mQ1 = 0.02;              % Learning rate for ctrl points
delta = 0.1;            % APA regularization parameters
if Lx < 30000,          % Batch for evaluating MSE
```

```
        B = 100;
else
        B = 4000;
end

% Spline activation function definition and initialization ----------------
afinit  = 0;      % Init act. func. -1 0 ... (ONLY -1, bip.sig. or  0 =↩
    linear )
aftype  = 5;      % Kind act. f. -1 0 1 2 4 5; (4 = CR-spline, 5 = B-↩
    spline )
Slope   = 1;      % Slope
x_range = 2;      % Range limit
% Definition of several values of Delta X parameter ------------------------
DeltaX1 = 0.15;
DeltaX2 = 0.3;
DeltaX3 = 0.45;
DeltaX4 = 0.6;

% Creating the nonlinearity ------------------------------------------------
af1 = create_activation_function(afinit, aftype, DeltaX1, x_range, Slope,↩
     M);   % Delta X 1
af2 = create_activation_function(afinit, aftype, DeltaX2, x_range, Slope,↩
     M);   % Delta X 2
af3 = create_activation_function(afinit, aftype, DeltaX3, x_range, Slope,↩
     M);   % Delta X 3
af4 = create_activation_function(afinit, aftype, DeltaX4, x_range, Slope,↩
     M);   % Delta X 4


%% Initialization

% --- SAF definition -----------------------------------------------------
F1 = create_SPL_apa_adaptive_filter_1(M,K,mu1,mQ1,delta,af1);   % Delta X↩
     1
F2 = create_SPL_apa_adaptive_filter_1(M,K,mu1,mQ1,delta,af2);   % Delta X↩
     2
F3 = create_SPL_apa_adaptive_filter_1(M,K,mu1,mQ1,delta,af3);   % Delta X↩
     3
F4 = create_SPL_apa_adaptive_filter_1(M,K,mu1,mQ1,delta,af4);   % Delta X↩
     4


% Initialize ---------------------------------------------------------------
N = Lx + M + 1;   % Total samples
for i = Lx+1:N
    x(i)=0;
end

dn = zeros(N,1);        % Noise output array
d  = zeros(N,1);        % Desired signal array
y1  = zeros(N,1);       % Output array
y2  = zeros(N,1);       % Output array
y3  = zeros(N,1);       % Output array
y4  = zeros(N,1);       % Output array
e1  = zeros(Lx,1);      % Error array
e2  = zeros(Lx,1);      % Error array
e3  = zeros(Lx,1);      % Error array
e4  = zeros(Lx,1);      % Error array
em1 = zeros(Lx,1);      % Mean square error
em2 = zeros(Lx,1);      % Mean square error
em3 = zeros(Lx,1);      % Mean square error
em4 = zeros(Lx,1);      % Mean square error
```

```
%% Main loop ------------------------------------------------------------------
disp('Algorithm start ... ');
t = clock;

for n = 0 : nRun-1
    fprintf('Test nr. %d/%d\n', n+1, nRun);

    % Nonlinear system signal generation -----------------------------------
    x = randn(Lx,1);          % x (Nx1) input signal array definition
    if a>0,
        x = filter( b, [1 -a], x);              % Colored input
    end
    z = filter([0.0154 0.0462 0.0465 0.0154], [1 -1.99 1.572 -0.4583], x)↩
        ;
    d = sin(z);
    %z = 0.1*(x.^3 - 3*x.^2 + x);                % Nonlinearity
    %d = filter([1 0.75 0.5], [1 -0.75 0.5], z); % Desired signal
    dn = out_noise_level*randn( size(x) );      % Noisy desired signal

    % SAF I.C. -------------------------------------------------------------
    F1.w(:) = 0;
    F1.w(1) = 1;  % Set filter i.c.
    F2.w(:) = 0;
    F2.w(1) = 1;  % Set filter i.c.
    F3.w(:) = 0;
    F3.w(1) = 1;  % Set filter i.c.
    F4.w(:) = 0;
    F4.w(1) = 1;  % Set filter i.c.

    % Set Activation Func I.C. ---------------------------------------------
    F1.af.Q = af1.Q;
    F2.af.Q = af2.Q;
    F3.af.Q = af3.Q;
    F4.af.Q = af4.Q;

    % WSAF Evaluation ------------------------------------------------------
    for k = 1 : Lx
        % Delta X 1 --------------------------------------------------------
        [F1, y1(k), e1(k)] = AF_APA_WSPL_F(F1, x(k), d(k) + dn(k) );

        % Delta X 2 --------------------------------------------------------
        [F2, y2(k), e2(k)] = AF_APA_WSPL_F(F2, x(k), d(k) + dn(k) );

        % Delta X 3 --------------------------------------------------------
        [F3, y3(k), e3(k)] = AF_APA_WSPL_F(F3, x(k), d(k) + dn(k) );

        % Delta X 4 --------------------------------------------------------
        [F4, y4(k), e4(k)] = AF_APA_WSPL_F(F4, x(k), d(k) + dn(k) );
    end

    % RUN time averaging ---------------------------------------------------
    em1 = em1 + e1.^2;    % Squared error
    em2 = em2 + e2.^2;    % Squared error
    em3 = em3 + e3.^2;    % Squared error
    em4 = em4 + e4.^2;    % Squared error

end

em1 = em1/nRun;    % MSE
em2 = em2/nRun;    % MSE
```

```
em3 = em3/nRun;     % MSE
em4 = em4/nRun;     % MSE

%-------------------------------------------------------------------------
% Average MSE evaluation
mse1 = mean(em1(end-B-M-1:end-M-1));  % Average MSE
mse2 = mean(em2(end-B-M-1:end-M-1));  % Average MSE
mse3 = mean(em3(end-B-M-1:end-M-1));  % Average MSE
mse4 = mean(em4(end-B-M-1:end-M-1));  % Average MSE
%-------------------------------------------------------------------------
fprintf('\n');


%% Results

% ------------------------------------------------------------------------
% Print table
% ------------------------------------------------------------------------

fprintf('\n');
fprintf('Mean Square Errors ↩
    ------------------------------------------\n');
fprintf('WSAF with Delta X = %3.2f Steady-state MSE = %5.7f, equal to ↩
    %5.3f dB\n',DeltaX1,mse1,10*log10(mse1));
fprintf('WSAF with Delta X = %3.2f Steady-state MSE = %5.7f, equal to ↩
    %5.3f dB\n',DeltaX2,mse2,10*log10(mse2));
fprintf('WSAF with Delta X = %3.2f Steady-state MSE = %5.7f, equal to ↩
    %5.3f dB\n',DeltaX3,mse3,10*log10(mse3));
fprintf('WSAF with Delta X = %3.2f Steady-state MSE = %5.7f, equal to ↩
    %5.3f dB\n',DeltaX4,mse4,10*log10(mse4));
fprintf('↩
    ------------------------------------------------------------------\n')↩
    ;


% ------------------------------------------------------------------------
% Plotting figures
% ------------------------------------------------------------------------

% Plot Spline functions ------------------------------------------------------
yLIM = 1.5;
xLIM = 3.0;
figure1 = figure('PaperSize',[20.98 29.68]);
box('on');
hold on;
hold('all');
grid on;
ylim([-yLIM yLIM]);
xlim([-xLIM xLIM]);
KK = 500;
yy1 = zeros(1,KK);
yy2 = zeros(1,KK);
yy3 = zeros(1,KK);
yy4 = zeros(1,KK);
xa1 = zeros(1,KK);
dx = 2*xLIM/KK;
xx = -xLIM;
for k = 1:KK
    yy1(k) = ActFunc(xx, F1.af);   % Delta X 1: Blue
    yy2(k) = ActFunc(xx, F2.af);   % Delta X 2: Black
    yy3(k) = ActFunc(xx, F3.af);   % Delta X 3: Green
    yy4(k) = ActFunc(xx, F4.af);   % Delta X 4: Magenta
```

```
    xa1(k) = xx;
    xx = xx + dx;
end
title('Profile of nonlinearities after learning','FontSize', 12, '↩
    FontWeight', 'demi');
xlabel('Nonlinearity input {\itx}[{\itn}] ','FontSize', 12, 'FontWeight',↩
     'demi');
ylabel('Nonlinearity output {\its}[{\itn}]','FontSize', 12, 'FontWeight',↩
     'demi');
plot(xa1,yy1,'b','LineWidth',2);   % Delta X 1: Blue
plot(xa1,yy2,'k','LineWidth',2);   % Delta X 2: Black
plot(xa1,yy3,'g','LineWidth',2);   % Delta X 3: Green
plot(xa1,yy4,'m','LineWidth',2);   % Delta X 4: Magenta
legend('Delta X 1','Delta X 2','Delta X 3','Delta X 4','Location','↩
    SouthEast');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);

% MSE dB ------------------------------------------------------------------
figure2 = figure('PaperSize',[20.98 29.68]);
box('on');
hold on;
grid on;
ylim([-out_noise_level_dB-5 0]);
[bb,aa] = butter(2, 0.005);
edb1 = 10*log10( em1 );
edb2 = 10*log10( em2 );
edb3 = 10*log10( em3 );
edb4 = 10*log10( em4 );
plot(filter(bb,aa,edb1),'b','LineWidth',2);       % Delta X 1: Blue
plot(filter(bb,aa,edb2),'k','LineWidth',2);       % Delta X 2: Black
plot(filter(bb,aa,edb3),'g','LineWidth',2);       % Delta X 3: Green
plot(filter(bb,aa,edb4),'m','LineWidth',2);       % Delta X 4: Magenta
title('Comparisons of MSE by using different \Delta_x values','FontSize',↩
     12, 'FontWeight', 'demi');
xlabel('Samples','FontSize', 12, 'FontWeight', 'demi');
ylabel('MSE [dB]   10log({\itJ}({\bfw},{\bfQ}))','FontSize', 12, '↩
    FontWeight', 'demi');
legend('Delta X 1','Delta X 2','Delta X 3','Delta X 4');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
```

3.11 WSAF_IIR_demo

This script implements the first experiment in [8]. The experiment consists in the identification of an unknown Wiener system composed by a linear component (see Figure 3.2), represented by the following recursive filter

$$H(z) = \frac{0.6 - 0.4z^{-1}}{1 + 0.2z^{-1} - 0.5z^{-2} + 0.1z^{-3}},$$

and a nonlinear memoryless target function implemented by a 23 points length LUT $\mathbf{q}_0$, interpolated by a uniform third degree spline with an interval sampling $\Delta x = 0.2$ and defined as

$$\mathbf{q}_0 = \{-2.2, -2, -1.8, \ldots, -1.0, -0.8, -0.91, 0.4, -0.2, \\ 0.05, 0, -0.15, 0.58, 1.0, 1.0, 1.2, \ldots, 2.0, 2.2\}.$$

The input signal $x[n]$ consists in 30.000 samples of the signal generated by the following relationship

$$x[n] = ax[n-1] + \sqrt{1-a^2}\xi[n],$$

where $\xi[n]$ is a zero mean white Gaussian noise with unitary variance and $a = 0.5$. In addition it is considered an additive white noise $v[n]$ with a signal to noise ratio $SNR = 30$ dB. The learning rates are set to $\mu_w = \mu_q = 0.01$ and B-spline basis is used. The orders of MA and AR parts of the IIR adaptive filter are set to $M = 2$ and $N = 3$ respectively.

The complete MATLAB code is reported below.

```
clear all;
close all;
disp('WSAF_IIR_demo');
% ------------------------------------------------------------------

%%  Parameters setting

% Input parameters ------------------------------------------------
Lx = 30000;                 % Length of input signal
nRun = 10;                  % Number of runs
out_noise_level_dB = 30;  % SNR
out_noise_level = 10^(-out_noise_level_dB/20);  % Noise level

x = zeros(Lx,1);            % x (Nx1) input signal array definition

% Colored signal generation ---------------------------------------
a = 0.5;
b = sqrt(1-a^2);
% x = filter( b, [1 -a], randn( size(x))) ; % H(z) = b/(1+a*z^-1)

% Adaptive filter definition --------------------------------------
M =  2;             % MA filter length
N =  3;             % AR filter length
K =  1;             % Affine Projection order; K=1 => NLMS
mu0  = 0.01;        % Learning rate for the linear IIR filter
mQ0  = 0.01;        % Learning rate for control points
if Lx < 30000,      % Batch for evaluating MSE
        B = 100;
else
        B = 4000;
end

% Spline activation function definition and initialization --------
afinit = 0;    % Init act. func. -1 0 ... (ONLY -1, bip.sig. or  0 =↩
    linear )
aftype = 5;    % Kind act. f. -1 0 1 2 4 5; (4 = CR-spline, 5 = B-spline ↩
    )
Slope  = 1;    % Slope
DeltaX = 0.2;  % Delta X
x_range = 2;   % Range limit

% Creating the nonlinearity ---------------------------------------
af0 = create_activation_function( afinit, aftype, DeltaX, x_range, Slope,↩
     2);
af1 = create_activation_function( afinit, aftype, DeltaX, x_range, Slope,↩
     M);

% Printing the control points -------------------------------------
```

```
fprintf('af.lut_len = %d \n', af0.lut_len );
for j=1 : af0.lut_len
    fprintf('%16.2f \n', af0.Q(j) );
end

%% Initialization

% ---- Model definition ------------------------------------------------------
% apa_af0 = create_SPL_apa_iir_adaptive_filter_1(M,N,K,mu0,mQ0,1e-2,af0);↩
     % Model APA
apa_af0 = create_SPL_lms_iir_adaptive_filter_1(2,3,mu0,mQ0,1e-2,af0); % ↩
    Model LMS

% TARGET: Nonlinear memoryless function implemented by Spline ↩
    interpolated LUT
apa_af0.af.Q = [
            -2.20
            -2.00
            -1.80
            -1.60
            -1.40
            -1.20
            -1.00
            -0.80
            -0.91
            -0.40
            -0.20
             0.05
             0.0
            -0.15
             0.58
             1.00
             1.00
             1.20
             1.40
             1.60
             1.80
             2.00
             2.20
];
QL = length(apa_af0.af.Q);   % Number of control points

% Linear filter ---------------------------------------------------------------
apa_af0.b =  [0.6   -0.4].';                % MA model
apa_af0.a =  [0.2   -0.1   0.1].';          % AR model
apa_af0.w =  [apa_af0.b.'  apa_af0.a.'].'; % ARMA model

% ---- SAF definition --------------------------------------------------------
apa_af1 = create_SPL_lms_iir_adaptive_filter_1(M,N,mu0,mQ0,1e-2,af1);  % ↩
    SAF

% Initialize ------------------------------------------------------------------
Nn = Lx + M + K;    % Total samples
for i = Lx+1:Nn
    x(i)=0;
end

dn = zeros(Nn,1);       % Noise output array
d  = zeros(Nn,1);       % Desired signal array
y  = zeros(Nn,1);       % Output array
e  = zeros(Nn,1);       % Error array
em = zeros(Nn,1);       % Mean square error array
```

```
wm   = zeros(M+N,1);    % Mean value of w
varW = zeros(M+N,1);    % Variance value of w
qm   = zeros( QL, 1 );  % Mean value Spline coeff
varQ = zeros( QL, 1 );  % Variance value Spline coeff


%% Main loop ------------------------------------------------------------------
disp('WSAF_IIR algorithm start ... ');
t = clock;

for n = 0 : nRun-1
    fprintf('Test nr. %d/%d\n', n+1, nRun);
    x = filter( b, [1 -a], randn( size(x)) ) ;   % H(z) = b/(1+a*z^-1)
    dn = out_noise_level * randn( size(x) ) ;   % Noise

    % SAF I.C. ------------------------------------------------------------------
    apa_af1.b    = zeros(M,1);
    apa_af1.b(1) = 1;
    apa_af1.a    = zeros(N,1);
    apa_af1.w    = [apa_af1.b.' apa_af1.a.'].';

    % Set Nonlinear Func I.C. ---------------------------------------------------
    apa_af1.af  = create_activation_function(afinit, aftype, DeltaX, ↩
        x_range, Slope, M);

    % WSAF_IIR Evaluation -------------------------------------------------------
    for k = 1 : Lx
        % Computing the desired output ------------------------------------------
        [apa_af0, d(k)] = FW_WSPL_IIR_F(apa_af0, x(k));    % Model

        % Updating WSAF_IIR -----------------------------------------------------
        [apa_af1, y(k), e(k)] = AF_LMS_WSPL_IIR_F(apa_af1, x(k), d(k) + ↩
            dn(k));  % SAF LMS (Eqs. (17) and (19) )
    end

    em   = em + e.^2;  % Squared error

    % SAF run-time mean and variance estimation ---------------------------------
    wm   = (1/(n+1))*apa_af1.w + (n/(n+1))*wm;
    varW = varW + (n/(n+1))*((apa_af1.w - wm).^2);

    qm   = (1/(n+1))*apa_af1.af.Q + (n/(n+1))*qm;
    varQ = varQ + (n/(n+1))*((apa_af1.af.Q - qm).^2);
end
em  = em/nRun;  % MSE
apa_af1.af.Q = qm;

%-------------------------------------------------------------------------------
% Average MSE evaluation
mse = mean(em(end-B-M-1:end-M-1));  % Average MSE
%-------------------------------------------------------------------------------
fprintf('\n');


%% Results

%-------------------------------------------------------------------------------
% Print tables
%-------------------------------------------------------------------------------
fprintf('\n');
fprintf('Number of iterations = %d\n',nRun);
fprintf('Learning rates:  muW = %5.3f   muQ = %5.3f\n', mu0, mQ0);
```

```
fprintf('a = %4.2f  b = %4.2f\n',a, b );
fprintf('Number of filter weights: M = %d and N = %d\n', M, N);
fprintf('Number of control points = %d\n', QL);
fprintf('AF type = %d\n',aftype);
fprintf('DeltaX  = %4.2f\n',DeltaX);
fprintf('SNR_dB  = %4.2f dB\n',out_noise_level_dB);
fprintf('Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse,10*log10(mse)↩
    );
fprintf('\n');
fprintf('Mean and Variance Tables ↩
    ----------\n');
for i=1:QL
    fprintf('i=%2d  q0 =%5.2f    qm =%9.6f    varQ = %10.3e \n', i,  ↩
        apa_af1.af.Q(i), qm(i), varQ(i) );
end
fprintf('\n');
fprintf('↩
    ----------\n')↩
    ;
for i=1:M
    fprintf('i=%d  w0 =%5.2f    wm =%9.6f    varW = %10.3e \n', i, apa_af1↩
        .w(i), wm(i), varW(i) );
end
fprintf('↩
    ----------\n')↩
    ;


% ------------------------------------------------------------------------
% Plotting figures
% ------------------------------------------------------------------------

% Plot Spline functions ------------------------------------------------------
yLIM = 1.5;
xLIM = 3.0;
figure1 = figure('PaperSize',[20.98 29.68]);
box('on');
hold on;
hold('all');
ylim([-yLIM yLIM]);
xlim([-xLIM xLIM]);
grid on;
KK = 500;
yy1 = zeros(1,KK);
yy2 = zeros(1,KK);
xa1 = zeros(1,KK);
dx = 2*xLIM/KK;
xx = -xLIM;
for k = 1:KK
    yy1(k) = ActFunc(xx, apa_af0.af);   % Model
    yy2(k) = ActFunc(xx, apa_af1.af);   % Adapted
    xa1(k)=xx;
    xx = xx + dx;
end
xlabel('Linear combiner output {\its}[{\itn}] ','FontSize', 12, '↩
    FontWeight', 'demi');
ylabel('SAF output {\ity}[{\itn}]','FontSize', 12, 'FontWeight', 'demi');
title('Profile of model and adapted nonlinearity \varphi(s[n])','FontSize↩
    ', 12, 'FontWeight', 'demi');
plot(xa1,yy1,'-.r','LineWidth',2);
plot(xa1,yy2,'k','LineWidth',2);
legend('Target','Adapted','Location','SouthEast');
```

```
set(gca, 'FontSize', 10, 'FontWeight', 'demi');

% Filter coefficients ------------------------------------------------------
figure2 = figure('PaperSize',[20.98 29.68]);
hold on;
plot(apa_af0.w,'LineWidth',2);
plot(apa_af1.w,'m','LineWidth',2);
xlabel('time {\itn}','FontSize', 12, 'FontWeight', 'demi');
ylabel('Linear combiner coefficients','FontSize', 12, 'FontWeight', 'demi↩
    ');
legend('Model','Adapted');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');

% MSE dB -------------------------------------------------------------------
figure3 = figure('PaperSize',[10 15]);
box('on');hold on;hold('all');
ylim([-out_noise_level_dB-5 10]);
grid on;
edb = 10*log10( em );
[bb,aa] = butter(2, 0.02 );
plot( filter(bb,aa,edb ),'Color',[1 0 0],'LineWidth',2);
noiseLevel(1: length(edb)-1 ) = -out_noise_level_dB;
plot( noiseLevel,'--','Color',[0 0 1],'LineWidth',2 );
title('Wiener IIR SAF convergence test','FontSize', 12, 'FontWeight', '↩
    demi');
xlabel('Samples','FontSize', 12, 'FontWeight', 'demi');
ylabel('MSE [dB]   10log({\itJ}({\bfw},{\bfQ}))','FontSize', 12, '↩
    FontWeight', 'demi');
legend('MSE', 'NoiseLevel' );
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);
```

3.12 WSAF_IIR_Volterra_compare_demo

This script implements the second experiment in [8] (see Figure 3.2). It consists in the identification of the Back and Tsoi NARMA model that is a cascade of the following 3-rd order IIR filter

$$H(z) = \frac{0.0154 + 0.0462z^{-1} + 0.0462z^{-2} + 0.0154z^{-3}}{1 - 1.99z^{-1} + 1.572z^{-2} - 0.4583z^{-3}},$$

and the following nonlinearity

$$y[n] = \sin(s[n]).$$

The input signal $x[n]$ consists of 50,000 samples of the signal generated by the following relationship

$$x[n] = ax[n-1] + \sqrt{1-a^2}\xi[n],$$

where $\xi[n]$ is a zero mean white Gaussian noise with unitary variance and $a = 0.9$. The learning rates are set to $\mu_w = \mu_q = 0.1$ and B-spline basis is used. The orders of MA and AR parts of the IIR adaptive filter are set to $M = 4$ and $N = 3$ respectively. The IIR WSAF is also compared with a full 3-rd order Volterra architecture with $M_v = 15$ coefficients and adapted by a LMS algorithm with $\mu_v = 0.01$, a simple

FIR WSAF approach [6] using 15 filter taps $\mu_w = \mu_q = 0.02$ and a conventional IIR polynomial filter [5] using $\mu_p = 0.01$, $M = 4$, $N = 3$ and a 5-th order polynomial.

The complete MATLAB code is reported below.

```
clear all;
close all;
disp('WSAF_IIR_Volterra_compare_demo');
% ------------------------------------------------------------------

%%  Parameters setting

% Input parameters ------------------------------------------------
Lx = 50000;                  % Length of input signal
nRun = 10;                   % Number of runs
out_noise_level_dB = Inf;    % SNR
out_noise_level = 10^(-out_noise_level_dB/20);  % Noise level

x = zeros(Lx,1);   % x (Nx1) input signal array definition

% Colored signal generation ---------------------------------------
a = 0.95;
b = sqrt(1-a^2);

% Adaptive filter definition --------------------------------------
% IIR_WSAF
M =  4;          % MA filter length
N =  3;          % AR filter length
K =  1;          % Affine Projection order; K=1 => NLMS
mu = 0.02;       % Learning rate for the linear filter
mQ = 0.02;       % Learning rate for control points
% WSAF
Mw = 5;          % Linear filter length
mu0 = 0.1;       % Learning rate APA
mQ0 = 0.1;       % Learning rate for control points
delta = 1e-2;    % APA regularization parameters
% Polynomial
muP = 0.005;     % Learning rate for the linear filter
mQP = 0.005;     % Learning rate for polynomial coeficients
% Volterra AF
MV1 = 15;        % Number of Volterra coefficients
Kv  = 1;         % Affine projection order
muV = 0.01;      % Learning rate for Volterra AF


% Spline activation function definition and initialization --------
afinit = 0;     % Init act. func. -1 0 ... (ONLY -1, bip.sig. or  0,↩
    linear )
aftype = 5;     % Kind act. f. -1 0 1 2 3 4 5
aftype2 = 3;    % Kind act. f. -1 0 1 2 3 4 5
Slope = 1;      % Slope
DeltaX = 0.2;   % Delta X
x_range = 2.0;  % Range limit
Pord = 5;       % Olynomial order

% Creating the nonlinearity ---------------------------------------
af1 = create_activation_function( afinit, aftype, DeltaX, x_range, Slope,↩
     M);
af2 = create_activation_function( afinit, aftype, DeltaX, x_range, Slope,↩
     Mw);
af3 = create_activation_function( afinit, aftype2, DeltaX, x_range, Slope↩
    , M, Pord);
```

```
%% Initialization

% ---- Model definition ------------------------------------------------------
apa_af1 = create_SPL_lms_iir_adaptive_filter_1(M,N,mu0,mQ0,delta,af1);  %↩
     IIR_WSAF LMS
apa_af2 = create_SPL_apa_adaptive_filter_1(M,K,mu,mQ,delta,af2);        %↩
     WSAF APA
%apa_af2 = create_SPL_lms_adaptive_filter_1(M,mu,mQ,delta,af);          %↩
     WSAF LMS
apa_af3 = create_SPL_lms_iir_adaptive_filter_1(M,N,muP,mQP,delta,af3);  %↩
     IIR Polynomial LMS
VF      = create_III_ord_Volterra_filter_1(MV1, Kv, muV, delta);        %↩
     III Order Volterra AF

% Initialize ------------------------------------------------------------------
Nn = Lx + M + K;   % Total number of samples
for i = Lx+1:Nn
    x(i)=0;
end

e1 = zeros(Nn,1);    % Error array IIR_WSAF
e2 = zeros(Nn,1);    % Error array III-order Volterra AF
e3 = zeros(Nn,1);    % Error array WSAF
e4 = zeros(Nn,1);    % Error array IIR_Polynomial AF
y  = zeros(Nn,1);    % Output array
dn = zeros(Nn,1);    % Noise output array
d  = zeros(Nn,1);    % Desired signal array


%% Main loop ------------------------------------------------------------------
disp('WSAF_IIR_Volterra_compare algorithm start ... ');
t = clock;

for n = 0 : nRun-1
    fprintf('Test nr. %d/%d\n', n+1, nRun);

    % Back & Tsoi NARMA system signal generation ------------------------------
    % Back, A. D., Tsoi, A. C. (1993). A simplified gradient algorithm
    % for IIR synapse Multilayer Perceptron. Neural Networks, 5, 456-462.
    x = filter( sqrt(1-b^2), [1 -b], randn( size(x))); % H(z) = a*z↩
        ^-1/1-b*z^-1
    x = x/max(abs(x));      % Normalize x
    z = filter([0.0154  0.0462  0.0465  0.0154], [1  -1.99  1.572  -0.4583], x)↩
        ;
    d = sin(z);  %d = sin((pi/2)*z);

    % Noise generation --------------------------------------------------------
    dn = out_noise_level * randn( size(x) ) ;

    % SAF I.C. ----------------------------------------------------------------
    apa_af1.b    = zeros(M,1);
    apa_af1.b(1) = 1;
    apa_af1.a    = zeros(N,1);
    apa_af1.w    = [apa_af1.b.' apa_af1.a.'].';
    apa_af1.af   = create_activation_function(afinit, aftype, DeltaX, ↩
        x_range, Slope, M);
    VF           = create_III_ord_Volterra_filter_1(MV1, Kv, muV, delta);↩
          % III Order Volterra
    apa_af2.w(:) = 0;
    apa_af2.w(1) = 1;
```

```
    apa_af2.af   = create_activation_function(afinit, aftype, DeltaX, ↩
        x_range, Slope, Mw);
    apa_af3.b    = zeros(M,1);
    apa_af3.b(1) = 1;
    apa_af3.a    = zeros(N,1);
    apa_af3.w    = [apa_af1.b.' apa_af1.a.'].';

    % Set Nonlinear Func I.C. ---------------------------------------------
    apa_af3.af   = create_activation_function(afinit, aftype2, DeltaX, ↩
        x_range, Slope, M, Pord);

    % Algorithms Comparison -----------------------------------------------
    for k = 1 : Lx
        % Updating WSAF_IIR -----------------------------------------------
        [apa_af1, y(k), ee] = AF_LMS_WSPL_IIR_F(apa_af1, x(k), d(k) + dn(↩
            k));  % SAF
        e1(k) = e1(k) +  ee^2;

        % Updating III-order Volterra AF ----------------------------------
        [VF, yDummy, ee] = AF_III_ord_VF_APA_F( VF, x(k), d(k) + dn(k) );↩
                % III Order Volterra AF
        e2(k) = e2(k) +  ee^2;

        % Updating WSAF ---------------------------------------------------
        [apa_af2, y(k), ee] = AF_APA_WSPL_F(apa_af2, x(k), d(k) + dn(k));↩
                % WSAF APA
        %[apa_af2, y(k), ee] = AF_LMS_WSPL_F(apa_af2, x(k), d(k) + dn(k))↩
            ;     % WSAF LMS
        e3(k) = e3(k) +  ee^2;

        % Updating IIR Polynomilal AF -------------------------------------
        [apa_af3, y(k), ee] = AF_LMS_WPOLY_IIR_F(apa_af3, x(k), d(k) + dn↩
            (k)); % IIR Polynomial AF
        e4(k) = e4(k) +  ee^2;
    end

end

% Mean square errors ------------------------------------------------------
e1 = e1/nRun;   % IIR_WSAF
e2 = e2/nRun;   % III-order Volterra AF
e3 = e3/nRun;   % WSAF
e4 = e4/nRun;   % IIR Polynomial AF

fprintf('\n');

%% Resuts

% Plot Spline functions ---------------------------------------------------
yLIM = 1.5;
xLIM = 3.0;
fig1 = figure('PaperSize',[20.98 29.68]);
box('on');
hold on;
hold('all');
ylim([-yLIM yLIM]);
xlim([-xLIM xLIM]);
grid on;
KK = 500;
yy1 = zeros(1,KK);
yy2 = zeros(1,KK);
yy3 = zeros(1,KK);
```

```
yy4 = zeros(1,KK);
xa1 = zeros(1,KK);
dx = 2*xLIM/KK;
xx = -xLIM;
for k = 1:KK
    yy1(k) = ActFunc(xx, apa_af1.af);      % IIR WSAF
    [VF,yy2(k)] = FW_III_ord_VF_F(VF, xx); % III-order Volterra AF
    yy3(k) = ActFunc(xx, apa_af2.af);      % WSAF
    yy4(k) = ActFunc(xx, apa_af3.af);      % IIR Polynomial AF
    xa1(k) = xx;
    xx = xx + dx;
end
title('Profile of spline nonlinearity after learning');
xlabel('Linear combiner output {\its}[{\itn}] ');
ylabel('SAF output {\ity}[{\itn}]');
plot(xa1,yy1,'-.k','LineWidth',2);
plot(xa1,yy2,'r','LineWidth',2);
plot(xa1,yy3,'b','LineWidth',2);
plot(xa1,yy4,'g','LineWidth',2);
legend('IIR WSAF', 'Volterra', 'WSAF', 'IIR Poly');

% MSE dB ----------------------------------------------------------------
fig2 = figure('PaperSize',[10 15]);
box('on');hold on;hold('all');
ylim([-45 0]);
grid on;
[bb,aa] = butter(3, 0.02 );
edb1 = 10*log10( e1 );
edb2 = 10*log10( e2 );
edb3 = 10*log10( e3 );
edb4 = 10*log10( e4 );
plot( filter(bb,aa,edb1),'-k','LineWidth',2);   % IIR_WSAF
plot( filter(bb,aa,edb2),'-.b','LineWidth',2);  % III-order Volterra AF
plot( filter(bb,aa,edb3),'--r','LineWidth',2);  % WSAF
plot( filter(bb,aa,edb4),':m','LineWidth',2);   % IIR Polynomial AF
title('Back and Tsoi NARMA model','FontSize', 12, 'FontWeight', 'demi');
xlabel('Samples','FontSize', 12, 'FontWeight', 'demi');
ylabel('MSE [dB]   10log({\itJ}({\bfw},{\bfq}))','FontSize', 12, '↩
    FontWeight', 'demi');
legend('IIR WSAF', 'Volterra', 'WSAF', 'IIR Poly');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
set(gcf, 'PaperSize', [20.98 29.68]);

% Filter coefficients ---------------------------------------------------
fig3 = figure('PaperSize',[20.98 29.68]);
hold on;
plot(apa_af1.w,'k','LineWidth',2);
plot(apa_af2.w,'r','LineWidth',2);
plot(apa_af3.w,'m','LineWidth',2);
title('Filter coefficients');
xlabel('time {\itn}');
ylabel('Linear combiner coefficients');
legend('IIR WSAF', 'WSAF', 'IIR Poly');
```

3.13 WSAF_Volterra_compare_demo

This script implements the second experiment in [6]. It consists in the identification of a nonlinear dynamic system composed by three blocks. The first and last blocks are two 4-th order IIR filter, Butterworth and Chebychev respectively, with

transfer functions

$$H_B(z) = \frac{(0.2851 + 0.5704z^{-1} + 0.2851z^{-2})}{(1 - 0.1024z^{-1} + 0.4475z^{-2})} \cdot \frac{(0.2851 + 0.5701z^{-1} + 0.2851z^{-2})}{(1 - 0.0736z^{-1} + 0.0408z^{-2})},$$

and

$$H_C(z) = \frac{(0.2025 + 0.2880z^{-1} + 0.2025z^{-2})}{(1 - 1.01z^{-1} + 0.5861z^{-2})} \cdot \frac{(0.2025 + 0.0034z^{-1} + 0.2025z^{-2})}{(1 - 0.6591z^{-1} + 0.1498z^{-2})},$$

while the second block is the following nonlinearity

$$y[n] = \frac{2s[n]}{1 + |s[n]|^2}.$$

This system is similar to radio frequency amplifiers for satellite communications (High Power Amplifier), in which the linear filters model the dispersive transmission paths, while the nonlinearity models the amplifier saturation. The input signal $x[n]$ consists of 100,000 samples of the signal generated by the following relationship

$$x[n] = ax[n-1] + \sqrt{1-a^2}\xi[n],$$

where $\xi[n]$ is a zero mean white Gaussian noise with unitary variance and $a = 0.9$. The learning rates are set to $\mu_w = \mu_q = 0.02$, while $\delta_w = 0.01$. We compare the proposed SAF architecture using $M = 15$ filter taps with two 3-rd order Volterra filters using $M = 5$ and $M = 15$ respectively.

The complete MATLAB code is reported below.

```
clear all;
close all;
disp('WSAF_Volterra_compare_demo');
%-------------------------------------------------------------------------

%% Parameters setting

% Input parameters ------------------------------------------------------
Lx = 100000;                    % Length of the input signal
nRun = 10;                      % Number of runs
out_noise_level_dB = Inf;       % SNR
out_noise_level = 10^(-out_noise_level_dB/20);   % Noise level

x = zeros(Lx,1);                % x (Nx1) input signal array definition

% Colored signal generation ---------------------------------------------
a = 0.9;
b = sqrt(1-a^2);
% x = filter(b, [1 -a], randn( size(x)));   % H(z) = b/(1+a*z^-1)
disp('..... ');

% Adaptive filter definition --------------------------------------------
M   = 15;           % Length of the linear filter
MV  = 15;           % Length of Volterra filter
K   = 1;            % Affine Projection order; K=1 => NLMS
KV  = 1;            % Volterra Affine Projection order; K=1 => NLMS
mu  = 0.001;        % Learning rate for the linear filter
mQ  = 0.001;        % Learning rate for ctrl points
muV = 0.001;        % Learning rate for Volterra
```

```
delta = 1e-2;       % APA regularization parameters
if Lx < 30000,      % Batch for evaluating MSE
        B = 100;
else
        B = 4000;
end

% Spline activation function definition and initialization ----------
afinit  = 0;        % Kind of init act. f. -1 0 1 2 */
aftype  = 5;        % Kind act. f. -1 0 1 2 4 5 */
Slope   = 1;        % Slope
DeltaX  = 0.2;      % DeltaX
x_range = 8;        % Range limit


%% Initialization

% Creating the nonlinearity -----------------------------------------
af = create_activation_function(afinit, aftype, DeltaX, x_range, Slope, M↩
    );

% --- Models definition -----------------------------------------------
apa_af = create_SPL_apa_adaptive_filter_1(M, K, mu, mQ, delta, af);   % ↩
    WSAF APA
VF     = create_III_ord_Volterra_filter_1(MV, KV, muV, delta);        % ↩
    III Order Volterra

% Initialize --------------------------------------------------------
N = Lx + M + K;
for i = Lx+1:N
    x(i)=0;
end

dn = zeros(N,1);        % Noise output array
d  = zeros(N,1);        % Desired signal array
y1  = zeros(N,1);       % Output array
y2  = zeros(N,1);       % Output array
e1  = zeros(Lx,1);      % Error array
e2  = zeros(Lx,1);      % Error array
em1 = zeros(Lx,1);      % Mean square error
em2 = zeros(Lx,1);      % Mean square error


%% Main loop ---------------------------------------------------------
disp('WSAF_Volterra_compare algorithm start ... ');
t = clock;

for n = 0 : nRun-1
    fprintf('Test nr. %d/%d\n', n+1, nRun);

    % Nonlinear system signal generation ----------------------------
    x = randn(Lx,1);          % x (Nx1) input signal array definition
    if a>0,
        x = filter( a, [1 -a], x);              % Colored input
    end
    y = filter(conv([0.2851  0.5704  0.2851], [0.2851  0.5701  0.2851]), ...
               conv([1      -0.1024  0.4475], [1      -0.0736  0.0408]), x) ↩
                   ;
    z = y./(1+abs(y.^2));
    d = filter(conv([0.2025  0.288  0.2025],[0.2025  0.0034  0.2025]), ...
               conv([1      -1.01   0.5861],[1      -0.6591  0.1498]), z);
    dn = out_noise_level*randn( size(x) );      % Noisy desired signal
```

```
    % SAF I.C. ------------------------------------------------------------
    apa_af.w (:) = 0;
    apa_af.w (1) = 0.1;

    % Set Activation Func I.C. ----------------------------------------------
    apa_af.af.Q = af.Q;


    % Models Evaluation -----------------------------------------------------
    for k = 1 : Lx

        % Updating WSAF -----------------------------------------------------
        [apa_af, y1(k), e1(k)] = AF_APA_WSPL_F(apa_af, x(k), d(k) + dn(k)↩
            );    % WSAF

        % Updating VAF ------------------------------------------------------
        [VF, y2(k), e2(k)] = AF_III_ord_VF_APA_F( VF, x(k), d(k) + dn(k) ↩
            );     % III-order Volterra
    end

    em1  = em1 + (e1.^2);   % Squared error
    em2  = em2 + (e2.^2);   % Squared error

end

em1 = em1/nRun;       % MSE
em2 = em2/nRun;       % MSE

%-------------------------------------------------------------------------
% Average MSE evaluation
mse1 = mean(em1(end-B-M-1:end-M-1));  % Average MSE
mse2 = mean(em2(end-B-M-1:end-M-1));  % Average MSE
%-------------------------------------------------------------------------
fprintf('\n');


%% Results

% ------------------------------------------------------------------------
% Print table
% ------------------------------------------------------------------------

fprintf('\n');
fprintf('Mean Square Errors ↩
    ------------------------------------------------\n');
fprintf('WSAF Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse1,10*↩
    log10(mse1));
fprintf(' VAF Steady-state MSE = %5.7f, equal to %5.3f dB\n',mse2,10*↩
    log10(mse2));
fprintf('↩
    ------------------------------------------------------------------\n')↩
    ;


% ------------------------------------------------------------------------
% Plotting figures
% ------------------------------------------------------------------------

% Plot Spline functions -------------------------------------------------
```

```
% MSE dB ----------------------------------------------------------------------
figure2 = figure('PaperSize',[20.98 29.68]);
box('on');
hold on;
grid on;
ylim([-out_noise_level_dB-5 0]);
[bb,aa] = butter(2, 0.005);
edb1 = 10*log10( em1 );
edb2 = 10*log10( em2 );
plot(filter(bb,aa,edb1),'Color',[0 0 0],'LineWidth',2);          % WSAF
plot(filter(bb,aa,edb2),'Color',[1 0 0],'LineWidth',2);          % Volterra
title('Comparisons of MSE','FontSize', 12, 'FontWeight', 'demi');
xlabel('Samples','FontSize', 12, 'FontWeight', 'demi');
ylabel('MSE [dB]    10log({\itJ}({\bfw},{\bfQ}))','FontSize', 12, '↩
    FontWeight', 'demi');
legend('WSAF','3-rd Order Volterra');
set(gca, 'FontSize', 10, 'FontWeight', 'demi');
```

References

[1] C. de Boor. *A Practical Guide to Splines*. Springer-Verlag, revised edition, 2001.

[2] V. Hegde, C. Radhakrishnan, D. J. Krusienski, and W. K. Jenkins. Series-cascade nonlinear adaptive filters. In *Proceedings of the 45-th Midwest Symposium on Circuits and Systems (MWSCAS2002)*, pages 219–222, 2002.

[3] J. Jeraj and V. J. Mathews. A stable adaptive Hammerstein filter employing partial orthogonalization of the input signals. *IEEE Transactions on Signal Processing*, 54(4):1412–1420, April 2006.

[4] L. Ljung. *System Identification - Theory for the User*. Prentice Hall, Upper Saddle River, 2nd edition, 1999.

[5] V. J. Mathews. Adaptive polynomial filters. *IEEE Signal Processing Magazine*, 8(3):10–26, July 1991.

[6] M. Scarpiniti, D. Comminiello, R. Parisi, and A. Uncini. Nonlinear spline adaptive filtering. *Signal Processing*, 93(4):772–783, April 2013.

[7] M. Scarpiniti, D. Comminiello, R. Parisi, and A. Uncini. Hammerstein uniform cubic spline adaptive filters: Learning and convergence properties. *Signal Processing*, 100:112–123, July 2014.

[8] M. Scarpiniti, D. Comminiello, R. Parisi, and A. Uncini. Nonlinear system identification using IIR spline adaptive filters. *Signal Processing*, 108:30–35, March 2015.

[9] M. Scarpiniti, D. Comminiello, R. Parisi, and A. Uncini. Novel cascade spline architectures for the identification of nonlinear systems. *IEEE Transactions on Circuits and Systems—I: Regular Papers*, 62(7):1825–1835, July 2015.

[10] M. Scarpiniti, D. Comminiello, G. Scarano, R. Parisi, and A. Uncini. Steady-state performance of spline adaptive filters. *IEEE Transactions on Signal Processing*, 64(4):816–828, February 2016.