The Adaptive Signal Processing Toolbox

For use with Matlab

Author:

Dr. Eng. John Garas
garas@dspalgorithms.com

ASPT User Manual
Version 2.1

Preface

Since the early days of adaptive signal processing, computer simulations have been used to examine the performance of adaptive systems, compare adaptive algorithms, and prove the feasibility of new adaptive applications. The difficulty of analytical analysis of adaptive signal processing systems explains the popularity of computer simulation in the research and development of adaptive filters. The "Adaptive Signal Processing Toolbox", or ASPT for short, has been designed to enhance the simulation process by isolating the user from the details of the adaptive algorithms implementation while giving the user a complete control on the algorithms parameters and behavior. ASPT users may use the adaptive filters as building blocks without the need to know how those blocks are constructed, or may edit the algorithms to optimize certain characteristics important to a specific application. This approach has proved to increase productivity, shorten time to market, and enhance the understanding of the systems being developed, and therefore increasing the potential of developing better adaptive systems.

The Adaptive Signal Processing Toolbox is a software package developed specifically for engineers and researchers involved in developing adaptive signal processing systems. ASPT is also an indispensable tool for class instructors to aid in teaching adaptive signal processing, and quickly and easily demonstrating the applications of adaptive filters. ASPT contains a continuously expanding collection of basic as well as advanced adaptive filters algorithms and many practical applications. ASPT isolates the user from the details of the algorithms internal implementation and allows using the adaptive algorithms as reliable, well tested, and well documented black box functions. ASPT contains adaptive algorithms for transversal, lattice, recursive, and nonlinear filters, with implementations in the time and frequency domains, as well as specialized algorithms for applications such as active noise and vibration control, and beam forming. ASPT also comes with simulation examples for applications of adaptive filters including echo cancelers, single channel and multichannel active noise and vibration control, beam forming, channel equalization, adaptive line enhancers, system identification, interference canceling, and linear prediction.

Researchers who develop new adaptive algorithms to meet specific requirements need to test their newly developed techniques against existing ones. They must be familiar with the existing state of the art and be able to easily and quickly experiment with several features of the existing algorithms. In many cases, the requirements of the new algorithm can be readily met by modifying an existing algorithm or combining the processing of several existing ones. For those researchers, ASPT provides the basic foundation on which they can build their developments. With the large set of existing state of the art algorithms provided by ASPT, the researcher has a better understanding of the known techniques. In a few minutes, he can measure the performance enhancement brought about by his newly developed algorithm compared to known ones. He can easily experiment with modifying the existing state of the art without the need to implement all those known techniques. ASPT, therefore, makes it possible for those researchers to concentrate on their value added work by removing redundant activities, and therefore, shortening development time.

Engineers who deploy adaptive filters to solve specific technical problems would probably benefit the most from a library of adaptive algorithms such as ASPT. The first step in developing an adaptive system is usually building a simulation of the system to explore the benefits of a specific filter structure, define the filter parameters, examine the system performance, and predict the problems that might come up in real-time implementation. For instance, an engineer designing a new Acoustic Echo Canceler (AEC) for a video conferencing system must choose the adaptive filter structure (FIR, IIR, Lattice, Volterra, etc) that is most suitable for typical conference rooms.

Next, the adaptive algorithm to be used in updating the AEC coefficients must be chosen. Usually, the choice of the adaptive algorithm is constrained by system resources such as memory usage and processor cycles (MIPS) as well as system performance requirements such as convergence time and Echo Return Loss Enhancement (ERLE). To make a sound choice, the AEC designer would probably need to set up a simulation system using measured speech fragments in a conference room, implement several adaptive filters that he knows from experience that may meet the system requirements, experiment with the parameters of each filters, and test against the required overall system performance. This process can be very lengthy, especially if experiments with different filter structures updated using different algorithms are required. Due to development time limitation, the situation usually encountered in practice, is that the designer chooses an algorithm that is already available, or one that is easy to implement, regardless of its suitability for the application at hand, or whether it gives the best performance in this specific application. In such design scenario, ASPT is a necessary tool that provides the system designer with a rich set of filter structures, each updated using a wide variety of algorithms. ASPT allows the designer to easily and quickly experiment with FIR, IIR, lattice, and nonlinear filter structures, update the filters using different algorithms in time and frequency domains, and as easily compare between the performance, limitations, and resource requirements of each filter. ASPT can save designers many man-months (if not years when evaluated over a company wide activities over a several projects) of low level programming activities allowing cheaper and faster product development cycle.

On the other hand, class and lab instructors and industry trainers might find ASPT an indispensable tool in teaching adaptive signal processing concepts using hands-on approach. Using ASPT, instructors can easily make their point clear using real-life demonstrations without writing lengthy software programs. Students can easily and quickly study the performance of complex algorithms, compare between traditional algorithms, and experiment for themselves with filter parameters. By bringing applications closer to the class, the teaching and learning experiences become more interesting than just fiddling with equations. In the past few years, ASPT has proved its benefits for both instructors and students. The base of ASPT itself has been developed at Eindhoven University of Technology, during the time the author was pursuing his PhD program. For this reason, the author is committed to education institutes, and hopes that one day ASPT will be the preferred tool in teaching adaptive filters. To encourage using ASPT in adaptive filters education, several algorithms that are only of theoretical interest have been added to enhance the understanding of adaptive techniques. High performance, efficient, and nonlinear filters can serve as good material for advanced courses in adaptive signal processing.

ASPT is the fruit of many years of research and development in the area of adaptive algorithms and their applications. We introduce this toolbox to our colleagues in the hope that it will enhance and encourage the development of adaptive systems. We very much hope that engineers, researchers, class instructors, and students will like it, use it, and help us to improve it in the years to come.

List of Figures

List of Tables

Contents

Chapter 1

Overview

This manual describes the Matlab version of the Adaptive Signal Processing Toolbox, ASPT for short. ASPT is a collection of adaptive algorithms, applications, and other helper functions designed to aid in simulating, developing, and analyzing adaptive systems. This chapter gives you general information such installation and configuration instructions, notational conventions, and lists the contents and organization of the rest of this document.

1.1 Hardware and Software requirements

The current implementation of the Matlab version of the Adaptive Signal Processing Toolbox is a cross platform version that runs within the Matlab environment. ASPT, therefore, will run on all platforms supported by Matlab 5.0 or higher including Windows, Linux, Macintosh, and Unix platforms. To use the Matlab version of ASPT you need to have Matlab 5.0 or higher installed. No other toolboxes are required since all functionalities needed by the toolbox are internally implemented.

1.2 Installing ASPT

To install the Adaptive Signal Processing Toolbox do the following.

1. Extract the package files you received on your disk drive where you would like to install ASPT. In the following it is assumed that you are using Windows operating system and you have extracted ASPT contents at `D:\`. If you are using an other operating system, Unix or Linux for instance, replace `D:\` with your home directory or where Matlab is installed. Extracting will create a directory named `D:\dspalgorithms\asptxxx`, where `xxx` is the ASPT version information.

2. Start Matlab and type the following at the command line prompt
   ```
   >> cd D:\dspalgorithms\asptxxx
   ```
   and press the `Enter` key,

3. Run the installation program located in the main ASPT directory by typing the following at the Matlab command line prompt
   ```
   >> asptinstall
   ```
   and press the `Enter` key. This will add the `asptdir` and `asptver` functions to the main ASPT directory and will add the ASPT directories to the Matlab path. You need to have write permission to the ASPT directory and to the `pathdef.m` file to successfully install ASPT and change the Matlab path. If you do not have write permission to those files, ASPT path information can not be saved and will be lost when you quit the current Matlab session. In this case you will need to run `asptinstall` each time you start Matlab. The installation program will inform you with any problems it might find during installation.

4. Quit Matlab and restart it again,

5. Run any of the test scripts located in the `D:\dspalgorithms\asptxxx\test` directory to make sure that your installation is successful and that Matlab can find the ASPT files on its path
   `>> testnlms`
   This should execute without errors. Receiving any error message when running any of the test scripts means that the installation has not been successfully completed. In this case, refer to the contact information given in Section 1.7 for support.

1.3 Uninstalling ASPT

To uninstall ASPT, type the following at the Matlab prompt
`>> asptuninstall`
and press the `Enter` key. This will remove the ASPT directories from Matlab path but will not delete any files. To successfully change the Matlab path you need to have write permission to the file `pathdef.m`. To remove ASPT completely from your computer you need to manually delete the ASPT directory and all its contents. Refer to Section 1.5 for a list of ASPT contents.

1.4 Registering Your ASPT Software

ASPT license is currently available per machine. Therefore, a license key is required for each computer you will use to run ASPT. To obtain your license key, follow the steps below.

1. After order the toolbox version of your choice, you will receive the software package. Unpack and install the ASPT software on your computer as explained in Section 1.2 and make sure that the toolbox is set up correctly and working properly before you proceed further. The installation you have at this point will allow you to work with ASPT for a limited time. To gain unlimited access to the toolbox you need to install a license file. The steps below explain how to obtain and install your license file.

2. Determine the ASPT ID CODE for your ASPT software and computer combination by typing the following at the Matlab command line prompt
   `>> asptidcode`
   and press the `Enter` key. This will print your ASPT ID CODE in the Matlab command window. Copy this code **as is** and paste it in the correct place in the order form or email it to `aspt@dspalgorithms.com`.

3. On receiving your ASPT ID CODE, A license file matching the version you purchased will be sent to you.

4. Copy the license file you received to the main ASPT directory, replacing the current limited license.

5. Make a reserve copy of this license file on a floppy and keep it in a save place. You can use this license file if you need to reinstall ASPT on the same computer in the future.

Note that the ASPT ID CODE is a code that uniquely identifies the ASPT software, the computer on which ASPT is running, and the Matlab software. This means that if you modify any of the above after obtaining your license key, ASPT will stop functioning and you will need to request a new license key that is suitable for your new installation.

1.5 ASPT Directory Structure

The current distribution of the Matlab version of the Adaptive Signal processing toolbox includes implementation of adaptive algorithms, initialization routines, application scripts, short test scripts, help files, and documentation. ASPT also comes with some data files, and audio files that are used by the application scripts. Table 1.1 summarizes the directory structure of the ASPT distribution and shows where are those components located. In this table, the ASPT installation directory is assumed to be `asptxxx`.

Directory	Contents
`asptxxx`	Adaptive algorithms, initialization, plotting, and helper routines.
`asptxxx\apps`	Applications scripts.
`asptxxx\data`	Data and transfer functions used by the application scripts.
`asptxxx\docs`	Documentation.
`asptxxx\help`	Matlab help files.
`asptxxx\test`	Short applications for testing purposes.
`asptxxx\wavin`	Audio files used as input signals to the application scripts.
`asptxxx\wavout`	Audio files generated by the application scripts.

Table 1.1: ASPT directory structure.

1.6 Getting Started with ASPT

ASPT has been designed to have a simple user interface and a single calling convention for all algorithms, while giving the user maximum control on the parameters of each algorithm. This homogeneous design makes ASPT very easy to use and makes new users immediately comfortable with the toolbox. In fact, the homogeneous calling procedure means that if you know how to use one algorithm you will have no hard time using any of the others.

Each adaptive algorithm supported by ASPT has an initialization function with its name starting with the `init_` prefix and a main function with name starting with the `aspt` prefix. For instance, the Least Mean Squares (LMS) algorithm has its initialization function named `init_lms()` and its main function named `asptlms()`. The initialization function is usually called only once to create and initialize the variables required for its algorithm, while the main function is usually called in a processing loop to performs the actual work of calculating the adaptive filter output and updating the filter coefficients. You can find this calling procedure repeated over and over again in all scripts located in the `apps` and `test` directories.

Probably the fastest way to learn how to use the Adaptive Signal Processing Toolbox is to examine the scripts in the `apps` and `test` directories.. Those scripts give example applications covering all adaptive algorithms included in the current release of ASPT. The scripts in the `test` directory are self contained short applications that are also listed in this documentation. Each of those scripts creates and initializes an adaptive filter, creates the input and desired signals, calls the adaptive algorithm to update the filter coefficients in a loop, and finally generates a plot demonstrating the functionality of the application. The scripts in the `apps` directory are more involved applications with real life data and usually take more time to execute. The input and output data for those applications are usually read from audio files located in the `wavin` directory and generate output audio files which will be stored in the `wavout` directory. Those applications use the initialization functions and adaptive algorithms in the same way the simple test applications do. Examining a few of those scripts will immediately make you comfortable with the toolbox. You are also encouraged to copy from those scripts and paste into your own applications.

Once you have examined a few applications, you are ready to build your own adaptive applications using the ASPT routines. You can obtain more information on a specific algorithm by examining

its reference page in this document or by typing
```
>> help function_name
```
at the Matlab command line prompt. To see a list of all functions and applications included in your ASPT distribution, use the following command
```
>> help aspt
```

1.7 Obtaining Support

Should you need any support in installing, upgrading, uninstalling, or otherwise using the Adaptive Signal Processing Toolbox, email to `aspt@dspalgorithms.com`. Also email to the same address for bug report, suggestions, comments, and feedback. The ASPT team will be glad to answer any question you might have. They also appreciate any suggestions that might lead to enhancing the toolbox.

1.8 ASPT Flavors and Related Products

Most of the adaptive algorithms included in the Matlab version of the Adaptive Signal Processing Toolbox are also available in other forms. ANSI C and C++ source code as well as object code versions are available for any platform having an ANSI C/C++ compiler. ASPT is also available for several Digital Signal Processors (DSPs). Off the shelf applications based on ASPT such as network and acoustic echo cancelers, active noise and vibration control, and interference cancelers are also directly available for licensing for many platforms. For more information on those and other adaptive algorithms and applications email to `support@dspalgorithms.com`.

1.9 ASPT Naming conventions

To manage the large and continuously increasing number of routines included in ASPT, and to protect naming conflicts with other packages, the Matlab implementation of ASPT uses several naming conventions. Each routine name consists of two parts. The first part is a prefix indicating the category to which the routine belongs. For example, all initialization routines start with the `init_` prefix. The second part of the routine name is a description of the algorithm or function performed by the routine. For example, the `init_lms()` routine initializes the variables and structures for the Least Mean Squares (LMS) adaptive filter. The following prefixes are currently in use.

- `init_` : used for initialization routines.
- `aspt` : used for adaptive filters routines.
- `plot_` : used for plotting routines.

Several non-adaptive routines that do not contribute to the set of adaptive algorithms but perform complementary functions required by the toolbox are also included in the ASPT distribution. Examples of those functions are `osfilter()` and `sovfilt()` that implement an overlap-save frequency domain fixed filter, and a second order Volterra fixed filter, respectively. To keep the names of those routines descriptive, the naming conventions rules mentioned above do not apply to those helper routines.

1.10 Notational Conventions

The following notational conventions are used throughout the rest of this document . Unless otherwise explicitly indicated, lower-case letters are used to represent time domain signals, while upper-case letters represent signals that have been processed by a transformation, such as the fast Fourier transform. Boldface characters (e.g. $\mathbf{X}$) represent matrices and underlined boldface characters (e.g. $\underline{\mathbf{w}}$) represent vectors. In adaptive filters contexts, the filter input signal is called $x(n)$, its output signal $y(n)$, the error signal $e(n)$, the desired signal $d(n)$, the vector of filter coefficients $\underline{\mathbf{w}}(n)$, and the adaptation constant (step size) is called μ. Code fragments, examples, and ASPT routine names are printed in fixed font (e.g. `asptlms()`).

1.11 Manual Organization

The rest of this document is organized as follows.

Chapter 2	Provides an introduction to adaptive filters, reviews the filter structures supported by ASPT, introduces the general adaptive filter model implemented by most ASPT routines, and explains the applications of adaptive filters.
Chapter 3	Gives a summery of all the routines and applications included in the toolbox grouped by filter structure.
Chapter 4	Includes the reference pages for routines used to create, initialize, and update transversal and linear combiner adaptive filters.
Chapter 5	Includes the reference pages for routines used to create, initialize, and update lattice adaptive filters.
Chapter 6	Includes the reference pages for routines used to create, initialize, and update recursive adaptive filters.
Chapter 7	Includes the reference pages for routines used to create, initialize, and update active noise and vibration control filters.
Chapter 8	Includes the reference pages for routines used to create, initialize, and update nonlinear adaptive filters.
Chapter 9	Includes the reference pages for non-adaptive routines, visualization routines, and other helper functions.
Chapter 10	Includes detailed description of the application scripts supplied with ASPT.

Chapter 2

Introduction to Adaptive Filters

2.1 Introduction

In signal processing and control applications where the signals and transfer functions involved are time invariant and known at design time, designing fixed filters and controllers to achieve the desired design goals is sufficient. In many applications, however, signals, transfer functions, and the environment in which the system operates are time-varying. In some applications, such as in active noise and vibration control, it is the rule rather than the exception that the system to be controlled is unknown at design time. In such situations a self designing or self adjusting filter/controller is necessary to achieve the desired system function in a changing environment. This is usually done by adjusting the coefficients of a digital filter/controller on-line in the operation field by optimizing predefined quantities.

Self adjusting filters, better known as adaptive filters, might have any underlying filter structure. The most widely used adaptive filter structure is the transversal structure due to the stability and simplicity of analysis of those filters. The linear combiner structure is a generalized version of the transversal structure, and is mainly used in array signal processing applications. Recursive adaptive filters have also found wide application in adaptive line enhancers, autoregressive signal modeling and channel equalization. A third structure which is widely used in adaptive linear prediction applications is the lattice structure. All above mentioned adaptive filter structures are well supported by the current release of the Adaptive Signal Processing Toolbox (ASPT). A short theoretical introduction to transversal filters, linear combiner filters, recursive filters, and lattice filters is given in Sections 2.2.1, 2.2.2, 2.2.3, and 2.2.4, respectively. Adaptive algorithms for adjusting the coefficients of those filter structures are documented in Chapters 4 to 7.

This chapter also includes a brief review of the theory behind adaptive signal processing. Adaptive algorithms can roughly be divided into two main categories. The first based on statistical optimization which leads to the the Least Mean Squares (LMS) algorithm and its derivatives. The second is based on deterministic optimization which leads to the Recursive Least Squares (RLS) algorithm and its derivatives. The basic optimization problems for statistic and deterministic approaches and the model on which all ASPT functions are based is introduced in Section 2.3.

A brief review of some common adaptive filters applications is given in Section 2.4. This is by far not a complete list of adaptive signal processing applications but gives the novice reader a good basis on which she can start building her own applications. System identification and forward modeling applications using adaptive filters are described in Section 2.4.1, equalization and inverse modeling in Section 2.4.2, adaptive linear prediction in Section 2.4.3, adaptive autoregressive spectrum analysis in Section 2.4.4, echo cancellation in Section 2.4.5, and finally adaptive interference canceling in Section 2.4.6

2.2 Filter Structures supported by ASPT

This section is a brief review of the digital filter structures supported by the Adaptive Signal Processing Toolbox. Each section describes the equations involved in calculating the filter output and gives specific information on how the filter structure can be used as an adaptive filter. Links to ASPT functions implementing adaptive algorithms based on each structure to adjust the filter coefficients are also given. The transversal filter structure is considered in Section 2.2.1, the linear combiner filters structure is considered in Section 2.2.2, the recursive filter structure in Section 2.2.3, the lattice filter structure in Section 2.2.4, and finally the nonlinear transversal filters in Section 2.2.5.

2.2.1 Transversal Filters

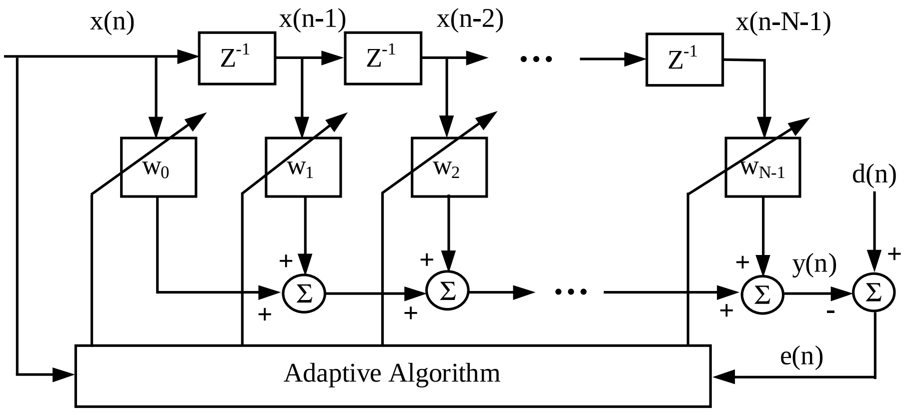

Figure 2.1: Transversal adaptive filter structure.

The most commonly used structure in implementing adaptive filters is the transversal structure shown in Fig. 2.1. The transversal adaptive filter can be split into two main parts, the filter part and the update part. The function of the former is to calculate the filter output $y(n)$, while the function of the latter is to adjust the set of N filter coefficients $w_i, i = 0, 1, \cdots, N-1$ (tap weights) so that the output $y(n)$ becomes as close as possible to a desired signal $d(n)$.

The filter part processes a single input sample $x(n)$ and produces a single output sample $y(n)$ (assuming sample per sample implementation). The filter output is calculated as a linear combination of the input sequence $x(n-i), i = 0, 1, \cdots, N-1$ composed of delayed samples of $x(n)$,

$$y(n) = \sum_{i=0}^{N-1} w_i(n) \cdot x(n-i). \tag{2.1}$$

Expressing the set of N filter coefficients at time index n and the sequence of delayed input samples in vector notations such that $\underline{\mathbf{w}}(n) = [w_0(n) \; w_1(n) \; \cdots \; w_{N-1}(n)]^T$ and $\underline{\mathbf{x}}(n) = [x(n) \; x(n-1) \; \cdots \; x(n-N+1)]^T$, where $(\cdot)^T$ is the vector transpose operator, eq (2.1) can be written as

$$y(n) = \underline{\mathbf{w}}(n)^T \cdot \underline{\mathbf{x}}(n) = \underline{\mathbf{x}}(n)^T \cdot \underline{\mathbf{w}}(n). \tag{2.2}$$

The transversal filter structure is, therefore, a linear temporal filter that processes the temporal samples of its input signal $x(n)$ to produce the temporally and consequently spectrally modified (filtered) output $y(n)$.

In fixed transversal filter applications, the set of filter coefficients are chosen at the system design time to achieve the required spectral filtering and remain constant during the filter operation. In adaptive filters applications, however, an adaptive algorithm is used to continuously adjust the filter coefficients so that a certain performance criterion is optimized in some sense. Regardless of the optimization method, it is usually desired to adjust the filter coefficients such that the filter output $y(n)$ resembles a desired signal $d(n)$, or equivalently, the error signal $e(n)$ must be minimized. The details of the optimization process defines the adaptive algorithm and its behavior.

The adaptive signal processing toolbox contains several transversal adaptive algorithms such as the Least Mean Squares (see Section 4.9), the Normalized Least Mean Squares (see Section 4.11), the leaky Normalized Least Mean Squares (see Section 4.7), the Variable Step Size Least Mean Squares (see Sections 4.10 and 4.20), and the Recursive Least Squares (see Section 4.16). When the number of filter coefficients N, is large, it is much more efficient to perform filtering and coefficient update in the frequency domain. This requires collecting a block of samples of the input signal before the fast Fourier transform (FFT) can be calculated. For this reason, a frequency domain transversal filter is usually a block processing filter that accepts a block of B input samples and produces a block of B output samples. Several implementations of block frequency domain adaptive filters are included in the adaptive signal processing toolbox, such as the Block Frequency Domain Adaptive Filter (see Section 4.2), the Partitioned Block Frequency Domain Adaptive Filter (see Section 4.12), and the Reduced Complexity Partitioned Block Frequency Domain Adaptive Filter (see Section 4.13).

2.2.2 Linear Combiner Filters

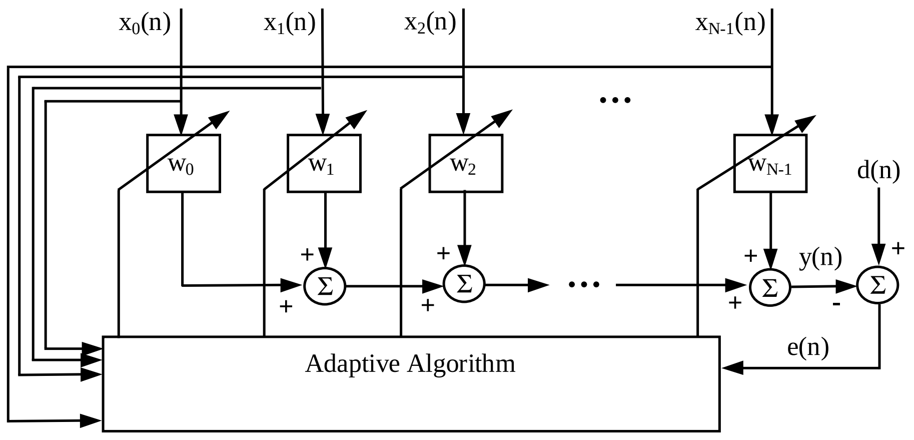

Figure 2.2: Linear combiner filter structure.

Linear combiner adaptive filters are very similar to transversal adaptive filters. The main difference is that the linear combiner input sequence is not necessarily temporal delayed samples of one single input, and it is therefore a generalized form of the transversal structure. The adaptive linear combiner filter structure is shown in Fig. 2.2. The input vector in the case of the linear combiner consists of temporal samples of several signals, that might be coming from an array of sensors for instance, and is expressed as $\underline{\mathbf{x}}(n) = [x_0(n)\ x_1(n)\ \cdots\ x_{N-1}(n)]^T$. Similar to the adaptive transversal filter, the adaptive linear combiner can be split into two main parts, the filter part and the update part. The function of the former is to calculate the filter output $y(n)$, while the function of the latter is to adjust the set of N filter coefficients $w_i, i = 0, 1, \cdots, N-1$ (tap weights) so that the output $y(n)$ becomes as close as possible to a desired signal $d(n)$.

The filter part processes the set of input signals at each time index n to produces a single output sample $y(n)$ (assuming sample per sample implementation). The filter output at time index n is calculated as a linear combination of the input signals sampled at that time instance as,

$$y(n) = \sum_{i=0}^{N-1} w_i(n) \cdot x_i(n). \tag{2.3}$$

Expressing the set of N filter coefficients at time index n in vector notations such that $\underline{\mathbf{w}}(n) = [w_0(n)\ w_1(n)\ \cdots\ w_{N-1}(n)]^T$, where $(\cdot)^T$ is the vector transpose operator, eq (2.3) can be written as

$$y(n) = \underline{\mathbf{w}}(n)^T \cdot \underline{\mathbf{x}}(n) = \underline{\mathbf{x}}(n)^T \cdot \underline{\mathbf{w}}(n). \tag{2.4}$$

When the input signals are samples of sensor signals with the sensors placed at different positions in space, the linear combiner is a linear spatial filter that processes its input signals to produce the spatially filtered output $y(n)$. The filter coefficients are usually chosen such that the signals arriving from certain directions are passed to the output while signals arriving from other directions are rejected. Such filter is usually referred to as a beam former or an array processor.

Similar to their transversal counterparts, adaptive linear combiner filters employ an adaptive algorithm to continuously adjust the filter coefficients so that a certain performance criterion is optimized in some sense. Regardless of the optimization method, it is usually desired to adjust the filter coefficients such that the filter output $y(n)$ resembles a desired signal $d(n)$ usually arriving from the "look direction", or equivalently, the error signal $e(n)$ must be minimized. The details of the optimization process defines the adaptive algorithm and its behavior.

Although the adaptive signal processing toolbox contains adaptive algorithms that are widely used with linear combiner structures, such as the Linearly Constrained Least Mean Squares (see Section 4.8), any of the adaptive algorithms mentioned in Section 2.2.1 can be used to update the linear combiner coefficients. The only necessary modification needed is to feed the adaptive algorithm with an input vector derived from temporal samples of N different signals rather than from delayed samples of a single signal.

2.2.3 Recursive Filters

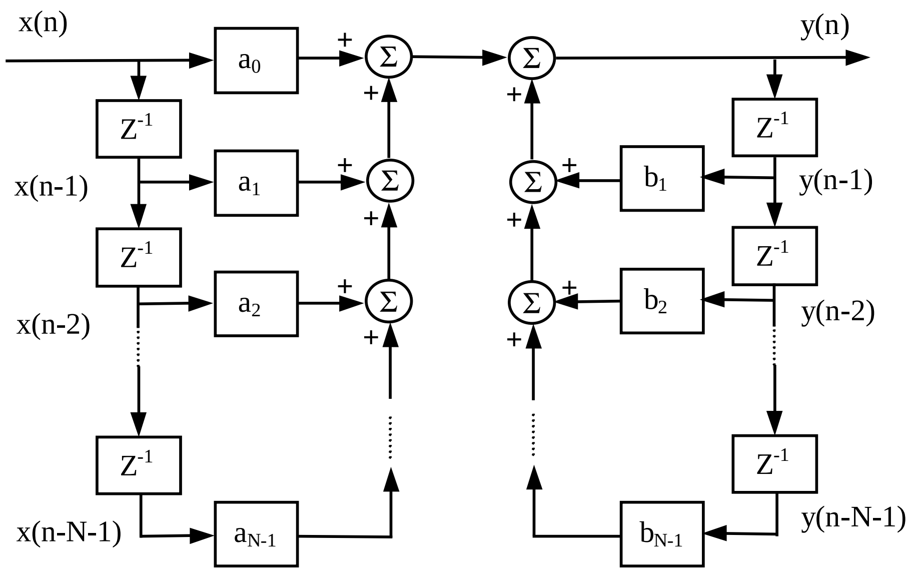

Figure 2.3: Recursive filter structure.

All filter structures mentioned so far are non-recursive structures that calculate the filter output from a linear combination of their input but do not make use of any feedback mechanism. Such non-recursive filters have impulse responses of limited duration and therefore known as Finite Impulse Response (FIR) filters. A filter structure that calculates its output as a linear combination of its current and previous input samples as well as previous samples of its output, such that shown in Fig. 2.3, is referred to as a recursive filter. Recursive filters usually have very long impulse response, therefore, they are referred to as Infinite Impulse Response (IIR) filters. The output $y(n)$ of an IIR filter is given by

$$y(n) = \sum_{i=0}^{N-1} a_i(n) \cdot x(n-i) + \sum_{j=1}^{M} b_j(n) \cdot y(n-j), \tag{2.5}$$

where $a_i(n); i = 0, 1, \cdots, N-1$ are the feed-forward coefficients and $b_j(n); j = 1, 2, \cdots, M$ are the feedback coefficients of the IIR filter. In vector notations, eq (2.5) can be written as

$$y(n) = \underline{\mathbf{a}}(n)^T \cdot \underline{\mathbf{x}}(n) + \underline{\mathbf{b}}(n)^T \cdot \underline{\mathbf{y}}(n). \tag{2.6}$$

where $\underline{\mathbf{a}}(n) = [a_0(n)\ a_1(n)\ \cdots\ a_{N-1}(n)]$ is the vector of feed-forward coefficients at time index n, $\underline{\mathbf{b}}(n) = [b_1(n)\ b_2(n)\ \cdots\ b_M(n)]$ is the vector of feedback coefficients at time index n, $\underline{\mathbf{x}}(n) = [x(n)\ \ x(n-1)\ \ \cdots\ \ x(n-N+1)]$, is the vector of current and past input samples, $\underline{\mathbf{y}}(n) = [y(n-1)\ y(n-2)\ \cdots\ y(n-M)]$, is the vector of past output samples, and $(\cdot)^T$ is the vector transpose operator.

Besides calculating the filter output, an adaptive IIR filter must also update the $N + M$ filter coefficients to optimize some performance function in the same manner as in the case of FIR adaptive filters. Adjusting the coefficients of an IIR filters, however, is complicated by two factors. The first is that the filter can run unstable very easily during adaptation if the filter poles shift outside the unit circle. The second is that the performance function to be optimized, in general, has many local minima which might lead to adjusting the filter coefficients to one of those minima and not to the desired global minimum. This is in contrast to the performance functions (the mean square error function for instance) usually encountered in adapting FIR filters which have a single global minimum. Despite those difficulties, recursive adaptive filters have found many practical applications, especially in control systems. In such applications, adaptive IIR filters offer great advantages when the physical system to be controlled or modeled is of a recursive nature as is the case of adaptive control of mechanical systems, and in active vibration control systems. In such applications, an adaptive IIR filter of a few coefficients can result in much better performance than an FIR of a few thousand coefficients. The adaptive signal processing toolbox provides many IIR adaptive algorithms such as the Equation Error (see Section 6.2), the Output Error (see Section 6.3), and the Simple Hyperstable Adaptive Recursive Filter (see Section 6.4). Furthermore, three algorithms for adapting second order IIR sections are provided, namely, SOIIR1, SOIIR2, and CSOIIR2 (see Sections 6.5, 6.6, and 6.1, respectively).

2.2.4 Lattice Filters

Lattice structures are widely used in prediction applications. Fig. 2.4 shows the lattice predictor structure of order M. Stage $m + 1$ of the lattice predictor has two inputs from the previous stage, namely the forward and backward prediction errors $e_{f_m}(n)$ and $e_{b_m}(n)$, respectively, and produces two outputs $e_{f_{m+1}}(n)$ and $e_{b_{m+1}}(n)$. The two outputs are given by the following order update equations

$$\begin{aligned} e_{f_{m+1}}(n) &= e_{f_m}(n) &&- k_{m+1} e_{b_m}(n-1), \\ e_{b_{m+1}}(n) &= e_{b_m}(n-1) &&- k_{m+1} e_{f_m}(n). \end{aligned} \tag{2.7}$$

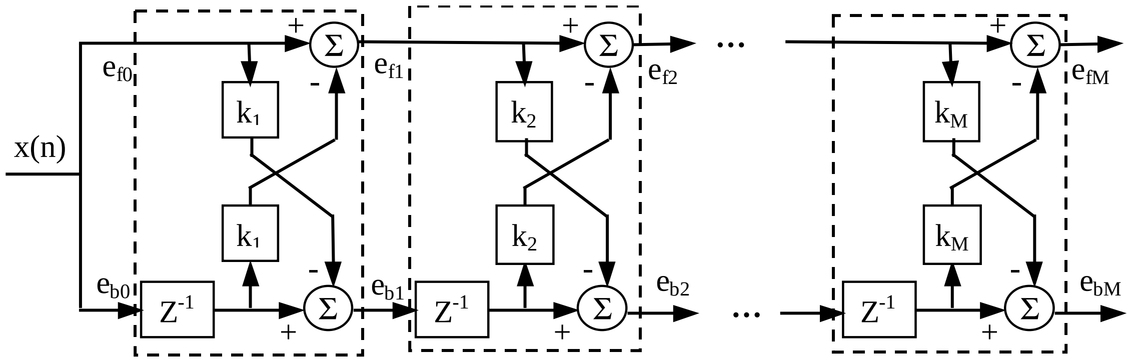

Figure 2.4: Block diagram of the lattice predictor.

The input of the first lattice stage has its forward and backward errors equal to the input signal

$$e_{f_0}(n) = e_{b_0}(n) = x(n). \tag{2.8}$$

The coefficient k_m of stage m is known as the partial correlation coefficient (PARCOR) or the reflection coefficient. The set of PARCOR coefficients for an M-stage lattice predictor are related to the coefficients of the transversal predictor of the same order (see Section 2.4.3). In fact the lattice and transversal predictors are equivalent. The Levinson-Durbin algorithm is an efficient procedure to calculate the transversal predictor coefficients $\underline{\mathbf{a}}$ from the autocorrelation function of the input sequence and it also provides the PARCOR coefficients for the corresponding lattice predictor. The following properties are well known for lattice structures

- The PARCOR coefficients always satisfy the relation $|k_m| \leq 1$.
- The power of the forward prediction error $E[e^2_{f_m}(n)]$ and the backward prediction error $E[e^2_{b_m}(n)]$ of the same stage are equal.
- The backward prediction errors $e_{b_0}(n), e_{b_1}(n), \cdots, e_{b_M}(n)$ are uncorrelated with one another for any input sequence $x(n)$. This property is very important since it shows that the lattice predictor can be seen as an orthogonal transformation with the input signal samples $x(n), x(n-1), \cdots, x(n-M+1)$ as input and the uncorrelated (orthogonal) output as the backward error from the M-stages.
- The power of the prediction error decreases with increasing lattice order. The error power decrease is controlled by the PARCOR coefficients according to the relation $P_{m+1} = (1 - k^2_{m+1})\ P_m$, where P_{m+1} is the power of the forward or backward prediction error at stage m. This indicates that the closer the value of k_{m+1} to unity the higher the contribution of stage m in reducing the prediction error. Usually the first few PARCOR coefficients have higher magnitude with the magnitude of the coefficients dropping to values close to zero for later stages.

Although the operation of lattice filters are usually described in the prediction context, the application of lattice filters is not limited to prediction applications. A traditional adaptive transversal filter can also be implemented using the lattice structure as shown in Fig. 2.5. The structure in Fig. 2.5 is known as the joint process estimator since it estimates a process $d(n)$ from another correlated process $x(n)$. The joint process estimator consists of two separate parts, the lattice predictor part and the linear combiner part. The lattice predictor part main function is to transform the input signal samples $x(n), x(n-1), \cdots, x(n-M+1)$ that might be well correlated to the uncorrelated backward prediction errors $e_{b_0}(n), e_{b_1}(n), \cdots, e_{b_M}(n)$. The linear combiner part calculates the equivalent transversal filter output according to the relationship

$$y(n) = \sum_{i=1}^{M} c_i\ e_{b_i}(n). \tag{2.9}$$

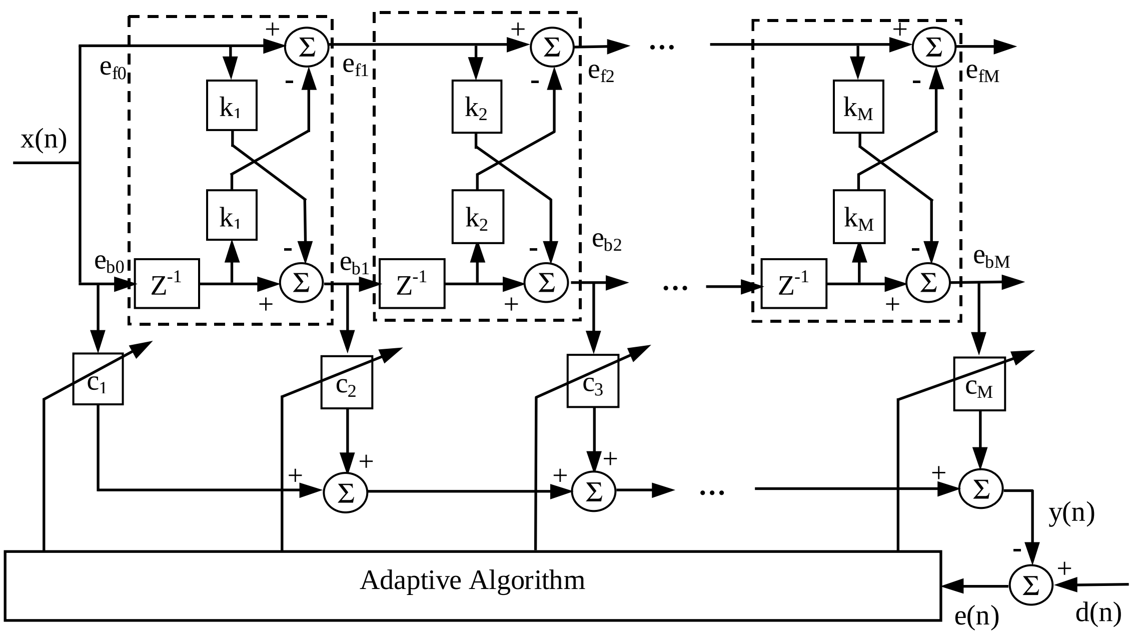

Figure 2.5: Block diagram of the joint process estimator.

An adaptive joint process estimator adjusts both the PARCOR coefficients k_i $i = 1, 2, \cdots, M$; and the linear combiner coefficients c_i; $i = 1, 2, \cdots, M$ simultaneously. The PARCOR coefficients are adjusted to minimize the forward and backward prediction error power and the linear combiner coefficients are adjusted to minimizes the square of error signal $e(n) = d(n) - x(n)$ as shown in Fig. 2.5.

2.2.5 Nonlinear Filters

In many applications linear filtering techniques do not provide satisfactory results and nonlinear filters must be used. For instance, in data communication applications it is known that the transmission errors on telephone lines at transmission rates higher than 4800 bits/s are caused almost entirely by nonlinear distortion. In applications such as subscriber lines that use much higher transmission rate, even a slight channel nonlinearity may cause problems for a linear echo canceler. The solution to such problems is to use nonlinear filters, equalizers, or controllers to manage the nonlinear systems at hand. Unlike linear systems that have a unique theory, there exists no unique method to model and characterize nonlinear systems. Different types of nonlinear systems have been developed to solve specific problems. The most commonly used nonlinear techniques are order statistics filters, polynomial filters, morphological filters, and homomorphic filters [7]. Of those techniques, the polynomial filters are supported by the current release of ASPT. The most widely used polynomial filters are the Volterra filters discussed below.

A causal nonlinear time-invariant continuous system with finite memory can be modeled by a continuous Volterra series of finite order and finite memory as follows,

$$\begin{aligned} y(t) &= h_0 + \int_0^T h_1(t_1)x(t-t_1)dt_1 + \int_0^T\int_0^T h_2(t_1,t_2)x(t-t_1)x(t-t_2)dt_1dt_2 + \cdots \\ &\quad + \int_0^T \cdots \int_0^T h_k(t_1,\cdots,t_k)x(t-t_1)\cdots x(t-t_k)dt_1\cdots dt_k, \end{aligned} \tag{2.10}$$

where the multidimensional functions $h_k(t_1, \cdots, t_k)$ are the Volterra kernel functions that are assumed to be symmetric with respect to their variables. Similarly, a causal nonlinear time-invariant discrete system with finite memory can be modeled by a discrete Volterra series of finite order and finite memory as follows,

$$\begin{aligned} y(n) &= h_0 + \sum_{m_1=0}^{N-1} h_1(m_1)x(n-m_1) + \sum_{m_1=0}^{N-1}\sum_{m_2=0}^{N-1} h_2(m_1,m_2)x(n-m_1)x(n-m_2) + \cdots \\ &\quad + \sum_{m_1=0}^{N-1}\cdots\sum_{m_k=0}^{N-1} h_k(m_1,\cdots,m_k)x(n-m_1)\cdots x(n-m_k). \end{aligned} \tag{2.11}$$

One of the important properties of the Volterra series based filters is that the output of both the continuous and discrete Volterra filters are linear functions with respect to the coefficients. The nonlinearity of the filter is introduced by forming the cross products of the input signals. This linearity property allows applying much of the linear systems theory, including linear adaptive filtering, to the Volterra series based nonlinear filters. A disadvantage of the Volterra filters, however, is that they can not be used to model strong nonlinearity such as saturation and discontinuity.

To illustrate, the second order Volterra filter is discussed in more details below. The output of the second order Volterra filter with symmetric coefficients is given by

$$y(n) = \sum_{m_1=0}^{N-1} w_1(m_1)x(n-m_1) + \sum_{m_1=0}^{N-1}\sum_{m_2=m_1}^{N-1} w_2(m_1,m_2)x(n-m_1)x(n-m_2), \tag{2.12}$$

where N is the filter memory length. The linear relationship mentioned above allows writing equation (2.12) in vector notation as follows,

$$y(n) = \underline{\mathbf{w}}(n)^T\,\underline{\mathbf{x}}(n), \tag{2.13}$$

where $\underline{\mathbf{w}}(n)$ is the vector of filter coefficients including the linear kernel, w_1 and the nonlinear kernel, w_2. As an example, the coefficients vector of a second order Volterra filter of memory length 3 are given by

$$\underline{\mathbf{w}}(n) = [w_1(0), w_1(1), w_1(2), w_2(0,0), w_2(0,1), w_2(0,2), w_2(1,1), w_2(1,2), w_2(2,2)]. \tag{2.14}$$

The vector $\underline{\mathbf{x}}(n)$ includes the current and past input samples as well as the cross products needed to calculate the nonlinear part of the filter output.

$$\underline{\mathbf{x}}(n) = [x(n), x(n-1), x(n-2), x^2(n), x(n)x(n-1), x(n)x(n-2), x^2(n-1), x(n-1)x(n-2), x^2(n-2)]. \tag{2.15}$$

The total length of $\underline{\mathbf{w}}(n)$ and $\underline{\mathbf{x}}(n)$ for the second order Volterra filter is $N + sum(1:N)$ or $3 + (1+2+3) = 9$ in the above example.

Besides calculating the filter output, an adaptive second order Volterra filters must also adjust the filter coefficients to optimize a certain performance index in some sense such that the filter output becomes as close as possible to the desired signal $d(n)$. Again, due to the linearity property mentioned above, methods used to adapt linear adaptive filters can readily be applied to adapt the coefficients of the Volterra adaptive filter. The current release of ASPT contains several functions that implement Volterra filters. For fixed filters, `sovfilt()` (Section 9.11) calculates the output of a second order Volterra filter. Second order Volterra adaptive filters can be implemented using `asptsovlms()` (Section 8.1), `asptsovnlms()` (Section 8.2), `asptsovrls()` (Section 8.3), `asptsovtdlms()` (Section 8.4), or `asptsovvsslms()` (Section 8.5).

2.3 Basic Adaptive Filter Model

The optimization problem on which all ASPT functions are designed to solve is shown in Fig. 2.6. The adaptive filter is the dotted box in this figure and consists of two parts. The filter part and the update part. The filter part (labeled "Adjustable filter" in Fig. 2.6), can be based on any of the filter structures mentioned in Section 2.2. The function of the filter part is to calculate the filter output signal $y(n)$ as shown in Sections 2.2.1 to 2.2.4. The set of filter coefficients are continuously adjusted by the update part. The update part (labeled "Adaptive algorithm" in Fig. 2.6) is responsible for adjusting the filter coefficients so that the filter output $y(n)$ becomes as close as possible to a desired signal $d(n)$. In most cases, the update part changes the filter coefficients in small steps to minimize a certain function of the error signal $e(n)$, defined as the difference between the desired signal $d(n)$ and the filter output $y(n)$,

$$e(n) = d(n) - y(n). \tag{2.16}$$

The function to be minimized is often referred to as the performance functions, also known as the performance index. The performance function can be chosen based on statistic or deterministic approaches.

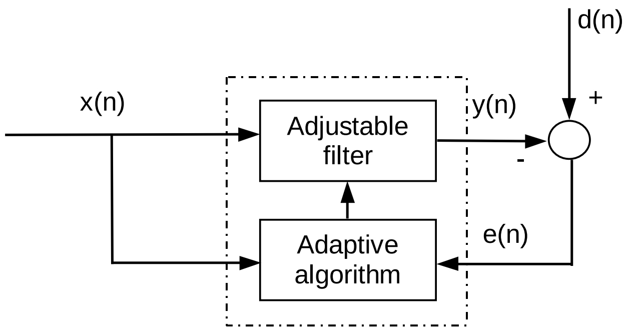

Figure 2.6: Block diagram of the general adaptive filtering problem.

The most commonly used statistical performance function is the mean square of the error signal $\zeta(n) = E\{e^2(n)\}$, where $E\{\cdot\}$ is the expectation operator. In this case, the update part of the adaptive filter adjusts the filter coefficients to minimize the mean square value of the error signal. On achieving this goal, and in ideal situations, the statistical average (mean value) of the error approaches zero, and the filter output approaches the desired signal. The adaptive filter has converged to its optimum solution in the mean square sense. When the input $x(n)$ and desired $d(n)$ are stationary signals, this optimization process leads to the well known Wiener filter [11]. The Least Mean Square (LMS) algorithms is a good example of a practical algorithm based on this statistical approach. The exact details on how the coefficients are adjusted define the time it takes to reach the final solution (the conversion time) and the difference between this final solution and the optimum solution (the final misadjustment). Many ASPT functions are based on this statistical framework. Examples are `asptlms()` (Section 4.9), `asptnlms()` (Section 4.11), `asptleakynlms()` (Section 4.7), `asptvsslms()` (Sections 4.10) for time domain sample per sample transversal and linear combiner filters. For time domain sample per sample recursive filters `aspteqerr()` (Section 6.2), and `asptouterr()` (Section 6.3). For block processing transversal filter `asptbfdaf()` (Section 4.2), `asptpbfdaf()` (Section 4.12), and `asptrcpbfdaf()` (Section 4.13). And finally for lattice filters `asptlmslattice()` (Section 5.4), `asptlbpef()` (Section 5.2), and `asptlfpef()` (Section 5.3).

A commonly used deterministic performance function is the weighted sum of the squared value of the previous error signal samples $\zeta(n) = \Sigma_{k=1}^{n} \lambda^{n-k} e^2(k)$, where λ is a constant close to, but less than one, and $k = 1, 2, \cdots, n$. This choice puts more emphasis on recent observed error samples and gradually forgets about the past samples, a good reason for calling the parameter λ the forgetting factor. Minimizing $\zeta(n)$ leads to an optimum set of filter coefficients for the given set of data that makes the filter output $y(n)$ as close as possible to the desired signal $d(n)$ in the least squares sense. It is worth mentioning, however, that if the set of data satisfy certain statistical properties, and a large data length is used, the optimum filter coefficients obtained from this deterministic optimization approaches the Wiener (statistical) solution [4]. The deterministic approach mentioned above is the basis for the Recursive Least Squares (RLS) algorithm and its derivatives known for fast convergence and fast tracking properties. Examples of ASPT functions that are based on the deterministic framework are `asptrls()` (Section 4.16). And finally for lattice filters `asptrlslattice()` (Section 5.5), `asptrlslbpef()` (Section 5.7), and `asptrlslfpef()` (Section 5.8).

ASPT functions are all designed to match the model shown in Fig. 2.6. Besides parameters specific for each algorithm, the adaptive algorithms take the input $x(n)$, desired $d(n)$, and the filter coefficients vector from previous iterations as input parameters and provide the filter output $y(n)$, the error signal $e(n)$, and the updated filter coefficients as output parameters. For sample per sample algorithms such as the `asptlms()`, the adaptive function is called each sample in a loop to process each pair of input and desired samples. For block processing functions such as the `asptbfdaf()`, the adaptive algorithm is called each B samples, where B is the block length to process B input and B desired samples and provides B output and B error samples. The filter coefficients are then updated each B samples, the frequency of calling the adaptive algorithm. This closely simulates real time implementations and provides insight into real time implementations performance. More details on the operation of adaptive algorithms, their characteristics, and properties are given in the reference page of each algorithm. For theoretical background on those algorithms, the reader is referred to one of the classical text books mentioned at the end of this document.

2.4 Adaptive Filters Applications

Adaptive filters have become invaluable system component in modern industry. Without adaptive filters, many of the systems we currently rely on in our daily life would not exist. This section summarizes the most common applications of adaptive filters currently used in commercial systems. System identification and forward modeling applications using adaptive filters are described in Section 2.4.1, equalization and inverse modeling in Section 2.4.2, adaptive linear prediction in Section 2.4.3, adaptive autoregressive spectrum analysis in Section 2.4.4, echo cancellation in Section 2.4.5, and finally adaptive interference canceling in Section 2.4.6

2.4.1 System Identification and Forward Modeling

System identification is an essential stage in control systems design. The goal of this stage is to establish a model of the physical system (plant) to be controlled. The control system is then designed to meet the design criteria based on the plant model. In many cases, the system to be controlled is slowly time varying and it is therefore of little use to design a control system based on the plant model. Instead, an adaptive control system is used and in most cases, the system identification is also performed on-line using an adaptive modeling or adaptive system identification as shown in Fig. 2.7. The controller in this case is referred to as an adaptive controller or a self-tuning regulator.

Another common application in which adaptive system identification is widely used is active noise and vibration control (ANVC), which usually employ adaptive controllers. All ANVC adaptive algorithms rely on an adaptive model for the secondary path between the control actuators and the error sensors. ANVC system might perform their system identification step in a stage prior to

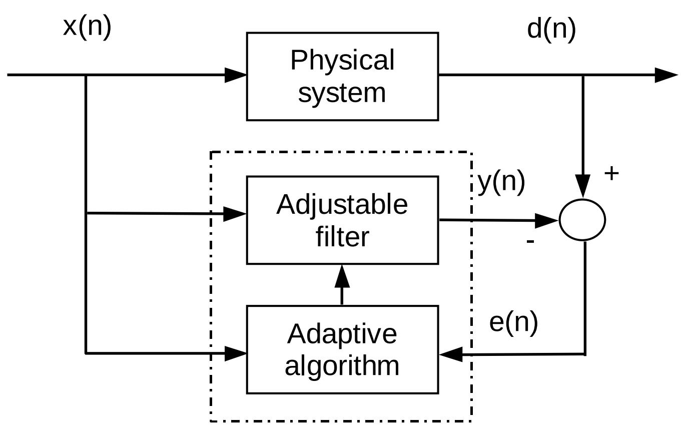

Figure 2.7: Block diagram of the general adaptive system identification (forward modeling) problem.

operation or update the secondary path model during operation depends on how fast the changes in the physical secondary path occur.

The adaptive model is obtained by exciting the physical system to be modeled and the adaptive filter by a spectrally rich and persistent input signal $x(n)$ as shown in Fig. 2.7. An adaptive algorithm is then used to minimize the difference $e(n)$ between the physical system output $d(n)$ and the adaptive filter output $y(n)$. On convergence, and in ideal cases, the error signal is reduced to zero and $y(n)$ approaches $d(n)$. This in turn means that the filter impulse response approaches the physical systems' response.

It is important to note that successful adaptive system identification start with correctly choosing the adaptive filter structure. When the plant response is oscillatory in nature, an infinite impulse response adaptive filter updated using the equation error algorithm (section 6.2) or the output error algorithm (Section 6.3) might be used. When the system response is short, an FIR transversal filter is usually preferred for stability reasons. An FIR adaptive model might be updated using the least mean squares or its normalized version (Section 4.9 and 4.11, respectively). For longer FIR models, frequency domain adaptive algorithms (such as BFDAF and PBFDAF, discussed in Sections 4.2 and 4.12, respectively) give superior performance and huge computational saving. Lattice adaptive filters have also been successfully and advantageously used in some applications such as modeling of the earth layers in seismic explorations. Adaptive lattice filters might be updated using the LMS-lattice or RLS-lattice algorithms (Sections 5.4 and 5.5, respectively). The Adaptive Signal Processing Toolbox includes many system identification examples for different physical systems and using different adaptive algorithms. Examples of those examples are `model_arlmsnewt` (Section 10.21), `model_eqerr` (Section 10.22), `model_lmslattice` (Section 10.23), `model_mvsslms` (Section 10.24), `model_outerr` (Section 10.25), `model_rlslattice` (Section 10.26), `model_sharf` (Section 10.27), `model_tdlms` (Section 10.28), and `model_vsslms` (Section 10.29).

2.4.2 Equalization and Inverse Modeling

The basic idea of inverse modeling, also known as deconvolution or equalization, is shown in Fig. 2.8. The input signal $u(n)$ is filtered through a physical system, which might be a communication channel for instance. The observed distorted signal $x(n)$ is filtered through an adaptive inverse model of the physical system such that the output $y(n)$ is as close as possible to the input signal $u(n)$. To achieve this goal, the coefficients of the adaptive filter are adjusted to minimize the difference between the filter output $y(n)$ and a delayed version of the input $u(n)$. The delay Δ is chosen to match the delay introduced by the combined physical system and the adaptive filter path. On convergence, the convolution of the adaptive filter response and the physical system response equals to a delayed impulse $\delta(n - \Delta)$. The frequency response of the adaptive filter

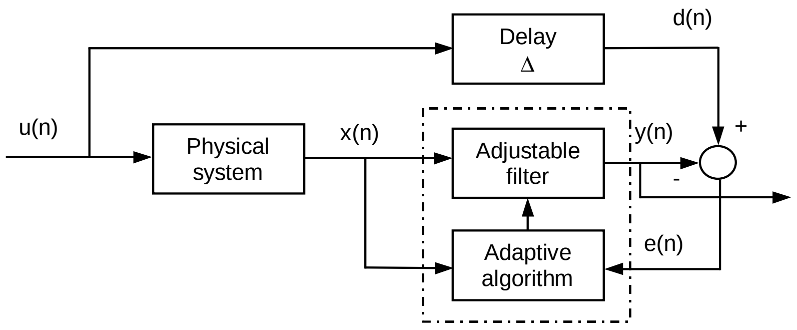

Figure 2.8: Block diagram of the general adaptive system identification (forward modeling) problem.

$\underline{\mathbf{W}}(z)$ is then an approximation of the inverse of the frequency response of the physical system $\underline{\mathbf{H}}(z)$ such that $\underline{\mathbf{W}}(z) \simeq z^{-\Delta}/\underline{\mathbf{H}}(z)$. The adaptive filter in such applications basically tries to undo the distortion introduced by the physical system to restore the input signal as much as possible.

Inverse modeling has found many practical applications in control systems and communication systems. The most widely used application of this technique is channel equalization, where the physical system is a communication channel and the adaptive filter is referred to as an adaptive channel equalizer filter. The input signal $u(n)$ in this case is the transmitted data (usually in the form of modulated pulses). The transmitted data is distorted by the communication channel in different ways. The most serious kind of distortion is the inter-symbol interference resulting from the fact that the channel response is never an impulse but one that is nonzero over many symbol periods. This results in interference between neighboring data symbols making symbol detection using a simple threshold detector unreliable and, therefore, increasing the detector symbol error rate. The adaptive equalizer is required to reduce the inter-symbol interference distortion while avoiding amplifying the additive noise usually present at the equalizer input.

The problem in the above channel equalizer setup is that the reference signal $u(n)$ is not available during normal transmission at the receiver side to be used as the desired signal, which is necessary for updating the equalizer coefficients. This is solved by introducing a training session prior to transmission. In the training session, the transmitter sends a sequence of training symbols that are known at the receiver side. The training sequence is locally generated at the receiver and used to adjust the equalizer coefficients to minimize the symbol error rate. Once the optimal coefficients have been found, the detected symbols are similar to the transmitted symbols and can be used as the desired signal for further adaptation of the equalizer coefficients to track any further changes in the channel. This mode of operation is usually referred to as the decision directed mode and works well as long as the changes in the communication channel is slow enough and the adaptive algorithm can successfully track the changes. Channel equalizers are usually implemented as adaptive transversal FIR filters. The Adaptive Signal Processing Toolbox includes several inverse modeling applications such as `equalizer_nlms` (Section 10.19), and `equalizer_rls` (Section 10.20).

2.4.3 Adaptive Linear Prediction

The general block diagram of a forward prediction system is shown in Fig. 2.9. In this application it is required to estimate the current sample of the input sequence $x(n)$ as a linear combinations of the past M input samples $x(n-\Delta), x(n-\Delta-1), \cdots, x(n-\Delta-M+1)$. To achieve this goal, the desired signal is taken as the system input $d(n) = x(n)$ and the adaptive filter input is a delayed version of the system input $x(n-\Delta)$. The filter output $y(n)$ is the required linear combination of the past input samples. The adaptive algorithm adjusts the coefficients of the adaptive filter so that the error signal $e(n) = d(n) - y(n)$ is minimized in some sense. Upon convergence the error signal $e(n)$ becomes uncorrelated with the filter input signal $x(n-\Delta)$. This indicates that $x(n)$

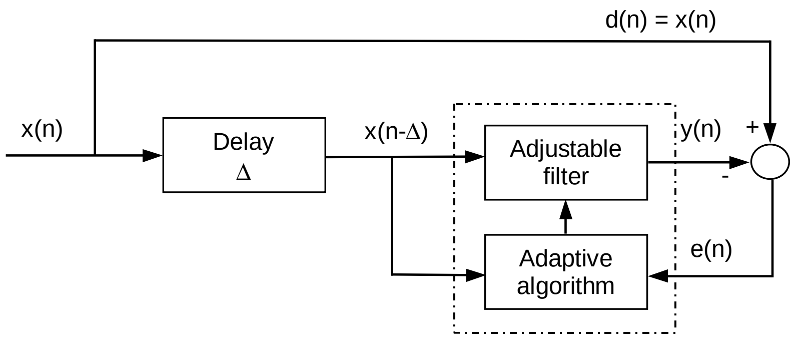

Figure 2.9: Block diagram of the general forward prediction problem.

can be uniquely expressed as the linear combination of $y(n)$ plus the residual uncorrelated term $e(n)$.

The adjustable filter in the above system is called the *forward predictor*, which might have any underlying filter structure. The most widely used filter structures in prediction applications are the transversal and lattice filters. Fig. 2.10 shows the forward predictor with a transversal adaptive filter of order M and the delay $\Delta = 1$. The filter output in this case is referred to as the M^{th} order forward prediction of the input $x(n)$ and is given by [11]

$$y(n) = \sum_{i=1}^{M} a_i \; x(n-i). \tag{2.17}$$

The error signal $e_f(n) = x(n) - y(n)$ is referred to as the M^{th} order forward prediction error. Minimizing $|e_f(n)|^2$ results in a conventional Wiener filtering problem with a solution for the optimal forward predictor coefficients $\underline{\mathbf{a}}$ given by

$$\underline{\mathbf{a}} = \mathbf{R}\;\underline{\mathbf{r}}. \tag{2.18}$$

Defining the autocorrelation function of the input at lag k as $r(k) = E[x(n)x(n-k)]$, then $\mathbf{R}$ and $\underline{\mathbf{r}}$ in (2.18) can be expressed as

$$\mathbf{R} = \begin{bmatrix} r(0) & r(1) & \cdots & r(m-1) \\ r(1) & r(0) & \cdots & r(m-2) \\ \vdots & \vdots & \ddots & \vdots \\ r(m-1) & r(m-2) & \cdots & r(0) \end{bmatrix}, \quad \underline{\mathbf{r}} = \begin{bmatrix} r(1) \\ r(2) \\ \vdots \\ r(m) \end{bmatrix} \tag{2.19}$$

The system that has its input as $x(n)$ and its output $e(n)$ is known as *the forward prediction-error filter*.

Similarly, the *backward predictor* of order M has its desired signal $d(n) = x(n-M)$ and its input $x(n)$ as shown in Fig. 2.11. The backward predictor estimates $x(n-M)$ as a linear combination of $x(n), x(n-1), \cdots, x(n-M+1)$ by minimizing the error signal $e_b(n) = x(n-M) - y(n)$ in some sense. The filter output in this case is referred to as the M^{th} order backward prediction of the input $x(n)$ and is given by

$$y(n) = \sum_{i=1}^{M} b_i \; x(n-i+1). \tag{2.20}$$

The error signal $e_b(n)$ is referred to as the M^{th} order backward prediction error. Minimizing $|e_b(n)|^2$ results in a conventional Wiener filtering problem with a solution for the optimal backward predictor coefficients $\underline{\mathbf{b}}$ given by

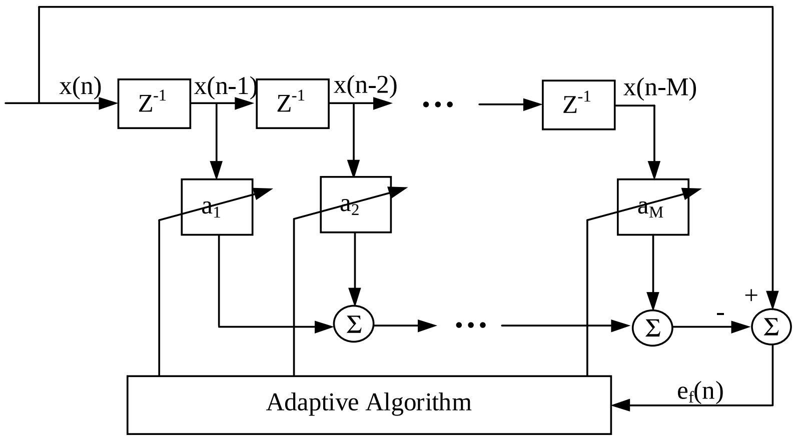

Figure 2.10: Block diagram of the transversal forward prediction problem.

$$\underline{\mathbf{b}} = \mathbf{R}\,\underline{\mathbf{r}}_b. \tag{2.21}$$

Where $\mathbf{R}$ is the same as in (2.19) since $x(n)$ is considered to be a stationary process and $\underline{\mathbf{r}}_b$ is given by

$$\underline{\mathbf{r}}_b = \begin{bmatrix} r(m) \\ r(m-1) \\ \vdots \\ r(1) \end{bmatrix}, \tag{2.22}$$

which is the same as $\underline{\mathbf{r}}$ in (2.19) with the elements arranged in reverse order. This indicates that the optimal backward predictor coefficients are the same as the optimal forward predictor coefficients of the same order but arranged in reverse order such that

$$b_i = a_{M+1-i}, \quad for\ i = 1, 2, \cdots, M. \tag{2.23}$$

The *backward prediction-error filter* is the system with input $x(n)$ and output $e(n)$. For the backward predictor with optimal coefficients it also holds that the input sequence $x(n)$ and the backward prediction error $e_b(n)$ are uncorrelated. Moreover, the J^{th} order backward prediction error $e_{b_J}(n)$ for $J = 0, 1, \cdots, M$ are uncorrelated with one another. This latter property is used in the lattice joint process estimators to decorrelate the input sequence samples as discussed in Section 2.2.4.

Besides the transversal predictors, the lattice predictor has also found a wide range of practical applications. Transversal and lattice predictors are closely related, namely there is a unique relationship between the coefficients of the optimum (forward and backward) transversal predictor of order M and the optimum reflection coefficients of the lattice predictor of the same order as mentioned in Section 2.2.4.

2.4.4 Adaptive Autoregressive Spectrum Analysis

The power spectrum of a discrete-time stochastic process can be estimated by assuming that the process can be modeled using a linear process as shown in Fig. 2.12. In this figure, the process $x(n)$ is modeled as the output of the linear filter $H(\omega)$, the input of which is a white noise $u(n)$. In this case, the power spectrum of $x(n)$ is given by

$$S_x(\omega) = \sigma_u^2 |H(\omega)|^2, \tag{2.24}$$

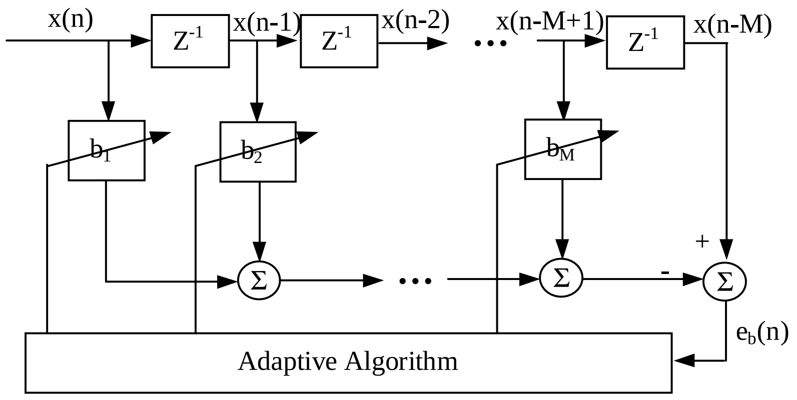

Figure 2.11: Block diagram of the transversal backward prediction problem.

where σ_u^2 is the variance of the white noise $u(n)$ which has a flat spectrum. From eq. (2.24) it is clear that the power spectrum of $x(n)$ can be obtained by estimating the transfer function $H(\omega)$. This can be done by using an adaptive prediction structure. A case of practical interest is when the filter $H(\omega)$ can be modeled as an all-pole autoregressive model with transfer function given by

$$H(\omega) = \frac{1}{1 - \Sigma_{k=1}^{M} a_k e^{-j\omega k}}. \tag{2.25}$$

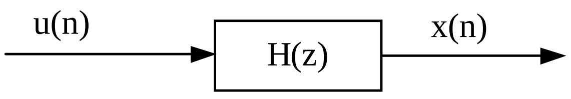

Figure 2.12: Autoregressive process modeling.

In this case, the AR parameters, $\{a_1, a_2, \cdots a_M\}$ can be estimated using an adaptive transversal prediction error filter as shown in Fig. 2.13. The power spectrum function of $x(n)$ can then be calculated from

$$S_x(\omega, n) = \frac{\sigma_u^2}{|1 - \Sigma_{k=1}^{M} a_k(n) e^{-j\omega k}|^2}. \tag{2.26}$$

In eq. (2.26), the power spectrum $S_x(\omega, n)$ as well as the autoregressive parameters $a_k; k = 1, 2, \cdots M$ are considered time varying. This provides a practical procedure for measuring the instantaneous frequency contents of the process $x(n)$ from the autoregressive parameters.

2.4.5 Echo Cancellation

Echo cancelers have become essential components in many applications, especially in communications. Echo cancelers can be divided into two main categories, namely Network Echo Cancelers (NEC) and Acoustic Echo Cancelers (AEC). Both types of echo cancelers rely on an adaptive filter to estimate the echo and subsequently use this estimate to reduce the echo in transmitted signals. The requirements, performance, underlying structure, and adaptive algorithms used to implement a NIC are usually different from those used to implement an AEC. The NEC and AEC are discussed separately in more details in Section 2.4.5.1 and 2.4.5.2, respectively.

2.4.5.1 Network Echo Cancelers

Network echo cancelers are essential in telephone networks, especially long-distance calls. Echos in telephone lines are generated mostly at the hybrid devices located in central switching offices.

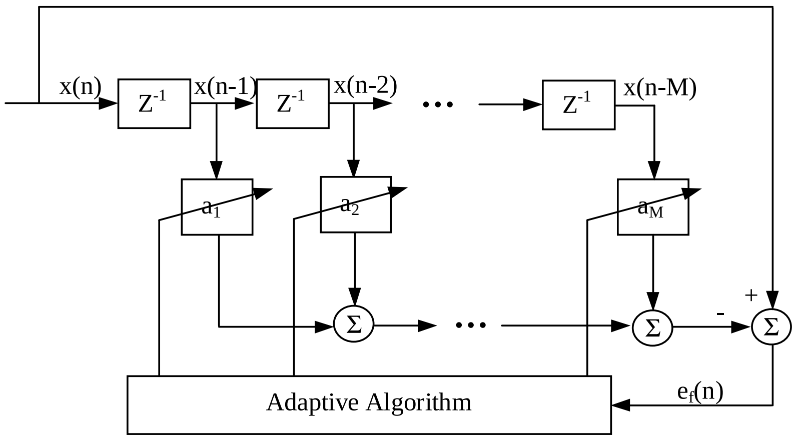

Figure 2.13: Block diagram of the adaptive transversal forward prediction error filter.

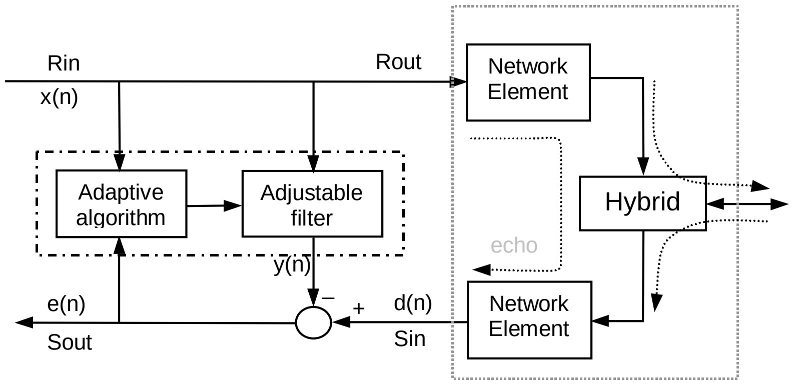

Figure 2.14: Block diagram of the network echo canceler.

The function of the hybrid is to convert the end of the two-wire subscriber link which connects the subscriber's telephone to the central office (at the right side of Fig. 2.14) to the four-wire inter-office trunk lines (at the left side of Fig. 2.14). A perfect hybrid would pass all incoming voice signals (Rin) through to the two-wire side without any leakage. Such a perfect hybrid, however, does not exist in practice and usually a portion of the incoming voice signal is leaked to the transmit terminal (Sin). The result is that the remote user will hear his own voice delayed by the network round trip (network delay). The longer the network delay, the more annoying the echo becomes, a phenomenon well observed in long-distance calls. When the telephone network is used for data communication (such as in modem and fax applications), both short and long delayed echo have severe consequences.

This problem can be solved by employing a network echo canceler at the four-wire side of the hybrid as shown in Fig. 2.14. The network echo canceler is usually an adaptive transversal filter the coefficients of which are adjusted to minimize the error signal $e(n)$ at the Sout terminal. By minimizing the error, the adaptive filter coefficients converge to a FIR model of the hybrid circuit since the hybrid and the adaptive filter share the input and the error is formed by taking the difference between the hybrid output and the adaptive filter output signals. Filtering the incoming signal through the hybrid model results in an estimate of the echo, shown as $y(n)$ in Fig. 2.14. Subtracting this echo estimate from the signal at the Sin terminal reduces the echo at the Sout terminal.

For sufficient echo reduction, the number of coefficients of the adaptive transversal filter should be chosen so that an accurate estimate of the hybrid impulse response can be estimated. Usually, the hybrid impulse response is assumed to last for 20 to 30 milliseconds. Assuming that the echo canceler is working at sampling frequency of 8 kHz, the number of coefficients of the adaptive filters are usually in the range of 240 coefficients. Such filter can be readily implemented on any general purpose DSP using the Normalized Least Mean Squares (see Section 4.11) or the Leaky NLMS algorithms (see Section 4.7). Several echo canceler application scripts are also included with ASPT such as `echo_nlms` (Section 10.16), `echo_leakynlms` (Section 10.15), `echo_bfdaf` (Section 10.14), and `echo_pbfdaf` (Section 10.17).

2.4.5.2 Acoustic Echo Cancelers

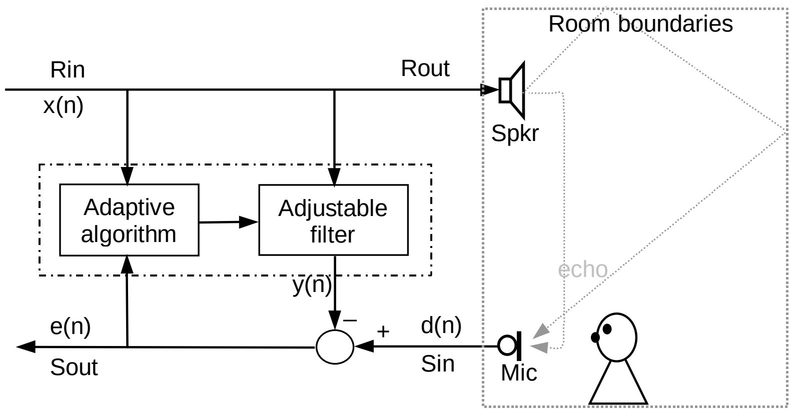

Figure 2.15: Block diagram of the acoustic echo canceler.

Acoustic echo cancelers are necessary in applications such as hands-free telephony, speakerphones, desktop communication, audio and video conferencing, voice command, desktop dictation, speech recognition, and many more. The problem addressed by the acoustic echo canceler in all those applications is illustrated in Fig. 2.15. In this figure, one side of a communication channel (of an audio conference for instance) is shown. The received speech or audio signal (Rout) is played through a loudspeaker so that all users in the local conference room can hear what the remote users say. The speech of the local users (Sin) is collected by one or more microphones and transmitted through the communication channel to the remote users. The problem in this setup is that the voice of the remote users played through the loudspeaker and its reflections off the room boundaries will also be collected by the microphones and transmitted back with the voice of the local user to the remote users who will hear their own voice delayed by the network and acoustic delay of the communication chain. The presence of the acoustic echo in the communication chain makes the users feel that they are being interrupted by their own echo, forcing them to stop speaking until the echo is faded away and the process is repeated over and over again. When the delay is large enough, which is usually the case in most communication applications especially mobile and voice over Internet protocol, this acoustic echo degrades the quality of the communication considerably.

The acoustic echo problem mentioned above can be considerably reduced by an acoustic echo canceler as shown in Fig. 2.15. The main function of the AEC is to estimate the acoustic echo and subtract this estimate from the microphone signal. In the best case, when the estimate is accurate, this leads to eliminating the acoustic echo completely. This best case, however, is not easy to realize in practical echo cancelers.

The AEC usually employs a transversal finite impulse response filter to estimate the acoustic echo. The FIR coefficients are adjusted using an adaptive algorithm to minimize the (Sout) signal, which is the adaptive filter error signal. The input (reference) signal for the adaptive filter is the far end speech (Rin) while its desired signal is the near end speech (Sin). After convergence, the adaptive filter coefficients will have an impulse response equals to the acoustic impulse response between the loudspeaker and microphone, including the reflections off the room boundaries. Since this acoustic impulse response can last several hundreds of milliseconds, acoustic echo cancelers usually employ adaptive FIR filters of several thousands coefficients. Adapting such a huge filter in real time is a real challenge, and therefore, advanced and efficient adaptive algorithms that reduce computational complexity and speed the convergence rate must be used. Examples of adaptive algorithms that have been successfully used in AEC are the Block Frequency Domain Adaptive Filter (see Section 4.2), and its partitioned version (see Section 4.12). The Normalized Least Mean Squares (see Section 4.11) and Leaky NLMS (4.7) have also been used in simple AEC applications when the acoustic impulse response is short. Several echo canceler application scripts are also included with ASPT such as `echo_nlms` (Section 10.16), `echo_leakynlms` (Section 10.15), `echo_bfdaf` (Section 10.14), and `echo_pbfdaf` (Section 10.17).

Finally, it should be noted that a communication channel of a speaker phone for instance should employ both types of echo cancelers as shown in Fig. 2.16.

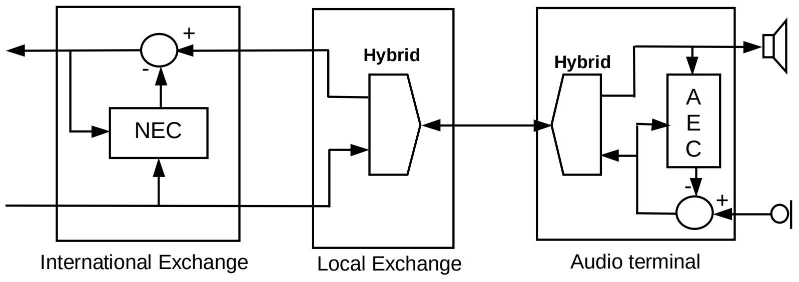

Figure 2.16: Block diagram of a communication channel employing both acoustic and network echo cancelers.

2.4.6 Adaptive Interference Canceling

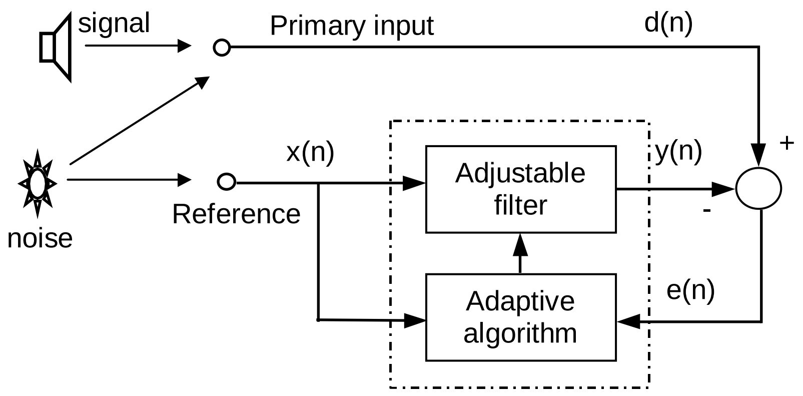

Figure 2.17: Block diagram of the adaptive interference canceling setup.

Several practical applications of adaptive filters such as active noise and vibration control, beam forming, and echo cancellation fall under the category of adaptive interference canceling. The basic principle of adaptive interference canceling is shown in Fig. 2.17 [11]. In all interference canceling application, a useful signal is corrupted with uncorrelated interference and it is desired to recover the signal from the observed corrupted signal. Adaptive interference cancelers rely on using two separate sensors. The first sensor is referred to as the primary input and receive a combination of signal and interference ($d(n) = s(n) + x1(n)$), where $s(n)$ is the signal generated by the signal source and transmitted through the channel between the source and the primary sensor, and $x1(n)$ is the interference at the primary sensor. The second sensor, referred to as the reference input, receives an interference $x(n)$ which is correlated with $x1(n)$ in some sense but uncorrelated with $s(n)$. The reference interference $x(n)$ is filtered by an adaptive filter to produce the filter output $y(n)$. The adaptive filter coefficients are adjusted so that $y(n)$ is as close as possible to the interference at the primary sensor $x1(n)$. The recovered signal is then the adaptive filter error $e(n) = d(n) - y(n)$.

The above principle can readily be extended to cancel multiple unwanted interferences by simply adding one extra reference branch for each interfering signal. Each reference branch includes a sensor that receives a signal correlated with one interference and uncorrelated with the useful signal. The output of each reference sensor is filtered by an adaptive filter. The outputs of all adaptive filters are added together to form the estimation of all interferences. This estimate is subsequently subtracted from the corrupted signal to produce the clean signal.

A frequently mentioned application of adaptive interference canceling is cleaning power-line interference from weak sensor signals. This is essential in applications such as recording electrocardiograms (ECGs), weak vibration measurements, audio frequency measurements using microphones, and many other applications that employ sensors to collect input data. The interference can be occasionally reduced by proper grounding and using shielded cables, but can not be completely eliminated. An adaptive interference canceler with a reference input taken from the power-line (with proper attenuation) and the primary input taken from the sensor as shown in Fig. 2.18 can be used to achieve much better results. The adaptive interference canceler will also track any changes in the amplitude and phase of the power-line. In this application, only two filter coefficients are needed to track the phase and amplitude of the reference interference. Fig. 2.19 shows the input and output signals of the adaptive interference canceler shown in Fig. 2.18. The result shown in Fig. 2.19 can be regenerated by typing `powerline` in Matlab after installing ASPT. Using ASPT, it is easy to simulate the power-line interference canceler, as shown in the code below. In this code, the LMS algorithm is used to adjust a two-coefficient adaptive linear combiner. The two inputs to the line combiner are the interference x1 and a 90° phase shifted version of the interference, x2. The cleaned signal is the error of the adaptive line combiner. The same technique has also been used in many other applications such as canceling donor-heart interference in heart transplant ECG, and canceling the maternal ECG in fetal ECG, canceling noise in speech signals, and canceling antenna sidelobe interference [11].

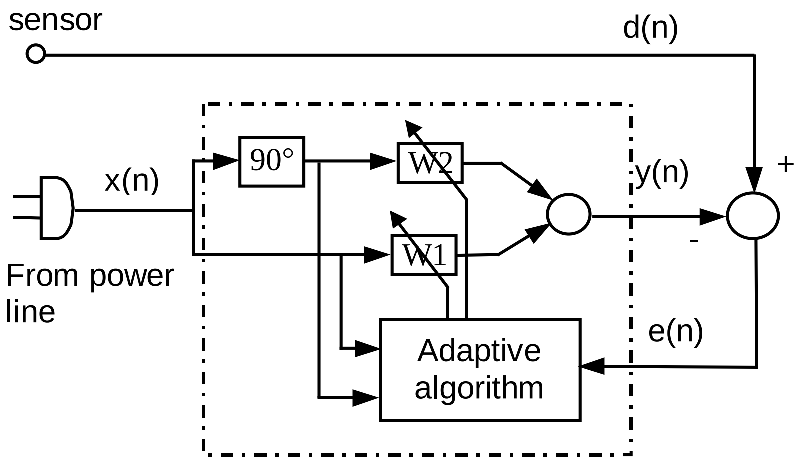

Figure 2.18: Block diagram of the power-line adaptive interference canceler.

```
% File = powerline.m
% This script simulates a power-line interference canceler.
iter = 1000;
Fs   = 200;                       % sampling frequency
t    = (1:iter) ./ Fs;            % sampled time
f    = 50 + (2*triang(iter)-1);   % interference freq, 50 +/- 1 Hz
```

```
x1   = 0.2 * cos(2*pi*f.*t');       % interference
x2   = 0.2 * cos(2*pi*f.*t'-pi/2); % 90 deg phase shifted

% Load corrupted signal
load(strcat(asptdir,'\data\powerline.mat'));
sc = s;
[w,x,d,y,e] = init_lms(2);
for n=1:iter
   [w,y,e] = asptlms([x1(n) ; x2(n)],w, s(n), 0.2);
   sc(n)  = e;
end
```

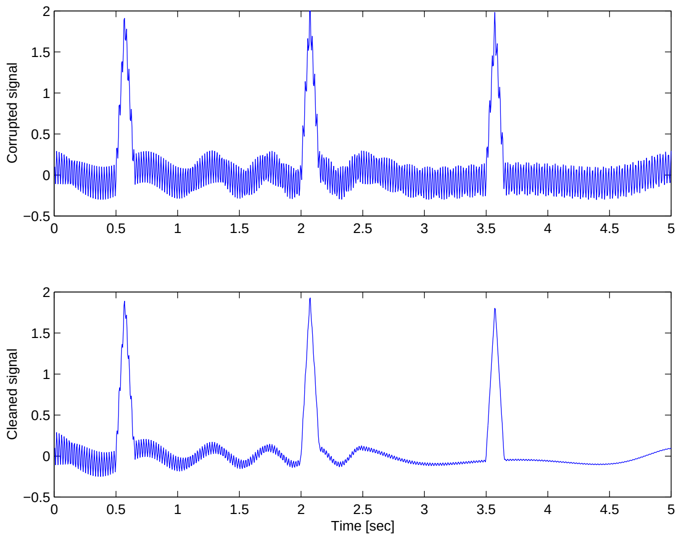

Figure 2.19: Input and output signals of an adaptive interference canceler.

Chapter 3

ASPT Quick Reference Guide

This Chapter summarizes the functions and application scripts included in the current release of the Adaptive Signal Processing Toolbox. Functions are grouped by filter structure. Section 3.1 summarizes the transversal adaptive algorithms. Section 3.2 summarizes the lattice adaptive algorithms. Section 3.3 summarizes the recursive adaptive algorithms. Section 3.4 summarizes the active noise and vibration control adaptive algorithms. ASPT also includes several non-adaptive filtering functions, plotting functions, and help functions that are used to manage the iteration progress window. Those functions are summarized in Section 3.6. Finally, a list of the scripts simulating adaptive filters applications are given in Section 3.7.

3.1 Summary of Transversal adaptive algorithms

Functions implementing transversal adaptive filters algorithms are summarized in Table 3.1. The initialization functions for those algorithms are summarized in Table 3.2. Each of the two tables includes a short description of each function and a pointer to the function reference section.

Function Name	Reference	Short Description
asptarlmsnewt	4.1	AR-modeling implementation of LMS-Newton method.
asptbfdaf	4.2	Block Frequency Domain Adaptive Filter (BFDAF) algorithm.
asptblms	4.3	Block Least Mean Squares
asptbnlms	4.4	Block Normalized Least Mean Squares
asptdrlms	4.5	Data Reusing Least Mean Squares
asptdrnlms	4.6	Data Reusing Normalized Least Mean Squares
asptleakylms	4.7	Leaky Normalized LMS algorithm.
asptlclms	4.8	Linearly Constrained LMS (LCLMS) algorithm.
asptlms	4.9	Least Mean Squares (LMS) and several of its variants.
asptmvsslms	4.10	Modified Variable Step Size LMS algorithm.
asptnlms	4.11	Normalized LMS (NLMS) algorithm.
asptpbfdaf	4.12	Partitioned Block Frequency Domain (PBFDAF) algorithm.
asptrcpbfdaf	4.13	Reduced Complexity PBFDAF (RCPBFDAF) algorithm.
asptrdrlms	4.14	Recent Data Reusing Least Mean Squares
asptrdrnlms	4.15	Recent Data Reusing Normalized Least Mean Squares
asptrls	4.16	Recursive Least Squares (RLS) algorithm.
aspttdftaf	4.17	Transform domain Fault Tolerant Adaptive Filter.
aspttdlms	4.18	Transform domain LMS algorithm.
asptvffrls	4.19	Variable Forgetting Factor RLS (VFFRLS).
asptvsslms	4.20	Variable Step Size LMS (VSSLMS) algorithm.

Table 3.1: Functions implementing transversal adaptive algorithms.

Function Name	Reference	Short Description
init_ arlmsnewt	4.21	Initialize AR modeling implementation of LMS-Newton method.
init_ bfdaf	4.22	Initialize Block Frequency Domain Adaptive Filter.
init_ blms	4.23	Initialize Block Least Mean Squares
init_ bnlms	4.24	Initialize Block Normalized Least Mean Squares
init_ drlms	4.25	Initialize Data Reusing Least Mean Squares
init_ drnlms	4.26	Initialize Data Reusing Normalized Least Mean Squares
init_ leakynlms	4.27	Initialize Leaky Normalized Least Mean Squares.
init_ lclms	4.28	Initialize Linearly Constrained LMS.
init_ lms	4.29	Initialize Least Mean Squares (LMS).
init_ mvsslms	4.30	Initialize Modified Variable Step Size LMS algorithm.
init_ nlms	4.31	Initialize Normalized LMS.
init_ pbfdaf	4.32	Initialize Partitioned Block Frequency Domain.
init_ rcpbfdaf	4.33	Initialize Reduced Complexity Partitioned BFDAF.
init_ rdrlms	4.34	Initialize Recent Data Reusing Least Mean Squares
init_ rdrnlms	4.35	Initialize Recent Data Reusing Normalized Least Mean Squares
init_ rls	4.36	Initialize Recursive Least Squares.
init_ tdftaf	4.37	Initialize Transform Domain Fault Tolerant Adaptive Filter.
init_ tdlms	4.38	Initialize Transform domain LMS.
init_ vffrls	4.39	Initialize Variable Forgetting Factor RLS (VFFRLS).
init_ vsslms	4.40	Initialize Variable Step Size LMS.

Table 3.2: Functions for creating and initializing the transversal adaptive filters.

3.2 Summary of Lattice Adaptive Algorithms

Functions implementing lattice adaptive filters algorithms are summarized in Table 3.3. The initialization functions for those algorithms are summarized in Table 3.4. Each of the two tables includes a short description of each function and a pointer to the function reference section.

Function Name	Reference	Short Description
asptftrls	5.1	Fast Transversal RLS algorithm.
asptlbpef	5.2	Lattice Backward Prediction Error Filter.
asptlfpef	5.3	Lattice Forward Prediction Error Filter.
asptlmslattice	5.4	LMS-Lattice Joint Process Estimator.
asptrlslattice	5.5	RLS-Lattice joint process estimator using a posteriori estimation errors.
asptrlslattice2	5.6	RLS-Lattice joint process estimator using a priori estimation errors with error feedback.
asptrlslbpef	5.7	Lattice Backward Prediction Error Filter.
asptrlslfpef	5.8	Lattice Forward Prediction Error Filter.

Table 3.3: Functions implementing lattice adaptive algorithms.

Function Name	Reference	Short Description
init_ ftrls	5.9	Initialize Fast Transversal RLS.
init_ lbpef	5.10	Initialize Lattice Backward Prediction Error Filter.
init_ lfpef	5.11	Initialize Lattice Forward Prediction Error Filter.
init_ lmslattice	5.12	Initialize LMS Lattice adaptive filter.
init_ rlslattice	5.13	Initialize RLS-Lattice joint process estimator using a posteriori estimation errors.
init_ rlslattice2	5.14	Initialize RLS-Lattice joint process estimator using a priori estimation errors with error feedback.
init_ rlslbpef	5.15	Initialize Recursive Least Squares Lattice Backward PEF.
init_ rlslfpef	5.16	Initialize Recursive Least Squares Lattice Forward PEF.

Table 3.4: Functions for creating and initializing lattice adaptive algorithms.

3.3 Summary of Recursive Adaptive Algorithms

Functions implementing recursive adaptive filters algorithms are summarized in Table 3.5. The initialization functions for those algorithms are summarized in Table 3.6. Each of the two tables includes a short description of each function and a pointer to the function reference section.

Function Name	Reference	Short Description
asptcsoiir2	6.1	Cascaded Second Order type-2 IIR adaptive filter.
aspteqerr	6.2	Equation Error IIR adaptive algorithm.
asptouterr	6.3	Output Error IIR adaptive algorithm.
asptsharf	6.4	Simple Hyperstable Adaptive Recursive Filter (SHARF).
asptsoiir1	6.5	Second Order IIR adaptive algorithm type-1.
asptsoiir2	6.6	Second Order IIR adaptive algorithm type-2.

Table 3.5: Functions implementing recursive adaptive algorithms.

Function Name	Reference	Short Description
init_ csoiir2	6.7	Initialize Cascaded Second Order IIR adaptive filter.
init_ eqerr	6.8	Initialize Equation Error IIR adaptive filter.
init_ outerr	6.9	Initialize Output Error IIR.
init_ sharf	6.10	Initialize Simple Hyperstable Adaptive Recursive Filter.
init_ soiir1	6.11	Initialize Second Order IIR adaptive algorithm type-1.
init_ soiir2	6.12	Initialize Second Order IIR adaptive algorithm type-2.

Table 3.6: Functions for creating and initializing recursive adaptive algorithms.

3.4 Summary of Active Noise and Vibration Control Algorithms

Functions implementing active noise and vibration control filters are summarized in Table 3.7. The initialization functions for those algorithms are summarized in Table 3.8. Each of the two tables includes a short description of each function and a pointer to the function reference section.

Function Name	Reference	Short Description
asptadjlms	7.1	Adjoint-LMS algorithm.
asptfdadjlms	7.2	Frequency Domain Adjoint LMS algorithm.
asptfdfxlms	7.3	Frequency Domain Filtered-x LMS algorithm.
asptfxlms	7.4	Filtered-x LMS algorithm.
asptmcadjlms	7.5	Multichannel Adjoint-LMS algorithms.
asptmcfdadjlms	7.6	Multichannel Frequency Domain Adjoint LMS algorithm.
asptmcfdfxlms	7.7	Multichannel Frequency Domain Filtered-x LMS algorithm.
asptmcfxlms	7.8	Multichannel Filtered-x LMS algorithm.

Table 3.7: Functions implementing active noise and vibration control filters.

Function Name	Reference	Short Description
init_ adjlms	7.9	Initialize Adjoint LMS.
init_ fdadjlms	7.10	Initialize Frequency Domain Adjoint LMS.
init_ fdfxlms	7.11	Initialize Frequency Domain Filtered-x LMS.
init_ fxlms	7.12	Initialize Filtered-x LMS.
init_ mcadjlms	7.13	Initialize Multichannel Adjoint LMS.
init_ mcfdadjlms	7.14	Initialize Multichannel Frequency Domain Adjoint LMS.
init_ mcfdfxlms	7.15	Initialize Multichannel Frequency Domain Filtered-x LMS.
init_ mcfxlms	7.16	Initialize Multichannel Filtered-x LMS.

Table 3.8: Functions for creating and initializing active noise and vibration control filters.

3.5 Summary of Nonlinear Adaptive Algorithms

Functions implementing nonlinear adaptive filters are summarized in Table 3.9. The initialization functions for those algorithms are summarized in Table 3.10. Each of the two tables includes a short description of each function and a pointer to the function reference section.

Function Name	Reference	Short Description
asptsovlms	8.1	Second Order Volterra LMS and several of its variants.
asptsovnlms	8.2	Second Order Volterra Normalized LMS algorithm.
asptsovrls	8.3	Second Order Volterra RLS algorithm.
asptsovtdlms	8.4	Second Order Volterra Transform domain LMS algorithm.
asptsovvsslms	8.5	Second Order Volterra Variable Step Size LMS algorithm.

Table 3.9: Functions implementing nonlinear adaptive filters.

Function Name	Reference	Short Description
init_ sovlms	8.6	Initialize Second Order Volterra LMS.
init_ sovnlms	8.7	Initialize Second Order Volterra NLMS.
init_ sovrls	8.8	Initialize Second Order Volterra RLS.
init_ sovtdlms	8.9	Initialize Second Order Volterra Transform domain LMS.
init_ sovvsslms	8.10	Initialize Second Order Volterra Variable Step Size LMS.

Table 3.10: Functions for creating and initializing nonlinear adaptive filters.

3.6 Summary of Non-adaptive, Visualization and Help Routines

Table 3.11 lists the visualization and help functions included in the current release of the Adaptive Signal Processing Toolbox, with a short description of each function, and a pointer to the function reference section.

Function Name	Reference	Short Description
init_ ipwin	9.1	Initializes iteration progress GUI window.
ipwin		Builds the iteration progress GUI window.
getStop		Returns the condition of the stop button in the IPWIN.
guifb		Handles the GUI feedback functions of the IPWIN.
mcmixr	9.2	Calculates the response of N speakers at M microphones.
osfilter	9.3	Fast FIR filter using overlap-save.
plot_ ale	9.4	Generates plots for the Adaptive Line Enhancer problems.
plot_ anvc	9.5	Generates plots for Active Noise and Vibration Control.
plot_ beam	9.6	Generates plots for beam forming problems.
plot_ echo	9.7	Generates plots for echo canceling applications.
plot_ invmodel	9.8	Generates plots for inverse modeling problems.
plot_ model	9.9	Generates plots for modeling problems.
plot_ predict	9.10	Generates plots for linear prediction problems.
sovfilt	9.11	Second Order Volterra filter.
update_ ipwin	9.12	Updates the iteration progress GUI window.

Table 3.11: Non-adaptive, visualization, and help functions.

3.7 Summary of adaptive applications

Table 3.12 lists the simulation scripts of adaptive applications included in the current release of the Adaptive Signal Processing Toolbox, with a short description of each script, and a pointer to the application reference section.

Script Name	Reference	Short Description
ale_ csoiir2	10.1	Adaptive Line Enhancer using CSOIIR2.
ale_ soiir1	10.2	Adaptive Line Enhancer using SOIIR1.
ale_ soiir2	10.3	Adaptive Line Enhancer using SOIIR2.
anvc_ adjlms	10.4	Active noise and vibration control using ADJLMS.
anvc_ fdadjlms	10.5	Active noise and vibration control using FDADJLMS.
anvc_ fdfxlms	10.6	Active noise and vibration control using FDFXLMS.
anvc_ fxlms	10.7	Active noise and vibration control using FXLMS.
anvc_ mcadjlms	10.8	Active noise and vibration control using MCADJLMS.
anvc_ mcfdadjlms	10.9	Active noise and vibration control using MCFDADJLMS.
anvc_ mcfdfxlms	10.10	Active noise and vibration control using MCFDFXLMS.
anvc_ mcfxlms	10.11	Active noise and vibration control using MCFXLMS.
beambb_ lclms	10.12	Beam former at base-band frequency using LCLMS.
beamrf_ lms	10.13	Beam former at RF frequency using LMS.
echo_ bfdaf	10.14	Echo canceler using BFDAF.
echo_ leakynlms	10.15	Echo canceler using LEAKYNLMS.
echo_ nlms	10.16	Echo canceler using NLMS.
echo_ pbfdaf	10.17	Echo canceler using PBFDAF.
echo_ rcpbfdaf	10.18	Echo canceler using RCPBFDAF.
equalizer_ nlms	10.19	Inverse modeling using NLMS.
equalizer_ rls	10.20	Inverse modeling using RLS.
model_ arlmsnewt	10.21	Modeling using LMS-NEWTON.
model_ eqerr	10.22	IIR modeling using EQERR.
model_ lmslattice	10.23	Modeling using LMSLATTICE.
model_ mvsslms	10.24	FIR modeling using MVSSLMS.
model_ outerr	10.25	IIR modeling using OUTERR.
model_ rlslattice	10.26	Modeling using RLSLATTICE.
model_ sharf	10.27	IIR modeling using SHARF.
model_ tdlms	10.28	FIR modeling using TDLMS.
model_ vsslms	10.29	FIR modeling using VSSLMS.
predict_ lbpef	10.30	Prediction using LBPEF.
predict_ lfpef	10.31	Prediction using LFPEF.
predict_ rlslbpef	10.32	Prediction using RLSLBPEF.
predict_ rlslfpef	10.33	Prediction using RLSLFPEF.

Table 3.12: Adaptive filters applications.

Chapter 4

Transversal and Linear Combiner Adaptive Algorithms

This chapter documents the functions used to create, initialize, and update the coefficients of transversal (Section 2.2.1) and linear combiner (Section 2.2.2) adaptive filters. Table 4.1 summarizes the transversal adaptive functions currently supported and gives a short description and a pointer to the reference page of each function.

Each function is documented in a separate section including the following information related to the function:

- **Purpose:** Short description of the algorithm implemented by this function.
- **Syntax:** Shows the function calling syntax. If the function has optional parameters, this section will have two calling syntaxes. One with only the required formal parameters and one with all the formal parameters.
- **Description:** Detailed description of the function usage with explanation of its input and output parameters.
- **Example:** A short example showing typical use of the function. The examples listed can be found in the ASPT/test directory of the ASPT distribution. The user is encouraged to copy from those examples and paste in her own applications.
- **Algorithm:** A short description of the operations internally performed by the function.
- **Remarks:** Gives more theoretical and practical remarks related to the usage, performance, limitations, and applications of the function.
- **Resources:** Gives a summary of the memory requirements and number of multiplications, addition/subtractions, and division operations required to implement the function in real time. This can be used to roughly calculate the MIPS (Million Instruction Per Second) required for a specific platform knowing the number of instructions the processor needs to perform each operation.
- **See Also:** Lists other functions that are related to this function.
- **Reference:** Lists literature for more information on the function.

Function Name	Reference	Short Description
asptarlmsnewt	4.1	AR-modeling implementation of LMS-Newton method.
asptbfdaf	4.2	Block Frequency Domain Adaptive Filter (BFDAF) algorithm.
asptblms	4.3	Block Least Mean Squares
asptbnlms	4.4	Block Normalized Least Mean Squares
asptdrlms	4.5	Data Reusing Least Mean Squares
asptdrnlms	4.6	Data Reusing Normalized Least Mean Squares
asptleakylms	4.7	Leaky Normalized LMS algorithm.
asptlclms	4.8	Linearly Constrained LMS (LCLMS) algorithm.
asptlms	4.9	Least Mean Squares (LMS) and several of its variants.
asptmvsslms	4.10	Modified Variable Step Size LMS algorithm.
asptnlms	4.11	Normalized LMS (NLMS) algorithm.
asptpbfdaf	4.12	Partitioned Block Frequency Domain (PBFDAF) algorithm.
asptrcpbfdaf	4.13	Reduced Complexity PBFDAF (RCPBFDAF) algorithm.
asptrdrlms	4.14	Recent Data Reusing Least Mean Squares
asptrdrnlms	4.15	Recent Data Reusing Normalized Least Mean Squares
asptrls	4.16	Recursive Least Squares (RLS) algorithm.
aspttdftaf	4.17	Transform domain Fault Tolerant Adaptive Filter.
aspttdlms	4.18	Transform domain LMS algorithm.
asptvffrls	4.19	Variable Forgetting Factor RLS (VFFRLS).
asptvsslms	4.20	Variable Step Size LMS (VSSLMS) algorithm.
init_ arlmsnewt	4.21	Initialize AR modeling implementation of LMS-Newton method.
init_ bfdaf	4.22	Initialize Block Frequency Domain Adaptive Filter.
init_ blms	4.23	Initialize Block Least Mean Squares
init_ bnlms	4.24	Initialize Block Normalized Least Mean Squares
init_ drlms	4.25	Initialize Data Reusing Least Mean Squares
init_ drnlms	4.26	Initialize Data Reusing Normalized Least Mean Squares
init_ leakynlms	4.27	Initialize Leaky Normalized Least Mean Squares.
init_ lclms	4.28	Initialize Linearly Constrained LMS.
init_ lms	4.29	Initialize Least Mean Squares (LMS).
init_ mvsslms	4.30	Initialize Modified Variable Step Size LMS algorithm.
init_ nlms	4.31	Initialize Normalized LMS.
init_ pbfdaf	4.32	Initialize Partitioned Block Frequency Domain.
init_ rcpbfdaf	4.33	Initialize Reduced Complexity Partitioned BFDAF.
init_ rdrlms	4.34	Initialize Recent Data Reusing Least Mean Squares
init_ rdrnlms	4.35	Initialize Recent Data Reusing Normalized Least Mean Squares
init_ rls	4.36	Initialize Recursive Least Squares.
init_ tdftaf	4.37	Initialize Transform Domain Fault Tolerant Adaptive Filter.
init_ tdlms	4.38	Initialize Transform domain LMS.
init_ vffrls	4.39	Initialize Variable Forgetting Factor RLS (VFFRLS).
init_ vsslms	4.40	Initialize Variable Step Size LMS.

Table 4.1: List of functions for creating, initializing, and updating transversal and linear combiner adaptive filters.

4.1 asptarlmsnewt

Purpose Efficient implementation of the LMS-Newton algorithm using autoregressive modeling.

Syntax `[k,w,b,u,P,y,e]=asptarlmsnewt(k,w,x,b,u,P,d,mu_p,mu_w,maxk)`

Description The LMS-Newton is a stochastic implementation of the Newton search method which solves the eigenvalue spread problem in adaptive filters with colored input signals. The update equation for the LMS-Newton is given by (see Fig. 2.5 and Fig. 2.6)

$$\underline{\mathbf{w}}(n+1) = \underline{\mathbf{w}}(n) + 2\mu\, e(n)\mathbf{R}^{-1}\, \underline{\mathbf{x}}(n), \tag{4.1}$$

where $\mathbf{R}$ is the autocorrelation matrix of the adaptive filter input signal $x(n)$. Direct implementation of the LMS-Newton update (4.1) requires estimation and inversion of $\mathbf{R}$ and the matrix vector multiplication $\mathbf{R}^{-1}\,\underline{\mathbf{x}}(n)$ each sample, which is of course very computational demanding. `asptarlmsnewt()` implements the LMS-Newton method efficiently by recursively estimating the term $u = \mathbf{R}^{-1}\,\underline{\mathbf{x}}(n)$ using autoregressive modeling. A lattice predictor of M stages is used for the autoregressive modeling part. When the input signal can be modeled with an autoregressive model of length M much less than the adaptive filter length L, a significant computational saving is obtained.
The input and output parameters of `asptarlmsnewt()` of M lattice predictor stages and L transversal filter coefficients are summarized below.

```
Input Parameters [Size]::
    k   : vector of lattice predictor coefficients [Mx1]
    w   : vector of linear combiner coefficients [Lx1]
    x   : vector of input samples [Lx1]
    b   : vector of backward prediction error [Lx1]
    u   : u = R^(-1)*x calculated recursively [Lx1]
    P   : vector of last estimated power of b [M+1x1]
    d   : desired response
    mu_p: adaptation constant for the predictor coefficients
    mu_w: adaptation constant for the combiner coefficient
    maxk: maximum allowed value of abs(k)
Output parameters::
    k   : updated lattice predictor coefficients
    w   : updated linear combiner coefficients
    b   : updated backward prediction error
    u   : updated {R^(-1)*x}
    P   : updated  power estimate of b
    y   : linear combiner output
    e   : error signal [e = d - y]
```

Example

```
% ARLMSNEWT used in a simple system identification application.
% By the end of this script the adaptive filter w
% should have the same coefficients as the unknown filter h.
iter = 5000;                      % samples to process
% Complex unknown impulse response
h  = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xn = 2*(rand(iter,1)-0.5);    % Input signal
% although xn is real, dn will be complex since h is complex
dn = osfilter(h,xn);            % Unknown filter output
en = zeros(iter,1);             % error signal
% Initialize ARLMSNEWT with M=2, L=10
M    = 2;                       % AR model length
L    = 10;                      % filter length
mu_w = .01;                     % linear combiner step size
mu_p = 0.001;                   % lattice predictor step size
[k,w,x,b,u,P,d,y,e]=init_arlmsnewt(L,M);

%% Processing Loop
for (m=1:iter)
   x = [xn(m,:);x(1:end-1,:) ]; % update the delay line
   d = dn(m,:) + 1e-3*rand;     % additive noise var = 1e-6
   [k,w,b,u,P,y,e]=asptarlmsnewt(k,w,x,b,u,P,d,mu_p,mu_w,0.99);
   % save the last error sample to plot later
   en(m,:) = e;
end;
% display the results
subplot(2,2,1);stem([real(w) imag(conj(w))]); grid;
subplot(2,2,2);
eb = filter(0.1,[1 -.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 4.1. The left side panel of this figure shows the adaptive filter coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB).

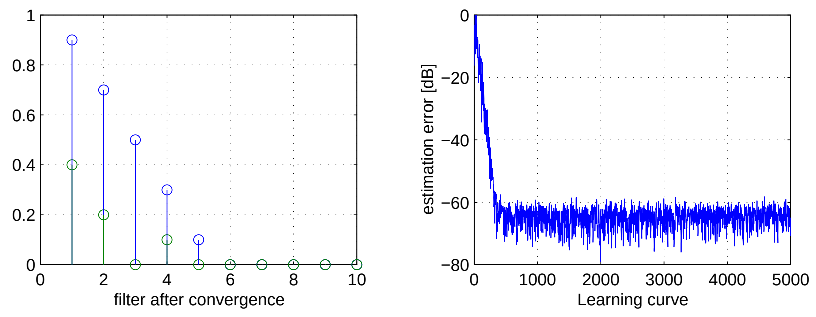

Figure 4.1: The adaptive filter coefficients after convergence and the learning curve for the complex FIR system identification problem using the ARLMSNEWT algorithm.

Algorithm `asptarlmsnewt()` performs the following operations every sample

- Calculates the forward and backward prediction errors for the lattice predictor,
- Calculates the power estimate of the backward prediction errors,
- Updates the PARCOR coefficients of the lattice predictor,
- Updates the estimate $u = \mathbf{R}^{-1}\ \underline{\mathbf{x}}(n)$,
- Evaluates the adaptive transversal filter output,
- Evaluates the error signal,
- Updates the adaptive transversal filter coefficients using the relationship $\underline{\mathbf{w}}(n+1) = \underline{\mathbf{w}}(n) + 2\mu\ e(n)\underline{\mathbf{u}}(n)$.

Resources The resources required to implement the ARLMSNEWT with a filter of length L and predictor of length M in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$4L + 0.5M^2 + 4.5M + 6$
MULTIPLY	$3L + 2.5M^2 + 9.5M + 5$
ADD	$2L + 2.5M^2 + 4.5M + 2$
DIVIDE	L + M

See Also INIT_ ARLMSNEWT, MODEL_ ARLMSNEWT.

Reference [2] and [4] for analysis of the adaptive Lattice filters, [2] and [11] for analysis of the LMS-Newton algorithm.

4.2 asptbfdaf

Purpose Block filtering and coefficient update in frequency domain using the Block Frequency Domain Adaptive Filter (BFDAF) algorithm.

Syntax `[W,x,y,e,Px,w]=asptbfdaf(M,x,xn,dn,W,mu,n,c,b,Px)`

Description `asptbfdaf()` is an efficient frequency domain implementation of the block NLMS algorithm. `asptbfdaf()` performs filtering and coefficient update in frequency domain using the overlap-save method, and therefore provide efficient implementation for long adaptive filters usually used in applications such as acoustic echo cancelers where the adaptive filter can be as long as a few thousand coefficients. `asptbfdaf()` is a block processing algorithm which is called every L samples. Every call processes L input and L desired samples (L is the block length), to produce L filter output samples and L error samples, besides updating all filter coefficients in the frequency domain. Fig. 4.2 shows the parameters of `asptbfdaf()` which are summarized below.

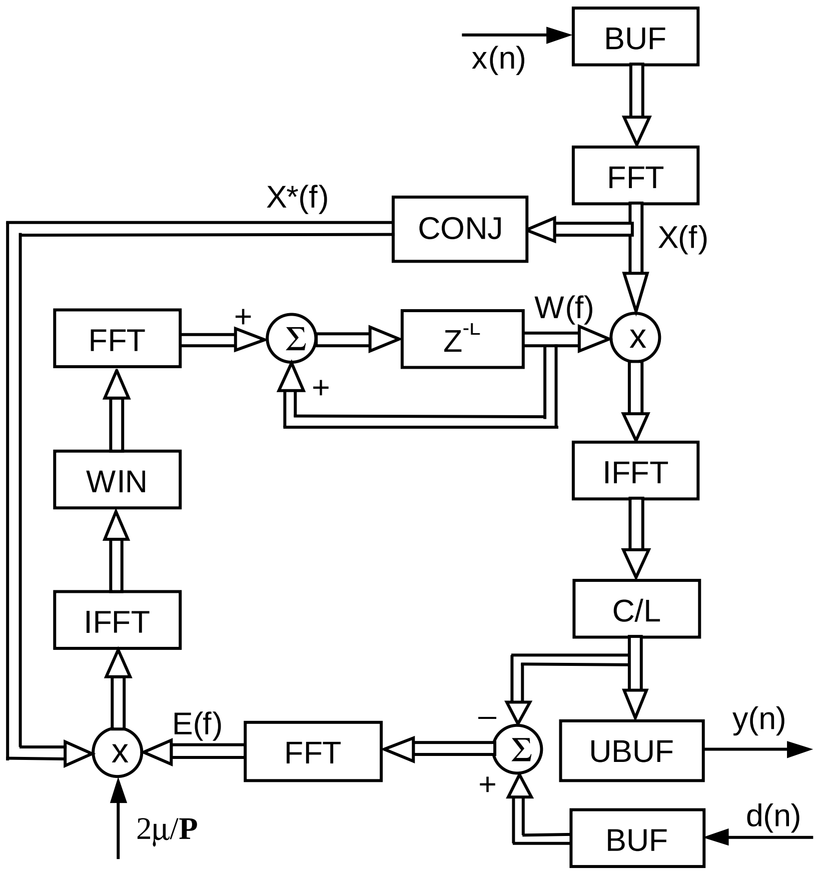

Figure 4.2: Block diagram of the Block Frequency Domain Adaptive Filter.

```
Input Parameters [Size]::
   M  : filter length [1 x 1]
   x  : previous overlap-save input vector [B x 1]
   xn : new input block [L x 1]
   dn : new desired block [L x 1]
   W  : F-domain filter coefficients vector [B x 1]
   mu : adaptation constant (step size) [1 x 1]
   n  : if not 0, normalization is performed
   c  : if not 0, constrains filter to length M
   b  : forgetting factor for estimation of Px
   Px : previous estimate of the power of x [B x 1]
Output parameters::
   W  : updated F-domain filter coefficients
   x  : updated overlap save input vector
   y  : filter output at block n (t-domain)
   e  : error vector (t-domain)
   Px : updated estimate of the power of X
```

Example

```
iter = 5000;        % Number of samples to process
% Complex unknown impulse response
h    = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];

xt   = 2*(rand(iter,1)-0.5);   % Input signal
% although xn is real, dn will be complex
dt   = osfilter(h,xt);         % Unknown filter output
en   = zeros(iter,1);          % estimation error

% Initialize BFDAF with a filter of 5 coef.
M = 5; L = 3;
[W,x,dn,e,y,Px,w]=init_bfdaf(L,M);

%% Processing Loop
for (m=1:L:iter-L)
   xn = xt(m:m+L-1,:);             % input block
   dn = dt(m:m+L-1,:)+ 1e-3*rand;  % desired block

   %  call BFDAF to calculate the filter output,
   %  estimation error and update the filter coef.
   [W,x,y,e,Px,w]=asptbfdaf(M,x,xn,dn,W,0.05,1,1,0.98,Px);
   % save the last error block to plot later
   en(m:m+L-1,:) = e;
end;

% display the results
subplot(2,2,1);stem([real(w) imag((w))]); grid;
subplot(2,2,2);
eb = filter(.1,[1 -.9], en(1:m) .* conj(en(1:m)));
plot(10*log10(eb ));grid
```

Running the above script will produce the graph shown in Fig. 4.3. The left side graph of the figure shows the adaptive filter coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the mean square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB).

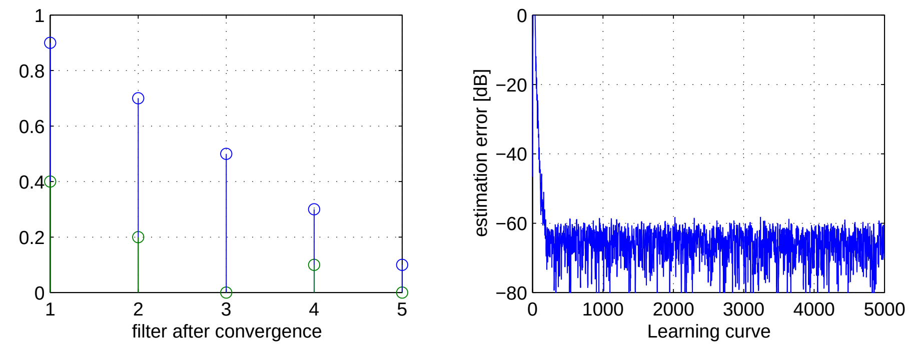

Figure 4.3: The adaptive filter coefficients after convergence and the learning curve for the complex FIR system identification problem using the BFDAF algorithm.

Algorithm `asptbfdaf()` performs the following operations (see Fig. 4.2).

- buffers L input samples, composes an overlap-save input vector $\underline{\mathbf{x}}(n)$ and computes its FFT, $\underline{\mathbf{X}}(f)$
- element-wise multiplies $\underline{\mathbf{X}}(f)$ by the adaptive filter coefficients vector $\underline{\mathbf{W}}(f)$ (circular convolution in time domain). The result is converted to time domain using IFFT and the linear convolution samples are extracted to produce the filter-output vector $\underline{\mathbf{y}}(n)$
- buffers L desired samples and evaluates the current error block $\underline{\mathbf{e}}(n) = \underline{\mathbf{d}}(n) - \underline{\mathbf{y}}(n)$. The error vector is padded with zeros and transformed to frequency domain giving $\underline{\mathbf{E}}(f)$
- estimates the input signal power at each frequency bin and normalizes the step size at each bin.
- evaluates the cross-correlation between $\underline{\mathbf{X}}(f)$ and $\underline{\mathbf{E}}(f)$ to produce the block gradient vector. This vector is used to update the frequency domain filter coefficients.
- constrains the filter if required. This is performed by first taking the IFFT of $\underline{\mathbf{W}}(f)$, applying a rectangular window on the time domain coefficients, and taking the FFT of the windowed coefficients.

Remarks

- Supports both real and complex signals.
- `asptbfdaf()` constrains the filter coefficients $\underline{\mathbf{W}}(f)$ rather than the gradient vector as in the official BFDAF algorithms since this proved to result in a more stable update.
- The unconstrained BFDAF (c = 0) saves two FFT operations on the cost of accuracy.
- The time domain filter coefficients $\underline{\mathbf{w}}(n)$ will be calculated and returned by `asptbfdaf()` only if the output variable w is given.
- The convergence properties of the BFDAF algorithm are superior to time domain algorithms since normalization is performed at each frequency bin which eliminates the eigenvalue spread problem.
- BFDAF introduces a processing delay between its input $x(n)$ and output $y(n)$ equals to the block length L, since the algorithm has to collect L samples before processing a block.
- Very efficient for long adaptive filters since convolution and correlation are performed in frequency domain. Maximum efficiency is obtained when the block length is chosen to be equal to the filter length $L = M$.
- Choosing L for maximum processing efficiency might result in a long delay for long filters. This can be solved either by reducing L on the cost of computational efficiency or using the Partitioned BFDAF (`asptpbfdaf()`) instead.

Resources The resources required for direct implement of the BFDAF algorithm in real time is given in the table below. The computations given are those required to process L samples using the constrained BFDAF. Unconstrained BFDAF uses two FFT operations less than the constrained BFDAF. In the table below $C(FFT_B)$ is used to indicate the number of operations required to implement an FFT or IFFT of length $B = 2^{nextpow2(M+L-1)}$

MEMORY	$6B + 3L + 3$
MULTIPLY	$6B + 5 * C(FFT_B)$
ADD	$3B + L + 5 * C(FFT_B)$
DIVIDE	$B + 5 * C(FFT_B)$

See Also INIT_ BFDAF, ECHO_ BFDAF, ASPTPBFDAF, ASPTRCPBFDAF.

Reference [3], Chapter 3 for detailed description of BFDAF, [8] for the overlap-save method, and [9] for frequency domain adaptive filters.

4.3 asptblms

Purpose Performs filtering and coefficient update using the Block Least Mean Squares (BLMS) algorithm. BLMS updates the N filter coefficients once every block of L samples.

Syntax

```
[w,x,y,e] = asptblms(x,xn,dn,w,mu)
[w,x,y,e] = asptblms(x,xn,dn,w,mu,alg)
```

Description Unlike `asptlms()` which updates all the filter coefficients every sample, `asptblms()` updates the coefficients every L samples. `asptblms()` takes a block of input samples $xn(k)$, a block of desired samples $dn(k)$, the vector of adaptive filter coefficients from the previous iteration $\underline{\mathbf{w}}(k-1)$, the step size μ, and returns a block of filter output samples $\underline{\mathbf{y}}(k)$, a block of error samples $\underline{\mathbf{e}}(k)$ and the updated vector of filter coefficients $\underline{\mathbf{w}}(k)$. The update equation is given by

$$\underline{\mathbf{w}}(k+1) = \underline{\mathbf{w}}(k) + \frac{\mu_B}{L}\sum_{i=0}^{L-1} e(kL+i)\underline{\mathbf{x}}(kL+i), \qquad (4.2)$$

where L is the block length, k is the block index, and μ_B is the algorithm step-size parameter. Coefficients update is performed according to the 'alg' input argument which can take any of the following values.

- `'lms'` : the default value, uses the LMS algorithm
- `'slms'` : uses the sign LMS algorithm, the sign of the error $e(k)$ is used in the update equation instead of the error.
- `'srlms'` : uses the signed regressor LMS algorithm, the sign of the input signal $x(k)$ is used in the update equation instead of the input signal.
- `'sslms'` : uses the sign-sign-LMS algorithm, the sign of the error $e(k)$ and the sign of the input signal $x(k)$ are used in the update equation instead of the error and the input signals.

The input and output parameters of `asptblms()` for an FIR adaptive filter of N coefficients are summarized below.

```
Input Parameters [Size]::
 x   : previous input delay line [N x 1]
 xn  : new block of input samples [L x 1]
 dn  : new block of desired samples [L x 1]
 w   : vector of filter coefficients [N x 1]
 mu  : adaptation constant (step size) [1 x 1]
 alg : specifies the variety of the lms to use in the
       update equation. Must be one of the following:
       'lms'     [default]
       'slms'  - sign LMS, uses sign(e)
       'srlms' - signed regressor LMS, uses sign(x)
       'sslms' - sign-sign LMS, uses sign(e) and sign(x)
Output parameters::
 w   : updated filter coefficients
 x   : updated delay-line of input signal
 y   : block of filter output samples
 e   : block of error samples.
```

Example

```
% BLMS used in a simple system identification application.
% By the end of this script the adaptive filter w
% should have the same coefficients as the unknown filter h.
iter = 5000;                      % Number of samples to process
% Complex unknown impulse response
h    = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xt   = 2*(rand(iter,1)-0.5);  % Input signal, zero mean random.
% although xn is real, dn will be complex since h is complex
dt   = osfilter(h,xt);            % Unknown filter output
en   = zeros(iter,1);             % vector to collect the error
% Initialize BLMS with a filter of 5 coef. and block of 5 samples.
[w,x,dn,e,y]=init_blms(5,5);

%% Processing Loop
for (m=1:L:iter-L)
   xn = xt(m:m+L-1,:);
   dn = dt(m:m+L-1,:)+ 1e-3*rand;
   % call BLMS to calculate the filter output, estimation error
   % and update the coefficients.
   [w,x,y,e]=asptblms(x,xn,dn,w,0.05,'srlms');
   % save the last error block to plot later
   en(m:m+L-1,:) = e;
end;

% display the results
subplot(2,2,1);stem([real(w) -imag((w))]); grid;
subplot(2,2,2);eb = filter(.1, [1 -.9], en(1:m) .* conj(en(1:m)));
plot(10*log10(eb +eps ));grid
```

Running the above script will produce the graph shown in Fig. 4.4. The left side graph of the figure shows the adaptive filter coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB).

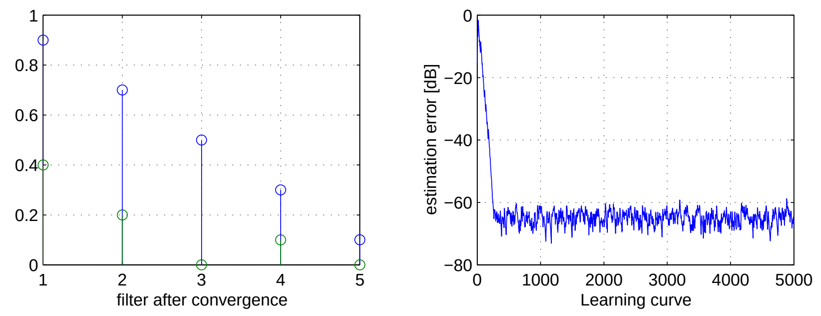

Figure 4.4: The adaptive filter coefficients after convergence and the learning curve for the complex FIR system identification problem using the BLMS algorithm.

Algorithm The current implementation of `asptblms()` performs the following operations

- Filters each of the new input samples through the adaptive filter $\underline{\mathbf{w}}(k-1)$ to produce the block of filter output samples $\underline{\mathbf{y}}(k)$.
- Calculates the block of error samples $\underline{\mathbf{e}}(n) = \underline{\mathbf{d}}(n) - \underline{\mathbf{y}}(n)$.
- Updates the adaptive filter coefficients once per block using the update equation (4.2).
- Updates the delay line for processing the next block.

Remarks The BLMS is a block implementation of the LMS algorithm, and therefore, BLMS has similar properties to those known for the LMS. The main difference between the BLMS and LMS is that the former updates all the filter coefficients once every L samples while the latter updates all the filter coefficients once every sample. The following remarks are also of interest.

- The convergence behaviors of the BLMS and LMS are identical.
- From equation 4.2, to obtain the same final misadjustment for LMS and BLMS, the following should hold $\mu_B/L = \mu$, where μ_B is the step size for BLMS and μ is the step size for the LMS algorithm. However, the current implementation of `asptblms()` assumes that the division by L in equation 4.2 has been absorbed in the algorithm step size.
- Like all block processing algorithms, the BLMS introduces a delay of L samples in the input-output path. This is the time required to collect a block of input data.
- `asptblms()` supports both real and complex data and filters. The adaptive filter for the complex BLMS algorithm converges to the complex conjugate of the optimum solution.
- `asptblms()` updates the input delay line internally.

Resources The resources required to implement the BLMS algorithm for a transversal adaptive FIR filter of N coefficients using a block of L samples in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$2N + 5L + 1$
MULTIPLY	$N(2L + 1)/L$
ADD	$N(2L + 1)/L$
DIVIDE	0

See Also INIT_ BLMS, ASPTBNLMS, ASPTBFDAF, ASPTLMS..

Reference [11] and [4] for extensive analysis of the LMS and the steepest-descent search method and [1] and [2] for BLMS.

4.4 asptbnlms

Purpose Performs filtering and coefficient update using the Block Normalized Least Mean Squares (BNLMS) algorithm. BNLMS updates the N filter coefficients once every block of L samples.

Syntax `[w,x,y,e,p] = asptbnlms(x,xn,dn,w,mu,p,b)`

Description Unlike `asptnlms()` which updates all the filter coefficients every sample, `asptbnlms()` updates the coefficients every L samples. `asptbnlms()` takes a block of input samples $xn(k)$, a block of desired samples $dn(k)$, the vector of adaptive filter coefficients from the previous iteration $\underline{\mathbf{w}}(k-1)$, the step size μ, the previous estimate of the input signal power p, and returns a block of filter output samples $\underline{\mathbf{y}}(k)$, a block of error samples $\underline{\mathbf{e}}(k)$, the updated power estimate, and the updated vector of filter coefficients $\underline{\mathbf{w}}(k)$. The update equation of `asptblms()` is given by

$$\underline{\mathbf{w}}(k+1) = \underline{\mathbf{w}}(k) + \frac{\mu_B}{Lp}\sum_{i=0}^{L-1} e(kL+i)\underline{\mathbf{x}}(kL+i), \tag{4.3}$$

where L is the block length, k is the block index, and μ_B is the algorithm step-size parameter.
The input and output parameters of `asptbnlms()` for an FIR adaptive filter of N coefficients are summarized below.

```
Input Parameters [Size]::
   x  : previous input delay line [N x 1]
   xn : new block of input samples [L x 1]
   dn : new block of desired samples [L x 1]
   w  : vector of filter coefficients [N x 1]
   mu : adaptation constant (step size) [1 x 1]
   p  : previous estimate of the input power [1 x 1]
   b  : pole of the IIR filter used to estimate the input signal power

Output parameters::
   w  : updated filter coefficients
   x  : updated delay line of input signal
   y  : block of filter output samples
   e  : block of error samples
   p  : updated estimate of the input signal power.
```

Example

```
% BNLMS used in a simple system identification application.
% By the end of this script the adaptive filter w
% should have the same coefficients as the unknown filter h.
iter = 5000;                    % Number of samples to process
% Complex unknown impulse response
h    = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xt   = 2*(rand(iter,1)-0.5);  % Input signal, zero mean random.
% although xn is real, dn will be complex since h is complex
dt   = osfilter(h,xt);          % Unknown filter output
en   = zeros(iter,1);           % vector to collect the error
% Initialize BNLMS with a filter of 5 coef. and block of 5 samples.
[w,x,dn,e,y,p]=init_bnlms(5,5);

%% Processing Loop
for (m=1:L:iter-L)
   xn = xt(m:m+L-1,:);
   dn = dt(m:m+L-1,:)+ 1e-3*rand;
   % call BNLMS to calculate the filter output, estimation error
   % and update the coefficients.
   [w,x,y,e,p] = asptbnlms(x,xn,dn,w,.05,p,.98);
   % save the last error block to plot later
   en(m:m+L-1,:) = e;
end;

% display the results
subplot(2,2,1);stem([real(w) -imag((w))]); grid;
subplot(2,2,2);eb = filter(.1, [1 -.9], en(1:m) .* conj(en(1:m)));
plot(10*log10(eb +eps ));grid
```

Running the above script will produce the graph shown in Fig. 4.5. The left side graph of the figure shows the adaptive filter coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB).

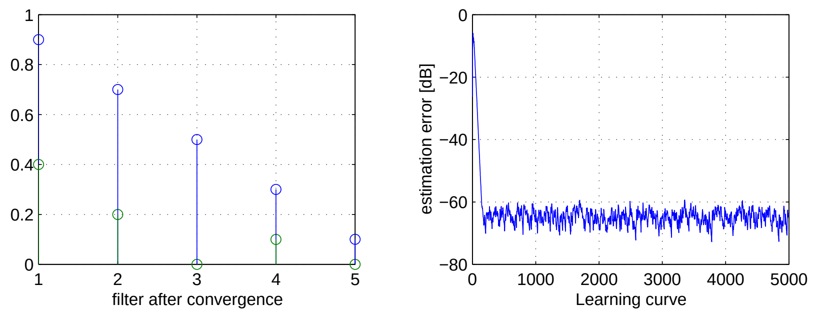

Figure 4.5: The adaptive filter coefficients after convergence and the learning curve for the complex FIR system identification problem using the BNLMS algorithm.

Algorithm The current implementation of `asptbnlms()` performs the following operations

- Filters each of the new input samples through the adaptive filter $\underline{\mathbf{w}}(k-1)$ to produce the block of filter output samples $\underline{\mathbf{y}}(k)$.
- Calculates the block of error samples $\underline{\mathbf{e}}(n) = \underline{\mathbf{d}}(n) - \underline{\mathbf{y}}(n)$.
- Updates the estimate of the input signal power.
- Updates the adaptive filter coefficients once per block using the update equation (4.3).
- Updates the delay line for processing the next block.

Remarks The BNLMS is a block implementation of the NLMS algorithm, and therefore, BNLMS has similar properties to those of the NLMS. The main difference between the BNLMS and NLMS is that the former updates all the filter coefficients once every L samples while the latter updates all the filter coefficients once every sample. The following remarks are also of interest.

- The convergence behaviors of the BNLMS and NLMS are identical.
- From equation 4.3, to obtain the same final misadjustment for NLMS and BNLMS, the following should hold, $\mu_B/L = \mu$, where μ_B is the step size for BNLMS and μ is the step size for the NLMS algorithm. However, the current implementation of `asptbnlms()` assumes that the division by L in equation 4.3 has been absorbed in the algorithm step size.
- Like all block processing algorithms, the BNLMS introduces a delay of L samples in the input-output path. This is the time required to collect a block of input data.
- `asptbnlms()` supports both real and complex data and filters. The adaptive filter for the complex BNLMS algorithm converges to the complex conjugate of the optimum solution.
- `asptbnlms()` updates the input delay line internally.

Resources The resources required to implement the BNLMS algorithm for a transversal adaptive FIR filter of N coefficients using a block of L samples in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$2N + 5L + 2$
MULTIPLY	$N(2L + 1)/L + 4$
ADD	$N(2L + 1)/L + 1$
DIVIDE	$1/L$

See Also INIT_ BNLMS, ASPTBLMS, ASPTBFDAF, ASPTNLMS.

Reference [11] and [4] for extensive analysis of the LMS and the steepest-descent search method and [1] and [2] for BNLMS.

4.5 asptdrlms

Purpose

Performs filtering and coefficient update using the Data Reusing Least Mean Squares (DRLMS) algorithm. DRLMS updates the filter coefficients k times each iteration using the same input data vector to speed the convergence process.

Syntax

```
[w,y,e] = asptdrlms(x,w,d,mu)
[w,y,e] = asptdrlms(x,w,d,mu,alg,k)
```

Description

`asptdrlms()` improves the convergence speed of the LMS algorithm by updating the filter coefficients several times using the same set of input and desired data. When the number of updates, $k = 0$, DRLMS falls back to the LMS algorithm. The coefficients update is performed according to the 'alg' input argument which can take any of the following values.

- `'lms'` : the default value, uses the LMS algorithm
- `'slms'` : uses the sign LMS algorithm, the sign of the error $e(k)$ is used in the update equation instead of the error.
- `'srlms'` : uses the signed regressor LMS algorithm, the sign of the input signal $x(k)$ is used in the update equation instead of the input signal.
- `'sslms'` : uses the sign-sign-LMS algorithm, the sign of the error $e(k)$ and the sign of the input signal $x(k)$ are used in the update equation instead of the error and the input signals.

The input and output parameters of `asptdrlms()` for an FIR adaptive filter of L coefficients are summarized below.

```
Input Parameters [size] ::
   x   : vector of input samples [L x 1]
   w   : vector of filter coefficients w(n-1) [L x 1]
   d   : desired output d(n) [1 x 1]
   mu  : adaptation constant
   alg : specifies the variety of the lms to use in the
         update equation. Must be one of the following:
         'lms'     [default]
         'slms'  - sign LMS, uses sign(e)
         'srlms' - signed regressor LMS, uses sign(x)
         'sslms' - sign-sign LMS, uses sign(e) and sign(x)
   k   : number of data reusing cycles
Output parameters ::
   w   : updated filter coefficients w(n)
   y   : filter output y(n)
   e   : error signal; e(n) = d(n) - y(n)
```

Example

```
% DRLMS used in a simple system identification application.
% The learning curves of the DRLMS is compared for several
% values of the number of data reusing cycles.
iter = 5000;                    % Number of samples to process
% Complex unknown impulse response
h    = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xn   = 2*(rand(iter,1)-0.5);   % Input signal, zero mean random.
xn   = filter(.05,[1 -.95], xn); % colored input
dn   = osfilter(h,xn);          % Unknown filter output
M    = [0, 2, 4, 8];            % data reusing cycles
en   = zeros(iter,length(M)); % vector to collect the error

%% Processing Loop
for n = 1:length(M)
  % Initialize the DRLMS algorithm with a filter of 10 coef.
  [w,x,d,y,e]=init_drlms(10);
  for (m=1:iter)
    x = [xn(m,:); x(1:end-1,:)];  % update the input delay line
    d = dn(m,:) + 1e-3*rand;      % additive noise of var = 1e-6
    [w,y,e]= asptdrlms(x,w,d,0.1,'lms',M(n));
    en(m,n) = e;                  % save the last error
  end;
end;

% display the results
subplot(2,2,1);stem([real(w) imag(conj(w))]); grid;
subplot(2,2,2);eb = filter(0.1, [1 -.9] , en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 4.6. The left side graph of the figure shows the adaptive filter coefficients after convergence. The right side graph shows the learning curve for DRLMS for $k = \{0, 2, 4, 8\}$. The case of $k = 0$ is equivalent to the LMS algorithm and is included as a reference. Fig. 4.6 suggests that the convergence speed improves as k increases.

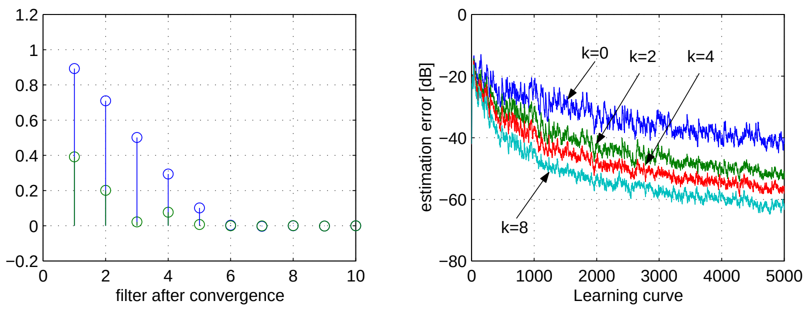

Figure 4.6: The adaptive filter coefficients after convergence and the learning curve for the complex FIR system identification problem using the DRLMS for several values of the data reusing parameter k.

Algorithm The current implementation of `asptdrlms()` performs the following operations

- Filters the input signal through the adaptive filter $w(n-1)$ to produce the filter's output sample $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$.
- Updates the adaptive filter coefficients k times according to the 'alg' input parameter.

Remarks

- DRLMS improves the convergence speed of the LMS by updating the filter coefficients more frequently, and therefore consumes more processor cycles.
- The DRLMS shows similar convergence properties to those known for the LMS algorithm.
- `asptdrlms()` supports both real and complex data and filters. The adaptive filter for the complex DRLMS algorithm converges to the complex conjugate of the optimum solution.
- `asptdrlms()` does not update the input delay line for $x(n)$, this has been chosen to provide more flexibility, so that the same function can be used with transversal as well as linear combiner structures. Delay line update, by inserting the newest sample at the beginning of the buffer and shifting the rest of the samples to the right, has to be done before calling `asptdrlms()` as in the example above.

Resources The resources required to implement the DRLMS algorithm for a transversal adaptive FIR filter of L coefficients and k data reusing cycles in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$2L + 5$
MULTIPLY	$(2L + 1) * [k + 1]$
ADD	$2L[k + 1]$
DIVIDE	0

See Also INIT_ DRLMS, ASPTLMS, ASPTDRNLMS, ASPTRDRLMS, ASPTR-DRNLMS..

Reference [11] and [4] for extensive analysis of the LMS and the steepest-descent search method and [7] for an introduction to the data reusing LMS algorithms.

4.6 asptdrnlms

Purpose Performs filtering and coefficient update using the Data Reusing Normalized Least Mean Squares (DRNLMS) algorithm. DRNLMS updates the filter coefficients k times each iteration using the same set of data to speed the convergence process.

Syntax

```
[w,y,e,p] = asptdrnlms(x,w,d,mu,p)
[w,y,e,p] = asptdrnlms(x,w,d,mu,p,b,k)
```

Description `asptdrnlms()` improves the convergence speed of the NLMS algorithm by updating the filter coefficients several times using the same set of input and desired data. When the number of updates, $k = 0$, DRNLMS falls back to the NLMS algorithm. The input and output parameters of `asptdrnlms()` for an FIR adaptive filter of L coefficients are summarized below.

```
Input Parameters [Size]::
   x   : input samples delay line [L x 1]
   w   : filter coefficients vector w(n-1) [L x 1]
   d   : desired output d(n) [1 x 1]
   mu  : adaptation constant
   p   : last estimated power of x p(n-1)
   b   : AR pole for recursive calculation of p
   k   : number of data reusing cycles
Output parameters::
   w   : updated filter coefficients w(n)
   y   : filter output y(n)
   e   : error signal; e(n) = d(n)-y(n)
   p   : new estimated power of x p(n)
```

Example

```
% DRNLMS used in a simple system identification application.
% The learning curves of the DRNLMS is compared for several
% values of the number of data reusing cycles.

iter = 5000;                    % Number of samples to process
% Complex unknown impulse response
h    = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xn   = 2*(rand(iter,1)-0.5);    % Input signal, zero mean random.
xn   = filter(.05,[1 -.95], xn); % Colored input
dn   = osfilter(h,xn);          % Unknown filter output
M    = [0, 2, 4, 8];            % Data reusing cycles
en   = zeros(iter,length(M)); % Vector to collect the error

%% Processing Loop
for n = 1:length(M)
  % Initialize the DRNLMS algorithm with a filter of 10 coef.
  [w,x,d,y,e,p]=init_drnlms(10);
  for (m=1:iter)
    x = [xn(m,:); x(1:end-1,:)];  % update the input delay line
    d = dn(m,:) + 1e-3*rand;      % additive noise of var = 1e-6
    [w,y,e,p]= asptdrnlms(x,w,d,0.005,p,0.98,M(n));
    en(m,n) = e;                  % save the last error sample
  end;
end;

% display the results
subplot(2,2,1);stem([real(w) imag(conj(w))]); grid;
subplot(2,2,2);eb = filter(0.1, [1 -.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 4.7. The left side graph of the figure shows the adaptive filter coefficients after convergence. The right side graph shows the learning curve for DRNLMS for $k = \{0, 2, 4, 8\}$. The case of $k = 0$ is equivalent to the NLMS algorithm and is included as a reference. Fig. 4.7 suggests that the convergence speed improves as k increases.

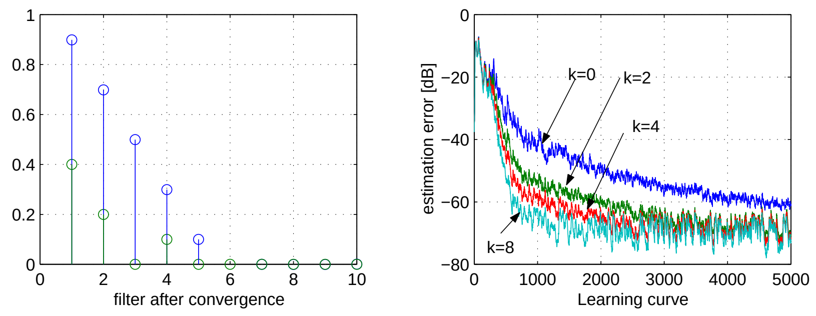

Figure 4.7: The adaptive filter coefficients after convergence and the learning curve for the complex FIR system identification problem using the DRNLMS for several values of the data reusing parameter k.

Algorithm The current implementation of `asptdrnlms()` performs the following operations

- Filters the input signal through the adaptive filter $w(n-1)$ to produce the filter's output sample $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$.
- Updates the estimate of the input signal power, p.
- Updates the adaptive filter coefficients k times.

Remarks

- DRNLMS improves the convergence speed of the NLMS by updating the filter coefficients more frequently, and therefore consumes more processor cycles.
- The DRNLMS shows similar convergence properties to those known for the NLMS algorithm.
- `asptdrnlms()` supports both real and complex data and filters. The adaptive filter for the complex DRNLMS algorithm converges to the complex conjugate of the optimum solution.
- `asptdrnlms()` does not update the input delay line for $x(n)$, this has been chosen to provide more flexibility, so that the same function can be used with transversal as well as linear combiner structures. Delay line update, by inserting the newest sample at the beginning of the buffer and shifting the rest of the samples to the right, has to be done before calling `asptdrnlms()` as in the example above.

Resources The resources required to implement the DRNLMS algorithm for a transversal adaptive FIR filter of L coefficients and k data reusing cycles in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$2L + 8$
MULTIPLY	$(2L + 1)[k + 1] + 4$
ADD	$(2L)[k + 1] + 1$
DIVIDE	k+1

See Also INIT_ DRNLMS, ASPTNLMS, ASPTDRLMS, ASPTRDRLMS, ASPTRDRNLMS.

Reference [11] and [4] for extensive analysis of the LMS and the steepest-descent search method and [7] for an introduction to the DRNLMS.

4.7 asptleakynlms

Purpose Sample per sample filtering and coefficient update using the Leaky Normalized LMS algorithm.

Syntax

```
[w,y,e,p]= asptleakynlms(x,w,d,mu,a)
[w,y,e,p]= asptleakynlms(x,w,d,mu,a,p,b)
```

Description `asptleakynlms()` implements the Leaky NLMS adaptive algorithm used to update transversal adaptive filters. Referring to the general adaptive filter shown in Fig. 2.6, `asptleakynlms()` takes an input samples delay line $x(n)$, a desired sample $d(n)$, the vector of the adaptive filter coefficients from previous iteration $w(n-1)$, the step size mu, and returns the filter output $y(n)$, the error sample $e(n)$ and the updated vector of filter coefficients $w(n)$. Similar to the NLMS, the Leaky NLMS also estimates the instantaneous power of the input signal $p(n)$ and normalizes the step size mu by this estimate to make the update algorithm independent of the input signal energy. If the input parameters p and b are given, an efficient recursive estimation of $x(n)$ is used, otherwise the inner product of $x(n)$ with itself is used instead. The update equation of `asptleakynlms()` is given by

$$\underline{\mathbf{w}}(n) = \alpha\underline{\mathbf{w}}(n) + (\frac{\mu}{p})e(n)\underline{\mathbf{x}}(n). \tag{4.4}$$

Where α is the leak factor, a scalar constant in the range $(0 < \alpha < 1)$. The effect of the leak is identical to adding white noise to the filter input with noise variance σ_n^2 given by $\sigma_n^2 = (1-\alpha)/(2\mu)$. This might be helpful in several applications such as antenna sidelobe cancelers and echo cancelers. The direct effect of the leak is that the filter coefficients tend to decay exponentially to zero when the step size μ is set to zero, so that the adaptation of the filter will not stall and the filter has to keep adapting to minimize the mean square error.
The input and output parameters of `asptleakynlms()` for an FIR adaptive filter of L coefficients are summarized below.

```
Input Parameters [Size]::
   x    : input samples delay line [L x 1]
   w    : filter coefficients vector w(n-1) [L x 1]
   d    : desired output d(n) [1 x 1]
   mu   : adaptation constant
   a    : leak factor (0 < a < 1)
   p    : last estimated power of x p(n-1)
   b    : AR pole for recursive calculation of p
Output parameters::
   w    : updated filter coefficients w(n)
   y    : filter output y(n)
   e    : error signal; e(n) = d(n)-y(n)
   p    : new estimated power of x p(n)
```

Example

```
% Leaky NLMS used in an inverse modeling application (channel
% equalizer). By the end of this script the adaptive filter w
% should have the inverse response of the filter h so that the
% cascade conv(w,h) = delta(t-D), is a pure delay of D samples.

iter = 5000;                  % Number of samples to process
h    = impz(.3,[1 -.7],10);   % channel to be equalized
x1   = 2*(rand(iter,1)-0.5);  % system input.
x2   = osfilter(h,x1);        % channel output = filter input
en   = zeros(iter,1);         % vector to collect the error
D    = 3;                     % inversion delay
% Initialize the Leaky NLMS algorithm with a filter of 10 coef.
[w,x,d,y,e,p]=init_leakynlms(10);

%% Processing Loop
for (m=1:iter-D)
   x = [x2(m+D,:); x(1:end-1,:)];  % update the input delay line
   d = x1(m,:);                    % desired = delayed sys input
   % call Leaky NLMS to calculate the filter output, estimation
   % error and update the coefficients.
   [w,y,e,p]= asptleakynlms(x,w,d,0.01,(1-1e-5),p,0.98);
   % save the last error sample to plot later
   en(m,:) = e;
end;

% display the results
subplot(2,2,1);stem(conv(h,w)); grid;
subplot(2,2,2);
eb = filter(.1,[1 -0.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 4.8. The left side graph of the figure shows the cascade of the channel and the adaptive filter coefficients after convergence. This cascade should be a pure delay equals to D for perfect equalization. The right side graph shows the mean square error in dB versus time during the adaptation process, which is usually called the learning curve.

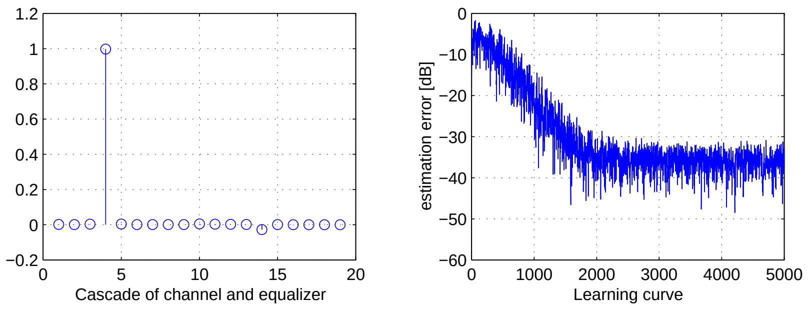

Figure 4.8: The cascade of the channel and the adaptive filter coefficients after convergence (left), and the learning curve for the inverse modeling problem using the Leaky NLMS algorithm (right).

Algorithm

Similar to `asptnlms()`, `asptleakynlms()` performs the following operations

- Filter the input signal $x(n)$ through the adaptive filter $w(n-1)$ to produce the filter output $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$.
- Estimates the input signal power p and normalizes the step size mu by this estimate.
- Updates the adaptive filter coefficients using the error $e(n)$ and the delay line of input samples $x(n)$ resulting in $w(n)$.

Remarks

The LEAKY NLMS is a stochastic implementation of the steepest-descent algorithm where the *mean* value of the filter coefficients converge towards their optimal solution biased by the variance of the equivalent input additive noise due to the leak. Therefore, the filter coefficients will fluctuate about their optimum values given by the Wiener solution. The amplitude of the fluctuations is partly controlled by the step size and partly by the leak factor. The smaller the step size, the smaller the fluctuations (less final misadjustment) but also the slower the adaptive coefficients converge to their optimal values. Note also the following.

- The LEAKY NLMS algorithm estimates the energy of the input signal each sample and normalizes (divides) the step size by this estimate, therefore selecting a step size inversely proportion to the instantaneous input signal power. Although this improves the convergence properties in comparison to the LMS, it does not solve the eigenvalue spread problem.
- The LEAKY NLMS algorithm shows stable convergence behavior only when the step size mu (convergence constant) takes a value between zero and an upper limit defined by the statistics of the filter's input signal. The fastest convergence will be achieved for a white noise input sequence. Such white input signal has all its eigenvalues equal to the noise variance σ^2 and therefore has a diagonal autocorrelation matrix with diagonal values equal to σ^2.
- The more colored the spectrum of the input signal, the slower the convergence will be. This is due to the large eigenvalue spread for such colored signals. This makes the convergence composed of several modes, each associated with one of the eigenvalues.
- `asptleakynlms()` supports both real and complex data and filters. The adaptive filter for the complex Leaky NLMS algorithm converges to the complex conjugate of the optimum solution.
- `asptleakynlms()` does not update the input delay line for $x(n)$, this has been chosen to provide more flexibility. Delay line update, by inserting the newest sample at the beginning of the buffer and shifting the rest of the samples to the right, has to be done before calling `asptleakynlms()` as in the example above.

Resources The resources required to implement the Leaky NLMS algorithm for a transversal adaptive FIR filter of L coefficients in real time is given in the table below. The computations given are those required to process one sample and assumes that recursive estimation of the input power is used.

MEMORY	$2L + 8$
MULTIPLY	$3L + 4$
ADD	$2L + 2$
DIVIDE	1

See Also INIT_ LEAKYNLMS, MODEL_ LEAKYNLMS, ASPTNLMS.

Reference [11] and [4] for extensive analysis of the NLMS and the steepest-descent search method.

4.8 asptlclms

Purpose Sample per sample filtering and coefficient update using the Linearly Constrained LMS (LCLMS) algorithm.

Syntax

```
[w,y,e]= asptlclms(x,w,d,mu,c,a)
```

Description `asptlclms()` implements the LMS adaptive algorithm used to update a transversal adaptive filters $\underline{\mathbf{w}}$ subject to the linear constraint $\underline{\mathbf{c}}^H\underline{\mathbf{w}} = a$. Referring to the general adaptive filter shown in Fig. 2.6, `asptlclms()` takes an input samples delay line $x(n)$, a desired sample $d(n)$, the vector of the adaptive filter coefficients from previous iteration $w(n-1)$, the step size mu, and returns the filter output $y(n)$, the error sample $e(n)$ and the updated vector of filter coefficients $w(n)$. The partial update equation of `asptlclms()` is given by

$$\underline{\mathbf{w}}(n) = \underline{\mathbf{w}}(n) + (\mu)e(n)\underline{\mathbf{x}}(n). \tag{4.5}$$

The linear constraint is then applied to the updated coefficients. An interesting case occurs when the desired signal is set to zero (blind adaptation) and the linear constraint is used to control the adaptation. This case is used in adaptive array signal processing to produce a beam in a certain look direction.
The input and output parameters of `asptlclms()` for an FIR adaptive filter of L coefficients are summarized below.

```
Input Parameters::
   x   : vector of input samples at time n
   w   : vector of filter coefficients w(n-1)
   d   : desired response d(n)
   mu  : adaptation constant
   c   : the weighting vector in the constraint equation
   a   : a scalar constant
Output parameters::
   w   : updated filter coefficients w(n)
   y   : filter output y(n)
   e   : error signal; e(n)=d(n)-y(n)
```

Example

```
% LCLMS used in a 2-element lambda/2 beam former application at
% baseband frequency. The desired signal coming from angle 0 deg
% and a jammer from angle P=30 deg.
iter = 5000;                  % samples to process
P    = 30;                    % Jammer angle of arrival
D    = pi * sin(pi*P/180);    % delay (L=lambda/2)
xn   = rand(iter,1);          % signal
nn   = rand(iter,1);          % Jammer
c    = [1 ; 1];               % linear constraint vector
a    = 1;                     % linear constraint scalar
% Initialize the LCLMS algorithm with a filter of 2 coef.
[w,x,d,y,e] = init_lclms(2);
d = 0;                        % no desired is needed
```

```
%% Processing Loop
for (m=1:iter)
   x(1)    = xn(m)+nn(m);            % element-1 input
   x(2)    = xn(m)+nn(m)*exp(-j*D);  % element-2 input
   [w,y,e] = asptlclms(x,w,d,0.05,c, a);
end;
% Plot the resulting sensitivity pattern
th = 2*pi*[0:0.001:1];
PG = zeros(size(th));
for k = 1:length(th)
  D     = pi * sin(th(k));  % delay in rad
  x(1)  = 1 ;               % test signal at element-1
  x(2)  = exp(-j* D ) ;     % and at element-2
  e     =  d - w' * x;      % error
  PG(k) = (e.*conj(e));     % Power Gain
end;
polar(th,(PG));             % Plot result
```

Running the above script will produce the graph shown in Fig. 4.9. It is clear from this sensitivity pattern that the array tries to reduce the jammer signal in the array output by producing a spatial notch at the angle of arrival of the jammer, which is 30° in the above example.

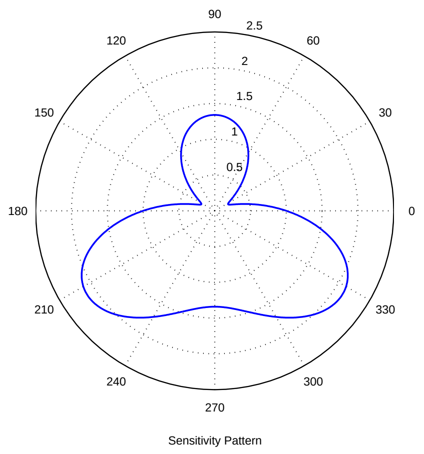

Figure 4.9: Sensitivity pattern for a 2-element adaptive array using LCLMS.

Algorithm The current implementation of `asptlclms()` performs the following operations

- Filter the input signal $x(n)$ through the adaptive filter $w(n-1)$ to produce the filter output $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$.
- Updates the adaptive filter coefficients using the error $e(n)$ and the input samples $x(n)$ resulting in $w(n)$.
- Applies the linear constraint on the newly calculated filter coefficients.

Remarks The LCLMS algorithm is a stochastic implementation of the steepest-descent algorithm where the *mean* value of the filter coefficients converge towards their constrained solution given by

$$\underline{\mathbf{w}}_c = \underline{\mathbf{w}}_{opt} + \frac{(a - \underline{\mathbf{w}}_{opt}^H \underline{\mathbf{c}}) * \mathbf{R}^{-1}\underline{\mathbf{c}}}{\underline{\mathbf{c}}^H \mathbf{R}^{-1}\underline{\mathbf{c}}}. \tag{4.6}$$

where $\underline{\mathbf{w}}_{opt}$ is the optimum unconstrained solution. The filter coefficients will fluctuate about their optimum values given above. The amplitude of the fluctuations is controlled by the step size. The smaller the step size, the smaller the fluctuations (less final misadjustment) but also the slower the adaptive coefficients converge to their optimal values. Note also the following.

- The LCLMS algorithm shows stable convergence behavior only when the step size *mu* (convergence constant) takes a value between zero and an upper limit defined by the statistics of the filter's input signal. The fastest convergence will be achieved for a white noise input sequence with zero mean and unit variance. Such white input signal has all its eigen values equal to unity and therefore has a diagonal autocorrelation matrix with diagonal values equal to unity.
- The more colored the spectrum of the input signal, the slower the convergence will be. This is due to the large eigen value spread for such colored signals. This makes the convergence composed of several modes, each associated with one of the eigen values.
- `asptlclms()` supports both real and complex data and filters. The adaptive filter for the complex LCLMS algorithm converges to the complex conjugate of the optimum solution.
- `asptlclms()` does not update the input delay line for $x(n)$, this has been chosen to provide more flexibility. Delay line update, by inserting the newest sample at the beginning of the buffer and shifting the rest of the samples to the right, has to be done before calling `asptlclms()`.

Resources The resources required to implement the LCLMS algorithm for a transversal adaptive FIR filter of L coefficients in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$3L + 5$
MULTIPLY	$6L$
ADD	$6L - 2$
DIVIDE	1

See Also INIT_ LCLMS, BEAMBB_ LCLMS, ASPTLMS.

Reference [11] for extensive analysis of the LMS and the steepest-descent search method. [2] for the Linearly Constrained LMS.

4.9 asptlms

Purpose Sample per sample filtering and coefficient update using the Least Mean Squares (LMS) or one of its variants. The variants currently implemented are the sign, sign-sign, and signed regressor algorithms.

Syntax

```
[w,y,e]= asptlms(x,w,d,mu)
[w,y,e]= asptlms(x,w,d,mu,alg)
```

Description `asptlms()` implements the LMS adaptive algorithm used to update transversal adaptive filters. Referring to the general adaptive filter shown in Fig. 2.6, `asptlms()` takes an input samples delay line $x(n)$, a desired sample $d(n)$, the vector of the adaptive filter coefficients from previous iteration $w(n-1)$, the step size mu, and returns the filter output $y(n)$, the error sample $e(n)$ and the updated vector of filter coefficients $w(n)$. The update equation of `asptlms()` is given by

$$\underline{\mathbf{w}}(n) = \underline{\mathbf{w}}(n) + \mu e(n)\underline{\mathbf{x}}(n). \tag{4.7}$$

Coefficients update is performed according to the 'alg' input argument which can take any of the following values.

- `'lms'` : the default value, uses the LMS algorithm
- `'slms'` : uses the sign LMS algorithm, the sign of the error $e(n)$ is used in the update equation instead of the error.
- `'srlms'` : uses the signed regressor LMS algorithm, the sign of the input signal $x(n)$ is used in the update equation instead of the input signal.
- `'sslms'` : uses the sign-sign-LMS algorithm, the sign of the error $e(n)$ and the sign of the input signal $x(n)$ are used in the update equation instead of the error and the input signals.

The input and output parameters of `asptlms()` for an FIR adaptive filter of L coefficients are summarized below.

```
Input Parameters [size] ::
   x   : vector of input samples x(n) [L x 1]
   w   : vector of filter coefficients w(n-1) [L x 1]
   d   : desired output d(n) [1 x 1]
   mu  : adaptation constant
   alg : specifies the variety of the lms to use in the
         update equation. Must be one of the following:
         'lms' [default]
         'slms'  - sign LMS, uses sign(e)
         'srlms' - signed regressor LMS, uses sign(x)
         'sslms' - sign-sign LMS, uses sign(e) and sign(x)
Output parameters ::
   w   : updated filter coefficients w(n)
   y   : filter output y(n)
   e   : error signal; e(n) = d(n) - y(n)
```

Example

```
% LMS used in a simple system identification application.
% By the end of this script the adaptive filter w should
% have the same coefficients as the unknown filter h.
%
iter = 5000;                    % Number of samples to process
% Complex unknown impulse response
h    = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xn   = 2*(rand(iter,1)-0.5);   % Input signal, zero mean random.
% although xn is real, dn will be complex since h is complex
dn   = osfilter(h,xn);          % Unknown filter output
en   = zeros(iter,1);           % vector to collect the error
% Initialize the LMS algorithm with a filter of 10 coef.
[w,x,d,y,e]=init_lms(10);

%% Processing Loop
for (m=1:iter)
   x = [xn(m); x(1:end-1)];  % update the input delay line
   d = dn(m,:) + 1e-3*rand;  % additive noise of var = 1e-6
   % call LMS to calculate the output, estimation error
   % and update the coefficients.
   [w,y,e]= asptlms(x,w,d,0.05);
   % save the last error sample to plot later
   en(m) = e;
end;

% display the results
% note that w converges to conj(h) for complex data
subplot(2,2,1);stem([real(w) imag(conj(w))]); grid;
subplot(2,2,2);eb = filter(.1,[1 -.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 4.10. The left side graph of the figure shows the adaptive filter coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB).

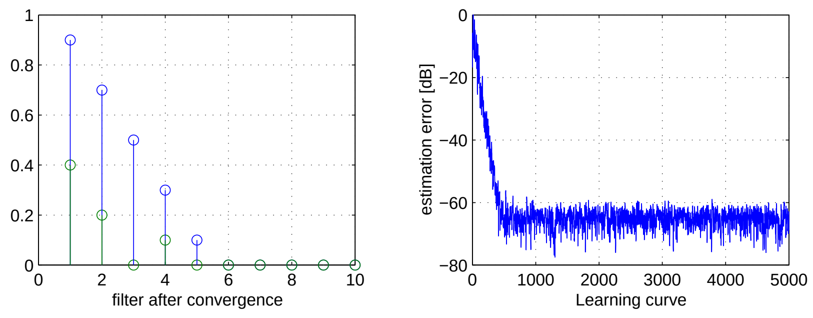

Figure 4.10: The adaptive filter coefficients after convergence and the learning curve for the complex FIR system identification problem using the LMS algorithm.

Algorithm The LMS algorithm and its normalized version NLMS are the most widely used adaptive algorithms in the industry due to their low complexity, good performance, and extensive existing analysis. The current implementation of `asptlms()` performs the following operations

- Filter the input signal $x(n)$ through the adaptive filter $w(n-1)$ to produce the filter output $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$.
- Updates the adaptive filter coefficients using the error $e(n)$ and the delay line of input samples $x(n)$ resulting in $w(n)$.

Remarks The LMS algorithm is a stochastic implementation of the steepest-descent algorithm where the *mean* value of the filter coefficients converge towards their optimal solution. Therefore, the filter coefficients will fluctuate about their optimum values given by the Wiener solution. The amplitude of the fluctuations is controlled by the step size. The smaller the step size, the smaller the fluctuations (less final misadjustment) but also the slower the adaptive coefficients converge to their optimal values. Note also the following.

- The LMS algorithm shows stable convergence behavior only when the step size *mu* (convergence constant) takes a value between zero and an upper limit defined by the statistics of the filter's input signal. The fastest convergence will be achieved for a white noise input sequence with zero mean and unit variance. Such white input signal has all its eigen values equal to unity and therefore has a diagonal autocorrelation matrix with diagonal values equal to unity.
- The more colored the spectrum of the input signal, the slower the convergence will be. This is due to the large eigen value spread for such colored signals. This makes the convergence composed of several modes, each associated with one of the eigen values.
- `asptlms()` supports both real and complex data and filters. The adaptive filter for the complex LMS algorithm converges to the complex conjugate of the optimum solution.
- `asptlms()` does not update the input delay line for $x(n)$, this has been chosen to provide more flexibility, so that the same function can be used with transversal as well as linear combiner structures. Delay line update, by inserting the newest sample at the beginning of the buffer and shifting the rest of the samples to the right, has to be done before calling `asptlms()` as in the example above.

Resources The resources required to implement the LMS algorithm for a transversal adaptive FIR filter of L coefficients in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$2L + 4$
MULTIPLY	$2L + 1$
ADD	$2L$
DIVIDE	0

See Also INIT_ LMS, BEAMRF_ LMS, ASPTNLMS, ASPTVSSLMS, ASPTLCLMS.

Reference [11] and [4] for extensive analysis of the LMS and the steepest-descent search method.

4.10 asptmvsslms

Purpose Sample per sample filtering and coefficient update using the Modified Variable Step Size LMS (MVSSLMS) algorithm.

Syntax

```
[w,g,mu,y,e]= asptmvsslms(x,w,g,d,mu,roh)
[w,g,mu,y,e]= asptmvsslms(x,w,g,d,mu,roh,mu_min,mu_max)
```

Description `asptmvsslms()` is a more resource efficient version of the Variable Step Size LMS adaptive algorithm, where the vector of step sizes in VSSLMS is replaced by a scalar step size. MVSSLMS does not only adjust the filter coefficients but also adjusts a scalar step size $mu(n)$ to obtain fast convergence rate as well as small final misadjustment, a combination impossible to achieve with constant step size. Referring to the general adaptive filter shown in Fig. 2.6, `asptmvsslms()` takes an input samples delay line $x(n)$, a desired sample $d(n)$, the vector of the adaptive filter coefficients from previous iteration $w(n-1)$, the previous step size value $mu(n-1)$, the previous gradient value $g(n-1)$ (used to update mu), and returns the filter output $y(n)$, the error sample $e(n)$, the updated gradient vector $g(n)$, the updated step size vector $mu(n)$, and the updated vector of filter coefficients $w(n)$. If the `mu_min` and `mu_max` optional input arguments are given, the new step size is constrained to those limits. The update equation of `asptmvsslms()` is given by

$$\underline{\mathbf{w}}(n) = \underline{\mathbf{w}}(n) + \mu(n)e(n)\underline{\mathbf{x}}(n). \tag{4.8}$$

The input and output parameters of `asptmvsslms()` for an FIR adaptive filter of L coefficients are summarized below. Note that the difference between VSSLMS and MVSSLMS is that mu and g in the former are vectors of size L and in the latter are scalars.

```
Input Parameters [Size] ::
   x      : input samples delay line [L x 1]
   d      : desired response [1 x 1]
   w      : filter coef. vector w(n-1) [L x 1]
   g      : previous gradient sample g(n-1) [1 x 1]
   mu     : previous step sizes value mu(n-1) [1 x 1]
   roh    : gradient step size [1 x 1]
   mu_min : lower bound for mu [1 x 1]
   mu_max : higher bound for mu [1 x 1]

Output parameters::
   w      : updated filter coefficients w(n)
   y      : filter output y(n)
   g      : updated gradient g(n)
   mu     : updated step size mu(n)
   e      : error sample, e(n)=d(n)-y(n)
```

Example

```
% MVSSLMS used in a system identification application.
% By the end of this script the adaptive filter w should
% have the same coefficients as the unknown filter h.
iter = 5000;                   % Number of samples to process
% Complex unknown impulse response
h    = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xn   = 2*(rand(iter,1)-0.5);   % Input signal, zero mean random.
% although xn is real, dn will be complex since h is complex
dn   = osfilter(h,xn);         % Unknown filter output
en   = zeros(iter,1);          % vector to collect the error
mu0  = 0.05;                   % initial step size (scalar)
muv  = zeros(iter,1);          % evolution of mu with time
% Initialize the MVSSLMS algorithm with a filter of 10 coef.
[w,x,d,y,e,g,mu]  = init_mvsslms(10,[],[],[],mu0);
%% Processing Loop
for (m=1:iter)
   % update the input delay line
   x = [xn(m,:); x(1:end-1,:)];
   d = dn(m,:) + 1e-3*rand;       % additive noise of var = 1e-6
   % call MVSSLMS to calculate the filter output, estimation error
   % and update the coefficients and step sizes.
   [w,g,mu,y,e] = asptmvsslms(x,w,g,d,mu,1e-3,1e-6,.99);
   % save the last error sample to plot later
   en(m,:) = e;  muv(m) = mu;
end;
% display the results
subplot(3,3,1);stem([real(w) imag(conj(w))]); grid;
eb = fftfilt(fir1(5,.05), en .* conj(en));
subplot(3,3,2);plot(10*log10(eb  ));grid
subplot(3,3,3);plot(muv); grid;
```

Running the above script will produce the graph shown in Fig. 4.11. The left-most graph of the figure shows the adaptive filter coefficients after convergence which are almost identical to the unknown filter h. The middle graph shows the mean square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB). The right-most graph shows the evolution of the scalar step size with time. Note that the step size increases at the beginning to speed up the convergence then decreases to decrease the final misadjustment.

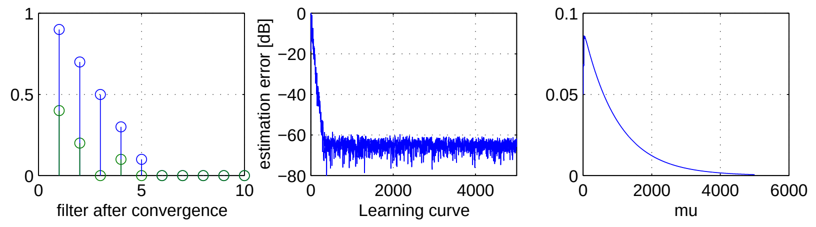

Figure 4.11: The adaptive filter coefficients after convergence, the learning curve, and the evolution of the step size for the complex FIR system identification problem using the MVSSLMS algorithm.

Remarks

Like the LMS, the MVSSLMS is also a stochastic implementation of the steepest-descent algorithm where the *mean* value of the filter coefficients converge towards their optimal solution. Therefore, the filter coefficients will fluctuate about their optimum values given by the Wiener solution. The amplitude of the fluctuations is controlled by the step size. The smaller the step size, the smaller the fluctuations (less final misadjustment) but also the slower the adaptive coefficients converge to their optimal values. The improvement the MVSSLMS introduces to the LMS is that the step size is also updated. When the filter coefficients are far from their optimal values, the step size is increased to speed up the convergence. Conversely, when the coefficients are near their optimal values, the step size is decreased to decrease the final misadjustment. Similar to the LMS, the following points also apply to the MVSSLMS.

- The MVSSLMS algorithm shows stable convergence behavior only when the step size $mu(n)$ takes a value between zero and an upper limit, at all time indexes n, defined by the statistics of the filter's input signal. The fastest convergence will be achieved for a white noise input sequence with zero mean and unit variance. Such white input signal has all its eigenvalues equal to unity and therefore has a diagonal autocorrelation matrix with diagonal values equal to unity.
- The more colored the spectrum of the input signal, the slower the convergence will be. This is due to the large eigenvalue spread for such colored signals. This makes the convergence composed of several modes, each associated with one of the eigenvalues.
- `asptmvsslms()` supports both real and complex data and filters. The adaptive filter for the complex MVSSLMS algorithm converges to the complex conjugate of the optimum solution.
- `asptmvsslms()` does not update the input delay line for $x(n)$, this has been chosen to provide more flexibility, so that the same function can be used with transversal as well as linear combiner structures. Delay line update, by inserting the newest sample at the beginning of the buffer and shifting the rest of the samples to the right, has to be done before calling `asptmvsslms()` as in the example above.

Resources

The resources required to implement the MVSSLMS algorithm for a transversal adaptive FIR filter of L coefficients in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$2L + 4$
MULTIPLY	$3L + 3$
ADD	$3L + 1$
DIVIDE	0

See Also

INIT_ MVSSLMS, MODEL_ MVSSLMS, ASPTVSSLMS, ASPTNLMS, ASPTLMS, ASPTLCLMS.

Reference

[11] for extensive analysis of the LMS and the steepest-descent search method.

4.11 asptnlms

Purpose Sample per sample filtering and coefficient update using the Normalized LMS (NLMS) algorithm.

Syntax

```
[w,y,e,p]= asptnlms(x,w,d,mu)
[w,y,e,p]= asptnlms(x,w,d,mu,p,b)
```

Description `asptnlms()` implements the NLMS adaptive algorithm used to update transversal adaptive filters. Referring to the general adaptive filter shown in Fig. 2.6, `asptnlms()` takes an input samples delay line $x(n)$, a desired sample $d(n)$, the vector of the adaptive filter coefficients from previous iteration $w(n-1)$, the step size mu, and returns the filter output $y(n)$, the error sample $e(n)$ and the updated vector of filter coefficients $w(n)$. The NLMS also estimates the instantaneous power $p(n)$ of the input signal and normalizes the step size mu by this estimate to make the update algorithm independent of the input signal energy. If the input parameters p and b are given, an efficient recursive estimation of $p(n)$ is used, otherwise the inner product of $x(n)$ with itself is used instead. The update equation of `asptnlms()` is given by

$$\underline{\mathbf{w}}(n) = \underline{\mathbf{w}}(n) + (\frac{\mu}{p})e(n)\underline{\mathbf{x}}(n). \tag{4.9}$$

The input and output parameters of `asptnlms()` for an FIR adaptive filter of L coefficients are summarized below.

```
Input Parameters [Size]::
   x    : input samples delay line [L x 1]
   w    : filter coefficients vector w(n-1) [L x 1]
   d    : desired output d(n) [1 x 1]
   mu   : adaptation constant
   p    : last estimated power of x(n), p(n-1)
   b    : AR pole for recursive calculation of p(n)

Output parameters::
   w    : updated filter coefficients w(n)
   y    : filter output y(n)
   e    : error signal; e(n) = d(n)-y(n)
   p    : new estimated power of x(n), p(n)
```

Example

```
% NLMS used in a simple system identification application.
% By the end of this script the adaptive filter w should
% have the same coefficients as the unknown filter h.
%
iter = 5000;                    % Number of samples to process
% Complex unknown impulse response
h    = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xn   = 2*(rand(iter,1)-0.5);   % Input signal, zero mean random.
% although xn is real, dn will be complex since h is complex
dn   = osfilter(h,xn);          % Unknown filter output
en   = zeros(iter,1);           % vector to collect the error
% Initialize the NLMS algorithm with a filter of 10 coef.
[w,x,d,y,e,p]=init_nlms(10);

%% Processing Loop
for (m=1:iter)
   x = [xn(m); x(1:end-1)];   % update the input delay line
   d = dn(m) + 1e-3*rand;     % additive noise of var = 1e-6
   % call NLMS to calculate the filter output, estimation error
   % and update the coefficients.
   [w,y,e,p]= asptnlms(x,w,d,0.05,p,0.98);
   % save the last error sample to plot later
   en(m) = e;
end;

% display the results
subplot(2,2,1);stem([real(w) imag(conj(w))]); grid;
subplot(2,2,2);
eb = filter(.1,[1 -.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 4.12. The left side graph of the figure shows the adaptive filter coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the mean square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error power is governed here by the additive noise at the output (-60 dB).

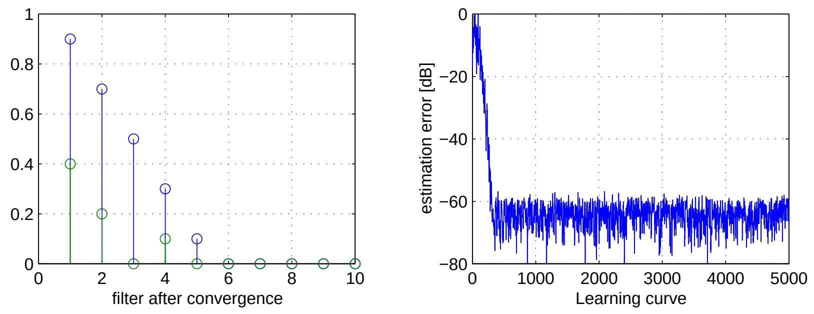

Figure 4.12: The adaptive filter coefficients after convergence and the learning curve for the complex FIR system identification problem using the NLMS algorithm.

Algorithm

The NLMS algorithm is a slightly improved version of the LMS algorithm. The current implementation of `asptnlms()` performs the following operations

- Filter the input signal $x(n)$ through the adaptive filter $w(n-1)$ to produce the filter output $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$.
- Estimates the input signal power p and normalizes the step size mu by this estimate (the improvement upon the LMS).
- Updates the adaptive filter coefficients using the error $e(n)$ and the delay line of input samples $x(n)$ resulting in $w(n)$.

Remarks

Like the LMS, the NLMS is also a stochastic implementation of the steepest-descent algorithm where the *mean* value of the filter coefficients converge towards their optimal solution. Therefore, the filter coefficients will fluctuate about their optimum values given by the Wiener solution. The amplitude of the fluctuations is controlled by the step size. The smaller the step size, the smaller the fluctuations (less final misadjustment) but also the slower the adaptive coefficients converge to their optimal values. Note also the following.

- The NLMS algorithm estimates the energy of the input signal each sample and normalizes (divides) the step size by this estimate, therefore selecting a step size inversely proportion to the instantaneous input signal power. Although this improves the convergence properties in comparison to the LMS, it does not solve the eigenvalue spread problem.
- The NLMS algorithm shows stable convergence behavior only when the step size mu (convergence constant) takes a value between zero and an upper limit defined by the statistics of the filter's input signal. The fastest convergence will be achieved for a white noise input sequence. Such white input signal has all its eigenvalues are equal to the noise variance σ^2 and therefore has a diagonal autocorrelation matrix with diagonal values equal to σ^2.
- The more colored the spectrum of the input signal, the slower the convergence will be. This is due to the large eigenvalue spread for such colored signals. This makes the convergence composed of several modes, each associated with one of the eigenvalues.
- `asptnlms()` supports both real and complex data and filters. The adaptive filter for the complex NLMS algorithm converges to the complex conjugate of the optimum solution.
- `asptnlms()` does not update the input delay line for $x(n)$, this has been chosen to provide more flexibility, so that the same function can be used with transversal as well as linear combiner structures. Delay line update, by inserting the newest sample at the beginning of the buffer and shifting the rest of the samples to the right, has to be done before calling `asptnlms()` as in the example above.

Resources The resources required to implement the NLMS algorithm for a transversal adaptive FIR filter of L coefficients in real time is given in the table below. The computations given are those required to process one sample and assumes that recursive estimation of the input power is used.

MEMORY	$2L+7$
MULTIPLY	$2L+4$
ADD	$2L+2$
DIVIDE	1

See Also INIT_ NLMS, ECHO_ NLMS, EQUALIZER_ NLMS, ASPTLMS, ASPTVSSLMS, ASPTLCLMS.

Reference [11] and [4] for extensive analysis of the NLMS and the steepest-descent search method.

4.12 asptpbfdaf

Purpose Block filtering and coefficient update in frequency domain using the Partitioned Block Frequency Domain Adaptive Filter (PBFDAF) algorithm.

Syntax `[W,X,x,y,e,Px,w]=asptpbfdaf(M,x,xn,dn,X,W,mu,n,c,b,Px)`

Description `asptpbfdaf()` solves the problem of long processing delay introduced by the BFDAF algorithm. This is achieved by splitting the adaptive filter into P partitions, from which only the first partition introduces delay, and therefore reducing the processing delay by a factor of P compared to the BFDAF for the same filter length. Similar to BFDAF, `asptpbfdaf()` performs filtering and coefficient update in the frequency domain using the overlap-save method, and therefore, provides an efficient implementation for long adaptive filters in applications such as acoustic echo cancelers where the adaptive filter can be a few thousand coefficients long. `asptpbfdaf()` is a block processing algorithm; every call processes L input and L desired samples (L is the block length), to produce L filter output samples and L error samples, besides updating all filter coefficients in frequency domain. The parameters of `asptpbfdaf()` are summarized below (see Fig. 4.2).

```
Input Parameters [Size]::
   M  : partition length
   x  : previous overlap-save vector [B x 1]
   xn : new input block [L x 1]
   dn : new desired block [L x 1]
   X  : previous matrix of F-domain input samples [B x P]
   W  : previous matrix of F-domain filter coef. [B x P]
   mu : adaptation constant
   n  : normalization flag, 0 means no normalization
   c  : constrain flag, 0 means use unconstrained PBFDAF,
        otherwise == all partitions are constrained.
   b  : forgetting factor for input power estimation
   Px : previous estimate of the power of X [B x 1]
Output parameters::
   W  : updated filter coefficients (F-domain)
   X  : updated matrix of past frequency input samples
   x  : updated overlap-save input vector
   y  : filter output block
   e  : error vector block
   Px : updated estimate of the power of X
   w  : time domain filter (calculated only if required)
```

Example

```
iter = 5000;          % Number of samples to process
% Complex unknown impulse response
h    = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xt   = 2*(rand(iter,1)-0.5); % Input signal
% although xn is real, dn will be complex
dt   = osfilter(h,xt);        % Unknown filter output
en   = zeros(iter,1);         % estimation error
% Initialize PBFDAF with a filter of 2*4 coef.
P = 2; L = 4; M = 4;
[W,x,d,e,y,Px,X,w]=init_pbfdaf(L,M,P);
%% Processing Loop
for (m=1:L:iter-L)
   xn = xt(m:m+L-1,:);              % input block
   dn = dt(m:m+L-1,:)+ 1e-3*rand;   % desired block
   % call BFDAF to calculate the filter output,
   % estimation error and update the filter coef.
   [W,X,x,y,e,Px,w] = asptpbfdaf(L,x,xn,dn,X,W,...
                      0.06,1,1,0.98,Px);
   % save the last error block to plot later
   en(m:m+L-1,:) = e;
end;
% display the results
subplot(2,2,1);stem([real(w) imag((w))]); grid;
subplot(2,2,2);
eb = filter(.1, [1 -.9], en(1:m) .* conj(en(1:m)));
plot(10*log10(eb ));grid
```

Running the above script will produce the graph shown in Fig. 4.13. The left side graph of the figure shows the adaptive filter coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB).

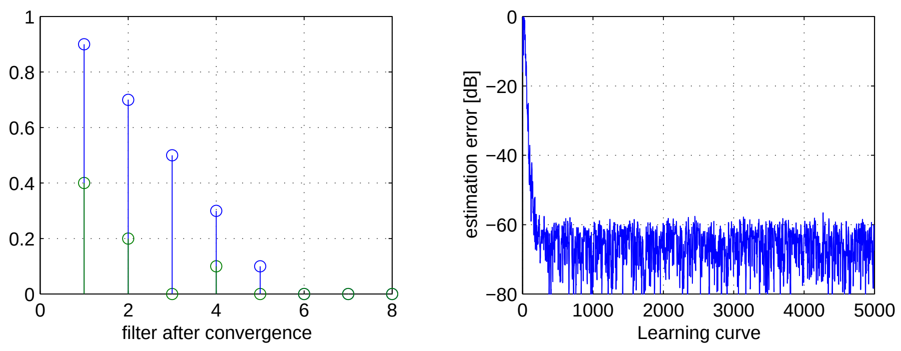

Figure 4.13: The adaptive filter coefficients after convergence and the learning curve for the complex FIR system identification problem using the PBFDAF algorithm.

Algorithm `asptpbfdaf()` performs the following operations (see Fig. 4.2).

- buffers L input samples, composes an overlap-save input vector $\underline{\mathbf{x}}(n)$ and computes its FFT, $\underline{\mathbf{X}}(f)$ and adds it to the input samples matrix $\mathbf{X}(f)$
- element-wise multiplies $\mathbf{X}(f)$ by the adaptive filter coefficients matrix $\mathbf{W}(f)$ and evaluates the sum at each frequency bin (circular convolution in time domain for the P partitions). The result is converted to time domain using IFFT and the linear convolution samples are extracted to produce the filter-output vector $\underline{\mathbf{y}}(n)$
- buffers L desired samples and evaluates the current error block $\underline{\mathbf{e}}(n) = \underline{\mathbf{d}}(n) - \underline{\mathbf{y}}(n)$. The error vector is padded with zeros and transformed to frequency domain giving $\underline{\mathbf{E}}(f)$
- estimates the input signal power at each frequency bin and normalizes the step size at each bin.
- evaluates the cross-correlation between $\mathbf{X}(f)$ and $\underline{\mathbf{E}}(f)$ to produce the block gradient vector. This vector is used to update the frequency domain filter coefficients.
- constrains the filter if required. This is performed by first taking the IFFT of $\mathbf{W}(f)$, applying a rectangular window on the time domain coefficients, and taking the FFT of the windowed coefficients.

Remarks

- Supports both real and complex signals.
- `asptpbfdaf()` constrains the filter coefficients $\mathbf{W}(f)$ rather than the gradient vector as in the official PBFDAF algorithms since this has been proven to result in a more stable update.
- The unconstrained PBFDAF (c $= 0$) saves $2P$ FFT operations each block on the cost of accuracy.
- The time domain filter coefficients $\underline{\mathbf{w}}(n)$ will be calculated and returned by `asptpbfdaf()` only if the output variable w is given.
- The convergence properties of the constrained PBFDAF algorithm are superior to time domain algorithms since normalization is performed at each frequency bin which eliminates the eigenvalue spread problem. Unconstrained PBFDAF suffer from eigenvalue spread within each bin due to the overlap in each FFT block which can be reduced by decreasing the overlap.
- Very efficient for long adaptive filters since convolution and correlation are performed in frequency domain. Maximum efficiency is obtained when the block length is chosen to be equal to the filter length $L = M$.
- PBFDAF introduces a processing delay between its input $x(n)$ and output $y(n)$ equals to the block length L, since the algorithm has to collect L samples before processing a block. This delay is however P times smaller than that introduced by BFDAF for the same total filter length.
- `asptpbfdaf` is optimized for block length equals to the partition length ($L = M$). For $L < M$ use `asptrcbfdaf()`.

Resources The resources required for direct implement of the PBFDAF algorithm in real time is given in the table below. The computations given are those required to process L samples using the constrained PBFDAF. Unconstrained PBFDAF uses $2P$ FFT operations less than the constrained PBFDAF. In the table below $C(FFT_B)$ is used to indicate the number of operations required to implement an FFT or IFFT of length $B = 2^{nextpow2(M+L-1)}$

MEMORY	$B(2P+4)+3L+3$
MULTIPLY	$6BP+(3+2P)C(FFT_B)$
ADD	$2BP+L+(3+2P)C(FFT_B)$
DIVIDE	$B+(3+2P)C(FFT_B)$

See Also INIT_ PBFDAF, ECHO_ PBFDAF, ASPTBFDAF, ASPTRCPBFDAF.

Reference [1] and [9] for detailed description of frequency domain adaptive filters.

4.13 asptrcpbfdaf

Purpose Block filtering and coefficient update in frequency domain using the Reduced Complexity Partitioned Block Frequency Domain (RCPBFDAF) algorithm.

Syntax `[W,X,x,y,e,Px,ci,w]=asptrcpbfdaf(M,x,xn,dn,X,W,mu,n,c,b,Px,ci)`

Description `asptrcpbfdaf()` is a reduced complexity and extended version of `asptpbfdaf()`. The computational complexity reduction is achieved by constraining one or more partition each call to `asptrcpbfdaf()` instead of constraining all partitions as in the case of `asptpbfdaf()`. This saves two FFT operations for each skipped partition while keeping the performance almost unaffected. No reduction in complexity is achieved for unconstrained filters. `asptrcpbfdaf()` is also designed to accommodate the general case of $M = g * L$, where M is the partition length, L is the block length, and g is an integer, which allows choosing L less than M to further reduce the processing delay. Similar to `asptpbfdaf()`, the adaptive filter is splitted into P partitions, from which only the first partition introduces a delay of L samples, and therefore, reducing the processing delay by a factor of P compared to the BFDAF for the same filter length. `asptrcpbfdaf()` performs filtering and coefficient update in the frequency domain using the overlap-save method, and therefore, provides an efficient implementation for long adaptive filters in applications such as acoustic echo cancelers where the adaptive filter can be a few thousand coefficients long. `asptrcpbfdaf()` is a block processing algorithm, every call processes L input and L desired samples (L is the block length), to produce L filter output samples and L error samples, besides updating all filter coefficients in the frequency domain. The parameters of `asptrcpbfdaf()` are summarized below (see Fig. 4.2).

```
Input Parameters [Size]::
   M  : partition length
   x  : previous overlap-save vector [B x 1]
   xn : new input block [L x 1]
   dn : new desired block [L x 1]
   X  : previous matrix of F-domain input samples [B x P]
   W  : previous matrix of F-domain filter coef. [B x P]
   mu : adaptation constant
   n  : normalization flag, 0 means no normalization
   c  : if 0, unconstrained PBFDAF is used,
        if -r (r +ve int), all partitions are constrained
        if +r (r +ve int), only r partitions are constrained
   b  : forgetting factor for input power estimation
   Px : previous estimate of the power of X [B x 1]
   ci : next partition to constrain
Output parameters::
   W  : updated filter coefficients (F-domain)
   X  : updated matrix of past frequency input samples
   x  : updated overlap-save input vector
   y  : filter output block
   e  : error vector block
   Px : updated estimate of the power of X
   ci : next partition to constrain
   w  : time domain filter (calculated only if required)
```

Example

```
iter = 5000;              % Number of samples to process
% Complex unknown impulse response
h    = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xt   = 2*(rand(iter,1)-0.5); % Input signal
% although xn is real, dn will be complex
dt   = osfilter(h,xt);        % Unknown filter output
en   = zeros(iter,1);         % estimation error
% Initialize RCPBFDAF with a filter of 2*4 coef.
P = 2; M = 4; L = M/2;
[W,x,d,e,y,Px,X,ci,w]=init_rcpbfdaf(L,M,P);
%% Processing Loop
for (m=1:L:iter-L)
   xn = xt(m:m+L-1,:);               % input block
   dn = dt(m:m+L-1,:)+ 1e-3*rand;  % desired block
   % call RCPBFDAF to calculate the filter output,
   % estimation error and update the filter coef.
   [W,X,x,y,e,Px,ci,w] = asptrcpbfdaf(M,x,xn,dn,...
                         X,W,0.06,1,1,0.98,Px,ci);

   % save the last error block to plot later
   en(m:m+L-1,:) = e;
end;
% display the results
subplot(2,2,1);stem([real(w) imag((w))]); grid;
subplot(2,2,2);
eb = filter(.1, [1 -.9], en(1:m) .* conj(en(1:m)));
plot(10*log10(eb ));grid
```

Running the above script will produce the graph shown in Fig. 4.14. The left side graph of the figure shows the adaptive filter coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB).

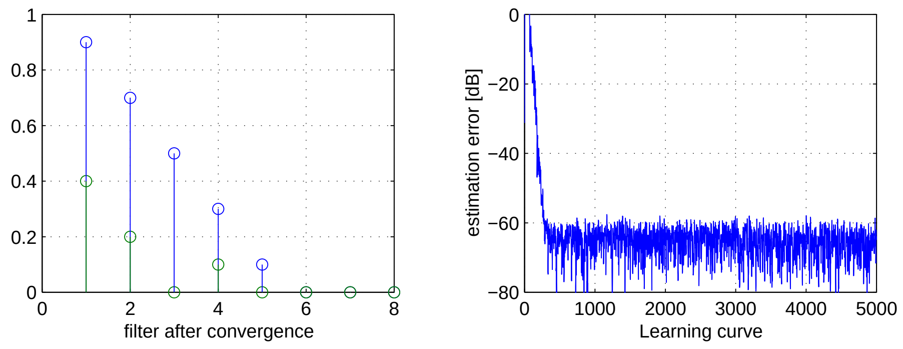

Figure 4.14: The adaptive filter coefficients after convergence and the learning curve for the complex FIR system identification problem using the RCPBFDAF algorithm.

Algorithm `asptrcpbfdaf()` performs the following operations (see Fig. 4.2).

- buffers L input samples, composes an overlap-save input vector $\underline{\mathbf{x}}(n)$ and computes its FFT, $\underline{\mathbf{X}}(f)$ and adds it to the input samples matrix $\mathbf{X}(f)$
- element-wise multiplies $\mathbf{X}(f)$ by the adaptive filter coefficients matrix $\mathbf{W}(f)$ and evaluates the sum at each frequency bin (circular convolution in time domain for the P partitions). The result is converted to time domain using IFFT and the linear convolution samples are extracted to produce the filter-output vector $\underline{\mathbf{y}}(n)$
- buffers L desired samples and evaluates the current error block $\underline{\mathbf{e}}(n) = \underline{\mathbf{d}}(n) - \underline{\mathbf{y}}(n)$. The error vector is padded with zeros and transformed to frequency domain giving $\underline{\mathbf{E}}(f)$
- estimates the input signal power at each frequency bin and normalizes the step size at each bin.
- evaluates the cross-correlation between $\mathbf{X}(f)$ and $\underline{\mathbf{E}}(f)$ to produce the block gradient vector. This vector is used to update the frequency domain filter coefficients.
- constrains only the partitioned given by the input argument c. This is performed by first taking the IFFT of the corresponding columns of $\mathbf{W}(f)$, applying a rectangular window on the time domain coefficients, and taking the FFT of the windowed coefficients.

Remarks

- Supports both real and complex signals.
- `asptrcpbfdaf()` constrains the filter coefficients $\mathbf{W}(f)$ rather than the gradient vector as in the official PBFDAF algorithms since this has been proven to result in a more stable update.
- The unconstrained RCPBFDAF (c = 0) saves $2P$ FFT operations each block on the cost of accuracy. Partially constrained filters ($c = r; r < P$) save $2(P - r)$ FFT operations each block.
- The time domain filter coefficients $\underline{\mathbf{w}}(n)$ will be calculated and returned by `asptrcpbfdaf()` only if the output variable w is given.
- The convergence properties of the constrained RCPBFDAF algorithm are superior to time domain algorithms since normalization is performed at each frequency bin which eliminates the eigen value spread problem. Unconstrained RCPBFDAF suffers from eigenvalue spread within each bin due to the overlap in each FFT block which can be reduced by decreasing the overlap.
- `asptrcpbfdaf()` introduces a processing delay between its input $x(n)$ and output $y(n)$ equals to the block length L, since the algorithm has to collect L samples before processing a block. This delay is however P times smaller than that introduced by BFDAF for the same total filter length.
- Very efficient for long adaptive filters since convolution and correlation are performed in frequency domain. Maximum efficiency is obtained when the block length is chosen to be equal to the filter length $L = M$.

Resources The resources required for direct implement of the RCPBFDAF algorithm in real time is given in the table below. The computations given are those required to process L samples using the fully constrained RCPBFDAF. Partially constrained RCPBFDAF saves 2 FFT operations for each skipped partition, for instance calling `asptrcpbfdaf` with P=16 and c=1 saves 30 FFT operations every L samples, which can be huge saving for filters composed of many partitions. In the table below $C(FFT_B)$ is used to indicate the number of operations required to implement an FFT or IFFT of length $B = 2^{nextpow2(M+L-1)}$

MEMORY	$B(G+P+4)+3L+4$
MULTIPLY	$6BP+(3+2P)C(FFT_B)$
ADD	$2BP+L+(3+2P)C(FFT_B)$
DIVIDE	$B+(3+2P)C(FFT_B)$

See Also INIT_ RCPBFDAF, ECHO_ RCPBFDAF, ASPTPBFDAF, ASPTBFDAF.

Reference [1] and [9] for detailed description of frequency domain adaptive filters.

4.14 asptrdrlms

Purpose Performs filtering and coefficient update using the Recent Data Reusing Least Mean Squares (RDRLMS) algorithm. RDRLMS updates the filter coefficients k times each iteration using the last k input and desired data sets to speed the convergence process.

Syntax

```
[w,y,e] = asptrdrlms(x,w,d,mu)
[w,y,e] = asptrdrlms(x,w,d,mu,alg)
```

Description `asptrdrlms()` improves the convergence speed of the LMS algorithm by updating the filter coefficients several times using the last few sets of input and desired data. When the number of data using cycles, $k = 0$, RDRLMS falls back to the LMS algorithm. Unlike the DRLMS which uses the current data set of input and desired signals, RDRLMS uses the current and past k data sets to update the filter coefficients. The coefficients update is performed according to the 'alg' input argument which can take any of the following values.

- `'lms'` : the default value, uses the LMS algorithm
- `'slms'` : uses the sign LMS algorithm, the sign of the error $e(k)$ is used in the update equation instead of the error.
- `'srlms'` : uses the signed regressor LMS algorithm, the sign of the input signal $x(k)$ is used in the update equation instead of the input signal.
- `'sslms'` : uses the sign-sign-LMS algorithm, the sign of the error $e(k)$ and the sign of the input signal $x(k)$ are used in the update equation instead of the error and the input signals.

The input and output parameters of `asptrdrlms()` for an FIR adaptive filter of L coefficients are summarized below.

```
Input Parameters ::
   x   : vector of input samples x(n)
   w   : vector of filter coefficients w(n-1)
   d   : desired output d(n)
   mu  : adaptation constant
   alg : specifies the variety of the lms to use in the
         update equation. Must be one of the following:
         'lms'     [default]
         'slms'  - sign LMS, uses sign(e)
         'srlms' - signed regressor LMS, uses sign(x)
         'sslms' - sign-sign LMS, uses sign(e) and sign(x)
Output parameters ::
   w   : updated filter coefficients w(n)
   y   : filter output y(n)
   e   : error signal; e(n) = d(n) - y(n)
```

Example

```
% RDRLMS used in a simple system identification application.
% The learning curves of the RDRLMS is compared for several
% values of the number of data reusing cycles.
iter = 5000;                    % Number of samples to process
% Complex unknown impulse response
h    = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xn   = 2*(rand(iter,1)-0.5);    % Input signal, zero mean random.
xn   = filter(.05,[1 -.95], xn); % colored input
dn   = osfilter(h,xn);          % Unknown filter output
M    = [0, 2, 4, 8];            % data reusing cycles
en   = zeros(iter,length(M)); % vector to collect the error

%% Processing Loop
for n = 1:length(M)
  % Initialize the DRLMS algorithm with a filter of 10 coef.
   [w,x,d,y,e]=init_rdrlms(10,M(n));
  for (m=1:iter)
    x = [xn(m,:); x(1:end-1,:)];  % update the input delay line
    d = [dn(m,:) + 1e-3*rand; d(1:end-1)];
    w,y,e]= asptrdrlms(x,w,d,.2,'lms');
    en(m,n) = e(1);                 % save the last error
  end;
end;

% display the results
subplot(2,2,1);stem([real(w) imag(conj(w))]); grid;
subplot(2,2,2);eb = filter(0.1, [1 -.9] , en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 4.15. The left side graph of the figure shows the adaptive filter coefficients after convergence. The right side graph shows the learning curve for RDRLMS for $k = \{0, 2, 4, 8\}$. The case of $k = 0$ is equivalent to the LMS algorithm and is included as a reference. Fig. 4.15 suggests that the convergence speed improves as k increases.

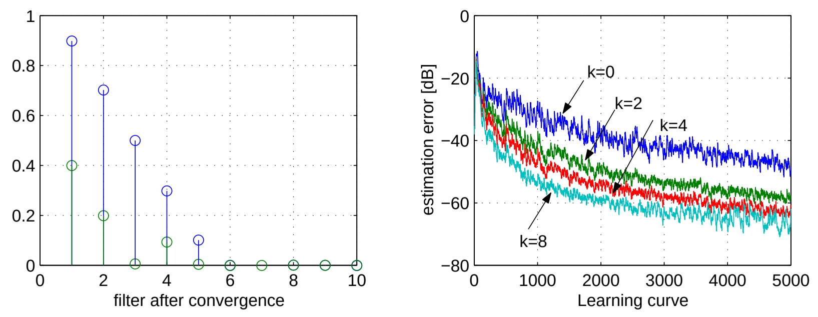

Figure 4.15: The adaptive filter coefficients after convergence and the learning curve for the complex FIR system identification problem using the RDRLMS for several values of the data reusing parameter k.

Algorithm The current implementation of `asptrdrlms()` performs the following operations

- Filters the input signal through the adaptive filter $w(n-1)$ to produce the filter's output sample $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$.
- Updates the adaptive filter coefficients k times according to the 'alg' input parameter.

Remarks

- RDRLMS improves the convergence speed of the LMS by updating the filter coefficients more frequently, and therefore consumes more processor cycles.
- The RDRLMS shows similar convergence properties to those known for the LMS algorithm.
- `asptrdrlms()` supports both real and complex data and filters. The adaptive filter for the complex RDRLMS algorithm converges to the complex conjugate of the optimum solution.
- `asptrdrlms()` does not update the input delay line for $x(n)$, this has been chosen to provide more flexibility, so that the same function can be used with transversal as well as linear combiner structures. Delay line update, by inserting the newest sample at the beginning of the buffer and shifting the rest of the samples to the right, has to be done before calling `asptrdrlms()` as in the example above.

Resources The resources required to implement the RDRLMS algorithm for a transversal adaptive FIR filter of L coefficients and k data reusing cycles in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$2L + 2k + 4$
MULTIPLY	$(2L + 1) * [k + 1]$
ADD	$2L[k + 1]$
DIVIDE	0

See Also INIT_ RDRLMS, ASPTLMS, ASPTDRLMS, ASPTRDRNLMS.

Reference [11] and [4] for extensive analysis of the LMS and the steepest-descent search method and [7] for an introduction to the RDRLMS.

4.15 asptrdrnlms

Purpose

Performs filtering and coefficient update using the Recent Data Reusing Normalized Least Mean Squares (RDRNLMS) algorithm. RDRNLMS updates the filter coefficients k times each iteration using the last k input and desired data sets to speed the convergence process.

Syntax

```
[w,y,e,p] = asptrdrnlms(x,w,d,mu,p)
[w,y,e,p] = asptrdrnlms(x,w,d,mu,p,b,k)
```

Description

`asptrdrnlms()` improves the convergence speed of the NLMS algorithm by updating the filter coefficients several times using the past few sets of input and desired data. When the number of data reusing cycles, $k = 0$, RDRNLMS falls back to the NLMS algorithm. Unlike the DRNLMS which uses the current data set of input and desired signals, RDRNLMS uses the current and past k data sets to update the filter coefficients.
The input and output parameters of `asptrdrnlms()` for an FIR adaptive filter of L coefficients are summarized below.

```
Input Parameters ::
   x    : input samples delay line
   w    : filter coefficients vector w(n-1)
   d    : desired output d(n)
   mu   : adaptation constant
   p    : last estimated power of x, p(n-1)
   b    : AR pole for recursive calculation of p
   k    : number of data reusing cycles
Output parameters::
   w    : updated filter coefficients w(n)
   y    : filter output y(n)
   e    : error signal; e(n) = d(n)-y(n)
   p    : new estimated power of x, p(n)
```

Example

```
% RDRNLMS used in a simple system identification application.
% The learning curves of the RDRNLMS is compared for several
% values of the number of data reusing cycles.

iter = 5000;                    % Number of samples to process
% Complex unknown impulse response
h    = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xn   = 2*(rand(iter,1)-0.5);  % Input signal, zero mean random.
xn   = filter([.05],[1 -.95], xn);
dn   = osfilter(h,xn);          % Unknown filter output
M    = [0, 2, 4, 8];            % data reusing cycles
en   = zeros(iter,length(M)); % vector to collect the error

%% Processing Loop
for n = 1:length(M)
  % Initialize the RDRNLMS algorithm with a filter of 10 coef.
  [w,x,d,y,e,p]=init_rdrnlms(10,M(n));
  for (m=1:iter)
    x = [xn(m,:); x(1:end-1,:)];  % update the input delay line
    d = [dn(m,:) + 1e-3*rand; d(1:end-1)];
    [w,y,e,p]= asptrdrnlms(x,w,d,.005,p,.98);
    en(m,n) = e(1);
  end;
end;

% display the results
subplot(2,2,1);stem([real(w) imag(conj(w))]); grid;
subplot(2,2,2);eb = filter(0.1, [1 -.9] , en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 4.16. The left side graph of the figure shows the adaptive filter coefficients after convergence. The right side graph shows the learning curve for RDRNLMS for $k = \{0, 2, 4, 8\}$. The case of $k = 0$ is equivalent to the NLMS algorithm and is included as a reference. Fig. 4.16 suggests that the convergence speed improves as k increases.

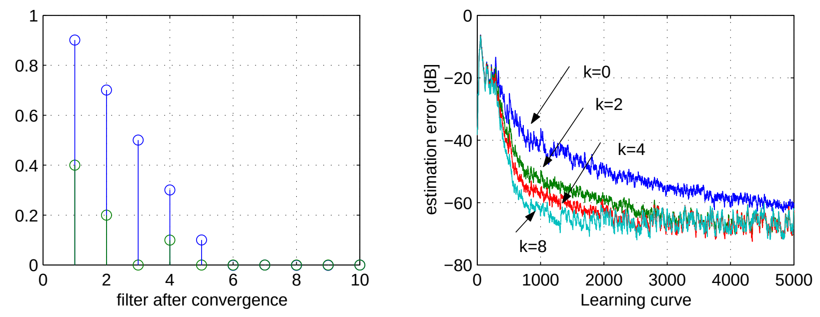

Figure 4.16: The adaptive filter coefficients after convergence and the learning curve for the complex FIR system identification problem using the RDRNLMS for several values of the data reusing parameter k.

Algorithm The current implementation of `asptrdrnlms()` performs the following operations

- Filters the input signal through the adaptive filter $w(n-1)$ to produce the filter's output sample $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$.
- Updates the estimate of the input signal power, p.
- Updates the adaptive filter coefficients k times.

Remarks

- RDRNLMS improves the convergence speed of the NLMS by updating the filter coefficients more frequently, and therefore consumes more processor cycles.
- The RDRNLMS shows similar convergence properties to those known for the NLMS algorithm.
- `asptrdrnlms()` supports both real and complex data and filters. The adaptive filter for the complex RDRNLMS algorithm converges to the complex conjugate of the optimum solution.
- `asptrdrnlms()` does not update the input delay line for $x(n)$, this has been chosen to provide more flexibility, so that the same function can be used with transversal as well as linear combiner structures. Delay line update, by inserting the newest sample at the beginning of the buffer and shifting the rest of the samples to the right, has to be done before calling `asptrdrnlms()` as in the example above.

Resources The resources required to implement the RDRNLMS algorithm for a transversal adaptive FIR filter of L coefficients and k data reusing cycles in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$2L + 2k + 6$
MULTIPLY	$(2L + 1)[k + 1] + 4$
ADD	$(2L)[k + 1] + 1$
DIVIDE	k+1

See Also INIT_ RDRNLMS, ASPTNLMS, ASPTDRNLMS, ASPTRDRLMS.

Reference [11] and [4] for extensive analysis of the LMS and the steepest-descent search method and [7] for an introduction to the RDRNLMS.

4.16 asptrls

Purpose Sample per sample filtering and coefficient update using the Recursive Least Squares (RLS) Adaptive algorithm.

Syntax `[w,y,e,R]=asptrls(x,w,d,R,a)`

Description `asptrls()` implements the recursive least squares adaptive algorithm used to update transversal adaptive filters. Referring to the general adaptive filter shown in Fig. 2.6, `asptrls()` takes an input samples delay line $x(n)$, a desired sample $d(n)$, the vector of the adaptive filter coefficients from previous iteration $w(n-1)$, the estimate of the inverse correlation matrix from previous iterations $R(n-1)$, the forgetting factor a, and returns the filter output $y(n)$, the error sample $e(n)$, the updated vector of filter coefficients $w(n)$, and the updated matrix $R(n)$. The update equation of `asptrls()` is given by

$$\underline{\mathbf{w}}(n) = \underline{\mathbf{w}}(n-1) + \frac{\mathbf{R}(n-1)\ \underline{\mathbf{x}}(n)\ e(n)}{a +\ \underline{\mathbf{x}}^T(n)\ \mathbf{R}(n-1)\underline{\mathbf{x}}(n)}. \tag{4.10}$$

The input and output parameters of `asptrls()` for an FIR adaptive filter of L coefficients are summarized below.

```
Input Parameters [Size]::
   x   : vector of input samples at time n, [L x 1]
   w   : vector of filter coefficients w(n-1), [L x 1]
   d   : desired response d(n), [1 x 1]
   R   : last estimate of the inverse of the weighted
         auto correlation matrix of x, [L x L]
   a   : forgetting factor, [1 x 1]
Output parameters::
   w   : updated filter coefficients w(n)
   y   : filter output y(n)
   e   : error signal, e(n)=d(n)-y(n)
   R   : updated R
```

Example

```
% RLS used in a simple system identification application.
% By the end of this script the adaptive filter w
% should have the same coefficients as the unknown filter h.
iter = 5000;                  % Number of samples to process
% Complex unknown impulse response
h    = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xn   = 2*(rand(iter,1)-0.5);  % Input signal, zero mean random.
% although xn is real, dn will be complex since h is complex
dn   = osfilter(h,xn);        % Unknown filter output
en   = zeros(iter,1);         % vector to collect the error
% Initialize RLS with a filter of 10 coef.
[w,x,d,y,e,R]=init_rls(10,0.1);

%% Processing Loop
for (m=1:iter)
   % update the input delay line
   x = [xn(m,:); x(1:end-1,:)];
```

```
    d = dn(m,:) + 1e-3*rand;      % additive noise of var = 1e-6
    % call RLS to calculate the filter output, estimation error
    % and update the filter coefficients.
    [w,y,e,R]=asptrls(x,w,d,R,0.98);
    % save the last error sample to plot later
    en(m,:) = e;
end;

% display the results
subplot(2,2,1);stem([real(w) imag(conj(w))]); grid;
subplot(2,2,2);
eb = filter(.1, [1 -.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 4.17. The left side graph of the figure shows the adaptive filter coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the mean square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error power is governed here by the additive noise at the output (-60 dB).

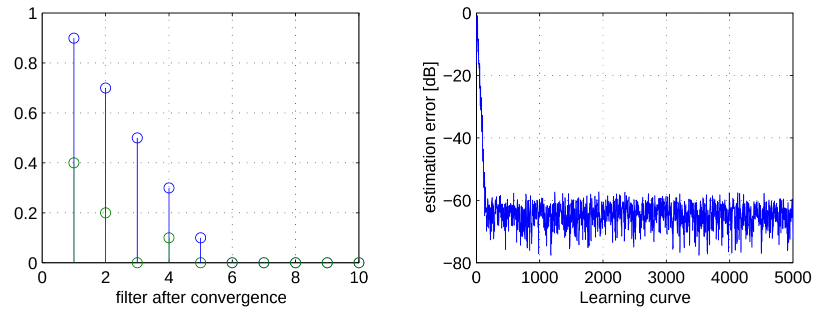

Figure 4.17: The adaptive filter coefficients after convergence and the learning curve for the complex FIR system identification problem using the RLS algorithm.

Remarks

- The closer the value of the forgetting factor λ to one, the longer the memory of the algorithm becomes. Roughly, the algorithm will take into account up to $1/(1-\lambda)$ past samples.
- The RLS algorithm has only one convergence mode, and does not suffer from the eigenvalue spread problem as in the LMS and its variants. In general, the RLS converges within $2L$ to $3L$ samples, where L is the filter length, and therefore very suitable for tracking applications
- `asptrls()` supports both real and complex data and filters. The adaptive filter for the complex RLS algorithm converges to the complex conjugate of the optimum solution.
- `asptrls()` does not update the input delay line for $x(n)$, this has been chosen to provide more flexibility, so that the same function can be used with transversal as well as linear combiner structures. Delay line update, by inserting the newest sample at the beginning of the buffer and shifting the rest of the samples to the right, has to be done before calling `asptrls()` as in the example above.

Algorithm

Unlike the LMS and its derivatives which use statistical (expected values) approach, the RLS is a deterministic algorithm based only on observed data. The practical implementation of the RLS algorithm adjusts the coefficients of an adaptive filter to minimize the following quantity

$$\xi(n) = \lambda^{n-k} e^2(n), \tag{4.11}$$

where $e(n)$ is the error signal and λ is a positive constant close to but less than one usually called the forgetting factor. This choice for the $\xi(n)$ in (4.11) puts more emphasis on recent observed data samples and exponentially less emphasis on past samples. The current implementation of `asptrls()` performs the following operations

- Filters the input signal $x(n)$ through the adaptive filter $w(n-1)$ to produce the filter output $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$.
- Recursively updates the gain vector $\underline{\mathbf{K}}(n)$ given by

$$\underline{\mathbf{K}}(n) = \frac{\mathbf{R}(n-1)\ \underline{\mathbf{x}}(n)}{a +\ \underline{\mathbf{x}}^T(n)\ \mathbf{R}(n-1)\underline{\mathbf{x}}(n)}. \tag{4.12}$$

- Updates the adaptive filter coefficients according to (4.10)

Resources

The resources required to implement the RLS algorithm for a transversal adaptive FIR filter of L coefficients in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$L^2 + 2L + 4$
MULTIPLY	$2L^2 + 4L$
ADD	$1.5L^2 + 2.5L$
DIVIDE	L

See Also

INIT_ RLS, EQUALIZER_ RLS.

Reference

[2] and [4] for analysis of the RLS algorithm and its variants.

4.17 asptdftaf

Purpose Performs sample-per-sample filtering and coefficient update using the Transform Domain Fault Tolerant Adaptive Filter (TDFTAF) algorithm. TDFTAF contains redundant filter coefficients to improve the filter robustness against partial hardware failure during operation.

Syntax

```
[W,y,e,p,w] = aspttdftaf(x,W,d,mu,p,b,T)
```

Description Fault tolerant adaptive filters address the issue of robustness against hardware failure. When a hardware failure occurs during the operation of a non fault tolerant adaptive filter, the filter diverges and will never converge again, unless special measures are taken to guarantee recovery. Fault tolerant adaptive filters makes sure that the filter can recover as quickly as possible after the occurrence of a hardware failure. The `aspttdftaf()` algorithm achieves hardware fault tolerance by introduces one or more redundant filter coefficients that will allow the filter to quickly recover to the optimal solution after the occurrence of a hardware failure in the underlying hardware running the filter. The type of hardware failure usually encountered in practice is partial memory failure. This type of hardware failure makes filter coefficients stored at the faulty memory locations appear to remain at an arbitrary constant value.

Similar to the `aspttdlms()`, `aspttdftaf()` implements the Transform Domain LMS adaptive algorithm used to update transversal adaptive filters. The algorithm performs filtering and coefficient update in the transform domain, T. Normalization of the step size by the input signal power is also performed in each band in the T-domain, which usually improves the convergence behavior compared to the conventional LMS when T is an orthogonal transformation. The only difference between TDFTAF and TDLMS is that the former updates $L+R$ filter coefficients, where L is the number of original filter coefficients and R is the number of redundant coefficients.

The block diagram of the TDFTAF is shown in Fig. 4.18. `aspttdftaf()` takes an input samples delay line $\underline{\mathbf{x}}(n)$ and applies the transformation T on this vector. This T-domain data vector is filtered through the vector of T-domain adaptive filter coefficients from previous iteration $\underline{\mathbf{W}}(n-1)$ to produce the time-domain filter output $y(n)$. The error $e(n) = d(n) - y(n)$ is then calculated and the power of the input vector is estimated in the T-domain at each band and used to normalize the step size. The update equation used by `aspttdftaf()` to update each T-domain coefficient is given by

$$W_i(n) = W_i(n-1) + \frac{\mu}{P_i(n)} e(n) X_i(n); \quad i = 0, 1, \cdots, L+R-1. \tag{4.13}$$

Where $W_i(n)$, $X_i(n)$, and $P_i(n)$ are the filter coefficient, input signal, and input signal power in band i at time index n, respectively.

The input and output parameters of `aspttdftaf()` for an FIR adaptive filter of L coefficients are summarized below.

```
Input Parameters [Size]::
  x   : input samples delay line [L+R x 1]
  W   : previous T-domain coef. vector W(n-1) [L+R x 1]
  d   : desired output d(n) [1 x 1]
  mu  : adaptation constant
  p   : last estimated power of x p(n-1) [L x 1]
  b   : AR pole for recursive calculation of p
  T   : The transform to be used {fft|dct|dst|...}
        user defined transforms are also supported.
        use transform T and its inverse iT.
Output parameters::
  W   : updated T-domain coef. vector
  y   : filter output y(n)
  e   : error signal; e(n) = d(n)-y(n)
  p   : new estimated power of x p(n)
  w   : updated t-domain coef. vector w(n), only
        calculated if this output argument is given.
```

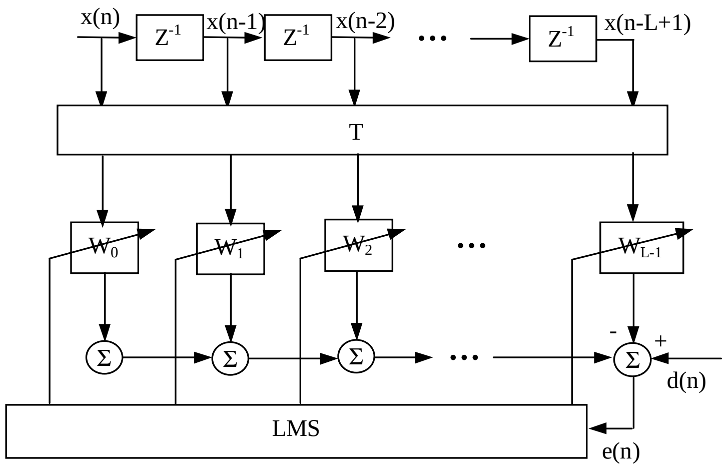

Figure 4.18: Block diagram of the Transform Domain Fault Tolerant Adaptive Filter.

Example

```
% TDFTAF used in a simple system identification application.
% During simulation two coefficients are fixed at arbitrary
% values to simulate a hardware failiar.
%
iter = 5000;                  % Number of samples to process
% Complex unknown impulse response
h    = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xn   = 2*(rand(iter,1)-0.5);  % Input signal, zero mean random.
% although xn is real, dn will be complex since h is complex
dn   = osfilter(h,xn);        % Unknown filter output
en   = zeros(iter,1);         % vector to collect the error
% Initialize the TDFTAF with a filter of 5 coef. and 3 redundant
[W,w,x,d,y,e,p]=init_tdftaf(5,3);
% Initialize a TDLMS filter for comparizon
[W1,w1,x,d,y1,e1,p1]=init_tdlms(5);
```

```
%% Processing Loop
for (m=1:iter)
   x = [xn(m,:); x(1:end-1,:)];  % update the input delay line
   d = dn(m,:) + 1e-3*rand;      % additive noise of var = 1e-6
   % call TDFTAF and TDLMS to calculate the filter output,
   % estimation error and update the coefficients.
   [W,y,e,p,w]     = asptdftaf(x,W ,d,0.05,p ,0.98,'fft');
   [W1,y1,e1,p1,w1] = asptdlms(x,W1,d,0.05,p1,0.98,'fft');
   % save the last error sample to plot later
   en(m,:)  = e;  en1(m,:) = e1;
   if (m > 2000), W(3) = 0.0; W1(3) = 0.0; end  % memory failiar
   if (m > 3000), W(4) = 1.0; W1(4) = 1.0; end  % memory failiar
end;
% display the results
eb  = filter(0.1, [1 -0.9], en  .* conj(en ));
eb1 = filter(0.1, [1 -0.9], en1 .* conj(en1));
subplot(2,2,1);plot(10*log10(eb1  )); grid
subplot(2,2,2); plot(10*log10(eb  )); grid
```

Running the above script will produce the graph shown in Fig. 4.19. The left side graph of the figure shows the learning curve of the TDLMS and the right side graph shows the learning curve of the TDFTAF. It is clear that the TDFTAF can recover after a failure while the TDLMS can not.

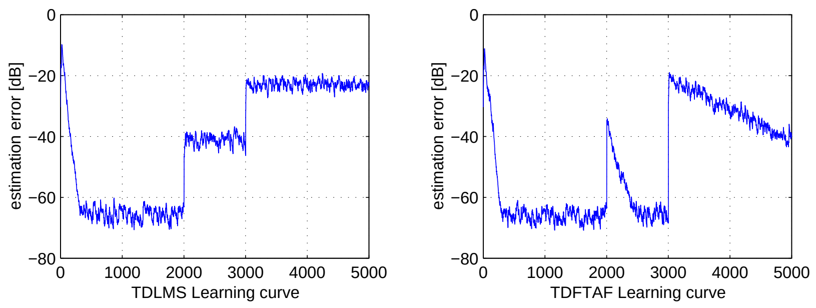

Figure 4.19: Learning curves for the TDLMS and TDFTAF when hardware failure is encountered.

Remarks

- `asptdftaf()` supports both real and complex data and filters. The adaptive filter for the complex TDFTAF algorithm converges to the complex conjugate of the optimum solution.
- `asptdftaf()` does not update the input delay line for $x(n)$, this has been chosen to provide more flexibility, so that the same function can be used with transversal as well as linear combiner structures. Delay line update, by inserting the newest sample at the beginning of the buffer and shifting the rest of the samples to the right, has to be done before calling `asptdftaf()` as in the example above.
- `asptdftaf()` not only supports standard transformations such as FFT, DCT, and DST, but also user-defined transformations. To use this feature, provide your transformation in two separate functions, one for the forward and the other for the backward transformation. For example if you implement a forward transformation in the file xyz.m you should implement its inverse transformation in the file ixyz.m and call `asptdftaf` with parameter T='xyz'. Care should be taken in scaling the transformation coefficients to ensure that the time-domain filter coefficients have the correct values.

Algorithm

`asptdftaf()` performs the following operations

- Calculates the transformation of $\underline{\mathbf{x}}(n)$ and filters this through the filter coefficient vector $\underline{\mathbf{W}}(n-1)$ to produce the filter output $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$ and the input power vector $\underline{\mathbf{P}}(n)$
- Calculates the updated T-domain adaptive coefficients $\underline{\mathbf{W}}(n)$ and their inverse transformation $\underline{\mathbf{w}}(n)$ if required.

Resources

The resources required to implement the TDFTAF algorithm for a transversal adaptive FIR filter of M coefficients, where $M = L + R$, in real time is given in the table below. The computations given are those required to process one sample. The complexity of the transformation T of length M is indicated as $C(T)$ in the table below.

MEMORY	$4M + 4$
MULTIPLY	$6M + C(T)$
ADD	$4M + C(T)$
DIVIDE	$M + C(T)$

See Also

INIT_ TDFTAF, ASPTTDLMS.

Reference

[7] for an introduction to fault tolerant adaptive filters.

4.18 asptdlms

Purpose Sample per sample filtering and coefficient update using the Transform Domain LMS algorithm. Filtering and coefficient update are performed in T-domain.

Syntax `[W,w,y,e,p] = asptdlms(x,W,d,mu,p,b,T)`

Description `asptdlms()` implements the Transform Domain LMS adaptive algorithm used to update transversal adaptive filters. TDLMS performs filtering and coefficient update in the transform domain, T. Normalization of the step size by the input signal power is also performed in T-domain in each band, which is usually improves the convergence behavior compared to the conventional LMS when T is an orthogonal transformation.
The block diagram of the TDLMS is shown in Fig. 4.20. `asptdlms()` takes an input samples delay line $\underline{\mathbf{x}}(n)$ and applies the transformation T on this vector. This T-domain data vector is filtered through the vector of T-domain adaptive filter coefficients from previous iteration $\underline{\mathbf{W}}(n-1)$ to produce the time-domain filter output $y(n)$. The error $e(n) = d(n) - y(n)$ is then calculated and the power of the input vector is estimated in the T-domain at each band and used to normalize the step size. The update equation used by `asptdlms()` to update each T-domain coefficient is given by

$$W_i(n) = W_i(n-1) + \frac{\mu}{P_i(n)} e(n) X_i(n); \quad i = 0, 1, \cdots, L-1. \tag{4.14}$$

Where $W_i(n)$, $X_i(n)$, and $P_i(n)$ are the filter coefficient, input signal, and input signal power in band i at time index n, respectively.
The input and output parameters of `asptdlms()` for an FIR adaptive filter of L coefficients are summarized below.

```
Input Parameters [Size]::
   x    : input samples delay line [L x 1]
   W    : previous T-domain coef. vector W(n-1) [L x 1]
   d    : desired output d(n) [1 x 1]
   mu   : adaptation constant
   p    : last estimated power of x, p(n-1) [L x 1]
   b    : AR pole for recursive calculation of p
   T    : The transform to be used {fft|dct|dst|...}
          user defined transforms are also supported.
          use transform T and its inverse iT.
Output parameters::
   W    : updated T-domain coef. vector
   y    : filter output y(n)
   e    : error signal; e(n) = d(n)-y(n)
   p    : new estimated power of x, p(n)
   w    : updated time-domain coef. vector w(n), only
          calculated if this output argument is given.
```

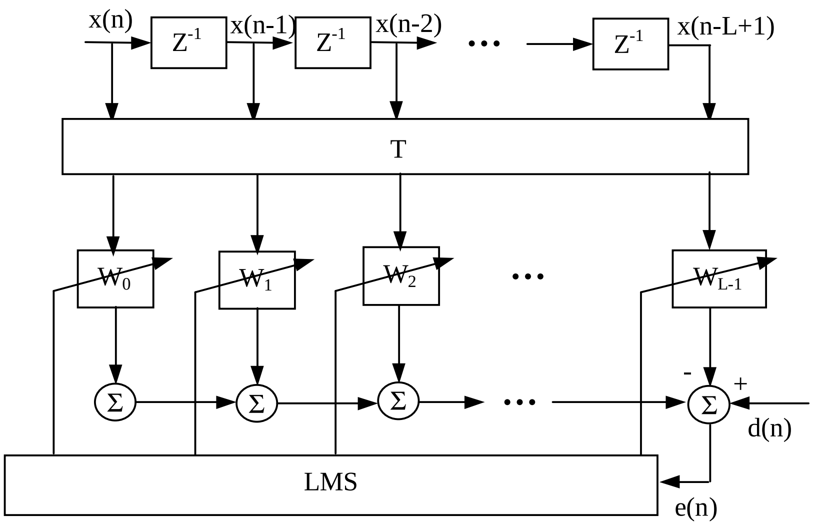

Figure 4.20: Block diagram of the Transform Domain LMS algorithm.

Example

```
% TDLMS used in a simple system identification application.
% By the end of this script the adaptive filter w should
% have the same coefficients as the unknown filter h.
%
iter = 5000;                % Number of samples to process
% Complex unknown impulse response
h    = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xn   = 2*(rand(iter,1)-0.5);  % Input signal, zero mean random.
% although xn is real, dn will be complex since h is complex
dn   = osfilter(h,xn);       % Unknown filter output
en   = zeros(iter,1);        % vector to collect the error
% Initialize the TDLMS algorithm with a filter of 10 coef.
[W,w,x,d,y,e,p]=init_tdlms(10);

%% Processing Loop
for (m=1:iter)
   x = [xn(m); x(1:end-1)];  % update the input delay line
   d = dn(m,:) + 1e-3*rand;  % additive noise of var = 1e-6
   % call TDLMS to calculate the output, estimation error
   % and update the coefficients.
   [W,y,e,p,w] = asptdlms(x,W,d,0.05,p,0.98,'fft');
   % save the last error sample to plot later
   en(m) = e;
end;

% display the results
% note that w converges to conj(h) for complex data
subplot(2,2,1);stem([real(w) imag(conj(w))]); grid;
subplot(2,2,2);eb = filter(.1, [1 -.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 4.21. The left side graph of the figure shows the adaptive filter coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB).

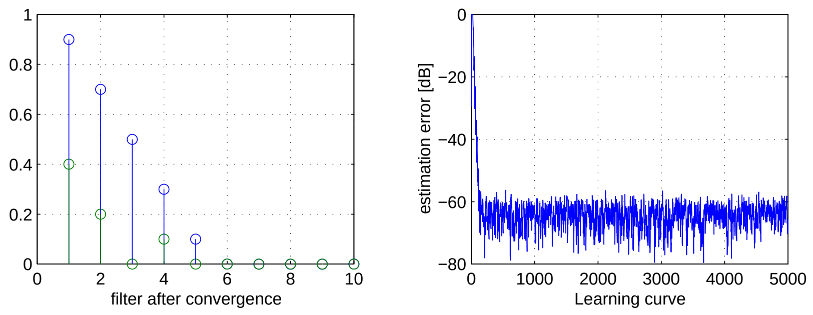

Figure 4.21: The adaptive filter coefficients after convergence and the learning curve for the complex FIR system identification problem using the TDLMS algorithm.

Remarks

The TDLMS algorithm solves the eigenvalue spread problem by first decorrelating the input signal samples using the transformation T and then normalizing the transformed data by its power in each band. This is equivalent to using a time varying step size in each band, the value of which is inversely proportional to the power of the input in this band. This will speed the convergence of the slow modes (those excited with relatively small input power) and improves the total convergence behavior of the adaptive filter. Note also that

- `asptdlms()` supports both real and complex data and filters. The adaptive filter for the complex TDLMS algorithm converges to the complex conjugate of the optimum solution.
- `asptdlms()` does not update the input delay line for $x(n)$, this has been chosen to provide more flexibility, so that the same function can be used with transversal as well as linear combiner structures. Delay line update, by inserting the newest sample at the beginning of the buffer and shifting the rest of the samples to the right, has to be done before calling `asptdlms()` as in the example above.
- `asptdlms()` not only supports standard transformations such as FFT, DCT, and DST, but also user-defined transformations. To use this feature, provide your transformation in two separate functions, one for the forward and the other for the backward transformation. For example if you implement a forward transformation in the file xyz.m you should implement its inverse transformation in the file ixyz.m and call `asptdlms` with parameter T='xyz'. Care should be taken in scaling the transformation coefficients to ensure that the time-domain filter coefficients have the correct values.
- The TDLMS is equivalent to an efficient implementation for the LMS-Newton algorithm when T is given by the Karhunen Loéve Transformation (KLT).

Algorithm `aspttdlms()` performs the following operations

- Calculates the transformation of the $\underline{\mathbf{x}}(n)$ and filters this through the filter coefficient vector $\underline{\mathbf{W}}(n-1)$ to produce the filter output $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$ and the input power vector $\underline{\mathbf{P}}(n)$
- Calculates the updated T-domain adaptive coefficients $\underline{\mathbf{W}}(n)$ and their inverse transformation $\underline{\mathbf{w}}(n)$ if required.

Resources The resources required to implement the TDLMS algorithm for a transversal adaptive FIR filter of L coefficients in real time is given in the table below. The computations given are those required to process one sample. The complexity of the transformation T is indicated as $C(T)$ in the table below.

MEMORY	$4L+4$
MULTIPLY	$6L+C(T)$
ADD	$4L+C(T)$
DIVIDE	$L+C(T)$

See Also INIT_ TDLMS, MODEL_ TDLMS, ASPTLMS, ASPTNLMS, ASPTVSSLMS.

Reference [11], [4], and [2] for extensive analysis of the LMS and the steepest-descent search method.

4.19 asptvffrls

Purpose Performs filtering and coefficient update using the Variable Forgetting Factor Recursive Least Squares (VFFRLS) Adaptive algorithm.

Syntax `[w,y,e,R,k,a] = asptvffrls(x,w,d,R,a,k,e,roh,a_min,a_max)`

Description `asptvffrls()` is an improved version of the conventional RLS algorithm optimized for tracking applications. VFFRLS not only optimizes the filter coefficients but also simultaneously optimizes the forgetting factor parameter for stability and fast tracking in a similar manner as performed in the variable step size LMS algorithm.
The input and output parameters of `asptvffrls()` of length L are summarized below.

```
Input Parameters [Size]::
   x     : vector of input samples at time n, [L x 1]
   w     : vector of filter coefficients w(n-1), [L x 1]
   d     : desired response d(n), [1 x 1]
   R     : last estimate of the inverse of the weighted
           auto-correlation matrix of x, [L x L]
   a     : forgetting factor, [1 x 1]
   k     : last gain vector
   e     : last error sample
   roh   : forgetting factor step size [1 x 1]
   a_min : lower bound for the forgetting factor [1 x 1]
   a_max : higher bound for forgetting factor  [1 x 1]

Output parameters::
   w   : updated filter coefficients w(n)
   y   : filter output y(n)
   e   : error signal, e(n)=d(n)-y(n)
   R   : updated R
```

Example

```
% VFFRLS used in a simple system identification application.
% By the end of this script the adaptive filter w
% should have the same coefficients as the unknown filter h.
iter = 5000;             % Number of samples to process
% Complex unknown impulse response
h    = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xn   = 2*(rand(iter,1)-0.5);  % Input signal, zero mean random.
% although xn is real, dn will be complex since h is complex
dn   = osfilter(h,xn);    % Unknown filter output
en   = zeros(iter,1);     % vector to collect the error
a    = 0.9;               % initial forgetting factor
av   = zeros(iter,1);     % storing changes in a

% Initialize the VFFRLS algorithm with a filter of 10 coef.
[w,x,d,y,e,R,k] = init_vffrls(10,.1);
```

```
%% Processing Loop
for (m=1:iter)
   % update the input delay line
   x = [xn(m,:); x(1:end-1,:)];
   d = dn(m,:) + 1e-3*rand;      % additive noise of var = 1e-6
   % call VFFRLS to calculate the filter output, estimation error
   % and update the coefficients.
   [w,y,e,R,k,a]=asptvffrls(x,w,d,R,a,k,e,.01,.8,1-.0001);


   en(m,:) = e;       % save the last error sample
   av(m)   = a;       % save the last value of a
end;

% display the results
subplot(3,3,1);stem([real(w) imag(conj(w))]); grid;
eb = filter(0.1,[1 -0.9], en .* conj(en));
subplot(3,3,2); plot(10*log10(eb ) );grid
subplot(3,3,3);plot(av); grid;
```

Running the above script will produce the graph shown in Fig. 4.22. The left side graph of the figure shows the adaptive linear combiner coefficients after convergence which are almost identical to the unknown filter h. The middle graph shows the square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB). The right side graph shows the evolution of the forgetting factor with time during the adaptation process.

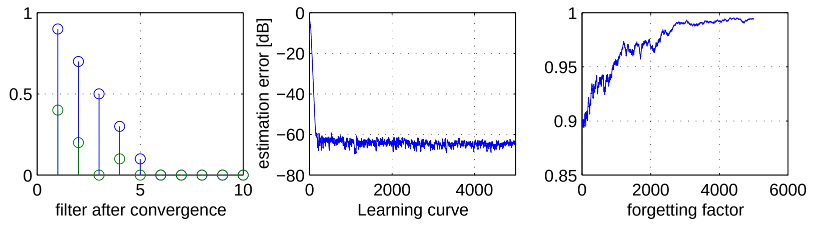

Figure 4.22: The adaptive filter coefficients after convergence, the learning curve, and the evolution of the forgetting factor for the complex system identification problem using the VFFRLS algorithm.

Algorithm `asptvffrls()` performs the following operations

- Filters the input signal $x(n)$ through the adaptive filter $w(n-1)$ to produce the filter output $y(n)$,
- Calculates the error sample $e(n) = d(n) - y(n)$,
- Recursively updates the gain vector $\underline{\mathbf{K}}(n)$ given by

$$\underline{\mathbf{K}}(n) = \frac{\mathbf{R}(n-1)\,\underline{\mathbf{x}}(n)}{a + \,\underline{\mathbf{x}}^T(n)\,\mathbf{R}(n-1)\underline{\mathbf{x}}(n)}. \tag{4.15}$$

- Updates the adaptive filter coefficients,
- Updates the forgetting factor.

Remarks

- `asptvffrls()` supports real as well as complex signals. The complex linear combiner VFFRLS filter converges to the complex conjugate of the Wiener solution.
- `asptvffrls()` does not update the delay line internally. The delay line must be updated before calling `asptvffrls()` as shown in the example listed above.

Resources The resources required to implement ta VFFRLS filter of length L in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$6L + 14$
MULTIPLY	$8L + 23$
ADD	$8L + 14$
DIVIDE	3

See Also INIT_ VFFRLS, ASPTRLS, ASPTVSSLMS.

Reference [2] and [4] for analysis of the adaptive Lattice filters.

4.20 asptvsslms

Purpose Sample per sample filtering and coefficient update using the Variable Step Size LMS (VSSLMS) algorithm.

Syntax

```
[w,g,mu,y,e] = asptvsslms(x,w,g,d,mu,roh)
[w,g,mu,y,e] = asptvsslms(x,w,g,d,mu,roh,ssa,mu_min,mu_max)
```

Description `asptvsslms()` implements the Variable Step Size LMS adaptive algorithm used to update transversal adaptive filters. VSSLMS does not only adjust the filter coefficients but also adjusts the step size *mu* to obtain fast convergence rate as well as small final misadjustment, a combination impossible to achieve with constant step size. Referring to the general adaptive filter shown in Fig. 2.6, `asptvsslms()` takes an input samples delay line $x(n)$, a desired sample $d(n)$, the vector of the adaptive filter coefficients from previous iteration $w(n-1)$, the previous vector of step sizes $mu(n-1)$, the previous gradient vector $g(n-1)$ (used to update mu), and returns the filter output $y(n)$, the error sample $e(n)$, the updated gradient vector $g(n)$, the updated step size vector $mu(n)$, and the updated vector of filter coefficients $w(n)$. If the `mu_min` and `mu_max` optional input arguments are given, each element of the step size vector is constrained to those limits. The update equation of `asptvsslms()` is given by

$$\underline{\mathbf{w}}(n) = \underline{\mathbf{w}}(n) + \underline{\mu}(n)e(n)\underline{\mathbf{x}}(n). \tag{4.16}$$

The input and output parameters of `asptvsslms()` for an FIR adaptive filter of L coefficients are summarized below.

```
Input Parameters [Size] ::
   x      : input samples delay line [L x 1]
   d      : desired response [1 x 1]
   w      : filter coef. vector w(n-1) [L x 1]
   g      : gradient vector g(n-1) [L x 1]
   mu     : vector of step sizes mu(n-1) [L x 1]
   roh    : gradient vector step size [1 x 1]
   ssa    : if 1, the sign-sign algorithm is used to update mu.
   mu_min : lower bound for mu [1 x 1]
   mu_max : higher bound for mu [1 x 1]

Output parameters::
   w      : updated filter coefficients w(n)
   y      : filter output y(n)
   g      : updated gradient vector g(n)
   mu     : updated vector of step sizes mu(n)
   e      : error sample, e(n)=d(n)-y(n)
```

Example

```
% VSSLMS used in a system identification application.
% By the end of this script the adaptive filter w should
% have the same coefficients as the unknown filter h.
iter = 5000;                     % Number of samples to process
% Complex unknown impulse response
h    = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xn   = 2*(rand(iter,1)-0.5);  % Input signal, zero mean random.
% although xn is real, dn will be complex since h is complex
dn   = osfilter(h,xn);           % Unknown filter output
en   = zeros(iter,1);            % vector to collect the error
mu0  = 0.05*ones(10,1);          % initial step size
muv  = zeros(iter,1);            % evolution of mu with time
% Initialize the VSSLMS algorithm with a filter of 10 coef.
[w,x,d,y,e,g,mu]  = init_vsslms(10,[],[],[],mu0);
%% Processing Loop
for (m=1:iter)
   % update the input delay line
   x = [xn(m,:); x(1:end-1,:)];
   d = dn(m,:) + 1e-3*rand;      % additive noise of var = 1e-6
   % call VSSLMS to calculate the filter output, estimation error
   % and update the coefficients and step sizes.
   [w,g,mu,y,e] = asptvsslms(x,w,g,d,mu,1e-3,1,1e-6,.99);
   % save the last error sample to plot later
   en(m,:) = e;   muv(m) = mean(mu);
end;
% display the results
% display the results
subplot(3,3,1);stem([real(w) imag(conj(w))]); grid;
eb = filter(.1, [1 -.9], en .* conj(en));
subplot(3,3,2);plot(10*log10(eb  ));grid
subplot(3,3,3);plot(muv); grid;
```

Running the above script will produce the graph shown in Fig. 4.23. The left-side graph of the figure shows the adaptive filter coefficients after convergence which are almost identical to the unknown filter h. The middle graph shows the mean square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB). The right-side graph shows the evolution of the mean value of the step size vector with time.

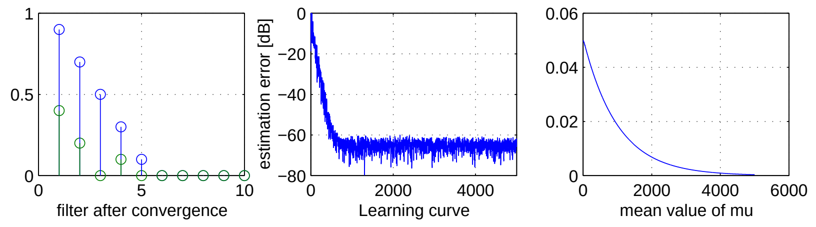

Figure 4.23: The adaptive filter coefficients after convergence, the learning curve, and the evolution of the mean value of the step size for the complex FIR system identification problem using the VSSLMS algorithm.

Remarks

Like the LMS, the VSSLMS is also a stochastic implementation of the steepest-descent algorithm where the *mean* value of the filter coefficients converge towards their optimal solution. Therefore, the filter coefficients will fluctuate about their optimum values given by the Wiener solution. The amplitude of the fluctuations is controlled by the step size. The smaller the step size, the smaller the fluctuations (less final misadjustment) but also the slower the adaptive coefficients converge to their optimal values. The improvement the VSSLMS introduces is that a separate step size is used for each filter coefficient, and the algorithm adapts those step sizes. When a coefficient is far from its optimal value, its corresponding step size is increased to converge faster. Conversely, when a coefficient is near its optimal value, the step size is decreased to decrease the final misadjustment. Similar to the LMS, the following points also apply to the VSSLMS.

- The VSSLMS algorithm shows stable convergence behavior only when all elements of the step size vector $mu(n)$ take values between zero and an upper limit, at all time indexes n, defined by the statistics of the filter's input signal. The fastest convergence will be achieved for a white noise input sequence with zero mean and unit variance. Such white input signal has all its eigenvalues equal to unity and therefore has a diagonal autocorrelation matrix with diagonal values equal to unity.
- The more colored the spectrum of the input signal, the slower the convergence will be. This is due to the large eigenvalue spread for such colored signals. This makes the convergence composed of several modes, each associated with one of the eigenvalues.
- `asptvsslms()` supports both real and complex data and filters. The adaptive filter for the complex VSSLMS algorithm converges to the complex conjugate of the optimum solution.
- `asptvsslms()` does not update the input delay line for $x(n)$, this has been chosen to provide more flexibility, so that the same function can be used with transversal as well as linear combiner structures. Delay line update, by inserting the newest sample at the beginning of the buffer and shifting the rest of the samples to the right, has to be done before calling `asptvsslms()` as in the example above.

Resources

The resources required to implement the VSSLMS algorithm for a transversal adaptive FIR filter of L coefficients in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$4L + 6$
MULTIPLY	$4L$
ADD	$3L$
DIVIDE	0

See Also

INIT_ VSSLMS, MODEL_ VSSLMS, ASPTNLMS, ASPTLMS, ASPTLCLMS.

Reference

[11] for extensive analysis of the LMS and the steepest-descent search method.

4.21 init_ arlmsnewt

Purpose Creates and initializes the variables required for the efficient implementation of the LMS-Newton algorithm using autoregressive modeling.

Syntax

```
[k,w,x,b,u,P,d,y,e]=init_arlmsnewt(L,M)
[k,w,x,b,u,P,d,y,e]=init_arlmsnewt(L,M,k0,w0,x0,b0,u0,P0,d0)
```

Description The LMS-Newton is a stochastic implementation of the Newton search method which solves the eigenvalue spread problem in adaptive filters with colored input signals. The update equation for the LMS-Newton is given by (see Fig. 2.6)

$$\underline{\mathbf{w}}(n+1) = \underline{\mathbf{w}}(n) + 2\mu\, e(n)\mathbf{R}^{-1}\,\underline{\mathbf{x}}(n), \tag{4.17}$$

where $\mathbf{R}$ is the autocorrelation matrix of the adaptive filter input signal $x(n)$. Direct implementation of the LMS-Newton update equation (4.17) requires estimation and inversion of $\mathbf{R}$ and matrix vector multiplication $\mathbf{R}^{-1}\,\underline{\mathbf{x}}(n)$ each sample which is of course very computational demanding. `arlmsnewt()` implements the LMS-Newton method efficiently by recursively estimating the term $u = \mathbf{R}^{-1}\,\underline{\mathbf{x}}(n)$ using autoregressive modeling. A lattice predictor of M stages is used for the autoregressive modeling part. When the input signal can be modeled with an autoregressive model of length M much less than the adaptive filter length L, a significant computational saving can be achieved. The variables of the ARLMSNEWT are summarized below.

```
Input Parameters::
   L   : number of adaptive filter coefficients
   M   : number of autoregressive model coefficients (M << L)
   k0  : vector of initial lattice predictor coefficients [Mx1]
   w0  : vector of initial filter coefficients [Lx1]
   x0  : vector of initial input samples [Lx1]
   b0  : vector of initial backward prediction errors [Lx1]
   u0  : vector of initial normalized gradients [Lx1]
   P0  : initial power of b [(M+1)x1]
   d0  : initial desired response [1x1]
Output parameters::
   k   : initialized lattice predictor coefficients [zeros]
   w   : initialized linear combiner coefficients [zeros]
   x   : initialized input samples vector [random]
   b   : initialized backward prediction errors [random]
   u   : initialized normalized gradient vector [zeros]
   P   : initialized estimated power of b [b .* b]
   d   : initialized desired response [random]
   y   : initialized filter output [w' * x]
   e   : initialized error signal [e = d - y]
```

Example

```
L  = 1024;                  % adaptive filter length
M  = 4;                     % lattice predictor stages
k0 = zeros(M,1);            % initial PARCOR coef.
w0 = zeros(L,1);            % initial filter coef.
b0 = rand(L,1);             % initial backward errors
P0 = b0(1:M+1).*b0(1:M+1); % initial power of b
d0 = .22;                   % initial desired sample

% Create and initialize an LMS-Newton filter
[k,w,x,b,u,P,d,y,e]=init_arlmsnewt(L,M,k0,w0,[],b0,[],P0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 3 through 9 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTARLMSNEWT, MODEL_ ARLMSNEWT.

4.22 init_ bfdaf

Purpose Creates and initializes the variables required for the Block Frequency Domain Adaptive Filter (BFDAF) algorithm.

Syntax

```
[W,x,d,e,y,Px,w]=init_bfdaf(L,M)
[W,x,d,e,y,Px,w]=init_bfdaf(L,M,W0,x0,d0)
```

Description The variables of the BFDAF are summarized below (see Fig. 4.2). The FFT length B is internally calculated using the equation $B = 2^{nextpow2(L+M-1)}$.

```
  L  : new samples per block (block length)
  M  : filter length in time domain
  W0 : initial frequency domain filter coef. vector [B x 1]
  x0 : initial overlap-save input vector [B x 1]
  d0 : initial desired response vector [L x 1]

Output parameters [default]::
  W  : initialized freq. domain filter coef. vector [zeros]
  x  : initialized input vector [zeros]
  d  : initialized desired response vector [white noise]
  e  : initialized error vector
  y  : initialized filter output
  Px : initialized estimate of the power of x
  w  : initialized time domain coefficients vector (optional)
```

Example

```
L  = 128;                 % Block length
M  = 128;                 % Filter Length
B  = 2^nextpow2(L+M-1);   % FFT length
w0 = zeros(B,1);          % initial filter coef.
x0 = rand(M,1);           % initial input buffer
d0 = rand(L,1);           % desired block

% Create and initialize a BFDAF FIR filter
[W,x,d,e,y,Px,w]=init_bfdaf(L,M,w0,x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 3 through 5 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTBFDAF, ECHO_ BFDAF.

4.23 init_ blms

Purpose Creates and initializes the variables required for the Block Least Mean Squares adaptive filter.

Syntax

```
[w,x,d,e,y] = init_blms(N,L)
[w,x,d,e,y] = init_blms(N,L,w0,x0,d0)
```

Description The variables of the BLMS are summarized below (see Fig. 2.6).

```
Input Parameters [Size]::
  N  : filter length
  L  : new samples per block (block length)
  w0 : initial filter coefficients vector [N x 1]
  x0 : initial input delay line [N x 1]
  d0 : initial desired response vector [L x 1]

Output parameters [default]::
  w  : initialized filter coefficients vector [zeros]
  x  : initialized input delay line [zeros]
  d  : initialized desired response vector [white noise]
  e  : initialized error vector
  y  : initialized filter output vector
```

Example

```
N  = 5;           % Block length
L  = 5;           % Number of coefficients
w0 = [0;0;1;0;0]; % initial filter coefficients
x0 = rand(N,1);   % initial delay line
d0 = x0;          % desired sample

% Create and initialize a BLMS FIR filter
[w,x,d,e,y]=init_blms(N,L,w0,x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 3 through 5 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTBLMS.

4.24 init_ bnlms

Purpose Creates and initializes the variables required for the Block Normalized Least Mean Squares adaptive filter.

Syntax

```
[w,x,d,e,y,p] = init_bnlms(N,L)
[w,x,d,e,y,p] = init_bnlms(N,L,w0,x0,d0)
```

Description The variables of the BNLMS are summarized below (see Fig. 2.6).

```
Input Parameters [Size]::
  N  : filter length
  L  : new samples per block (block length)
  w0 : initial filter coefficients vector [N x 1]
  x0 : initial input delay line [N x 1]
  d0 : initial desired response vector [L x 1]

Output parameters [default]::
  w  : initialized filter coefficients vector [zeros]
  x  : initialized input delay line [zeros]
  d  : initialized desired response vector [white noise]
  e  : initialized error vector
  y  : initialized filter output vector
  p  : initialized estimate of the power of x
```

Example

```
N  = 5;           % Block length
L  = 5;           % Number of coefficients
w0 = [0;0;1;0;0]; % initial filter coefficients
x0 = rand(N,1);   % initial delay line
d0 = x0;          % desired sample

% Create and initialize a BNLMS FIR filter
[w,x,d,e,y,p]=init_bnlms(N,L,w0,x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 3 through 5 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTBNLMS.

4.25 init_ drlms

Purpose Creates and initializes the variables required for the Data Reusing Least Mean Squares algorithm.

Syntax

```
[w,x,d,y,e] = init_drlms(L)
[w,x,d,y,e] = init_drlms(L,w0,x0,d0)
```

Description The variables of the DRLMS are summarized below (see Fig. 2.6).

```
Input Parameters [Size] ::
  L   : number of filter coefficients
  w0  : initial coefficient vector [L x 1]
  x0  : initial input samples vector [L x 1]
  d0  : initial desired sample [1 x 1]

Output parameters [default] ::
  w   : initialized filter coefficients [zeros]
  x   : initialized input vector [zeros]
  d   : initialized desired sample [white noise]
  y   : Initialized filter output
  e   : initialized error sample [e = d - y]
```

Example

```
L  = 5;             % Number of coefficients
w0 = [0;0;1;0;0];   % initial filter coefficients
x0 = rand(L,1);     % initial delay line
d0 = 0;             % desired sample

% Create and initialize a DRLMS FIR filter
[w,x,d,y,e]=init_drlms(L,w0,x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 2 through 4 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTDRLMS.

4.26 init_ drnlms

Purpose Creates and initializes the variables required for the Data Reusing Normalized Least Mean Squares algorithm.

Syntax

```
[w,x,d,y,e,p] = init_drnlms(L)
[w,x,d,y,e,p] = init_drnlms(L,w0,x0,d0)
```

Description The variables of the DRNLMS are summarized below (see Fig. 2.6).

```
Input Parameters [Size]::
  L   : number of filter coefficients
  w0 : initial coefficient vector [L x 1]
  x0 : initial input samples vector [L x 1]
  d0 : initial desired sample [1 x 1]

Output parameters [default]::
  w   : initialized filter coefficients [zeros]
  x   : initialized input vector [white noise]
  d   : initialized desired sample [white noise]
  y   : Initialized filter output
  e   : initialized error sample [e = d - y]
  p   : initialized power estimate
```

Example

```
L  = 5;           % Number of coefficients
w0 = [0;0;1;0;0]; % initial filter coefficients
x0 = rand(L,1);   % initial delay line
d0 = 0;           % desired sample

% Create and initialize a DRNLMS FIR filter
[w,x,d,y,e,p]=init_drnlms(L,w0,x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 2 through 4 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTDRNLMS.

4.27 init_ leakynlms

Purpose Creates and initializes the variables required for the Leaky Normalized Least Mean Squares (LEAKY NLMS) Adaptive algorithm

Syntax

```
[w,x,d,y,e,p]=init_leakynlms(L)
[w,x,d,y,e,p]=init_leakynlms(L,w0,x0,d0)
```

Description The variables of the Leaky NLMS are summarized below (see Fig. 2.6).

```
Input Parameters::
   L   : number of filter coefficients
   w0  : initial coefficient vector [L x 1]
   x0  : initial input samples vector [L x 1]
   d0  : initial desired sample [L x 1]

Output parameters [default]::
   w   : initialized filter coefficients [zeros]
   x   : initialized input vector [white noise]
   d   : initialized desired sample [white noise]
   y   : Initialized filter output
   e   : initialized error sample [e = d - y]
   p   : initialized power estimate
```

Example

```
L  = 5;            % Number of coefficients
w0 = [0;0;1;0;0]; % initial filter coefficients
x0 = rand(5,1);    % initial delay line
d0 = 0;            % desired sample

% Create and initialize a Leaky NLMS FIR filter
[w,x,d,y,e,p]=init_leakynlms(L,w0,x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 2 through 4 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTLEAKYNLMS, ECHO_ LEAKYNLMS, ASPTNLMS.

4.28 init_ lclms

Purpose Creates and initializes the variables required for the Linearly Constrained Least Mean Squares (LMS) Adaptive algorithm

Syntax

```
[w,x,d,y,e]=init_lclms(L)
[w,x,d,y,e]=init_lclms(L,w0,x0,d0)
```

Description The variables of the LCLMS are summarized below (see Fig. 2.6).

```
Input Parameters [Size] ::
   L   : number of filter coefficients
   w0  : initial coefficient vector [L x 1]
   x0  : initial input samples vector [L x 1]
   d0  : initial desired sample [a x 1]
Output parameters [default] ::
   w   : initialized filter coefficients [zeros]
   x   : initialized input vector [zeros]
   d   : initialized desired sample [white noise]
   y   : Initialized filter output
   e   : initialized error sample [e = d - y]
```

Example

```
L  = 5;            % Number of coefficients
w0 = [0;0;1;0;0]; % initial filter coefficients
x0 = rand(5,1);    % initial delay line
d0 = 0;            % desired sample

% Create and initialize a LCLMS FIR filter
[w,x,d,y,e]=init_lclms(L,w0,x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 2 through 4 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTLCLMS, BEAMBB_ LCLMS.

4.29 init_ lms

Purpose Creates and initializes the variables required for the Least Mean Squares (LMS) Adaptive algorithm

Syntax

```
[w,x,d,y,e]=init_lms(L)
[w,x,d,y,e]=init_lms(L,w0,x0,d0)
```

Description The variables of the LMS are summarized below (see Fig. 2.6).

```
Input Parameters [Size] ::
   L   : number of filter coefficients
   w0  : initial coefficient vector [L x 1]
   x0  : initial input samples vector [L x 1]
   d0  : initial desired sample [a x 1]
Output parameters [default] ::
   w   : initialized filter coefficients [zeros]
   x   : initialized input vector [zeros]
   d   : initialized desired sample [white noise]
   y   : Initialized filter output
   e   : initialized error sample [e = d - y]
```

Example

```
L  = 5;            % Number of coefficients
w0 = [0;0;1;0;0]; % initial filter coefficients
x0 = rand(5,1);    % initial delay line
d0 = 0;            % desired sample

% Create and initialize an LMS FIR filter
[w,x,d,y,e]=init_lms(L,w0,x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 2 through 4 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTLMS, MODEL_ LMS.

4.30 init_ mvsslms

Purpose Creates and initializes the variables required for the Modified Variable Step Size Least Mean Squares (MVSSLMS) Adaptive algorithm

Syntax

```
[w,x,d,y,e,g,mu] = init_mvsslms(L)
[w,x,d,y,e,g,mu] = init_mvsslms(L,w0,x0,d0,mu0,g0)
```

Description The MVSSLMS is a simplified version of the VSSLMS. The variables of the MVSSLMS are summarized below (see Fig. 2.6).

```
Input Parameters [Size] ::
  L    : adaptive filter length
  w0   : initial vector of filter coefficients [Lx1]
  x0   : initial input samples delay line [Lx1]
  d0   : initial desired sample [1x1]
  mu0  : initial step-size [1x1]
  g0   : initial gradient[1x1]

Output parameters [default]::
   w   : initialized filter coefficients [zeros]
   x   : initialized input delay line [zeros]
   d   : initialized desired sample [white noise]
   y   : Initialized filter output
   e   : initialized error sample [e = d - y]
   g   : initialized gradient vector [zero]
   mu  : initialized step-size vector [zero]
```

Example

```
L   = 5;              % Number of coefficients
w0  = [0;0;1;0;0];    % initial filter coefficients
x0  = rand(5,1);      % initial delay line
mu0 = 0.1;            % initial step sizes

% Create and initialize an MVSSLMS FIR filter
[w,x,d,y,e,g,mu] = init_mvsslms(L,w0,x0,[],mu0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 2 through 6 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTMVSSLMS, ASPTVSSLMS, MODEL_ MVSSLMS.

4.31 init_ nlms

Purpose Creates and initializes the variables required for the Normalized Least Mean Squares (NLMS) Adaptive algorithm

Syntax

```
[w,x,d,y,e,p]=init_nlms(L)
[w,x,d,y,e,p]=init_nlms(L,w0,x0,d0)
```

Description The variables of the NLMS are summarized below (see Fig. 2.6).

```
Input Parameters::
   L   : number of filter coefficients
   w0  : initial coefficient vector [L x 1]
   x0  : initial input samples vector [L x 1]
   d0  : initial desired sample [L x 1]

Output parameters [default]::
   w   : initialized filter coefficients [zeros]
   x   : initialized input vector [white noise]
   d   : initialized desired sample [white noise]
   y   : Initialized filter output
   e   : initialized error sample [e = d - y]
   p   : initialized power estimate
```

Example

```
L  = 5;           % Number of coefficients
w0 = [0;0;1;0;0]; % initial filter coefficients
x0 = rand(5,1);   % initial delay line
d0 = 0;           % desired sample

% Create and initialize an NLMS FIR filter
[w,x,d,y,e,p]=init_nlms(L,w0,x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 2 through 4 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTNLMS, MODEL_ NLMS, ASPTLMS.

4.32 init_ pbfdaf

Purpose

Creates and initializes the variables required for the Partitioned Block Frequency Domain Adaptive Filter (PBFDAF) algorithm.

Syntax

```
[W,x,d,e,y,Px,X,w]=init_pbfdaf(L,M,P)
[W,x,d,e,y,Px,X,w]=init_pbfdaf(L,M,P,W0,X0,d0)
```

Description

The variables of the PBFDAF are similar to those of the BFDAF (see Fig. 4.2) and are summarized below . The FFT length B is internally calculated using the equation $B = 2^{nextpow2(L+M-1)}$.

```
Input Parameters [Size]::
   L  : number of new input samples per block
   M  : filter partition length (in time domain)
   P  : number of partitions (total filter length = P*M)
   W0 : initial matrix of filter coefficients [B x P]
   X0 : initial matrix of frequency domain input signal [B x P]
   d0 : initial desired response vector [L x 1]

Output parameters [default]::
   W  : initialized matrix of filter coef. [zeros]
   x  : initialized overlap-save input buffer
   d  : initialized desired response [white noise]
   e  : initialized error vector in t-domain
   y  : initialized filter output in t-domain
   Px : initialized estimate of the input power (Bx1)
   X  : initialized input samples matrix [zeros]
   w  : time domain filter coefficients vector
```

Example

```
P  = 16;                  % partitions
L  = 128;                 % Block length
M  = 128;                 % 2048-long filter
B  = 2^nextpow2(L+M-1);   % FFT length
W0 = zeros(B,P);          % initial filter coef.
X0 = rand(B,P);           % initial input buffer
d0 = rand(L,1);           % desired block

% Create and initialize a PBFDAF FIR filter
[W,x,d,e,y,Px,X,w]=init_pbfdaf(L,M,P,W0,X0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 4 through 6 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also

ASPTPBFDAF, ECHO_ PBFDAF.

4.33 init_ rcpbfdaf

Purpose Creates and initializes the variables required for the Reduced Complexity Partitioned Block Frequency Domain Adaptive Filter (RCPBFDAF) algorithm.

Syntax

```
[W,x,d,e,y,Px,X,ci,w]=init_rcpbfdaf(L,M,P)
[W,x,d,e,y,Px,X,ci,w]=init_rcpbfdaf(L,M,P,W0,X0,d0)
```

Description The variables of the RCPBFDAF are similar to those of the BFDAF (see Fig. 4.2) and are summarized below . The FFT length B is internally calculated using the equation $B = 2^{nextpow2(L+M-1)}$.

```
Input Parameters [Size]::
   L  : block length (M = g * L)
   M  : filter partition length, must be int multiple of L
   P  : number of partitions (total filter length = P*M)
   W0 : initial matrix of filter coefficients [B x P]
   X0 : initial matrix of f-domain input signal [B x G]
   d0 : initial desired response vector [L x 1]
Output parameters [default]::
   W  : initialized matrix of filter coef. [zeros]
   x  : initialized overlap-save input buffer
   d  : initialized desired response [white noise]
   e  : initialized error vector in t-domain
   y  : initialized filter output in t-domain
   Px : initialized estimate of the input power (Bx1)
   X  : initialized input samples matrix [zeros]
   ci : index of the next partition to be constrained
   w  : time domain filter coefficients vector
```

Example

```
P  = 16;                  % partitions
L  = 32;                  % Block length
M  = L*4;                 % 2048-long filter
B  = 2^nextpow2(L+M-1);   % FFT length
G  = (P-1)*4+1;           % depth of X
W0 = zeros(B,P);          % initial filter coef.
X0 = rand(B,G);           % initial input buffer
d0 = rand(L,1);           % desired block

% Create and initialize a RCPBFDAF FIR filter
[W,x,d,e,y,Px,X,ci,w]=init_rcpbfdaf(L,M,P,W0,X0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 4 through 6 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTRCPBFDAF, ECHO_ RCPBFDAF.

4.34 init_ rdrlms

Purpose

Creates and initializes the variables required for the Recent Data Reusing Least Mean Squares algorithm.

Syntax

```
[w,x,d,y,e] = init_rdrlms(L,k)
[w,x,d,y,e] = init_rdrlms(L,k,w0,x0,d0)
```

Description

The variables of the RDRLMS are summarized below (see Fig. 2.6).

```
Input Parameters [Size] ::
  L  : number of filter coefficients
  k  : number of data reusing cycles.
  w0 : initial coefficient vector [L x 1]
  x0 : initial input samples vector [L+k x 1]
  d0 : initial desired sample [k+1 x 1]

Output parameters [default] ::
  w  : initialized filter coefficients [zeros]
  x  : initialized input vector [zeros]
  d  : initialized desired vector [white noise]
  y  : Initialized filter output
  e  : initialized error sample [d - y]
```

Example

```
L  = 5;              % Number of coefficients
k  = 2;              % number of reusing cycles
w0 = [0;0;1;0;0];    % initial filter coefficients
x0 = rand(L+k,1);    % initial delay line
d0 = rand(k+1,1);    % desired sample

% Create and initialize a RDRLMS FIR filter
[w,x,d,y,e]=init_rdrlms(L,k,w0,x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 3 through 5 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also

ASPTRDRLMS.

4.35 init_ rdrnlms

Purpose

Creates and initializes the variables required for the Recent Data Reusing Normalized Least Mean Squares algorithm.

Syntax

```
[w,x,d,y,e,p] = init_rdrnlms(L,k)
[w,x,d,y,e,p] = init_rdrnlms(L,k,w0,x0,d0)
```

Description

The variables of the RDRNLMS are summarized below (see Fig. 2.6).

```
Input Parameters [Size]::
  L  : number of filter coefficients
  k  : number of data reusing cycles.
  w0 : initial coefficient vector [L x 1]
  x0 : initial input samples vector [L+k x 1]
  d0 : initial desired sample [k+1 x 1]

Output parameters [default]::
  w  : initialized filter coefficients [zeros]
  x  : initialized input vector [white noise]
  d  : initialized desired sample [white noise]
  y  : Initialized filter output
  e  : initialized error vector [d - y]
  p  : initialized power estimate
```

Example

```
L  = 5;             % Number of coefficients
k  = 2;             % number of reusing cycles
w0 = [0;0;1;0;0];   % initial filter coefficients
x0 = rand(L+k,1);   % initial delay line
d0 = rand(k+1,1);   % desired sample

% Create and initialize a RDRNLMS FIR filter
[w,x,d,y,e,p]=init_rdrnlms(L,k,w0,x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 3 through 5 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also

ASPTRDRNLMS.

4.36 init_ rls

Purpose Creates and initializes the variables required for the Recursive Least Squares (RLS) adaptive algorithm.

Syntax

```
[w,x,d,y,e,R]=init_rls(L,b)
[w,x,d,y,e,R]=init_rls(L,b,w0,x0,d0)
```

Description The variables of the RLS are summarized below (see Fig. 2.6).

```
Input Parameters::
   L  : Adaptive filter length
   b  : a small +ve constant to initialize R
   w0 : initial coefficient vector
   x0 : initial input samples vector
   d0 : initial desired sample
Output parameters [default]::
   w  : Initialized filter coefficients [zeros]
   x  : Initialized input vector [zeros]
   d  : Initialized desired sample [white noise]
   y  : Initialized filter output [y = w' * x]
   e  : Initialized error sample [e = d - y]
   R  : Initialized inverse of the weighted
        auto correlation matrix of x, [R=b*eye(L)]
```

Example

```
L  = 5;            % Number of coefficients
w0 = [0;0;1;0;0];  % initial filter coefficients
x0 = rand(5,1);    % initial delay line
d0 = 0;            % desired sample

% Create and initialize the RLS FIR filter
[w,x,d,y,e,R]=init_rls(L,0.1,w0,x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 3 through 5 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTRLS, EQUALIZER_ RLS.

4.37 init_ tdftaf

Purpose Creates and initializes the variables required for the Transform Domain Fault Tolerant Adaptive Filter (TDFTAF). TDFTAF contains R redundant coefficients to guarantee that the filter will recover after the occurrence of a partial hardware failure during operation.

Syntax

```
[W,w,x,d,y,e,p] = init_tdftaf(L,R)
[W,w,x,d,y,e,p] = init_tdftaf(L,R,W0,x0,d0)
```

Description The variables of the TDFTAF are summarized below (see Fig. 2.6 and Fig. 4.18).

```
Input Parameters [Size]::

   L   : number of filter coefficients
   R   : number of redundant coefficients
   w0  : initial T-domain coef. vector [L+R x 1]
   x0  : initial input samples vector [L x 1]
   d0  : initial desired sample [1 x 1]
Output parameters [default]::

   W   : initialized T-domain coef. vector [zeros]
   w   : initialized time-domain coef. vector [zeros]
   x   : initialized input vector [white noise]
   d   : initialized desired sample [white noise]
   y   : Initialized filter output
   e   : initialized error sample [e = d - y]
   p   : initialized power estimate
```

Example

```
L  = 5;            % Number of coefficients
R  = 3;            % Redundant coefficients
x0 = rand(5,1);    % initial delay line
d0 = 0;            % desired sample

% Create and initialize a TDFTAF FIR filter
[W,w,x,d,y,e,p]=init_tdftaf(L,R,[],x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 3 through 5 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTTDFTAF, ASPTTDLMS.

4.38 init_ tdlms

Purpose Creates and initializes the variables required for the Transform Domain Least Mean Squares (TDLMS) Adaptive algorithm

Syntax

```
[W,w,x,d,y,e,p]=init_tdlms(L)
[W,w,x,d,y,e,p]=init_tdlms(L,W0,x0,d0)
```

Description The variables of the TDLMS are summarized below (see Fig. 2.6 and Fig. 4.20).

```
Input Parameters [Size]::
   L   : number of filter coefficients
   W0  : initial T-domain coef. vector [L x 1]
   x0  : initial input samples vector [L x 1]
   d0  : initial desired sample [L x 1]

Output parameters [default]::
   W   : initialized T-domain coef. vector [zeros]
   w   : initialized time-domain coef. vector [zeros]
   x   : initialized input vector [white noise]
   d   : initialized desired sample [white noise]
   y   : Initialized filter output
   e   : initialized error sample [e = d - y]
   p   : initialized power estimate
```

Example

```
L  = 5;            % Number of coefficients
W0 = [1;0;0;0;0]; % initial filter coefficients
x0 = rand(5,1);    % initial delay line
d0 = 0;            % desired sample

% Create and initialize a TDLMS FIR filter
[W,w,x,d,y,e,p]=init_tdlms(L,W0,x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 2 through 4 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTTDLMS, MODEL_ TDLMS.

4.39 init_ vffrls

Purpose Creates and initializes the variables required for the Variable Forgetting Factor Recursive Least Squares (VFFRLS) adaptive algorithm.

Syntax

```
[w,x,d,y,e,R,k] = init_vffrls(L,b)
[w,x,d,y,e,R,k] = init_vffrls(L,b,w0,x0,d0)
```

Description The variables of the VFFRLS are summarized below.

```
Input Parameters::

   L  : Adaptive filter length
   b  : a small +ve constant to initialize R
   w0 : initial coefficient vector
   x0 : initial input samples vector
   d0 : initial desired sample

Output parameters [default]::

   w  : Initialized filter coefficients [zeros]
   x  : Initialized input vector [zeros]
   d  : Initialized desired sample [white noise]
   y  : Initialized filter output [y = w' * x]
   e  : Initialized error sample [e = d - y]
   R  : Initialized inverse of the weighted
        auto correlation matrix of x, [R=b*eye(L)]
   k  : Initialized gain vector.
```

Example

```
L  = 5;           % Number of coefficients
w0 = [0;0;1;0;0]; % initial filter coefficients
x0 = rand(5,1);   % initial delay line
d0 = 0;           % desired sample

% Create and initialize the VFFRLS FIR filter
[w,x,d,y,e,R,k]=init_vffrls(L,0.1,w0,x0,d0);
```

Remarks

- `asptvffrls()` supports both real and complex signals and filters.
- Use input parameters 3 through 5 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTVFFRLS, TEST_ VFFRLS.

4.40 init_ vsslms

Purpose Creates and initializes the variables required for the Variable Step Size Least Mean Squares (VSSLMS) Adaptive algorithm

Syntax

```
[w,x,d,y,e,g,mu] = init_vsslms(L)
[w,x,d,y,e,g,mu] = init_vsslms(L,w0,x0,d0,mu0,g0)
```

Description The variables of the VSSLMS are summarized below (see Fig. 2.6).

```
Input Parameters [Size] ::
  L    : adaptive filter length
  w0   : initial vector of filter coefficients [Lx1]
  x0   : initial input samples delay line [Lx1]
  d0   : initial desired sample [1x1]
  mu0  : initial step-size vector [Lx1]
  g0   : initial gradient vector [Lx1]

Output parameters [default]::
   w   : initialized filter coefficients [zeros]
   x   : initialized input delay line [zeros]
   d   : initialized desired sample [white noise]
   y   : Initialized filter output
   e   : initialized error sample [e = d - y]
   g   : initialized gradient vector [zeros]
   mu  : initialized step-size vector [zeros]
```

Example

```
L   = 5;               % Number of coefficients
w0  = [0;0;1;0;0];     % initial filter coefficients
x0  = rand(5,1);       % initial delay line
mu0 = 0.1*ones(L,1);   % initial step sizes

% Create and initialize an VSSLMS FIR filter
[w,x,d,y,e,g,mu] = init_vsslms(L,w0,x0,[],mu0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 2 through 6 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTVSSLMS, MODEL_ VSSLMS.

Chapter 5

Lattice Adaptive Algorithms

This chapter documents the functions used to create, initialize, and update the coefficients of lattice adaptive filters (Section 2.2.4). Table 5.1 summarizes the lattice adaptive functions and gives a short description and a pointer to the reference page of each function.

Function Name	Reference	Short Description
asptftrls	5.1	Fast Transversal RLS algorithm.
asptlbpef	5.2	Lattice Backward Prediction Error Filter.
asptlfpef	5.3	Lattice Forward Prediction Error Filter.
asptlmslattice	5.4	LMS-Lattice Joint Process Estimator.
asptrlslattice	5.5	RLS-Lattice joint process estimator using a posteriori estimation errors.
asptrlslattice2	5.6	RLS-Lattice joint process estimator using a priori estimation errors with error feedback.
asptrlslbpef	5.7	Lattice Backward Prediction Error Filter.
asptrlslfpef	5.8	Lattice Forward Prediction Error Filter.
init_ ftrls	5.9	Initialize Fast Transversal RLS.
init_ lbpef	5.10	Initialize Lattice Backward Prediction Error Filter.
init_ lfpef	5.11	Initialize Lattice Forward Prediction Error Filter.
init_ lmslattice	5.12	Initialize LMS Lattice adaptive filter.
init_ rlslattice	5.13	Initialize RLS-Lattice joint process estimator using a posteriori estimation errors.
init_ rlslattice2	5.14	Initialize RLS-Lattice joint process estimator using a priori estimation errors with error feedback.
init_ rlslbpef	5.15	Initialize Recursive Least Squares Lattice Backward PEF.
init_ rlslfpef	5.16	Initialize Recursive Least Squares Lattice Forward PEF.

Table 5.1: Functions for creating, initializing, and updating lattice adaptive filters.

Each function is documented in a separate section including the following information related to the function:

- **Purpose:** Short description of the algorithm implemented by this function.
- **Syntax:** Shows the function calling syntax. If the function has optional parameters, this section will have two calling syntaxes. One with only the required formal parameters and one with all the formal parameters.
- **Description:** Detailed description of the function usage with explanation of its input and output parameters.
- **Example:** A short example showing typical use of the function. The examples listed can be found in the ASPT/test directory of the ASPT distribution. The user is encouraged to copy from those examples and paste in her own applications.

- **Algorithm:** A short description of the operations internally performed by the function.
- **Remarks:** Gives more theoretical and practical remarks related to the usage, performance, limitations, and applications of the function.
- **Resources:** Gives a summary of the memory requirements and number of multiplications, addition/subtractions, and division operations required to implement the function in real time. This can be used to roughly calculate the MIPS (Million Instruction Per Second) required for a specific platform knowing the number of instructions the processor needs to perform each operation.
- **See Also:** Lists other functions that are related to this function.
- **Reference:** Lists literature for more information on the function.

5.1 asptftrls

Purpose Performs filtering and coefficient update using the Fast Transversal Recursive Least Squares (FTRLS) algorithm, also known as the Fast Transversal Filter (FTF) algorithm.

Syntax `[ff,bb,k,cf,c,g,w,e,y] = asptftrls(ff,bb,k,cf,c,g,w,a,x,d)`

Description `asptftrls()` is an efficient implementation of the joint process estimator, and is therefore an improvement over the `asptrlslattice()` and `asptrlslattice2()` functions. In the RLSLATTICE functions, the problem of prediction and filtering (joint process estimation) is solved for lattice sections and linear combiner of length $1, 2, \cdots L$ simultaneously, where L is the linear combiner length. The improvement brought about by the FTRLS algorithm is obtained by concentrating on solving the problem only for a filter of length L, and therefore removing redundant calculations. Although the FTRLS has about half the computational complexity of the RLSLATTICE algorithms, it is known to be sensitive to round-off errors. This problem derive the algorithm unstable when finite precision calculations are used, for instance when implemented on a fixed-point DSP platform. There exist two solutions for this round-off error problem. The first is to detect the instability before occurring and reinitializing the algorithm with the current filter coefficients (soft initialization). This approach is implemented in `asptftrls()`. The second is to calculate the sensitive quantities in different ways (introduce computation redundancy), an approach followed by the Stabilized Fast Transversal RLS (SFTRLS) algorithm.
The input and output parameters of `asptftrls()` of length L are summarized below.

```
Input Parameters::
   ff  : last autocorrelation of forward prediction error (f)
   bb  : last autocorrelation of backward prediction error (b)
   k   : normalized gain vector
   cf  : last conversion factor
   c   : forward transversal predictor coefficients vector
   g   : backward transversal predictor coefficients vector
   w   : transversal linear combiner coefficients vector
   a   : forgetting factor
   x   : input samples vector
   d   : desired response d(n)

Output parameters::
   ff  : updated autocorrelation of forward prediction error
   bb  : updated autocorrelation of backward prediction error
   k   : updated normalized gain vector
   cf  : updated conversion factor vector
   c   : updated forward predictor coefficients vector
   g   : updated backward predictor coefficients vector
   w   : updated linear combiner coefficients vector
   y   : linear combiner output
   e   : error signal
```

Example

```
% FTRLSL used in a simple system identification application.
% By the end of this script the adaptive filter w
% should have the same coefficients as the unknown filter h.

iter = 5000;                  % samples to process
% Complex unknown impulse response
h  = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xn = 2*(rand(iter,1)-0.5);   % Input signal
% although xn is real, dn will be complex since h is complex
dn = osfilter(h,xn);         % Unknown filter output
en = zeros(iter,1);          % error signal
% Initialize FTRLS with a filter of 10 coef.
L   = 10;                    % filter length
a   = .9;                    % forgetting factor
[ff,bb,k,cf,c,g,w,x,d,e,y]=init_ftrls(L);

%% Processing Loop
for (m=1:iter)

   x = [xn(m,:); x(1:end-1)];  % input samples vector
   d = dn(m,:) + 1e-3*rand;    % additive noise var = 1e-6
   [ff,bb,k,cf,c,g,w,e,y]=asptftrls(ff,bb,k,cf,c,g,w,a,x,d);
   % save the last error sample to plot later
   en(m,:) = e;
end;

% display the results
subplot(2,2,1);stem([real(w) imag(conj(w))]); grid;
subplot(2,2,2);
eb = filter(.1, [1 -.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 5.1. The left side graph of the figure shows the adaptive linear combiner coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the square error in dB versus time during the adaptation process. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB).

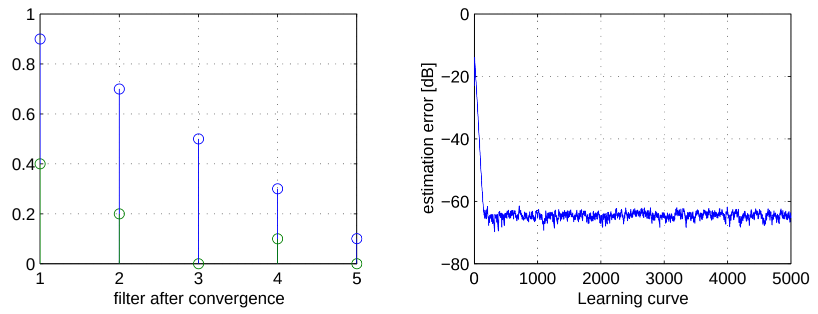

Figure 5.1: The adaptive linear combiner coefficients after convergence and the learning curve for the complex system identification problem using the FTRLS algorithm.

Algorithm `asptftrls()` performs the following operations each iteration

- Calculates the forward and backward prediction errors for the L^{th} lattice stage,
- Calculates the backward (bb) and forward (ff) autocorrelations for the L^{th} lattice stage,
- Calculates the conversion factor of the L^{th} and $L + 1^{th}$ lattice stages,
- Updates the normalized gain vector, and the forward and backward prediction coefficient vectors,
- Evaluates the linear combiner output and error signals,
- Updates the linear combiner coefficients,

Remarks

- `asptftrls()` supports real as well as complex signals and filters. The complex linear combiner FTRLS filter converges to the complex conjugate of the optimal solution.
- FTRLS algorithms have been found to be more sensitive to numerical rounding errors compared to RLS Lattice algorithms. `asptftrls()` detects the sign of instability, and reinitializes the algorithm before such instability occurs using the current coefficients vector.
- Stabilized versions of the FTRLS exist that introduce redundant calculations to improve robustness against numerical rounding errors, on the cost of extra computation. Even those stabilized versions have unstable modes that are excited when λ, the forgetting factor, is not close enough to unity. Since λ should be set smaller for fast tracking applications, this certainly can be seen as a limiting factor.

Resources The resources required to implement an FTRLS filter of length L in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$5L + 9$
MULTIPLY	$7L + 20$
ADD	$7L + 12$
DIVIDE	3

See Also INIT_ FTRLS, ASPTRLSLATTICE.

Reference [2] and [4] for analysis of the adaptive Lattice filters.

5.2 asptlbpef

Purpose Implements the adaptive LMS Lattice Backward Prediction Error Filter.

Syntax `[k,b,P,e,y,c] = asptlbpef(k,b,P,x,mu_p)`

Description `asptlbpef` is a lattice implementation of the backward prediction error filter shown in Fig. 5.2. Instead of updating the coefficients of a transversal filter, `asptlbpef()` updates the PARCOR coefficients of a lattice predictor of the same order. The backward prediction error of the last lattice stage is returned as the error signal $e(n)$ of the prediction error filter. If the number of output arguments are more than 5, the coefficients of the equivalent transversal prediction error filter are also calculated and returned in the variable c such that

$$e(n) = [1; -\underline{\mathbf{c}}]^T \ \underline{\mathbf{x}}(n-L) \tag{5.1}$$

where $[1; -\underline{\mathbf{c}}]^T$ represents the impulse response between the PEF output e(n) and its input x(n) and $\underline{\mathbf{x}}(n-L)$ is the input signal delay line delayed by L samples.
The input and output parameters of `asptlbpef()` of L stages are summarized below.

```
Input Parameters::
   k   : vector of lattice predictor coefficients
   b   : vector of backward prediction error
   P   : vector of last estimated power of b
   x   : input delay line
   mu_p: adaptation constant for the predictor coefficient

Output parameters::
   k   : updated lattice predictor coefficients
   b   : updated backward prediction error vector
   P   : updated power estimate of b
   e   : backward prediction error
   y   : predictor output
   c   : Equivalent transversal predictor coefficients.
```

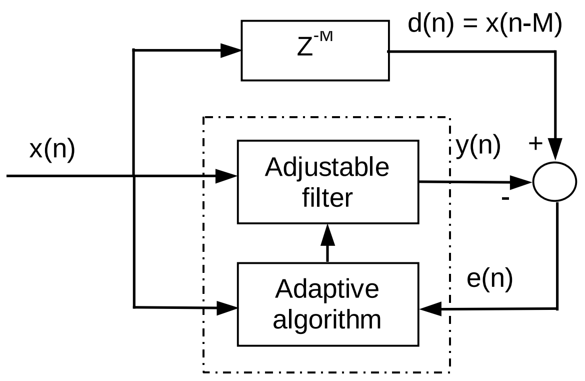

Figure 5.2: Block diagram of the backward prediction error filter.

Example

```
% LBPEF used in a adaptive line enhancer application.
% By the end of this script, the backward prediction error
% is the wide-band signal and the output of the equivalent
% transversal predictor is the narrow-band signal.
iter = 5000;
t    = (1:iter)/1000;           % time index @ 1kHz
xn   = 2*(rand(iter,1)-0.5)  ;  % Input signal, zero mean random.
xn   = xn + 1 * cos(2*pi*50*t');
yn   = zeros(iter,1);           % narrow-band signal
en   = yn;                      % error signal
% Initialize LBPEF
M    = 10;                      % filter length
mu_p = 0.01;                    % Step size
[k,b,P,e,y,x,c]=init_lbpef(M);  % Init LBPEF

%% Processing Loop
for (m=1:iter)
   x = [xn(m); x(1:end-1)];
   [k,b,P,e,y,c] = asptlbpef(k,b,P,x,mu_p);
   yn(m,:) = y;              % save narrow-band
   en(m,:) = e;              % save wide-band
end;

% Transfer function between e(n) and x(n-L).
h = filter([ 1;(-c)],1,[1;zeros(1023,1)]);
f = (0:512)*500/512;
H = 20*log10(abs(fft(h)));
% display the results
subplot(2,2,1);plot(f,H(1:513,:)); grid;
subplot(2,2,2);
plot([yn(4800:5000)]);grid
```

Running the above script will produce the graph shown in Fig. 5.3. The left side graph of the figure shows the frequency response of the equivalent transversal prediction error filter after convergence. This frequency response shows that the predictor adjusts itself to pass the narrow-band signal at 50 Hz and attenuate all other input components so that the error signal contains the wide-band signal only. The right side graph shows the last 200 samples of the filter output which shows that the filter output coincide with the narrow-band 50 Hz superimposed on the white noise input signal.

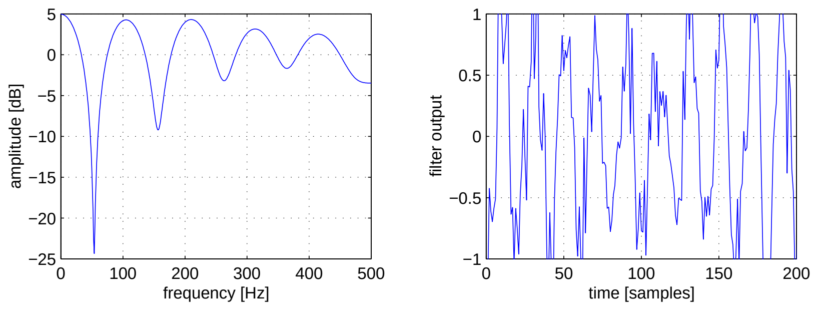

Figure 5.3: The frequency response of the PEF after convergence and the filter output for the adaptive line enhancer using LBPEF.

Algorithm `asptlbpef()` performs the following operations

- Calculates the forward and backward prediction errors for the lattice structure stages using the order update equations (see section 2.2.4),
- Calculates the power estimate of the backward prediction errors,
- Updates the PARCOR coefficients of the lattice predictor,
- Calculates the equivalent transversal predictor coefficients from the updated PARCOR coefficients,
- Calculates the equivalent transversal predictor output.

Remarks The adaptive Lattice Prediction Error Filter is a useful tool in linear prediction and autoregressive modeling applications. A few of the features that make the LBPEF so popular are:

- The PARCOR coefficients always satisfy the relation $|k_m| \leq 1$.
- The power of the forward prediction error $E[e^2_{fm}(n)]$ and the backward prediction error $E[e^2_{bm}(n)]$ of the same stage are equal.
- The backward prediction errors $e_{b0}(n), e_{b1}(n), \cdots, e_{bM}(n)$ are uncorrelated with one another for any input sequence $x(n)$. This property is very important since it shows that the lattice predictor can be seen as an orthogonal transformation with the input signal samples $x(n), x(n-1), \cdots, x(n-M+1)$ as input and the backward errors $e_{b0}(n), e_{b1}(n), \cdots, e_{bM}(n)$ as the uncorrelated (orthogonal) output.
- The power of the prediction error decreases with increasing predictor order. The error power decrease is controlled by the magnitude of the PARCOR coefficients. The closer the value of a PARCOR coefficient to unity, the higher the contribution of its stage in reducing the prediction error. The first few stages usually have PARCOR coefficients of high magnitudes. The magnitude of the coefficients decreases with increasing stage number.

Resources The resources required to implement the LBPEF of L stages in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$6L + 7$
MULTIPLY	$10L + 5 + sum(1 : L - 1)$
ADD	$7L + 2 + sum(1 : L - 1)$
DIVIDE	L

See Also INIT_ LBPEF, PREDICT_ LBPEF, ASPTLFPEF, ASPTLMSLATTICE.

Reference [2] and [4] for analysis of the adaptive Lattice filters.

5.3 asptlfpef

Purpose Implements an adaptive LMS Lattice Forward Prediction Error Filter.

Syntax `[k,b,P,e,y,c] = asptlfpef(k,b,P,x,mu_p)`

Description `asptlfpef()` is a lattice implementation of the forward prediction error filter shown in Fig. 5.4. Instead of updating the coefficients of a transversal filter, `asptlfpef()` updates the PARCOR coefficients of a lattice predictor of the same order. The forward prediction error of the last lattice stage is returned as the error signal $e(n)$ of the prediction error filter. If the number of output arguments are more than 5, the coefficients of the equivalent transversal prediction error filter are also calculated and returned in the variable c such that

$$e(n) = [1; -\underline{\mathbf{c}}]^T \; \underline{\mathbf{x}}(n-1) \qquad (5.2)$$

where $[1; -\underline{\mathbf{c}}]^T$ represents the impulse response between the PEF output e(n) and its input x(n) and $\underline{\mathbf{x}}(n-1)$ is the input signal delay line delayed by one sample.
The input and output parameters of `asptlfpef()` of L stages are summarized below.

```
Input Parameters::
   k   : vector of lattice predictor coefficients
   b   : vector of backward prediction error
   P   : vector of last estimated power of b
   x   : new input sample
   mu_p: adaptation constant for the predictor coefficient

Output parameters::
   k   : updated lattice predictor coefficients
   b   : updated backward prediction error vector
   P   : updated power estimate of b
   e   : forward prediction error
   y   : predictor output
   c   : Equivalent transversal predictor coefficients.
```

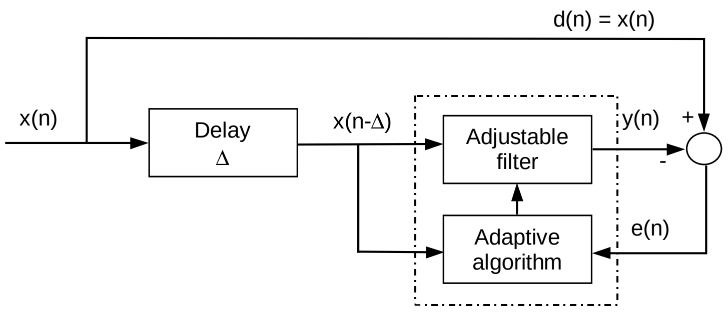

Figure 5.4: Block diagram of the forward prediction error filter.

Example

```
% LFPEF used in a adaptive line enhancer application.
% By the end of this script, the forward prediction error
% is the wide-band signal and the output of the equivalent
% transversal predictor is the narrow-band signal.
iter = 5000;
t    = (1:iter)/1000;             % time index @ 1kHz
xn   = 2*(rand(iter,1)-0.5)  ;   % Input signal, zero mean random
xn   = xn + 1 * cos(2*pi*50*t');% add 50 Hz narrow-band to input
yn   = zeros(iter,1);             % narrow-band signal
en   = yn;                        % error signal
% Initialize LFPEF
M    = 10;                        % filter length
mu_p = 0.01;                      % Step size
[k,b,P,e,y,c]=init_lfpef(M);     % Init LFPEF

%% Processing Loop
for (m=1:iter)
   [k,b,P,e,y,c] = asptlfpef(k,b,P,xn(m),mu_p);
   yn(m,:) = y;                % save narrow-band
   en(m,:) = e;                % save wide-band
end;

h = filter([1 ; -c],1,[1;zeros(1023,1)]);
f = (0:512)*500/512;
H = 20*log10(abs(fft(h)));
% display the results
subplot(2,2,1);plot(f,H(1:513,:)); grid;
subplot(2,2,2);
plot([yn(4800:5000)]);grid
```

Running the above script will produce the graph shown in Fig. 5.5. The left side graph of the figure shows the frequency response of the equivalent transversal prediction error filter after convergence. This frequency response shows that the predictor adjusts itself to pass the narrow-band signal at 50 Hz and attenuate all other input components so that the error signal contains the wide-band signal only. The right side graph shows the last 200 samples of the filter output which shows that the filter output coincide with the narrow-band 50 Hz superimposed on the white noise input signal.

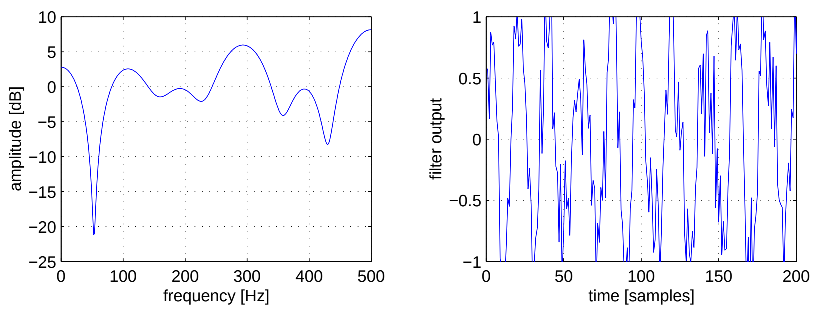

Figure 5.5: The frequency response of the PEF after convergence and the filter output for the adaptive line enhancer using LFPEF.

Algorithm `asptlfpef()` performs the following operations

- Calculates the forward and backward prediction errors for the lattice structure stages using the order update equations (see section 2.2.4),
- Calculates the power estimate of the backward prediction errors,
- Updates the PARCOR coefficients of the lattice predictor,
- Calculates the equivalent transversal predictor coefficients from the updated PARCOR coefficients,
- Calculates the equivalent transversal predictor output.

Remarks The adaptive Lattice Prediction Error Filter is a useful tool in linear prediction and autoregressive modeling applications. A few of the features that make the LFPEF so popular are:

- The PARCOR coefficients always satisfy the relation $|k_m| \leq 1$.
- The power of the forward prediction error $E[e^2_{fm}(n)]$ and the backward prediction error $E[e^2_{bm}(n)]$ of the same stage are equal.
- The backward prediction errors $e_{b0}(n), e_{b1}(n), \cdots, e_{bM}(n)$ are uncorrelated with one another for any input sequence $x(n)$. This property is very important since it shows that the lattice predictor can be seen as an orthogonal transformation with the input signal samples $x(n), x(n-1), \cdots, x(n-M+1)$ as input and the backward errors $e_{b0}(n), e_{b1}(n), \cdots, e_{bM}(n)$ as the uncorrelated (orthogonal) output.
- The power of the prediction error decreases with increasing predictor order. The error power decrease is controlled by the magnitude of the PARCOR coefficients. The closer the value of a PARCOR coefficient to unity, the higher the contribution of its stage in reducing the prediction error. The first few stages usually have PARCOR coefficients of high magnitudes. The magnitude of the coefficients decreases with increasing stage number.

Resources The resources required to implement the LFPEF of L stages in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$5L + 6$
MULTIPLY	$10L + 5 + sum(1 : L - 1)$
ADD	$7L + 2 + sum(1 : L - 1)$
DIVIDE	L

See Also INIT_ LFPEF, PREDICT_ LFPEF, ASPTLBPEF, ASPTLMSLATTICE.

Reference [2] and [4] for analysis of the adaptive Lattice filters.

5.4 asptlmslattice

Purpose Performs filtering and coefficient update for the LMS Lattice (joint process estimator) adaptive filter.

Syntax `[k,c,b,P,y,e]=asptlmslattice(k,c,b,P,x,d,mu_p,mu_c,upk)`

Description `asptlmslattice()` implements the joint process estimator shown in Fig. 5.6. The joint process estimator estimates a process $d(n)$ from another correlated process $x(n)$. It consists of two separate parts, the lattice predictor part and the linear combiner part. The main function of the lattice predictor part is to transform the input signal samples $x(n), x(n-1), \cdots, x(n-M+1)$ that might be well correlated to the uncorrelated backward prediction errors $e_{b0}(n), e_{b1}(n), \cdots, e_{bM}(n)$. The linear combiner part calculates the equivalent transversal filter output according to the relationship

$$y(n) = \sum_{i=1}^{M} c_i \, e_{bi}(n). \tag{5.3}$$

The adaptive joint process estimator adjusts both the PARCOR coefficients k_i; $i = 1, 2, \cdots, M$; and the linear combiner coefficients c_i; $i = 1, 2, \cdots, M$ simultaneously. The PARCOR coefficients are adjusted to minimize the forward and backward prediction error powers and the linear combiner coefficients are adjusted to minimize the mean square of the error signal $e(n) = d(n) - y(n)$.

The input and output parameters of `asptlmslattice()` of L stages are summarized below.

```
Input Parameters::
   k   : vector of lattice predictor coefficients (PARCOR)
   c   : vector of linear combiner coefficients
   b   : vector of backward prediction error
   P   : vector of last estimated power of b
   x   : new input sample
   d   : new desired sample
   mu_p: adaptation constant for the predictor coefficients
   mu_c: adaptation constant for the linear combiner coef.
   upk : flag if 0 will not update k
Output parameters::
   k   : updated lattice predictor coefficients
   c   : updated linear combiner coefficients
   b   : updated backward prediction error
   P   : updated power estimate of b
   y   : linear combiner output
   e   : error signal [e = d - y]
```

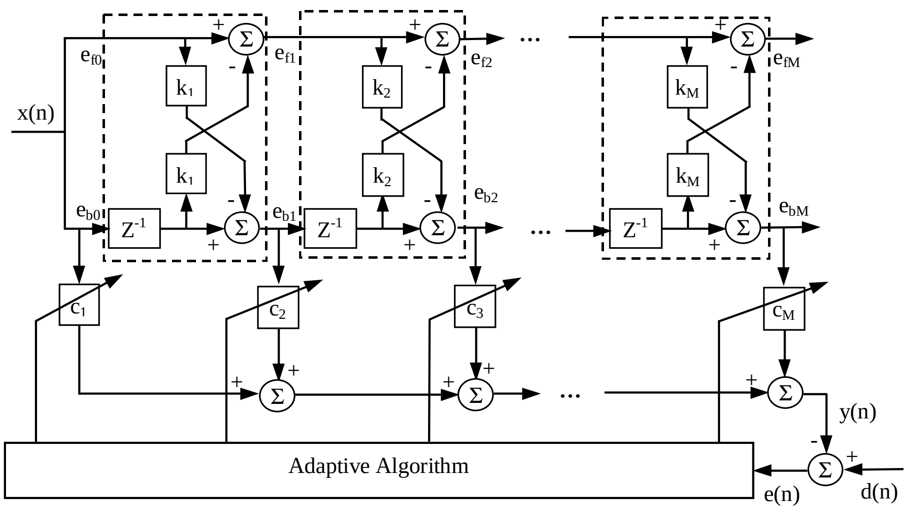

Figure 5.6: Block diagram of the adaptive Joint Process Estimator.

Example

```
% LMSLATTICE used in a simple system identification application.
% By the end of this script the adaptive filter w
% should have the same coefficients as the unknown filter h.

iter = 5000;                   % samples to process
% Complex unknown impulse response
h  = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xn = 2*(rand(iter,1)-0.5);    % Input signal
% although xn is real, dn will be complex since h is complex
dn = osfilter(h,xn);          % Unknown filter output
en = zeros(iter,1);           % error signal
% Initialize LMSLATTICE with a filter of 10 coef.
L    = 10;                    % filter length
mu_c = .01;                   % linear combiner step size
mu_p = 0.001;                 % lattice predictor step size
uk   = 1;
[k,w,b,P,d,y,e] = init_lmslattice(L);

%% Processing Loop
for (m=1:iter)
   % stop updating the PARCOR coef. after 2000 samples
   if (m == 2000), uk=0; end
   x = xn(m,:);                % new input sample
   d = dn(m,:) + 1e-3*rand;    % additive noise var = 1e-6
   [k,w,b,P,y,e]=asptlmslattice(k,w,b,P,x,d,mu_p,mu_c,uk);

   % save the last error sample to plot later
   en(m,:) = e;
end;

% display the results
subplot(2,2,1);stem([real(w) imag(conj(w))]); grid;
subplot(2,2,2);
eb = filter(.1,[1 -.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 5.7. The left side graph of the figure shows the adaptive linear combiner coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the mean square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB). Note that the final misadjustment is greatly affected by the fluctuations of the PARCOR coefficients when they are adapted. The estimation error drops sharply when adaptation of the PARCOR is stopped after 2000 samples. By that time the PARCOR should have reached their optimal values and should be fixed to allow lower final misadjustment.

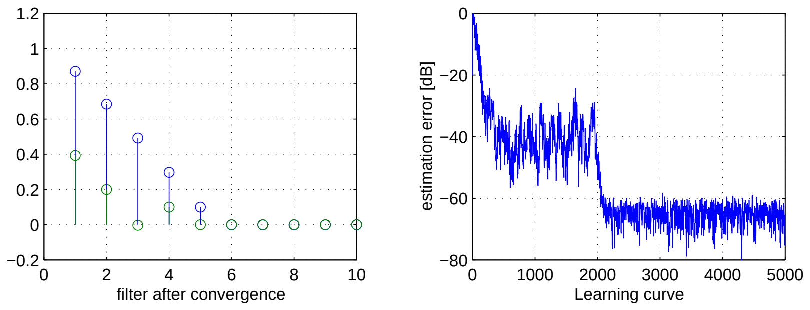

Figure 5.7: The adaptive linear combiner coefficients after convergence and the learning curve for the complex system identification problem using the LMSLATTICE algorithm.

Remarks

The joint process estimator simultaneously updates the PARCOR coefficients of the lattice predictor and the coefficients of the linear combiner. Updating the PARCOR coefficients result in changing all the backward prediction errors which are the inputs to the linear combiner part. This has the undesirable effect of increasing the final misadjustment due to the perturbation of the PARCOR coefficients. When the input is stationary, this problem can be solved by stopping the adaptation of the PARCOR coefficients after some time (as was done in the above example). This however does not help when the input $x(n)$ is non-stationary. The following points are also of interest.

- `asptlmslattice()` supports real as well as complex signals.
- The PARCOR coefficients always satisfy the relation $|k_m| \leq 1$.
- The power of the forward prediction error $E[e_{fm}^2(n)]$ and the backward prediction error $E[e_{bm}^2(n)]$ of the same stage are equal.
- The backward prediction errors $e_{b0}(n), e_{b1}(n), \cdots, e_{bM}(n)$ are uncorrelated with one another for any input sequence $x(n)$. This accelerates the convergence of the linear combiner coefficients.

Algorithm `asptlmslattice()` performs the following operations

- Calculates the forward and backward prediction errors for the lattice structure stages using the order update equations (see section 2.2.4),
- Calculates the power estimate of the backward prediction errors,
- Updates the PARCOR coefficients of the lattice predictor,
- Evaluates the linear combiner output,
- Evaluates the error signal,
- Updates the linear combiner coefficients.

Resources The resources required to implement the LMSLATTICE of length L in real time is given in the table below. The computations given are those required to process one sample.

	upk = 0	upk = 1
MEMORY	$6L+7$	$6L+7$
MULTIPLY	$10L-2$	$30L-2$
ADD	$7L-2$	$9L-2$
DIVIDE	L	2L

See Also INIT_ LMSLATTICE, MODEL_ LMSLATTICE.

Reference [2] and [4] for analysis of the adaptive Lattice filters.

5.5 asptrlslattice

Purpose Performs filtering and coefficient update for the Recursive Least Squares Lattice (joint process estimator) using the a posteriori estimation errors.

Syntax

```
[ff,bb,fb,be,cf,b,y,e,kf,kb,c]=
     asptrlslattice(ff,bb,fb,be,cf,b,a,x,d)
```

Description `asptrlslattice()` implements the joint process estimator shown in Fig. 5.8. Similar to the LMS joint process estimator (Fig. 5.6), it estimates a process $d(n)$ from another correlated process $x(n)$ and consists of two separate parts, the lattice predictor part and the linear combiner part. Unlike the LMS lattice however, the forward and backward PARCOR coefficients are not equal in the case of the RLS lattice structure.

The adaptive joint process estimator adjusts the forward PARCOR coefficients kf_i; $i = 1, 2, \cdots, M$, the backward PARCOR coefficients kb_i; $i = 1, 2, \cdots, M$, and the linear combiner coefficients c_i; $i = 1, 2, \cdots, M$ simultaneously. The PARCOR coefficients are adjusted to minimize the forward and backward prediction errors, while the linear combiner coefficients are adjusted to minimize the error signal $e(n)$ in the RLS sense.

The input and output parameters of `asptrlslattice()` of L stages are summarized below.

```
Input Parameters::
   ff   : last autocorrelation of forward prediction error (f)
   bb   : last autocorrelation of backward prediction error (b)
   fb   : last crosscorrelation of f and b
   be   : last crosscorrelation of b and e
   cf   : last conversion factor vector
   b    : last backward prediction error vector
   a    : forgetting factor
   x    : newest input sample x(n)
   d    : desired response d(n)
Output parameters::
   ff   : updated autocorrelation of forward prediction error
   bb   : updated autocorrelation of backward prediction error
   fb   : updated crosscorrelation of f and b
   be   : updated crosscorrelation of b and e
   cf   : updated conversion factor vector
   b    : updated backward prediction error vector
   y    : linear combiner output
   e    : error signal
   kf   : updated forward lattice coefficients kf(n)
   kb   : updated backward lattice coefficients kb(n)
   c    : updated linear combiner coefficients c(n)
```

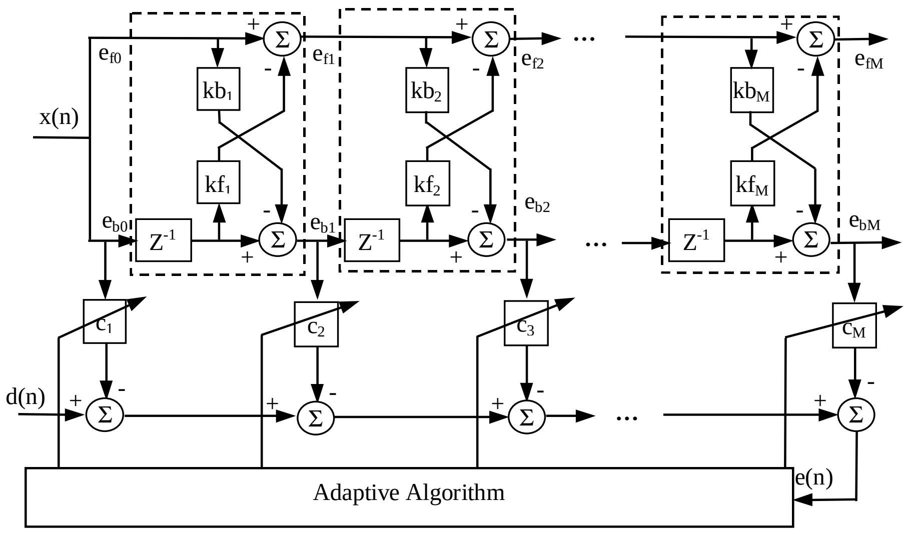

Figure 5.8: Block diagram of the RLS adaptive Joint Process Estimator.

Example

```
% RLSLATTICE used in a simple system identification application.
% By the end of this script the adaptive filter w
% should have the same coefficients as the unknown filter h.

iter = 5000;                  % samples to process
% Complex unknown impulse response
h  = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xn = 2*(rand(iter,1)-0.5);   % Input signal
% although xn is real, dn will be complex since h is complex
dn = osfilter(h,xn);         % Unknown filter output
en = zeros(iter,1);          % error signal

% Initialize RLSLATTICE with a filter of 10 coef.
L   = 10;                    % filter length
a   = .99;                   % forgetting factor
[ff,bb,fb,be,cf,b,d,y,e,kf,kb,w] = init_rlslattice(L);

%% Processing Loop
for (m=1:iter)

   x = xn(m,:);                % new input sample
   d = dn(m,:) + 1e-3*rand;    % additive noise var = 1e-6
   [ff,bb,fb,be,cf,b,y,e,kf,kb,w] = asptrlslattice(ff,bb,...
                                  fb,be,cf,b,a,x,d);
   % save the last error sample to plot later
   en(m,:) = e;
end;

% display the results
subplot(2,2,1);stem([real(w) imag(w)]); grid;
subplot(2,2,2);
eb = filter(.1, [1 -.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 5.9. The left side graph of the figure shows the adaptive linear combiner coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB). Note that the RLSLATTICE does not suffer from the fluctuations of the PARCOR coefficients and the final misadjustment is not affected by this fluctuations. The filter also shows very fast convergence rate and high degree of stability once it converged.

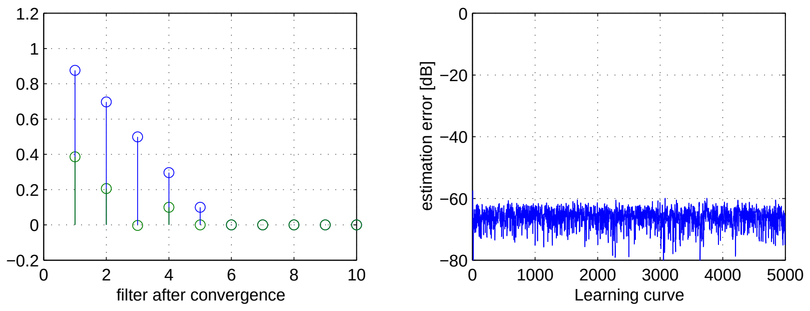

Figure 5.9: The adaptive linear combiner coefficients after convergence and the learning curve for the complex system identification problem using the RLSLATTICE algorithm.

Remarks

The joint process estimator simultaneously updates the PARCOR coefficients of the lattice predictor and the coefficients of the linear combiner. Updating the PARCOR coefficients result in changing all the backward prediction errors which are the inputs to the linear combiner. In the case of the LMSLATTICE this has the undesirable effect of increasing the final misadjustment due to the perturbation of the PARCOR coefficients. This problem does not appear to be of concern in the case of RLSLATTICE since the backward and forward prediction errors are minimized in the RLS sense using the exponentially windowed observed past prediction errors. The following points are also of interest.

- `asptrlslattice()` supports real as well as complex signals.
- The backward prediction errors $e_{b0}(n), e_{b1}(n), \cdots, e_{bM}(n)$ are uncorrelated with one another for any input sequence $x(n)$. This accelerates the convergence of the linear combiner coefficients.

Algorithm `asptrlslattice()` performs the following operations

- Calculates the backward (bb) and forward (ff) autocorrelations for each lattice stage,
- Calculates the crosscorrelation (fb) between forward and backward prediction errors for each lattice stage,
- Calculates the crosscorrelation (be) between backward prediction error and linear combiner error for each lattice stage,
- Calculates the forward and backward prediction errors for the lattice structure stages using the order update equations,
- Updates the forward and backward PARCOR coefficients of the lattice predictor and the conversion factor $e_{bn}/e_{b(n-1)}$,
- Evaluates the linear combiner output,
- Evaluates the error signal,
- Updates the linear combiner coefficients.

Resources The resources required to implement the RLSLATTICE of length L in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$10L+4$
MULTIPLY	$12L$
ADD	$8L+1$
DIVIDE	8L

See Also INIT_ RLSLATTICE, MODEL_ RLSLATTICE, ASPTRLSLATTICE2.

Reference [2] and [4] for analysis of the adaptive Lattice filters.

5.6 asptrlslattice2

Purpose Performs filtering and coefficient update for the Recursive Least Squares Lattice (joint process estimator) using the a priori estimation errors with error feedback

Syntax

```
[ff,bb,cf,b,y,e,kf,kb,c] =
     asptrlslattice2(ff,bb,kf,kb,c,cf,b,a,x,d)
```

Description `asptrlslattice2()` implements the joint process estimator shown in Fig. 5.10. Similar to the LMS joint process estimator (Fig. 5.6), it estimates a process $d(n)$ from another correlated process $x(n)$ and consists of two separate parts, the lattice predictor part and the linear combiner part. Unlike the LMS lattice however, the forward and backward PARCOR coefficients are not equal in the case of the RLS lattice structure.
The adaptive joint process estimator adjusts the forward PARCOR coefficients $kf_i;\ i = 1, 2, \cdots, M$, the backward PARCOR coefficients $kb_i;\ i = 1, 2, \cdots, M$, and the linear combiner coefficients $c_i;\ i = 1, 2, \cdots, M$ simultaneously. The PARCOR coefficients are adjusted to minimize the forward and backward prediction errors, while the linear combiner coefficients are adjusted to minimize the error signal $e(n)$ in the RLS sense.

The input and output parameters of `asptrlslattice2()` of L stages are summarized below.

```
Input Parameters::
   ff   : last autocorrelation of forward prediction error (f)
   bb   : last autocorrelation of backward prediction error (b)
   kf   : last forward lattice coefficients kf(n-1)
   kb   : last backward lattice coefficients kb(n-1)
   c    : last linear combiner coefficients c(n-1)
   cf   : last conversion factor vector
   b    : last backward a priori prediction error vector
   a    : forgetting factor
   x    : newest input sample x(n)
   d    : desired response d(n)

Output parameters::
   ff   : updated autocorrelation of forward prediction error
   bb   : updated autocorrelation of backward prediction error
   cf   : updated conversion factor vector
   b    : updated backward a priori estimation error vector
   y    : linear combiner output
   e    : error signal
   kf   : updated forward lattice coefficients kf(n)
   kb   : updated backward lattice coefficients kb(n)
   c    : updated linear combiner coefficients c(n)
```

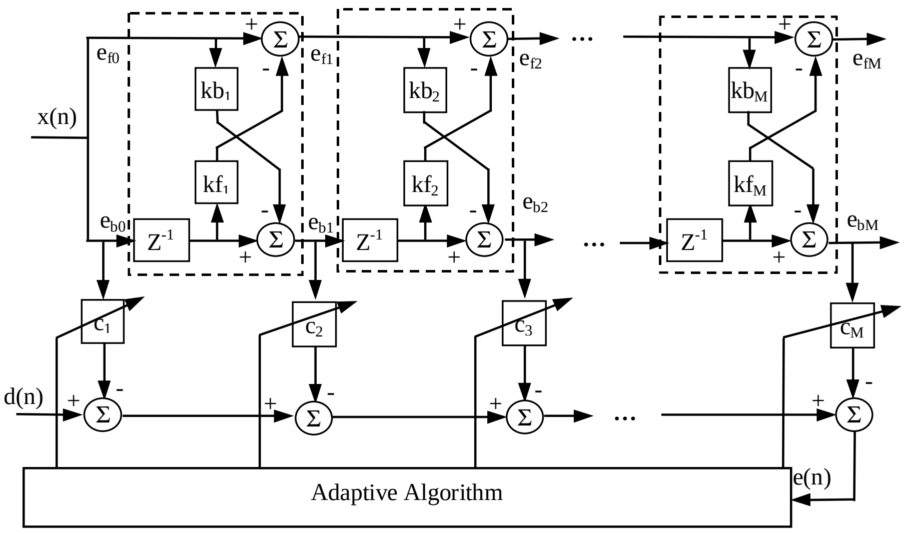

Figure 5.10: Block diagram of the RLS adaptive Joint Process Estimator.

Example

```
% RLSLATTICE2 used in a simple system identification application.
% By the end of this script the adaptive filter w
% should have the same coefficients as the unknown filter h.

iter = 5000;                 % samples to process
% Complex unknown impulse response
h  = [.9 + i*.4; 0.7+ i*.2; .5; .3+i*.1; .1];
xn = 2*(rand(iter,1)-0.5);   % Input signal
% although xn is real, dn will be complex since h is complex
dn = osfilter(h,xn);         % Unknown filter output
en = zeros(iter,1);          % error signal

% Initialize RLSLATTICE2 with a filter of 10 coef.
L   = 10;                    % filter length
a   = .99;                   % forgetting factor
[ff,bb,cf,b,d,y,e,kf,kb,w]=init_rlslattice2(L);

%% Processing Loop
for (m=1:iter)

   x = xn(m,:);               % new input sample
   d = dn(m,:) + 1e-3*rand;   % additive noise var = 1e-6
   [ff,bb,cf,b,y,e,kf,kb,w] =
                asptrlslattice2(ff,bb,kf,kb,w,cf,b,a,x,d);
   % save the last error sample to plot later
   en(m,:) = e;
end;

% display the results
subplot(2,2,1);stem([real(w) imag(w)]); grid;
subplot(2,2,2);
eb = filter(.1, [1 -.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 5.11. The left side graph of the figure shows the adaptive linear combiner coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB). Note that the `asptrlslattice2()` does not suffer from the fluctuations of the PARCOR coefficients and the final misadjustment is not affected by this fluctuations. The filter also shows very fast convergence rate and high degree of stability once it converged.

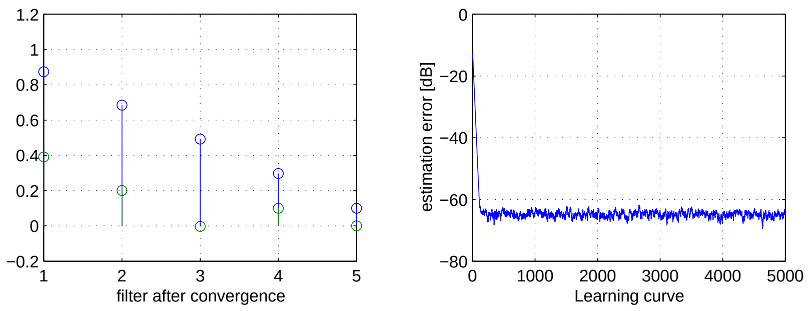

Figure 5.11: The adaptive linear combiner coefficients after convergence and the learning curve for the complex system identification problem using the RLSLATTICE-2 algorithm.

Remarks

The joint process estimator simultaneously updates the PARCOR coefficients of the lattice predictor and the coefficients of the linear combiner. Updating the PARCOR coefficients results in changing all the backward prediction errors which are the inputs to the linear combiner. In the case of the LMSLATTICE this has the undesirable effect of increasing the final misadjustment due to the perturbation of the PARCOR coefficients. This problem does not appear to be of concern in the case of RLSLATTICE algorithms since the backward and forward prediction errors are minimized in the RLS sense using the exponentially windowed observed past prediction errors. The following points are also of interest.

- `asptrlslattice2()` supports real as well as complex signals.
- The backward prediction errors $e_{b0}(n), e_{b1}(n), \cdots, e_{bM}(n)$ are uncorrelated with one another for any input sequence $x(n)$. This accelerates the convergence of the linear combiner coefficients.
- `asptrlslattice2()` has been found to be less sensitive to numerical rounding errors compared to `asptrlslattice()`.

Algorithm `asptrlslattice2()` performs the following operations

- Calculates the backward (bb) and forward (ff) autocorrelations for each lattice stage,
- Calculates the forward and backward prediction errors for the lattice structure stages using the order update equations,
- Updates the forward and backward PARCOR coefficients of the lattice predictor and the conversion factor $e_{bn}/e_{b(n-1)}$,
- Evaluates the linear combiner output and error signal,
- Updates the linear combiner coefficients,
- Updates the conversion factor vector.

Resources The resources required to implement an RLSLATTICE-2 filter of length L in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$9L + 6$
MULTIPLY	$9L$
ADD	$18L + 1$
DIVIDE	4L

See Also INIT_ RLSLATTICE2, ASPTRLSLATTICE.

Reference [2] and [4] for analysis of the adaptive Lattice filters.

5.7 asptrlslbpef

Purpose Implements the adaptive RLS Lattice Backward Prediction Error Filter.

Syntax `[ff,bb,fb,cf,b,y,e,kf,kb,c]=asptrlslbpef(ff,bb,fb,cf,b,a,x)`

Description `asptrlslbpef()` is a lattice implementation of the backward prediction error filter shown in Fig. 5.12. `asptrlslbpef` updates the PARCOR coefficients of the lattice predictor by minimizing the backward and forward prediction errors in the recursive least squares sense. The backward prediction error of the last lattice stage is returned as the error signal $e(n)$ of the prediction error filter. If the number of output arguments are more than 9, the coefficients of the equivalent transversal prediction error filter are also calculated and returned in the variable c such that $e(n) = [1; -\underline{\mathbf{c}}]^T\ \underline{\mathbf{x}}(n-L)$; where $[1; -\underline{\mathbf{c}}]^T$ represents the impulse response between the PEF output e(n) and its input x(n) and $\underline{\mathbf{x}}(n-L)$ is the input signal delay line delayed by L samples.

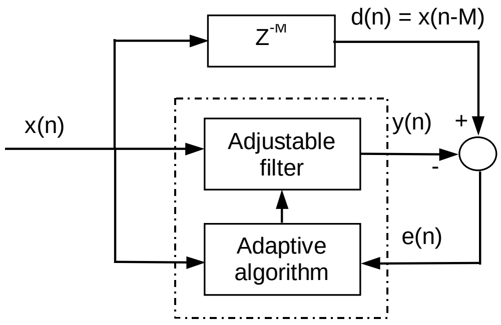

Figure 5.12: Block diagram of the backward prediction error filter.

The input and output parameters of `asptrlslbpef()` of L stages are summarized below.

```
Input Parameters::
   ff   : last autocorrelation of forward prediction error (f)
   bb   : last autocorrelation of backward prediction error (b)
   fb   : last crosscorrelation of f and b
   cf   : last conversion factor vector
   b    : last backward prediction error vector
   a    : forgetting factor
   x    : newest input sample x(n)
Output parameters::
   ff   : updated autocorrelation of forward prediction error
   bb   : updated autocorrelation of backward prediction error
   fb   : updated crosscorrelation of f and b
   cf   : updated conversion factor vector
   b    : updated backward prediction error vector
   y    : linear combiner output
   e    : forward prediction error
   kf   : updated forward lattice coefficients kf(n)
   kb   : updated backward lattice coefficients kb(n)
   c    : equivalent transversal backward predictor coefficients.
```

Example

```
% RLSLBPEF used in a adaptive line enhancer application.
% By the end of this script, the backward prediction error
% is the wide-band signal and the output of the equivalent
% transversal predictor is the narrow-band signal.

iter = 5000;
t    = (1:iter)/1000;           % time index @ 1kHz
xn   = 2*(rand(iter,1)-0.5)  ;  % Input signal, zero mean random.
xn   = xn + 1 * cos(2*pi*50*t');
yn   = zeros(iter,1);           % narrow-band signal
en   = yn;                      % error signal

% Initialize RLSLBPEF
M    = 10;                      % filter length
a    = 0.99;                    % forgetting factor
[ff,bb,fb,cf,b,y,e,kf,kb,x] = init_rlslbpef(M);

%% Processing Loop
for (m=1:iter)
   x = [xn(m); x(1:end-1)];
   [ff,bb,fb,cf,b,y,e,kf,kb,c]=asptrlslbpef(ff,bb,fb,cf,b,a,x);
   yn(m,:) = y;              % save narrow-band
   en(m,:) = e;              % save wide-band
end;

% Transfer function between e(n) and x(n-L).
h = filter([ 1;(-c)],1,[1; zeros(1023,1)]);
f = (0:512)*500/512;
H = 20*log10(abs(fft(h)));
% display the results
subplot(2,2,1);plot(f,H(1:513,:)); grid;
subplot(2,2,2);
plot([yn(4800:5000)]);grid
```

Running the above script will produce the graph shown in Fig. 5.13. The left side graph of the figure shows the frequency response of the equivalent transversal prediction error filter after convergence. This frequency response shows that the predictor adjusts itself to pass the narrow-band signal at 50 Hz and attenuate all other input components so that the error signal contains the wide-band signal only. The right side graph shows the last 200 samples of the filter output which shows that the filter output coincide with the narrow-band 50 Hz superimposed on the white noise input signal.

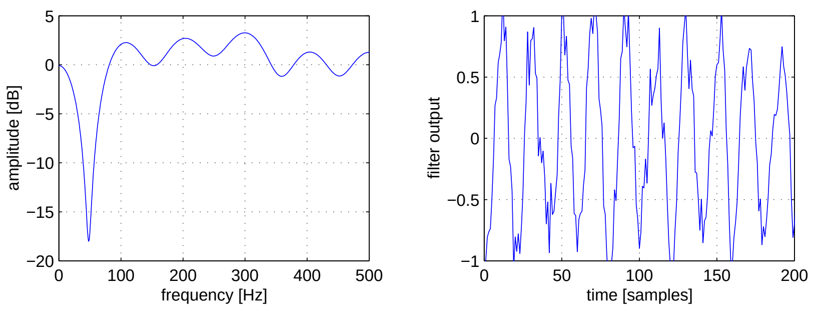

Figure 5.13: The frequency response of the PEF after convergence and the filter output for the adaptive line enhancer using RLSLBPEF.

Algorithm

`asptrlslbpef()` performs the following operations

- Calculates the backward (bb) and forward (ff) autocorrelations for each lattice stage,
- Calculates the crosscorrelation (fb) between forward and backward prediction errors for each lattice stage,
- Calculates the forward and backward prediction errors for the lattice structure stages using the order update equations,
- Updates the forward and backward PARCOR coefficients of the lattice predictor and the conversion factor $e_{bn}/e_{b(n-1)}$,
- Evaluates the error signal and equivalent transversal predictor output,
- Evaluates the equivalent transversal predictor coefficients if required.

Remarks

The adaptive Lattice Prediction Error Filter is a useful tool in linear prediction and autoregressive modeling applications. The following remarks apply to the RLS lattice backward prediction error filter:

- Unlike the LMS LBPEF, the forward and backward lattice prediction coefficients of the same lattice stage are not equal.
- The linear combiner coefficients are optimized sequentially starting with c_1 and ending with c_M.
- The backward prediction errors $e_{b0}(n), e_{b1}(n), \cdots, e_{bM}(n)$ are uncorrelated with one another for any input sequence $x(n)$. This property is very important since it shows that the lattice predictor can be seen as an orthogonal transformation with the input signal samples $x(n), x(n-1), \cdots, x(n-M+1)$ as input and the backward errors $e_{b0}(n), e_{b1}(n), \cdots, e_{bM}(n)$ as the uncorrelated (orthogonal) output.
- The power of the prediction error decreases with increasing predictor order. The error power decrease is controlled by the magnitude of the PARCOR coefficients. The closer the value of a PARCOR coefficient to unity, the higher the contribution of its stage in reducing the prediction error. The first few stages usually have PARCOR coefficients of high magnitudes. The magnitude of the coefficients decreases with increasing stage number.

Resources The resources required to implement the RLSLBPEF of L stages in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$8L + 3$
MULTIPLY	$10L$
ADD	$7L$
DIVIDE	$7L$

See Also INIT_ RLSLBPEF, PREDICT_ RLSLBPEF, ASPTRLSLFPEF, ASPTRLS-LATTICE.

Reference [2] and [4] for analysis of the adaptive Lattice filters.

5.8 asptrlslfpef

Purpose Implements the adaptive RLS Lattice Forward Prediction Error Filter.

Syntax `[ff,bb,fb,cf,b,y,e,kf,kb,c]=asptrlslfpef(ff,bb,fb,cf,b,a,x)`

Description `asptrlslfpef()` is a lattice implementation of the forward prediction error filter shown in Fig. 5.14. `asptrlslfpef()` updates the PARCOR coefficients of the lattice predictor by minimizing the backward and forward prediction errors in the recursive least squares sense. The forward prediction error of the last lattice stage is returned as the error signal $e(n)$ of the prediction error filter. If the number of output arguments are more than 9, the coefficients of the equivalent transversal prediction error filter are also calculated and returned in the variable c such that $e(n) = [1; -\underline{\mathbf{c}}]^T\ \underline{\mathbf{x}}(n-1)$; where $[1; -\underline{\mathbf{c}}]^T$ represents the impulse response between the PEF output e(n) and its input x(n) and $\underline{\mathbf{x}}(n-1)$ is the input signal delay line delayed by one sample.

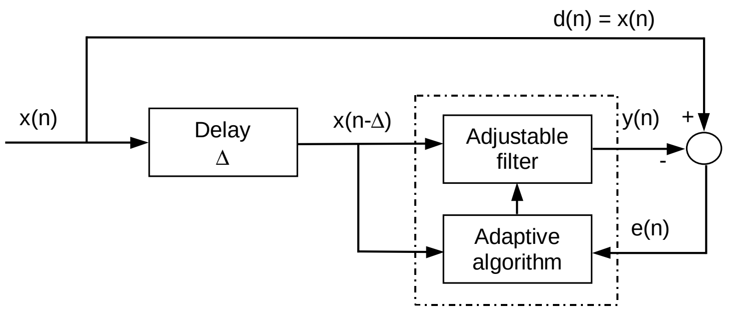

Figure 5.14: Block diagram of the forward prediction error filter.

The input and output parameters of `asptrlslfpef()` of L stages are summarized below.

```
Input Parameters::
   ff  : last autocorrelation of forward prediction error (f)
   bb  : last autocorrelation of backward prediction error (b)
   fb  : last crosscorrelation of f and b
   cf  : last conversion factor vector
   b   : last backward prediction error vector
   a   : forgetting factor
   x   : newest input sample x(n)
Output parameters::
   ff  : updated autocorrelation of forward prediction error
   bb  : updated autocorrelation of backward prediction error
   fb  : updated crosscorrelation of f and b
   cf  : updated conversion factor vector
   b   : updated backward prediction error vector
   y   : linear combiner output
   e   : forward prediction error
   kf  : updated forward lattice coefficients kf(n)
   kb  : updated backward lattice coefficients kb(n)
   c   : equivalent transversal forward predictor coefficients.
```

Example

```
% RLSLFPEF used in a adaptive line enhancer application.
% By the end of this script, the backward prediction error
% is the wide-band signal and the output of the equivalent
% transversal predictor is the narrow-band signal.

iter = 5000;
t    = (1:iter)/1000;            % time index @ 1kHz
xn   = 2*(rand(iter,1)-0.5)  ;   % Input signal, zero mean random.
xn   = xn + 1 * cos(2*pi*50*t');
yn   = zeros(iter,1);            % narrow-band signal
en   = yn;                       % error signal

% Initialize RLSLFPEF
M    = 10;                       % filter length
a    = 0.99;                     % forgetting factor
[ff,bb,fb,cf,b,y,e,kf,kb] = init_rlslfpef(M);

%% Processing Loop
for (m=1:iter)
   x = xn(m);
   [ff,bb,fb,cf,b,y,e,kf,kb,c]=asptrlslfpef(ff,bb,fb,cf,b,a,x);
   yn(m,:) = y;               % save narrow-band
   en(m,:) = e;               % save wide-band
end;

% Transfer function between e(n) and x(n-L).
h = filter([ 1;(-c)],1,[1; zeros(1023,1)]);
f = (0:512)*500/512;
H = 20*log10(abs(fft(h)));
% display the results
subplot(2,2,1);plot(f,H(1:513,:)); grid;
subplot(2,2,2);
plot([yn(4800:5000)]);grid
```

Running the above script will produce the graph shown in Fig. 5.15. The left side graph of the figure shows the frequency response of the equivalent transversal prediction error filter after convergence. This frequency response shows that the predictor adjusts itself to pass the narrow-band signal at 50 Hz and attenuate all other input components so that the error signal contains the wide-band signal only. The right side graph shows the last 200 samples of the filter output which shows that the filter output coincide with the narrow-band 50 Hz superimposed on the white noise input signal.

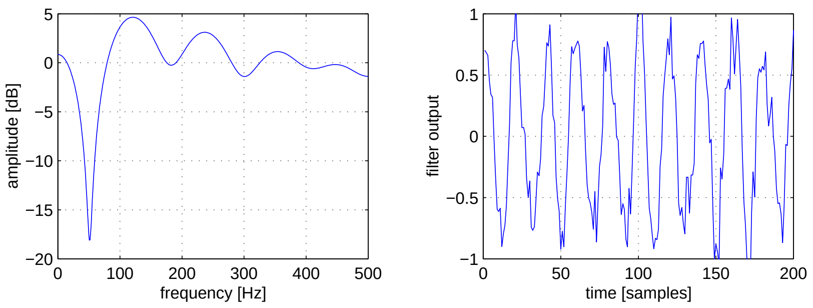

Figure 5.15: The frequency response of the PEF after convergence and the filter output for the adaptive line enhancer using RLSLFPEF.

Algorithm

`asptrlslfpef()` performs the following operations

- Calculates the backward (bb) and forward (ff) autocorrelations for each lattice stage,
- Calculates the crosscorrelation (fb) between forward and backward prediction errors for each lattice stage,
- Calculates the forward and backward prediction errors for the lattice structure stages using the order update equations,
- Updates the forward and backward PARCOR coefficients of the lattice predictor and the conversion factor $e_{bn}/e_{b(n-1)}$,
- Evaluates the error signal and equivalent transversal predictor output,
- Evaluates the equivalent transversal predictor coefficients if required.

Remarks

The adaptive Lattice Prediction Error Filter is a useful tool in linear prediction and autoregressive modeling applications. The following remarks apply to the RLS lattice forward prediction error filter:

- Unlike the LMS LFPEF, the forward and backward lattice prediction coefficients of the same lattice stage are not equal.
- The linear combiner coefficients are optimized sequentially starting with c_1 and ending with c_M.
- The backward prediction errors $e_{b0}(n), e_{b1}(n), \cdots, e_{bM}(n)$ are uncorrelated with one another for any input sequence $x(n)$. This property is very important since it shows that the lattice predictor can be seen as an orthogonal transformation with the input signal samples $x(n), x(n-1), \cdots, x(n-M+1)$ as input and the backward errors $e_{b0}(n), e_{b1}(n), \cdots, e_{bM}(n)$ as the uncorrelated (orthogonal) output.
- The power of the prediction error decreases with increasing predictor order. The error power decrease is controlled by the magnitude of the PARCOR coefficients. The closer the value of a PARCOR coefficient to unity, the higher the contribution of its stage in reducing the prediction error. The first few stages usually have PARCOR coefficients of high magnitudes. The magnitude of the coefficients decreases with increasing stage number.

Resources The resources required to implement the RLSLFPEF of L stages in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$8L + 3$
MULTIPLY	$10L$
ADD	$7L$
DIVIDE	$7L$

See Also INIT_ RLSLFPEF, PREDICT_ RLSLFPEF, ASPTRLSLBPEF, ASPTRLS-LATTICE.

Reference [2] and [4] for analysis of the adaptive Lattice filters.

5.9 init_ ftrls

Purpose Creates and initializes the variables required for the Fast Transversal Recursive Least Squares algorithm.

Syntax

```
[ff,bb,k,cf,c,g,w,x,d,e,y] = init_ftrls(L)
[ff,bb,k,cf,c,g,w,x,d,e,y] =
       init_ftrls(L,ff0,bb0,k0,cf0,c0,g0,w0,x0,d0)
```

Description The variables of the FTRLS algorithm are summarized below.

```
Input Parameters [Size]::
  L   : Linear combiner length
  ff0 : initial autocorrelation of forward prediction error [1x1]
  bb0 : initial autocorrelation of backward prediction error [1x1]
  k0  : initial normalized gain vector [Lx1]
  cf0 : initial conversion factor [1x1]
  c0  : initial forward predictor coefficients [L+1 x 1]
  g0  : initial backward predictor coefficients [L+1 x 1]
  w0  : initial linear combiner coefficients [L x 1]
  x0  : initial input samples vector [L+1 x 1]
  d0  : initial desired response [1x1]

Output parameters [Default]::
  ff  : autocorrelation of forward prediction error [.001]
  bb  : autocorrelation of backward prediction error [.001]
  k   : normalized gain vector [ones].
  cf  : conversion factor [1]
  c   : forward predictor coefficients [.1*ones(L+1,1)]
  g   : backward predictor coefficients [.1*ones(L+1,1)]
  w   : linear combiner coefficients [zeros]
  x   : input samples vector [zeros]
  d   : desired output [rand]
  y   : Linear combiner output [w' * x(1:end-1)]
  e   : Linear combiner error [d-y]
```

Example

```
L  = 5;                  % Number of lattice stages
ff = 0.01;               % autocorr. of forward prediction error
bb = 0.01;               % autocorr. of backward prediction error
k  = zeros(L,1);         % normalized gain vector
cf = 1;                  % conversion factor
c  = .01*ones(L+1,1);    % forward prediction coef. vector
g  = zeros(L+1,1);       % backward prediction coef. vector
w  = zeros(L,1);         % linear combiner coef. vector

% Create and initialize an RLS lattice filter
[ff,bb,k,cf,c,g,w,x,d,e,y]=init_ftrls(L,ff,bb,k,cf,c,g,w,[],0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 2 through 9 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations and is also used for soft initialization when instability signs are detected.

See Also ASPTFTRLS, ASPTRLSLATTICE2.

5.10 init_ lbpef

Purpose Creates and initializes the variables required for the Least Mean Squares Lattice Backward Prediction Error Filter.

Syntax

```
[k,b,P,e,y,x,c]=init_lbpef(L)
[k,b,P,e,y,x,c]=init_lbpef(L,k0,x0,b0,P0)
```

Description The block diagram of the lattice backward prediction error filter is shown in Fig. 5.2 while the details of the lattice structure showing its internal variables can be seen in Fig. 5.16. A summary of those variables is given below.

```
Input Parameters [Size]::
   L   : number of lattice coefficients
   k0  : initial lattice predictor coefficients [L x 1]
   x0  : initial input delay line [(L+1)x1]
   b0  : initial backward prediction errors [(L+1)x1]
   P0  : initial power of b [(L+1)x1]
Output parameters::
   k   : lattice predictor coefficients [zeros]
   b   : backward prediction errors [random]
   P   : estimated power of b [b .* b]
   e   : forward prediction error [random]
   y   : predictor output [0]
   x   : input delay line [random(L+1,1)]
   c   : equivalent transversal predictor coef.
```

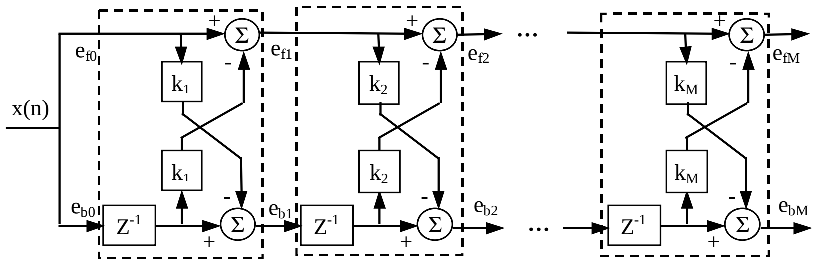

Figure 5.16: Block diagram of the lattice predictor.

Example

```
L  = 5;                  % Number of lattice stages
k0 = zeros(L,1);         % initial PARCOR coefficients
b0 = rand(L+1,1);        % initial backward errors
P0 = b0 .* conj(b0);     % initial power of b

% Create and initialize a lattice FPEF
[k,b,P,e,y,x,c]=init_lbpef(L,k0,[],b0,P0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 2 through 5 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTLBPEF, PREDICT_ LBPEF.

5.11 init_ lfpef

Purpose Creates and initializes the variables required for the Least Mean Squares Lattice Forward Prediction Error Filter.

Syntax

```
[k,b,P,e,y,c]=init_lfpef(L)
[k,b,P,e,y,c]=init_lfpef(L,k0,b0,P0)
```

Description The block diagram of the lattice forward prediction error filter is shown in Fig. 5.4 while the details of the lattice structure showing its internal variables can be seen in Fig. 5.17. A summary of those variables is given below.
The variables of the LFPEF are summarized below (see Fig. 5.17).

```
Input Parameters [Size]::
    L   : number of filter coefficients
    k0  : initial lattice predictor coefficients [L x 1]
    b0  : initial backward prediction errors [(L+1)x1]
    P0  : initial power of b [(L+1)x1]

Output parameters::
    k   : lattice predictor coefficients [zeros]
    b   : backward prediction errors [random]
    P   : estimated power of b [b .* b]
    e   : forward prediction error [random]
    y   : predictor output [0]
    c   : equivalent transversal predictor coef.
```

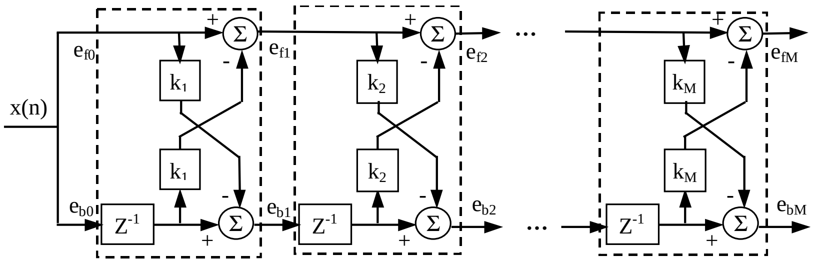

Figure 5.17: Block diagram of the lattice predictor.

Example

```
L  = 5;                 % Number of lattice stages
k0 = zeros(L,1);        % initial PARCOR coefficients
b0 = rand(L+1,1);       % initial backward errors
P0 = b0 .* conj(b0);    % initial power of b

% Create and initialize a lattice FPEF
[k,b,P,fM,y,c]=init_lfpef(L,k0,b0,P0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 2 through 4 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTLFPEF, PREDICT_ LFPEF.

5.12 init_ lmslattice

Purpose Creates and initializes the variables required for the LMS Lattice Joint Process Estimator.

Syntax

```
[k,c,b,P,d,y,e]=init_lmslattice(L)
[k,c,b,P,d,y,e]=init_lmslattice(L,k0,c0,b0,P0,d0)
```

Description The LMS-Lattice joint process estimator simultaneously adapts the PARCOR coefficients of a lattice predictor and the coefficients of the linear combiner as shown in Fig. 5.18. The variables of the LMS-LATTICE algorithms are summarized below.

```
Input Parameters [Size]::
   L   : number of linear combiner coefficients
   k0  : initial lattice predictor coefficients [Lx1]
   c0  : initial linear combiner coefficients [Lx1]
   b0  : initial backward prediction errors [Lx1]
   P0  : initial power of b [Lx1]
   d0  : initial desired sample [1x1]

Output parameters [default]::
   k   : lattice predictor coefficients [zeros]
   c   : linear combiner coefficients [zeros]
   b   : backward prediction errors [white noise]
   P   : estimated power of b [b .* b]
   d   : desired response [white noise]
   y   : linear combiner output [c’ * b]
   e   : error signal [e = d - y]
```

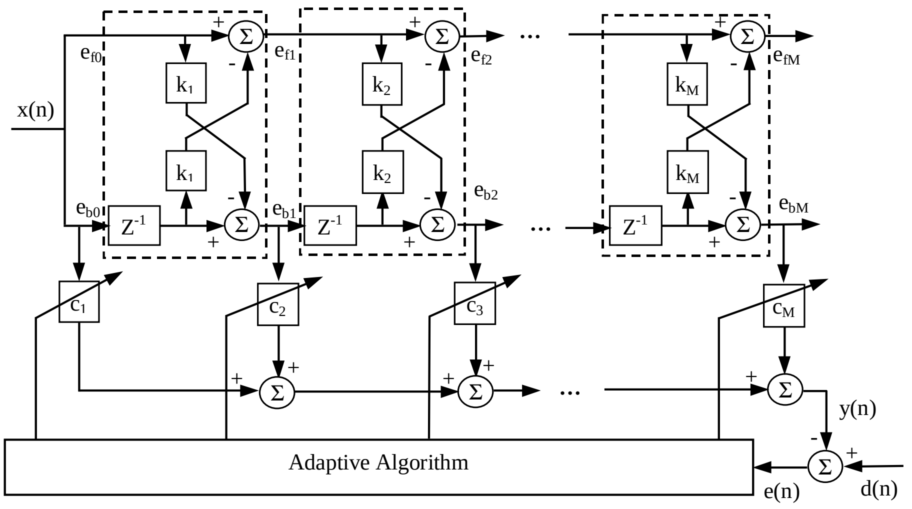

Figure 5.18: Block diagram of the LMS-LATTICE Joint Process Estimator.

Example

```
L  = 5;                  % Number of lattice stages
k0 = zeros(L,1);         % initial PARCOR coefficients
c0 = zeros(L,1);         % initial linear combiner coef.
b0 = rand(L,1);          % initial backward errors
P0 = b0 .* conj(b0);     % initial power of b
d0 = .22;                % initial desired sample

% Create and initialize an LMS lattice filter
[k,c,b,P,d,y,e]=init_lmslattice(L,k0,c0,b0,P0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 2 through 6 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTLMSLATTICE, MODEL_ LMSLATTICE.

5.13 init_ rlslattice

Purpose Creates and initializes the variables required for the RLS-Lattice Joint Process Estimator using the a posteriori estimation errors.

Syntax

```
[ff,bb,fb,be,cf,b,d,y,e,kf,kb,c] = init_rlslattice(L)
[ff,bb,fb,be,cf,b,d,y,e,kf,kb,c] = init_rlslattice(L,ff0,
                                   bb0,fb0,be0,cf0,b0,d0)
```

Description The variables of the RLS LATTICE are summarized below (see Fig. 5.19).

```
Input Parameters [Size]::
  L   : Linear combiner length
  ff0 : initial autocorr. of forward prediction error [Lx1]
  bb0 : initial autocorr. of backward prediction error [Lx1]
  fb0 : initial crosscorrelation of f and b [Lx1]
  be0 : initial crosscorrelation of b and e [Lx1]
  cf0 : initial conversion factor [Lx1]
  b0  : initial vector of backward prediction error [Lx1]
  d0  : initial desired response [1x1]
Output parameters [Default]::
  ff  : autocorr. of forward prediction error [.001*ones(L,1)]
  bb  : autocorr. of backward prediction error [.001*ones(L,1)]
  fb  : crosscorrelation of f and b [zeros]
  be  : crosscorrelation of b and e [zeros]
  cf  : conversion factor [ones]
  b   : vector of backward prediction error [zeros]
  d   : desired output [rand]
  y   : Linear combiner output [c' * b]
  e   : Linear combiner error [d-y]
  kf  : forward lattice coefficients [zeros]
  kb  : backward lattice coefficients [zeros]
  c   : linear combiner coefficients [zeros]
```

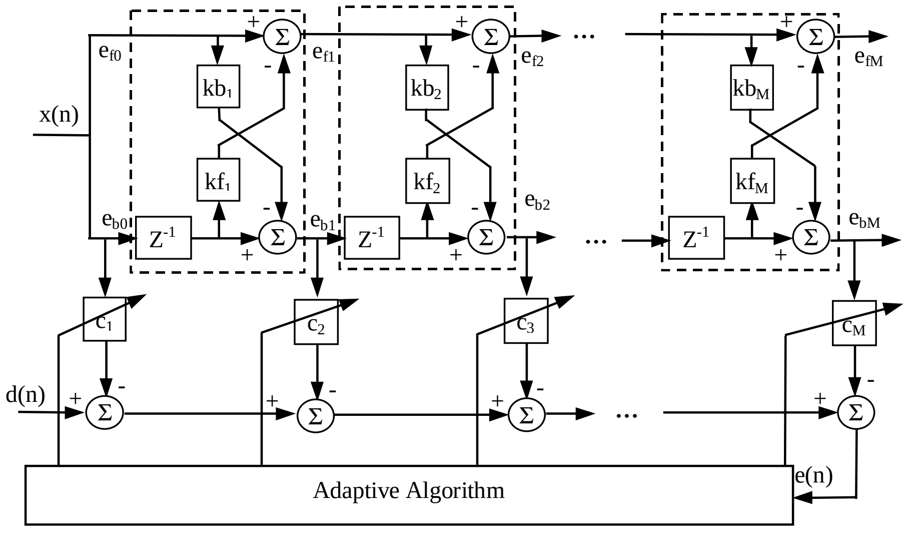

Figure 5.19: Block diagram of the RLS-LATTICE adaptive Joint Process Estimator.

Example

```
L  = 5;                  % Number of lattice stages
ff = 0.001*ones(L,1);    % autocorr. for forward prediction error
bb = 0.001*ones(L,1);    % autocorr. for backward prediction error
fb = zeros(L,1);         % crosscorr. between f and b
be = zeros(L,1);         % crosscorr. between b and e
cf = zeros(L,1);         % conversion factor
b  = zeros(L,1);         % backward prediction error
d  = .22;                % initial desired sample

% Create and initialize an RLS lattice filter
[ff,bb,fb,be,cf,b,d,y,e,kf,kb,c] = init_rlslattice(L,ff,...
                                   bb,fb,be,cf,b,d);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 2 through 8 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTRLSLATTICE, MODEL_ LMSLATTICE, ASPTRLSLATTICE2.

5.14 init_ rlslattice2

Purpose Creates and initializes the variables required for the RLS-Lattice Joint Process Estimator using the a priori estimation errors with error feedback.

Syntax

```
[ff,bb,cf,b,d,y,e,kf,kb,c] = init_rlslattice2(L)
[ff,bb,cf,b,d,y,e,kf,kb,c] =
    init_rlslattice2(L,ff0,bb0,kf0,kb0,c0,cf0,b0,d0)
```

Description The variables of the RLS LATTICE-2 are summarized below (see Fig. 5.20).

```
Input Parameters [Size]::
  L   : Linear combiner length
  ff0 : autocorrelation of forward prediction error [Lx1]
  bb0 : autocorrelation of backward prediction error [Lx1]
  kf0 : forward lattice coefficients [Lx1]
  kb0 : backward lattice coefficients [Lx1]
  c0  : linear combiner coefficients [Lx1]
  cf0 : conversion factor [Lx1]
  b0  : vector of backward a priori estimation error [Lx1]
  d0  : desired response [1x1]
Output parameters [Default]::
  ff  : initialized ff [.001*ones(L,1)]
  bb  : initialized bb [.001*ones(L,1)]
  cf  : conversion factor [ones]
  b   : vector of a priori backward estimation error [zeros]
  d   : desired output [rand]
  y   : Linear combiner output [c' * b]
  e   : Linear combiner error [d-y]
  kf  : forward lattice coefficients [zeros]
  kb  : backward lattice coefficients [zeros]
  c   : linear combiner coefficients [zeros]
```

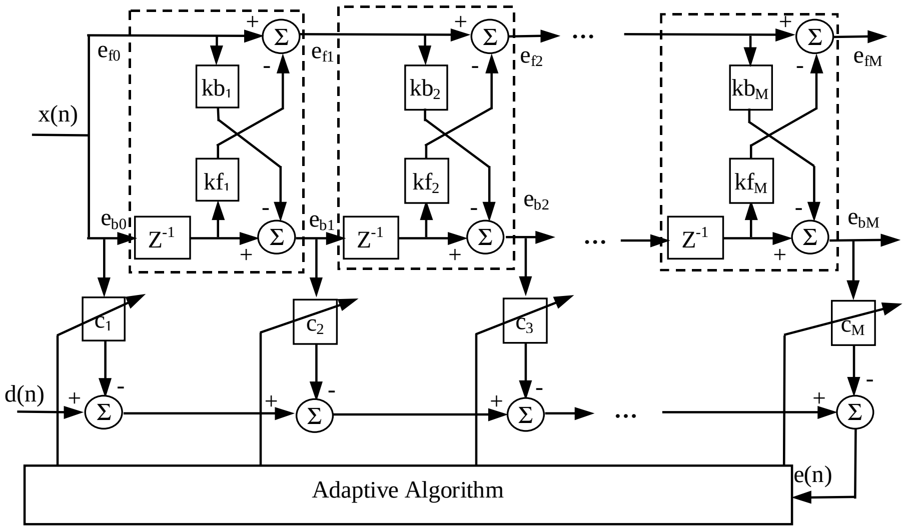

Figure 5.20: Block diagram of the RLSLATTICE-2 adaptive Joint Process Estimator.

Example

```
L  = 5;               % Number of lattice stages
ff = 0.001*ones(L,1); % autocorr. of forward prediction error
bb = 0.001*ones(L,1); % autocorr. of backward prediction error
kf = zeros(L,1);      % forward lattice coefficients
kb = zeros(L,1);      % backward lattice coefficients
c  = zeros(L,1);      % Linear combiner coefficients
cf = zeros(L,1);      % conversion factor
b  = zeros(L,1);      % backward prediction error
d  = .22;             % initial desired sample

% Create and initialize an RLS lattice-2 filter
[ff,bb,cf,b,d,y,e,kf,kb,c] =
           init_rlslattice2(L,ff0,bb0,kf0,kb0,c0,cf0,b0,d0)
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 2 through 9 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTRLSLATTICE2.

5.15 init_ rlslbpef

Purpose Creates and initializes the variables required for the RLS Lattice Backward Prediction Error Filter.

Syntax

```
[ff,bb,fb,cf,b,y,e,kf,kb,x]=init_rlslbpef(L)
[ff,bb,fb,cf,b,y,e,kf,kb,x]=init_rlslbpef(L,ff0,bb0,fb0,cf0,b0,x0)
```

Description The block diagram of the RLS-Lattice backward prediction error filter is shown in Fig. 5.21 while the details of the RLS lattice structure showing its internal variables can be seen in Fig. 5.19. A summary of those variables is given below.

```
Input Parameters [Size]::
  L   : number of predictor stages
  ff0 : initial autocorr. of forward prediction error [Lx1]
  bb0 : initial autocorr. of backward prediction error [Lx1]
  fb0 : initial crosscorrelation of f and b [Lx1]
  cf0 : initial conversion factor [Lx1]
  b0  : initial vector of backward prediction error [L+1 x 1]
  x0  : initial input delay line [L+1 x 1]
Output parameters [Default]::
  ff  : autocorr. of forward prediction error [.001*ones(L,1)]
  bb  : autocorr. of backward prediction error [.001*ones(L,1)]
  fb  : crosscorrelation of f and b [zeros]
  cf  : conversion factor [ones]
  b   : vector of backward prediction error [zeros]
  y   : Linear combiner output [zero]
  e   : Linear combiner error [rand]
  kf  : forward lattice coefficients [zeros]
  kb  : backward lattice coefficients [zeros]
  x   : initialized input delay line [zeros].
```

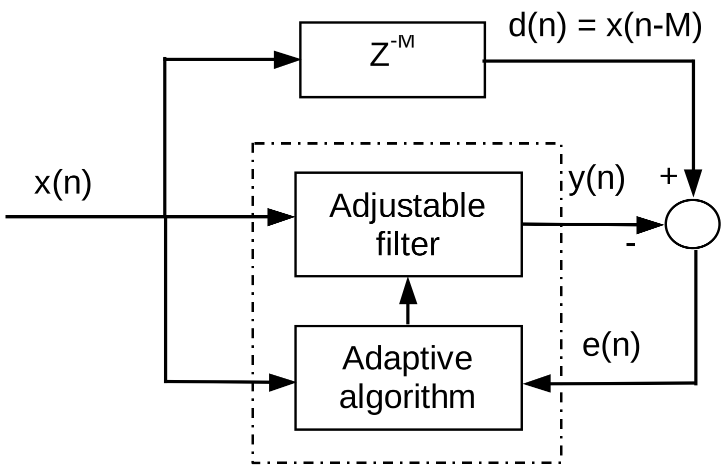

Figure 5.21: Block diagram of the backward prediction error filter.

Example

```
L  = 5;                   % Number of lattice stages
ff = zeros(L,1);          % autocorr. for forward prediction error
bb = zeros(L,1);          % autocorr. for backward prediction error
fb = zeros(L,1);          % crosscorr. between f and b
cf = zeros(L,1);          % conversion factor
b  = zeros(L+1,1);        % backward prediction error
x  = zeros(L+1,1);        % input delay line

% Create and initialize a lattice RLSLBPEF
[ff,bb,fb,cf,b,y,e,kf,kb,x] = init_rlslbpef(L,ff,bb,fb,cf,b,x);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 2 through 7 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTRLSLBPEF, PREDICT_ RLSLBPEF.

5.16 init_ rlslfpef

Purpose Creates and initializes the variables required for the RLS Lattice Forward Prediction Error Filter.

Syntax

```
[ff,bb,fb,cf,b,y,e,kf,kb] = init_rlslfpef(L)
[ff,bb,fb,cf,b,y,e,kf,kb] = init_rlslfpef(L,ff0,bb0,fb0,cf0,b0)
```

Description The block diagram of the RLS-Lattice forward prediction error filter is shown in Fig. 5.22 while the details of the RLS lattice structure showing its internal variables can be seen in Fig. 5.19. A summary of those variables is given below.

```
Input Parameters [Size]::
  L   : number of predictor stages
  ff0 : initial autocorr. of forward prediction error [Lx1]
  bb0 : initial autocorr. of backward prediction error [Lx1]
  fb0 : initial crosscorrelation of f and b [Lx1]
  cf0 : initial conversion factor [Lx1]
  b0  : initial vector of backward prediction error [L+1 x 1]

Output parameters [Default]::
  ff  : autocorr. of forward prediction error [.001*ones(L,1)]
  bb  : autocorr. of backward prediction error [.001*ones(L,1)]
  fb  : crosscorrelation of f and b [zeros]
  cf  : conversion factor [ones]
  b   : vector of backward prediction error [zeros]
  y   : Linear combiner output [zero]
  e   : Linear combiner error [rand]
  kf  : forward lattice coefficients [zeros]
  kb  : backward lattice coefficients [zeros]
```

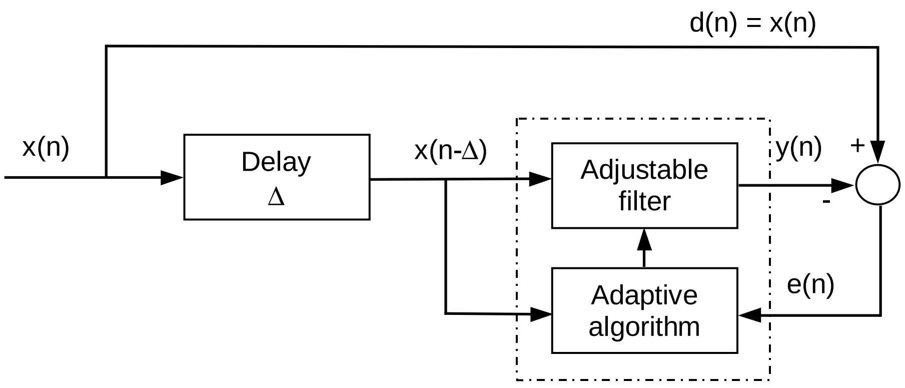

Figure 5.22: Block diagram of the forward prediction error filter.

Example

```
L  = 5;                 % Number of lattice stages
ff = zeros(L,1);        % autocorr. for forward prediction error
bb = zeros(L,1);        % autocorr. for backward prediction error
fb = zeros(L,1);        % crosscorr. between f and b
cf = zeros(L,1);        % conversion factor
b  = zeros(L+1,1);      % backward prediction error

% Create and initialize a lattice RLSLFPEF
[ff,bb,fb,cf,b,y,e,kf,kb] = init_rlslfpef(L,ff,bb,fb,cf,b);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 2 through 6 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTRLSLFPEF, PREDICT_ RLSLFPEF.

Chapter 6

Recursive Adaptive Algorithms

This chapter documents the functions used to create, initialize, and update the coefficients of recursive adaptive filters (Section 2.2.3). Table 6.1 summarizes the recursive adaptive functions functions and gives a short description and a pointer to the reference page of each function.

Function Name	Reference	Short Description
asptcsoiir2	6.1	Cascaded Second Order type-2 IIR adaptive filter.
aspteqerr	6.2	Equation Error IIR adaptive algorithm.
asptouterr	6.3	Output Error IIR adaptive algorithm.
asptsharf	6.4	Simple Hyperstable Adaptive Recursive Filter (SHARF).
asptsoiir1	6.5	Second Order IIR adaptive algorithm type-1.
asptsoiir2	6.6	Second Order IIR adaptive algorithm type-2.
init_ csoiir2	6.7	Initialize Cascaded Second Order IIR adaptive filter.
init_ eqerr	6.8	Initialize Equation Error IIR adaptive filter.
init_ outerr	6.9	Initialize Output Error IIR.
init_ sharf	6.10	Initialize Simple Hyperstable Adaptive Recursive Filter.
init_ soiir1	6.11	Initialize Second Order IIR adaptive algorithm type-1.
init_ soiir2	6.12	Initialize Second Order IIR adaptive algorithm type-2.

Table 6.1: Functions for creating, initializing, and updating recursive adaptive filters.

Each function is documented in a separate section including the following information related to the function:

- **Purpose:** Short description of the algorithm implemented by this function.
- **Syntax:** Shows the function calling syntax. If the function has optional parameters, this section will have two calling syntaxes. One with only the required formal parameters and one with all the formal parameters.
- **Description:** Detailed description of the function usage with explanation of its input and output parameters.
- **Example:** A short example showing typical use of the function. The examples listed can be found in the ASPT/test directory of the ASPT distribution. The user is encouraged to copy from those examples and paste in her own applications.
- **Algorithm:** A short description of the operations internally performed by the function.
- **Remarks:** Gives more theoretical and practical remarks related to the usage, performance, limitations, and applications of the function.
- **Resources:** Gives a summary of the memory requirements and number of multiplications, addition/subtractions, and division operations required to implement the function in real

time. This can be used to roughly calculate the MIPS (Million Instruction Per Second) required for a specific platform knowing the number of instructions the processor needs to perform each operation.

- **See Also:** Lists other functions that are related to this function.
- **Reference:** Lists literature for more information on the function.

6.1 asptcsoiir2

Purpose Performs filtering and parameter update for the Cascaded band-pass Second Order IIR type-2 adaptive filter. Each filter has a transfer function given by

$$H(z) = \frac{(1-s)(cos(t) - z^{-1})}{1-(1+s)cos(t)z^{-1} + sz^{-2}}. \tag{6.1}$$

where the adaptive parameter s controls the filter bandwidth and parameter t controls the filter center frequency.

Syntax

```
[y,a,b,u,t,s,p] = asptcsoiir2(xn,u,y,a,b,
                  t,s,p,mu_t,mu_s,t_lim,s_lim)
```

Description `asptcsoiir2()` is a cascade of M SOIIR2 sections with each section tracking one narrow-band signal. Fig. 6.1 shows the block diagram of the CSOIIR2 adaptive line enhancer, where the input of each stage is the error signal of its previous stage. In this arrangement, the first section adapts and tracks the strongest narrow-band component in the system input signal. The error of the first section is therefore free from this component which allows the next stage to converge to the second strongest narrow-band component and so on. The output of the last stage is the wide-band signal and the sum of the sections' outputs is the system output and contains all the estimated narrow-band signals. `asptcsoiir2()` takes a set of input delay lines $u(n)$, output delay lines $y(n)$, the two adaptive filter coefficients for each stage from previous iteration $t(n-1)$ and $s(n-1)$, and the previous gradient vectors $a(n-1)$ and $b(n-1)$, and returns the updated filters' output delay lines $y(n)$, the error sample $e(n)$ and the updated filters' parameters $t(n)$ and $s(n)$.
The input and output parameters of `asptcsoiir2()` are summarized below.

```
Input arguments [Size]:
   xn    : new input sample [1 x 1]
   u     : last 3 input samples for each stage [3 x M+1]
   y     : last 3 output samples for each stage [3 x M]
   a     : last 3 t gradients for each stage [3 x M]
   b     : last 3 s gradients for each stage [3 x M]
   t     : section center freq. parameter {0 pi} [1 x M]
   s     : section bandwidth parameter {0 1} [1 x M]
   p     : estimate of the input signal power [1 x M]
   mu_t  : adaptation constant for t [1 x M]
   mu_s  : adaptation constant for s [1 x M]
   t_lim : [t_min t_max]; min. & max. bounds for t
   s_lim : [s_min s_max]; min. & max. bounds for s
Output Parameters:
   y     : updated output buffer
   a     : updated t-gradient buffer
   b     : updated s-gradient buffer
   u     : updated input/error buffer
   t     : updated filter CF parameters
   s     : updated filter BW parameters
   p     : updated input power estimate
```

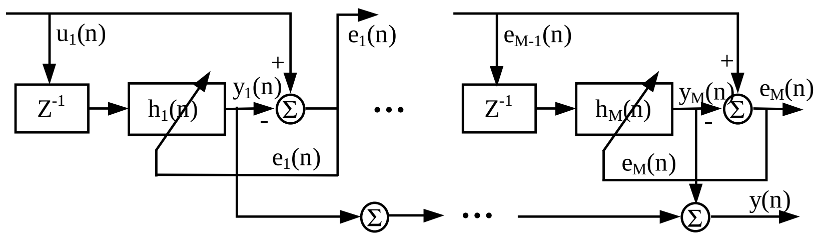

Figure 6.1: Block diagram of the cascaded second order IIR adaptive line enhancer.

Example

```
iter = 5000;
t    = (1:iter)/1000;             % time index @ 1kHz
xn   = 2*(rand(iter,1)-0.5)  ;    % Input signal.
xn   = xn + 1 * cos(2*pi*50*t') + .5 * cos(2*pi*150*t');
yn   = zeros(iter,1);             % narrow-band signal
en   = yn;                        % error signal

% Initialize CSOIIR2
M      = 2;                       % No. of harmonics.
s0     = 0.25*ones(1,M);          % initial s
t0     = 0.1*ones(1,M);           % initial t
mu_s   = 0.01*ones(1,M);          % s-parameter step size
mu_t   = 0.05*ones(1,M);          % t-parameter step size
s_lim  = [.1 .9];                 % bounds for s
t_lim  = [0.05 3.1];              % bounds for t
[s,t,u,y,a,b,p]=init_csoiir2(M,s0,t0);

%% Processing Loop
for (m=2:iter)
   [y,a,b,u,t,s,p] = asptcsoiir2(xn(m),u,y,a,b,t,s...
                     ,p,mu_t,mu_s,t_lim,s_lim);
   yn(m,:) = sum(y(1,:),2);  % sections' outputs
   en(m)   = u(1,M+1);       % error of last section
end;
h = [ zeros(1024,M)];
for m = 1:M,
   h(:,m) = impz([cos(t(m))*(1-s(m)) -(1-s(m))],...
                 [1 -cos(t(m))*(1+s(m)) s(m)],1024);
end
% display the results
H = 20*log10(abs(fft(h)));
subplot(2,2,1);plot(H(1:513,:)); grid;
subplot(2,2,2);
plot([yn(4800:5000)]);grid
```

Running the above script will produce the graph shown in Fig. 6.2. The left side graph of the figure shows the frequency responses of the two adaptive second order sections after convergence. It is clear that the two sections converge to band pass filters centered at 50 and 150 Hz, the narrow-band components in the input signals. The right side graph shows the last 200 samples of the cascaded filter output which coincides with the two narrow-band components at 50 and 150 Hz superimposed on the white noise input signal.

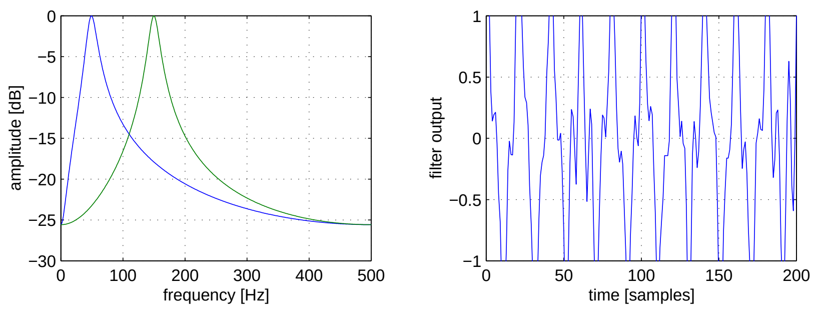

Figure 6.2: The adaptive filters frequency responses after convergence and the filter output for the cascaded adaptive line enhancer.

Algorithm `asptcsoiir2()` performs the following operations

- Calculates the output of each section $y(n)$ from the previous and current input samples $\underline{\mathbf{u}}(n)$ and previous output samples $\underline{\mathbf{y}}(n-1)$.
- Calculates the error sample for each section and updates the input/error vector $u(n)$.
- Calculates the gradient samples $a(n)$ and $b(n)$ and updates the gradient vectors.
- Updates the adaptive coefficients $s(n)$ and $t(n)$ and limits their values if necessary.

Remarks

- Being an IIR filter, the adaptive filter might become unstable during adaptation.
- Each second order filter $h(n)$ always has a zero dB gain at its center frequency.
- The filters center frequencies are given by $\omega_c = t$.
- `asptcsoiir2()` updates the input vector $u(n)$ internally.

Resources The resources required to implement the CSOIIR2 recursive adaptive line enhancer composed of cascade of M sections in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$20M$
MULTIPLY	$25M$
ADD	$16M$
DIVIDE	M
COS	M
SIN	M

See Also INIT_ CSOIIR2, ALE_ CSOIIR2, ASPTSOIIR2, ASPTSOIIR1.

Reference [2] and [10] for introduction to recursive adaptive filters.

6.2 aspteqerr

Purpose Sample per sample filtering and coefficient update using the Equation Error recursive adaptive algorithm. The filter transfer function is given by

$$H(z) = \frac{A(z)}{1 - B(z)}, \tag{6.2}$$

Syntax

```
[u,w,y,e,Px,Pd]=aspteqerr(N,M,u,w,y,x,d,mu,Px,Pd)
```

Description `aspteqerr()` implements the equation error LMS adaptive algorithm used to update recursive adaptive filters. The equation error algorithm adjusts the composite filter coefficients vector by minimizing the error signal as shown in Fig. 6.3. `aspteqerr()` takes an input sample $x(n)$, a desired sample $d(n)$, the vector of the adaptive filter coefficients from previous iteration $w(n-1)$, the composite input vector $u(n-1)$, the step size vector mu, and returns the filter output $y(n)$, the error sample $e(n)$ and the updated vector of filter coefficients $w(n)$.
The input and output parameters of `aspteqerr()` for a recursive adaptive filter of N numerator coefficients and M denumerator coefficients are summarized below.

```
Input arguments:
   N    : Number of coefficients of A(z)
   M    : Number of coefficients of B(z)
   u    : composite input vector
   w    : filter coefficient vector
   y    : [y(n-1) y(n-2) ... y(n-M)]^T
   x    : new input sample
   d    : new desired sample
   mu   : adaptation constant
   Px   : variance of x(n)
   Pd   : variance of d(n)
Output Parameters:
   u,w,y,Px,Py are the updated variables defined above
   e    : error signal e(n)
```

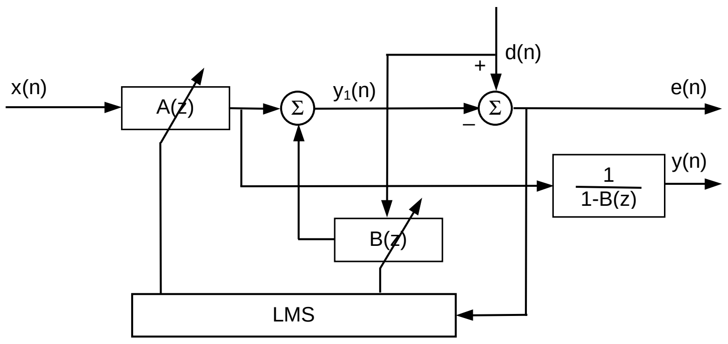

Figure 6.3: Block diagram of the equation error algorithm.

Example

```
iter = 5000;                        % Number of samples to process
xn   = 2*(rand(iter,1)-0.5)   ;    % Input signal, zero mean random.
dn   = filter([0.6 -.01],[1 -0.4 0.6],xn); % Filter output
en   = zeros(iter,1);              % error signal

% Initialize EQERR
N = 2; M = 2;
[u,w,y,e,mu,Px,Pd]=init_eqerr(N,M);

%% Processing Loop
for (m=1:iter)
   x = xn(m);
   d = dn(m) + 1e-3*rand;
   % update the filter
   [u,w,y,e,Px,Pd]=aspteqerr(N,M,u,w,y,x,d,mu,Px,Pd);
   % save the last error sample to plot later
   en(m,:) = e;
end;

wp = filter(w(1:N),[1 ; -w(N+1:N+M)],[1;zeros(19,1)]);

% display the results
subplot(2,2,1);stem(wp); grid;
xlabel('filter response after convergence')
subplot(2,2,2);
eb = filter(0.1,[1 -.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 6.4. The left side graph of the figure shows the adaptive filter impulse response after convergence. The right side graph shows the mean square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB).

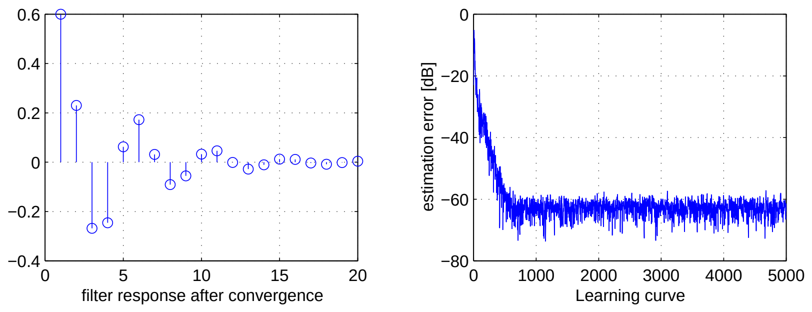

Figure 6.4: The adaptive filter impulse response after convergence and the learning curve for the IIR system identification problem using the equation error algorithm.

Algorithm

The equation error algorithm uses the desired signal $d(n)$ (instead of the output signal $y(n)$ as in the output error algorithm) as input to the recursive part of the filter. This makes the performance index function quadratic in the filter coefficients and results in a single global minimum similar to that found in FIR adaptive algorithms. The filter transfer function is given by

$$w(z) = \frac{A(z)}{1 - B(z)} \tag{6.3}$$

where

$$\begin{aligned} A(z) &= a_0 + a_1 z^{-1} + ... + a_{N-1} z^{-N+1} & (6.4) \\ B(z) &= b_1 z^{-1} + b_2 z^{-2} + ... + b_M z^{-M} & (6.5) \end{aligned}$$

The current implementation of `aspteqerr()` performs the following operations

- Updates the composite input vector $\underline{\mathbf{u}}(n)$ using the current and previous samples of $x(n)$ and $d(n)$.
- Filters the composite input vector $\underline{\mathbf{u}}(n)$ through the adaptive filter coefficients $\underline{\mathbf{w}}(n-1)$ to produce the update filter output $y_1(n)$.
- Calculates the error sample $e(n) = d(n) - y_1(n)$.
- Calculates the actual filter output $y(n)$ as shown in Fig. 6.3
- Updates the adaptive filter coefficients using the error $e(n)$ and the composite input vector $\underline{\mathbf{u}}(n)$ using the relationship

$$\underline{\mathbf{w}}(n) = \underline{\mathbf{w}}(n-1) + 2\,\mu\, e(n)\, \underline{\mathbf{u}}(n-1). \tag{6.6}$$

Remarks

- Being an IIR filter, the adaptive filter $w(n)$ might become unstable during adaptation. This can be avoided by checking that the poles of the filter remain within the unit circle after each call to `aspteqerr()`.
- Unlike the output error algorithm, the performance surface searched by the equation error algorithm is quadratic in the filter coefficients and has a single minimum. This guarantees that the filter will asymptotically converge to its optimal solution.
- `aspteqerr()` supports both real and complex data and filters.
- `aspteqerr()` updates the composite input vector internally.

Resources

The resources required to implement the EQERR algorithm for a recursive adaptive filter of N numerator coefficients and M denumerator coefficients in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$3N + 4M + 4$
MULTIPLY	$4N + 5M + 6$
ADD	$3N + 4M + 1$
DIVIDE	N+M

See Also

INIT_ EQERR, MODEL_ EQERR, ASPTOUTERR.

Reference

[2] and [10] for introduction to recursive adaptive filters.

6.3 asptouterr

Purpose Sample per sample filtering and coefficient update using the Output Error recursive adaptive algorithm. The filter transfer function is given by

$$H(z) = \frac{A(z)}{1 - B(z)}, \tag{6.7}$$

Syntax `[u,w,c,y,e,Px,Py]=asptouterr(N,M,u,w,c,x,d,mu,Px,Py)`

Description `asptouterr()` implements the output error LMS adaptive algorithm used to update recursive adaptive filters. The output error algorithm adjusts the composite filter coefficients vector by minimizing the error signal as shown in Fig. 6.5. `asptouterr()` takes an input samples $x(n)$, a desired sample $d(n)$, the vector of the adaptive filter coefficients from previous iteration $w(n-1)$, the composite input vector $u(n)$, the step size vector mu, and returns the filter output $y(n)$, the error sample $e(n)$ and the updated vector of filter coefficients $w(n)$.
The input and output parameters of `asptouterr()` for a recursive adaptive filter of N numerator coefficients and M denumerator coefficients are summarized below.

```
Input arguments:
  N   : Number of coefficients of A(z)
  M   : Number of coefficients of B(z)
  u   : composite input vector
  w   : vector of adaptive filter coefficients
  c   : composite gradient vector
  x   : new input sample
  d   : new desired sample
  mu  : adaptation constant vector
  Px  : variance of x(n)
  Py  : variance of y(n)
 Output Parameters:
  u,w,c,Px,Py are the updated input variables
  y   : filter output y(n)
  e   : error signal e(n)
```

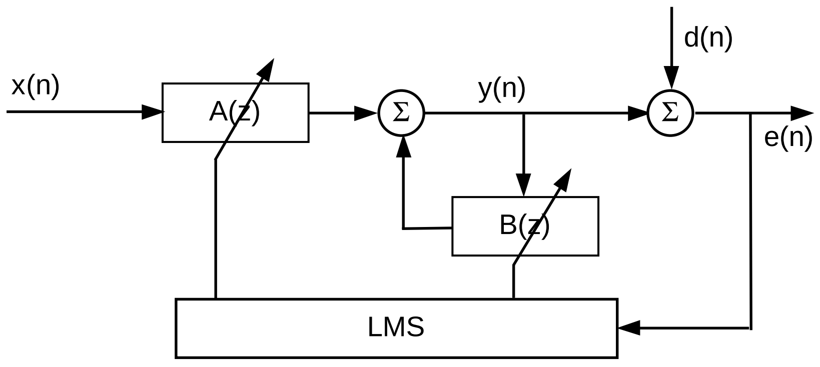

Figure 6.5: Block diagram of the output error algorithm.

Example

```
iter = 5000;                    % Number of samples to process
xn   = 2*(rand(iter,1)-0.5)  ;  % Input signal, zero mean random.
dn   = filter([0.6 -.01],[1 -0.4 0.6],xn); % Filter output
en   = zeros(iter,1);           % error signal

% Initialize OUTERR
N = 2; M = 2;
[u,w,c,y,d,e,mu,Px,Py]=init_outerr(N,M);

%% Processing Loop
for (m=1:iter)
   x = xn(m);
   d = dn(m) + 1e-3*rand;
   % update the filter
   [u,w,c,y,e,Px,Py]=asptouterr(N,M,u,w,c,x,d,mu,Px,Py);
   % save the last error sample to plot later
   en(m,:) = e;
end;

wp = filter(w(1:N),[1 ; -w(N+1:N+M)],[1; zeros(19,1)]);

% display the results
subplot(2,2,1);stem(wp); grid;
xlabel('filter response after convergence')
subplot(2,2,2);
eb = filter(.1,[1 -.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 6.6. The left side graph of the figure shows the adaptive filter impulse response after convergence. The right side graph shows the mean square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB).

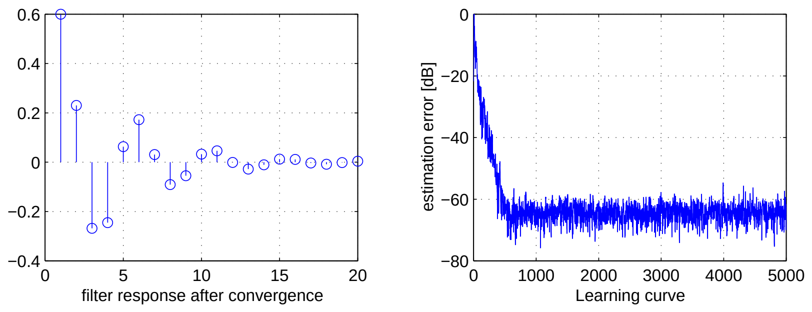

Figure 6.6: The adaptive filter response after convergence and the learning curve for the IIR system identification problem using the output error algorithm.

Algorithm

The output error algorithm is a direct extension of the Wiener filter theory to recursive filters. The filter transfer function is given by

$$w(z) = \frac{A(z)}{1 - B(z)} \tag{6.8}$$

where

$$\begin{aligned} A(z) &= a_0 + a_1 z^{-1} + ... + a_{N-1} z^{-N+1} && (6.9) \\ B(z) &= b_1 z^{-1} + b_2 z^{-2} + ... + b_M z^{-M} && (6.10) \end{aligned}$$

The current implementation of `asptouterr()` performs the following operations

- Updates the composite input vector $\underline{\mathbf{u}}(n)$ using the current and previous samples of $x(n)$ and $y(n)$.
- Filters the composite input vector $\underline{\mathbf{u}}(n)$ through the adaptive filter coefficients $\underline{\mathbf{w}}(n-1)$ to produce the filter output $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$.
- Calculates the gradient vector $\underline{\mathbf{c}}(n)$
- Updates the adaptive filter coefficients using the error $e(n)$ and the gradient vector $\underline{\mathbf{c}}(n)$ using the relationship

$$\underline{\mathbf{w}}(n) = \underline{\mathbf{w}}(n-1) + 2\,\mu\, e(n)\, \underline{\mathbf{c}}(n). \tag{6.11}$$

Remarks

- Being an IIR filter, the adaptive filter $w(n)$ might become unstable during adaptation. This can be avoided by checking that the poles of the filter remain within the unit circle after each call to `asptouterr()`.
- The performance surface searched by the output error algorithm usually has local minima and maxima. Therefore, it is not guaranteed that the filter will converge to a global minimum. This problem is alleviated in the equation error algorithm (see Section 6.2.
- `asptouterr()` supports both real and complex data and filters.
- `asptouterr()` updates the composite input vector internally.

Resources

The resources required to implement the OUTERR algorithm for a recursive adaptive filter of N numerator coefficients and M denumerator coefficients in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$4N + 4M + 5$
MULTIPLY	$3N + 5M + 6$
ADD	$3N + 5M + 2$
DIVIDE	N+M

See Also

INIT_ OUTERR, MODEL_ OUTERR, ASPTEQERR.

Reference

[2] and [10] for introduction to recursive adaptive filters.

6.4 asptsharf

Purpose Sample per sample IIR filtering and coefficient update using the Simple Hyperstable Adaptive Recursive Filter (SHARF) algorithm.

Syntax `[w,u,y,e,Px,Py]=asptsharf(N,M,w,u,xn,d,e,c,mu,Px,Py)`

Description `asptsharf()` implements the Simple Hyperstable Adaptive Recursive Filter (SHARF) algorithm. The SHARF algorithm adjusts the composite filter coefficients vector by minimizing the error signal as shown in Fig. 6.7. The main difference between SHARF and other LMS based IIR adaptive algorithms is that the SHARF algorithm uses a low pass filter $C(z)$ to smooth the error signal and uses the smoothed error as gradient. `asptsharf()` takes an input sample $x(n)$, a desired sample $d(n)$, the vector of the adaptive filter coefficients from previous iteration $w(n-1)$, the composite input vector $u(n-1)$, the step size vector mu, and returns the filter output $y(n)$, the error sample $e(n)$ and the updated vector of filter coefficients $w(n)$. The input and output parameters of `asptsharf()` for a recursive adaptive filter of N numerator coefficients and M denumerator coefficients are summarized below.

```
Input Parameters:
   N   : Number of coefficients of A(z)
   M   : Number of coefficients of B(z)
   w   : vector of adaptive filter coefficients
   u   : composite input / output delay line
   xn  : new input sample
   d   : new desired sample
   e   : error vector
   c   : error smoothing coefficients vector
   mu  : adaptation constant vector
   Px  : previously estimated power of input signal x(n)
   Py  : previously estimated power of output signal y(n)
Output Parameters:
   w   : updated adaptive coefficients
   u   : updated composite delay line
   y   : filter output y(n)
   e   : error signal e(n)
   Px  : updated power of input signal x(n)
   Py  : updated power of output signal y(n)
```

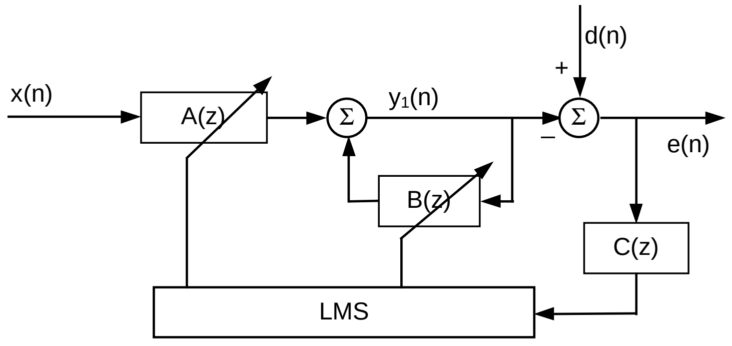

Figure 6.7: Block diagram of the SHARF algorithm.

Example

```
iter = 5000;                      % Number of samples to process
xn   = 2*(rand(iter,1)-0.5)  ;  % Input signal, zero mean random.
dn   = filter([0.6 -.01],[1 -0.4 0.6],xn); % Filter output
en   = zeros(iter,1);            % error signal

% Initialize SHARF.
N=2; M=2; L = 7;
[u,w,e,c,d,mu,Px,Py]=init_sharf(N,M,L);
mu = [0.02;0.02;0.02;0.02];

%% Processing Loop
for (m=1:iter)
x = xn(m);
   d = dn(m)+ 1e-3*rand;
   % update the filter
   [w,u,y,e,Px,Py]=asptsharf(N,M,w,u,x,d,e,c,mu,Px,Py);
   % save the last error sample to plot later
   en(m,:) = e(1);
end;

wp = filter(w(1:N),[1 ; -w(N+1:N+M)],[1; zeros(19,1)]);

% display the results
subplot(2,2,1);stem(wp); grid;
xlabel('filter response after convergence')
subplot(2,2,2);
eb = filter(.1, [1 -.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 6.8. The left side graph of the figure shows the adaptive filter impulse response after convergence. The right side graph shows the mean square error in dB versus time during the adaptation process, which is usually called the learning curve. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB).

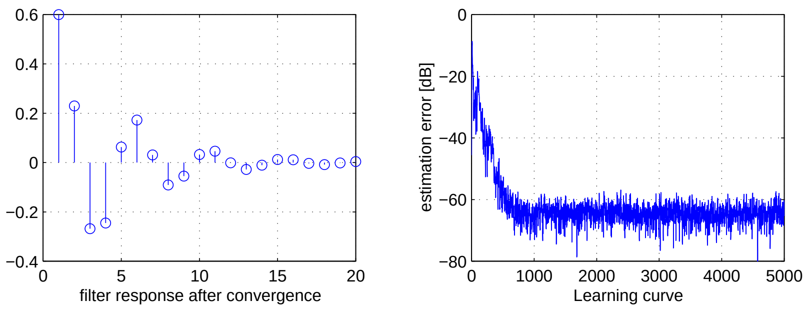

Figure 6.8: The adaptive filter impulse response after convergence and the learning curve for the IIR system identification problem using the SHARF algorithm.

Algorithm

The SHARF algorithm uses a smoothed version of the error signal as the gradient vector. The filter transfer function is given by

$$w(z) = \frac{A(z)}{1 - B(z)} \tag{6.12}$$

where

$$\begin{aligned} A(z) &= a_0 + a_1 z^{-1} + ... + a_{N-1} z^{-N+1} && (6.13) \\ B(z) &= b_1 z^{-1} + b_2 z^{-2} + ... + b_M z^{-M} && (6.14) \end{aligned}$$

The current implementation of `asptsharf()` performs the following operations

- Updates the composite input vector $\underline{\mathbf{u}}(n)$ using the current and previous samples of $x(n)$ and $y(n)$.
- Filters the composite input vector $\underline{\mathbf{u}}(n)$ through the adaptive filter coefficients $\underline{\mathbf{w}}(n-1)$ to produce the filter output $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$.
- Calculates the smoothed error signal as shown in Fig. 6.7.
- Updates the adaptive filter coefficients using the smoothed error and the composite input vector $\underline{\mathbf{u}}(n)$.

Remarks

- Being an IIR filter, the adaptive filter $w(n)$ might become unstable during adaptation. This can be avoided by checking that the poles of the filter remain within the unit circle after each call to `asptsharf`.
- `asptsharf()` supports both real and complex data and filters.
- `asptsharf()` updates the composite input vector internally.

Resources

The resources required to implement the SHARF algorithm for a recursive adaptive filter of N numerator coefficients and M denumerator coefficients in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	$3N + 3M + 2L + 4$
MULTIPLY	$3N + 3M + L + 6$
ADD	$N + M + L + 2$
DIVIDE	N+M

See Also

INIT_ SHARF, MODEL_ SHARF.

Reference

[2] and [10] for introduction to recursive adaptive filters.

6.5 asptsoiir1

Purpose Performs filtering and parameter update for the band-pass Second Order IIR type-1 adaptive filter. The filter transfer function is given by

$$H(z) = \frac{(1-s)(w-z^{-1})}{1-(1+s)wz^{-1}+sz^{-2}}. \tag{6.15}$$

where parameter s controls the filter bandwidth and parameter w controls the filter center frequency.

Syntax

```
[y,a,b,e,w,s] = asptsoiir1(u,y,a,b,e,w,s,mu_w,mu_s,w_lim,s_lim)
```

Description `asptsoiir1()` is a special second order IIR adaptive filter algorithm optimized for extracting and tracking narrow-band signals buried in a wide-band signal. Therefore, it is widely used in applications such as Adaptive Line Enhancers (ALE) where a weak carrier signal is required to be recovered from a strong wide-band noise. Another common application of `asptsoiir1()` is removing the 50/60 Hz power line noise usually introduced into weak sensor signals. Fig. 6.9 shows the block diagram of the applications mentioned above which is basically an IIR forward prediction configuration. `asptsoiir1()` takes an input delay line $u(n)$, an output delay line $y(n)$, the two adaptive filter coefficients from previous iteration $w(n-1)$ and $s(n-1)$, and the previous gradient vectors $a(n-1)$ and $b(n-1)$, and returns the updated filter output delay line $y(n)$, the error sample $e(n)$ and the updated filter parameters $w(n)$ and $s(n)$. The input and output parameters of `asptsoiir1()` are summarized below.

```
Input arguments [Size]:
  u     : the last 3 input samples [3 x 1]
  y     : the last 3 output samples [3 x 1]
  a     : the last 3 w gradients [3 x 1]
  b     : the last 3 s gradients [3 x 1]
  e     : the last 3 error samples [3 x 1]
  w     : filter center freq. parameter {-1 1} [1 x 1]
  s     : filter bandwidth parameter {0 1} [1 x 1]
  mu_w  : adaptation constant for w [1x1]
  mu_s  : adaptation constant for s [1x1]
  w_lim : [w_min w_max]; min. & max. bounds for w
  s_lim : [s_min s_max]; min. & max. bounds for s
Output Parameters:
  y     : updated output buffer
  a     : updated t-gradient buffer
  b     : updated s-gradient buffer
  e     : updated error buffer
  s     : updated filter BW parameters
  w     : updated filter CF parameters
```

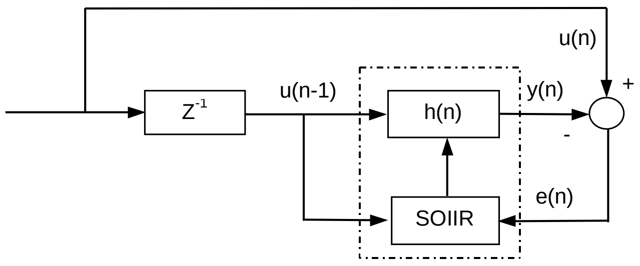

Figure 6.9: Block diagram of the second order IIR algorithm in an adaptive line enhancer configuration.

Example

```
iter = 5000;
t    = (1:iter)/1000;           % time index @ 1kHz
xn   = 2*(rand(iter,1)-0.5)  ;  % Input signal, zero mean random.
xn   = xn + .5 * cos(2*pi*50*t');
yn   = zeros(iter,1);           % error signal
en   = yn;

% Initialize SOIIR1
w_lim  = [-.99 0.99];           % bounds for w
s_lim  = [0.1 .9];              % bounds for s
mu_w   = 0.1;                   % step size for w
mu_s   = 0.01;                  % step size for s
[s,w,u,y,a,b,e] = init_soiir1(0.3,0.1); % initialize soiir1

%% Processing Loop
for (m=1:iter)
u = [xn(m); u(1:2)];
   [y,a,b,e,w,s] = asptsoiir1(u,y,a,b,e,w,s,mu_w,mu_s,w_lim,s_lim);
   yn(m,:) = y(1);
   en(m) = e(1);
end;
% display the results
h = filter([w*(1-s) -(1-s)],[1 -w*(1+s) s],[1;zeros(255,1)]);
f = (0:128)*500/128;
H = 20*log10(abs(fft(h)));
subplot(2,2,1);plot(f,H(1:129)); grid;
subplot(2,2,2);
plot([yn(4800:5000)]);grid
```

Running the above script will produce the graph shown in Fig. 6.10. The left side graph of the figure shows the adaptive filter frequency response after convergence. The right side graph shows the last 200 samples of the filter output which shows that the filter output coincide with the narrow-band 50 Hz superimposed on the white noise input signal.

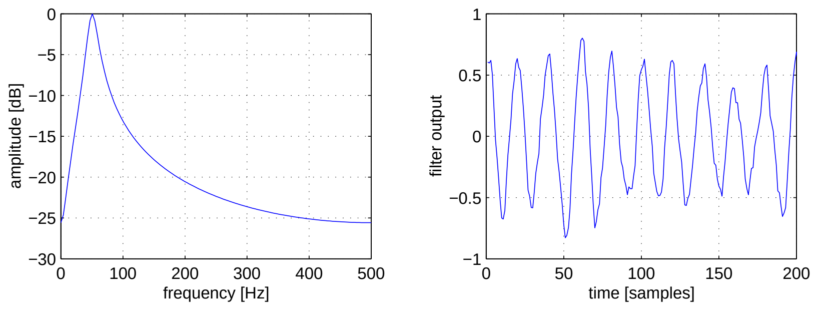

Figure 6.10: The adaptive filter frequency response after convergence and the filter output for the adaptive line enhancer problem using the second order IIR type-1 algorithm.

Algorithm `asptsoiir1()` performs the following operations

- Calculates the filter output $y(n)$ from the previous and current input samples $\underline{\mathbf{u}}(n)$ and previous output samples $\underline{\mathbf{y}}(n-1)$.
- Calculates the error sample $e(n) = d(n) - y(n)$ and updates the error vector.
- Calculates the gradient samples $a(n)$ and $b(n)$ and updates the gradient vectors.
- Updates the adaptive coefficients $s(n)$ and $w(n)$ and limits their values if necessary.

Remarks

- Being an IIR filter, the adaptive filter might become unstable during adaptation.
- The second order filter $h(n)$ always has a zero dB gain at its center frequency.
- The filter center frequency is given by $\omega_c = cos^{-1}w$.
- `asptsoiir1()` suffers from slow convergence when the center frequency of the narrow-band signal is close to zero or close to π. This problem is solved in `asptsoiir2()`.
- The configuration in Fig. 6.9 will function as a notch filter at ω_c when the error signal is taken as output. This will remove the narrow-band signal from the input signal.
- `asptsoiir1()` does not update the input vector internally. This has to be done before calling `asptsoiir1()` as shown in the example above.

Resources The resources required to implement the SOIIR1 recursive adaptive filter in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	20
MULTIPLY	24
ADD	16
DIVIDE	1

See Also INIT_ SOIIR1, ALE_ SOIIR1, ASPTSOIIR2.

Reference [2] and [10] for introduction to recursive adaptive filters.

6.6 asptsoiir2

Purpose Performs filtering and parameter update for the band-pass Second Order IIR type-2 adaptive filter. The filter is derived from type-1 by substituting $w = cos(t)$ which results in the transfer function

$$H(z) = \frac{(1-s)(cos(t) - z^{-1})}{1-(1+s)cos(t)z^{-1} + sz^{-2}}. \tag{6.16}$$

where parameter s controls the filter bandwidth and parameter t controls the filter center frequency.

Syntax `[y,a,b,e,t,s] = asptsoiir2(u,y,a,b,e,t,s,mu_t,mu_s,t_lim,s_lim)`

Description `asptsoiir2()` is a special second order IIR adaptive filter algorithm optimized for extracting and tracking narrow-band signals buried in a wide-band signal. Therefore, it is widely used in applications such as Adaptive Line Enhancers (ALE) where a weak carrier signal is required to be recovered from a strong wide-band noise. Another common application of `asptsoiir2()` is removing the 50/60 Hz power line noise usually introduced into weak sensor signals. `asptsoiir2()` is derived from `asptsoiir1()` by substituting $w = cos(t)$ and adapting t instead of adapting w to overcome the slow convergence when the center frequency of the narrow-band signal is close to zero or close to π. Fig. 6.11 shows the block diagram of the applications mentioned above which is basically an IIR forward prediction configuration. `asptsoiir2()` takes an input delay line $u(n)$, an output delay line $y(n)$, the two adaptive filter coefficients from previous iteration $t(n-1)$ and $s(n-1)$, and the previous gradient vectors $a(n-1)$ and $b(n-1)$, and returns the updated filter output delay line $y(n)$, the error sample $e(n)$ and the updated filter parameters $t(n)$ and $s(n)$. The input and output parameters of `asptsoiir2()` are summarized below.

```
Input arguments [Size]:
  u     : the last 3 input samples [3 x 1]
  y     : the last 3 output samples [3 x 1]
  a     : the last 3 t gradients [3 x 1]
  b     : the last 3 s gradients [3 x 1]
  e     : the last 3 error samples [3 x 1]
  t     : filter center freq. parameter {-1 1} [1 x 1]
  s     : filter bandwidth parameter {0 1} [1 x 1]
  mu_t  : adaptation constant for t [1x1]
  mu_s  : adaptation constant for s [1x1]
  t_lim : [t_min t_max]; min. & max. bounds for t
  s_lim : [s_min s_max]; min. & max. bounds for s
Output Parameters:
  y     : updated output buffer
  a     : updated t-gradient buffer
  b     : updated s-gradient buffer
  e     : updated error buffer
  s     : updated filter BW parameters
  t     : updated filter CF parameters
```

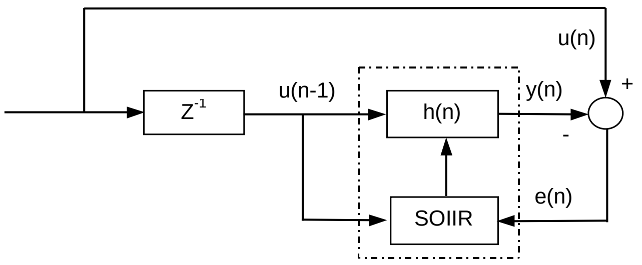

Figure 6.11: Block diagram of the second order IIR algorithm in an adaptive line enhancer configuration.

Example

```
iter = 5000;
t    = (1:iter)/1000;           % time index @ 1kHz
xn   = 2*(rand(iter,1)-0.5)  ;  % Input signal, zero mean random.
xn   = xn + .5 * cos(2*pi*50*t');
yn   = zeros(iter,1);           % error signal
en   = yn;

% Initialize SOIIR2
t_lim  = [-.99 0.99];           % bounds for w
s_lim  = [0.1 .9];              % bounds for s
mu_t   = 0.1;                   % step size for w
mu_s   = 0.01;                  % step size for s
[s,t,u,y,a,b,e]=init_soiir2(0.3,0.1); % initialize soiir1

%% Processing Loop
for (m=1:iter)
u = [xn(m); u(1:2)];
   [y,a,b,e,t,s] = asptsoiir2(u,y,a,b,e,t,s,mu_t,mu_s,t_lim,s_lim);
   yn(m,:) = y(1);
   en(m) = e(1);
end;
% display the results
h = filter([cos(t)*(1-s) -(1-s)],[1 -cos(t)*(1+s) s],[1;zeros(255,1)]);
f = (0:128)*500/128;
H = 20*log10(abs(fft(h)));
subplot(2,2,1);plot(f,H(1:129)); grid;
subplot(2,2,2);
plot([yn(4800:5000)]);grid
```

Running the above script will produce the graph shown in Fig. 6.12. The left side graph of the figure shows the adaptive filter frequency response after convergence. The right side graph shows the last 200 samples of the filter output which shows that the filter output coincide with the narrow-band 50 Hz superimposed on the white noise input signal.

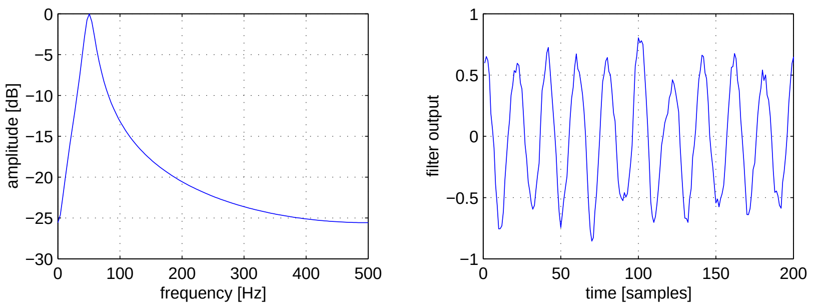

Figure 6.12: The adaptive filter frequency response after convergence and the filter output for the adaptive line enhancer problem using the second order IIR type-2 filter.

Algorithm `asptsoiir2()` performs the following operations

- Calculates the filter output $y(n)$ from the previous and current input samples $\underline{\mathbf{u}}(n)$ and previous output samples $\underline{\mathbf{y}}(n-1)$.
- Calculates the error sample $e(n) = d(n) - y(n)$ and updates the error vector.
- Calculates the gradient samples $a(n)$ and $b(n)$ and updates the gradient vectors.
- Updates the adaptive coefficients $s(n)$ and $t(n)$ and limits their values if necessary.

Remarks

- Being an IIR filter, the adaptive filter might become unstable during adaptation.
- The second order filter $h(n)$ always has a zero dB gain at its center frequency.
- The filter center frequency is given by $\omega_c = t$.
- The configuration in Fig. 6.11 will function as a notch filter at ω_c when the error signal is taken as output. This will remove the narrow-band signal from the input signal.
- `asptsoiir2()` does not update the input vector internally. This has to be done before calling `asptsoiir2()` as shown in the example above.

Resources The resources required to implement the SOIIR2 recursive adaptive filter in real time is given in the table below. The computations given are those required to process one sample.

MEMORY	20
MULTIPLY	25
ADD	16
DIVIDE	1
COS	1
SIN	1

See Also INIT_ SOIIR2, ALE_ SOIIR2, ASPTSOIIR1.

Reference [2] and [10] for introduction to recursive adaptive filters.

6.7 init_ csoiir2

Purpose Creates and initializes the variables for the Cascaded Second Order IIR type-2 adaptive filter. The transfer function of each section is given by

$$H(z) = \frac{(1-s)(cos(t)-z^{-1})}{1-(1+s)cos(t)z^{-1}+sz^{-2}}. \tag{6.17}$$

where parameter s controls the filter bandwidth and parameter t controls the filter center frequency.

Syntax

```
[s,t,u,y,a,b,p]=init_csoiir2(M,s0,t0)
[s,t,u,y,a,b,p]=init_csoiir2(M,s0,t0,u0,y0,a0,b0,p0)
```

Description The CSOIIR2 algorithm is used to simultaneously estimate and track any changes in multiple spectral lines (multiple harmonic signals). The variables of the CSOIIR2 algorithm are summarized below (see Fig. 6.1).

```
Input arguments [Size]:
   M   : number of second order sections
   s0  : initial adaptive bandwidth parameters [1xM]
   t0  : initial adaptive center frequency parameters [1xM]
   u0  : initial last 3 input samples [3xM]
   y0  : initial last 3 output samples [3xM]
   a0  : initial last 3 w-gradients [3xM]
   b0  : initial last 3 s-gradients [3xM]
   p0  : initial input power estimate [1xM]

Output Parameters [default]:
   s   : initialized adaptive bandwidth parameters [zeros]
   t   : initialized adaptive center frequency parameters [zeros]
   u   : initialized input buffer [zeros]
   y   : initialized output buffer [zeros]
   a   : initialized w-gradient buffer [zeros]
   b   : initialized s-gradient buffer [zeros]
   p   : initialized power [zeros]
```

Example

```
M     = 2;               % No. of harmonics.
s0    = 0.25*ones(1,M);  % initial s
t0    = 0.1*ones(1,M);   % initial t

% initialize the csoiir2 filter
[s,t,u,y,a,b,p]=init_csoiir2(M,s0,t0);
```

Remarks Use input parameters 4 through 8 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTCSOIIR2, ALE_ CSOIIR2.

6.8 init_ eqerr

Purpose Creates and initializes the variables required for the Equation Error recursive adaptive algorithm. The filter transfer function is given by

$$w(z) = \frac{A(z)}{1 - B(z)} \tag{6.18}$$

where

$$\begin{aligned} A(z) &= a_0 + a_1 z^{-1} + ... + a_{N-1} z^{-N+1} &(6.19)\\ B(z) &= b_1 z^{-1} + b_2 z^{-2} + ... + b_M z^{-M} &(6.20)\end{aligned}$$

Syntax

```
[u,w,y,e,mu,Px,Pd] = init_eqerr(N,M)
[u,w,y,e,mu,Px,Pd] = init_eqerr(N,M,u0,w0,y0,d0,mu0)
```

Description The variables of the equation error algorithm are summarized below (see Fig. 6.3).

```
Input Parameters:
   N   : Number of coefficients of A(z)
   M   : Number of coefficients of B(z)
   u0  : [x(0) x(-1) ... x(-N+1) d(0) ... d(-M)]'
   w0  : [a_0 a_1 ... a_(N-1) b_1 ... b_M]'
   y0  : [y(-1) y(-2) ... y(-M)]'
   d0  : desired signal at time index 0
   mu0 : vector of step sizes
Output Parameters [default]:
   u   : initialized composite input [zeros]
   w   : initialized filter coef. [zeros]
   y   : initialized filter output vector [zeros]
   d   : Initialized desired signal [white noise]
   e   : Initial error signal [e=d-y]
   mu  : step size vector [.01 ... .01)].
   Px  : power of x(n).
   Pd  : power of d(n).
```

Example

```
N  = 2;                     % Number of numerator coef.
M  = 2;                     % Number of denumerator coef.
u0 = rand(4,1);             % initial composite input vector
y0 = zeros(2,1);            % initial output vector
d0 = 0;                     % desired sample
mu = [0.1;0.1;0.01;0.01]; % step size vector

% Create and initialize an Output Error filter
[u,w,y,e,mu,Px,Pd]=init_eqerr(N,M,u0,[],y0,d0,mu);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 3 through 7 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTEQERR, MODEL_ EQERR.

6.9 init_ outerr

Purpose Creates and initializes the variables required for the Output Error recursive adaptive algorithm. The filter transfer function is given by

$$w(z) = \frac{A(z)}{1 - B(z)} \tag{6.21}$$

where

$$A(z) = a_0 + a_1 z^{-1} + ... + a_{N-1} z^{-N+1} \tag{6.22}$$
$$B(z) = b_1 z^{-1} + b_2 z^{-2} + ... + b_M z^{-M} \tag{6.23}$$

Syntax

```
[u,w,c,y,d,e,mu,Px,Py]=init_outerr(N,M)
[u,w,c,y,d,e,mu,Px,Py]=init_outerr(N,M,u0,w0,c0,d0,mu0)
```

Description The variables of the output error algorithm are summarized below (see Fig. 6.5).

```
Input arguments:
   N   : Number of coefficients of A(z).
   M   : Number of coefficients of B(z).
   u0  : composite input vector [N+M x 1]
   w0  : composite filter coefficients vector
   c0  : composite gradient vector
   d0  : initial desired sample
   mu0 : vector of step sizes
Output Parameters [default]:
   u   : initialized composite input [zeros]
   w   : initialized filter coef. vector [zeros]
   c   : initialized gradient vector [zeros]
   y   : filter output [zeros]
   d   : Initialized desired signal [white noise]
   e   : Initial error signal [e=d-y]
   mu  : step size vector [.01 ... .01)].
   Px  : power of x(n).
   Py  : power of y(n).
```

Example

```
N  = 2;                    % Number of numerator coef.
M  = 2;                    % Number of denumerator coef.
w0 = [1;0;0;0];            % initial filter coef.
u0 = rand(4,1);            % initial composite input vector
c0 = zeros(4,1);           % initial gradient vector
d0 = 0;                    % desired sample
mu = [0.1;0.1;0.01;0.01]; % step size vector
% Create and initialize an Output Error filter
[u,w,c,y,d,e,mu,Px,Py]=init_outerr(N,M,u0,w0,c0,d0,mu);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 3 through 7 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTOUTERR, MODEL_ OUTERR.

6.10 init_ sharf

Purpose Creates and initializes the variables required for the Simple Hyperstable Adaptive Recursive Filter (SHARF) algorithm. The filter transfer function is given by

$$w(z) = \frac{A(z)}{1 - B(z)} \tag{6.24}$$

where

$$A(z) = a_0 + a_1 z^{-1} + ... + a_{N-1} z^{-N+1} \tag{6.25}$$
$$B(z) = b_1 z^{-1} + b_2 z^{-2} + ... + b_M z^{-M} \tag{6.26}$$

Syntax

```
[u,w,e,c,d,mu,Px,Py]=init_sharf(N,M,L)
[u,w,e,c,d,mu,Px,Py]=init_sharf(N,M,L,u0,w0,e0,c0,d0,mu0)
```

Description The variables of the SHARF algorithm are summarized below (see Fig. 6.7).

```
Input arguments:
   N   : Number of coefficients of A(z)
   M   : Number of coefficients of B(z)
   L   : Number of coefficients of error smoothing filter c
   u0  : initial composite input/output delay line [N+M x 1]
   w0  : initial composite filter coefficients vector [N+M x 1]
   e0  : initial error vector [L x 1]
   c0  : smoothing filter coefficients [L x 1]
   d0  : desired sample at time index 0 [1 x 1]
   mu0 : step size vector [N+M x 1]
Output Parameters:
   u   : Initialized composite delay line [zeros].
   w   : Initialized filter vector [zeros].
   e   : Initialized error vector [e=d-y].
   c   : Initialized error smoothing filter [fir1(L-1,.05)]
   d   : Initialized desired sample
   mu  : Initialized step size vector 0.01*[1 ... 1].
   Px  : Initialized power of x(n).
   Pd  : Initialized power of d(n).
```

Example

```
N  = 2;                    % Number of numerator coef.
M  = 2;                    % Number of denumerator coef.
L  = 5;                    % error smoothing filter length
u0 = rand(4,1);            % initial composite input vector
c0 = fir1(L-1,0.1);        % error smoothing filter
d0 = 0;                    % desired sample
mu = [0.1;0.1;0.01;0.01];  % step size vector

% Create and initialize an Output Error filter
[u,w,e,c,d,mu,Px,Py]=init_sharf(N,M,L,u0,[],[],c0,d0,mu);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 4 through 9 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTSHARF, MODEL_ SHARF.

6.11 init_ soiir1

Purpose Creates and initializes the variables for the Second Order IIR type-1 adaptive filter. The filter transfer function is given by

$$H(z) = \frac{(1-s)(w-z^{-1})}{1-(1+s)wz^{-1}+sz^{-2}}. \quad (6.27)$$

where parameter s controls the filter bandwidth and parameter w controls the filter center frequency.

Syntax

```
[s,w,u,y,a,b,e]=init_soiir1(s0,w0)
[s,w,u,y,a,b,e]=init_soiir1(s0,w0,u0,y0,a0,b0,e0)
```

Description The variables of the SOIIR1 algorithm are summarized below (see Fig. 6.9).

```
Input arguments [Size]:
  s0  : initial adaptive bandwidth parameter [1x1]
  w0  : initial adaptive center frequency parameter [1x1]
  u0  : last 3 input samples [3x1]
  y0  : last 3 output samples [3x1]
  a0  : last 3 w-gradients [3x1]
  b0  : last 3 s-gradients [3x1]
  e0  : last 3 error samples [3x1]

Output Parameters [default]:
  s   : initialized bandwidth parameter [zero]
  w   : initialized center frequency parameter [zero]
  u   : initialized input buffer [zeros]
  y   : initialized output buffer [zeros]
  a   : initialized w-gradient buffer [zeros]
  b   : initialized s-gradient buffer [zeros]
  e   : initialized error buffer [zeros]
```

Example

```
u0 = [.9; .5;.3];
y0 = zeros(3,1);
a0 = [.01;.05;.01];
b0 = zeros(3,1);
% Create and initialize a SOIIR1 filter
[s,w,u,y,a,b,e]=init_soiir1(0.3,0.1,u0,y0,a0,b0);
```

Remarks Use input parameters 3 through 7 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTSOIIR1, ALE_ SOIIR1.

6.12 init_ soiir2

Purpose Creates and initializes the variables for the Second Order IIR type-2 adaptive filter. The filter transfer function is given by

$$H(z) = \frac{(1-s)(cos(t) - z^{-1})}{1-(1+s)cos(t)z^{-1} + sz^{-2}}. \tag{6.28}$$

where parameter s controls the filter bandwidth and parameter t controls the filter center frequency.

Syntax

```
[s,t,u,y,a,b,e]=init_soiir2(s0,t0)
[s,t,u,y,a,b,e]=init_soiir2(s0,t0,u0,y0,a0,b0,e0)
```

Description The variables of the SOIIR2 algorithm are summarized below (see Fig. 6.11).

```
Input arguments [Size]:
  s0  : initial adaptive bandwidth parameter [1x1]
  t0  : initial adaptive center frequency parameter [1x1]
  u0  : last 3 input samples [3x1]
  y0  : last 3 output samples [3x1]
  a0  : last 3 w-gradients [3x1]
  b0  : last 3 s-gradients [3x1]
  e0  : last 3 error samples [3x1]

Output Parameters [default]:
  s   : initialized bandwidth parameter [zero]
  t   : initialized center frequency parameter [zero]
  u   : initialized input buffer [zeros]
  y   : initialized output buffer [zeros]
  a   : initialized w-gradient buffer [zeros]
  b   : initialized s-gradient buffer [zeros]
  e   : initialized error buffer [zeros]
```

Example

```
u0 = [.9; .5;.3];
y0 = zeros(3,1);
a0 = [.01;.05;.01];
b0 = zeros(3,1);
% Create and initialize a SOIIR2 filter
[s,t,u,y,a,b,e]=init_soiir2(0.3,0.1,u0,y0,a0,b0);
```

Remarks Use input parameters 3 through 7 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTSOIIR2, ALE_ SOIIR2.

Chapter 7

Active Noise and Vibration Control Algorithms

This chapter documents the functions used to create, initialize, and update the coefficients of active noise and vibration control filters. Table 7.1 summarizes the ANVC functions and gives a short description and a pointer to the reference page of each function.

Function Name	Reference	Short Description
asptadjlms	7.1	Adjoint-LMS algorithm.
asptfdadjlms	7.2	Frequency Domain Adjoint LMS algorithm.
asptfdfxlms	7.3	Frequency Domain Filtered-x LMS algorithm.
asptfxlms	7.4	Filtered-x LMS algorithm.
asptmcadjlms	7.5	Multichannel Adjoint-LMS algorithms.
asptmcfdadjlms	7.6	Multichannel Frequency Domain Adjoint LMS algorithm.
asptmcfdfxlms	7.7	Multichannel Frequency Domain Filtered-x LMS algorithm.
asptmcfxlms	7.8	Multichannel Filtered-x LMS algorithm.
init_ adjlms	7.9	Initialize Adjoint LMS.
init_ fdadjlms	7.10	Initialize Frequency Domain Adjoint LMS.
init_ fdfxlms	7.11	Initialize Frequency Domain Filtered-x LMS.
init_ fxlms	7.12	Initialize Filtered-x LMS.
init_ mcadjlms	7.13	Initialize Multichannel Adjoint LMS.
init_ mcfdadjlms	7.14	Initialize Multichannel Frequency Domain Adjoint LMS.
init_ mcfdfxlms	7.15	Initialize Multichannel Frequency Domain Filtered-x LMS.
init_ mcfxlms	7.16	Initialize Multichannel Filtered-x LMS.

Table 7.1: Functions for creating, initializing, and updating active noise and vibration control filters.

Each function is documented in a separate section including the following information related to the function:

- **Purpose:** Short description of the algorithm implemented by this function.
- **Syntax:** Shows the function calling syntax. If the function has optional parameters, this section will have two calling syntaxes. One with only the required formal parameters and one with all the formal parameters.
- **Description:** Detailed description of the function usage with explanation of its input and output parameters.
- **Example:** A short example showing typical use of the function. The examples listed can be found in the ASPT/test directory of the ASPT distribution. The user is encouraged to copy from those examples and paste in her own applications.

- **Algorithm:** A short description of the operations internally performed by the function.
- **Remarks:** Gives more theoretical and practical remarks related to the usage, performance, limitations, and applications of the function.
- **Resources:** Gives a summary of the memory requirements and number of multiplications, addition/subtractions, and division operations required to implement the function in real time. This can be used to roughly calculate the MIPS (Million Instruction Per Second) required for a specific platform knowing the number of instructions the processor needs to perform each operation.
- **See Also:** Lists other functions that are related to this function.
- **Reference:** Lists literature for more information on the function.

Active noise and vibration control systems usually operate in two phases. The first phase is an identification phase in which models of the system secondary paths are obtained and stored in memory. The second phase is the control phase in which the coefficients of an adaptive controller are adjusted to reduce the noise or vibration at the error sensors. When the secondary paths are continuously changing, it is also necessary to continuously update the models of the secondary paths during the control phase. The functions documented here assume that the models of the secondary paths are obtained using an external identification process and perform the control task only. The identification process can be performed using any of the transversal or recursive adaptive algorithms in a system identification setup. The functions, therefore, distinguish between the physical secondary paths (usually named s) and the estimated secondary paths se. The physical model is used to calculate the response of the secondary actuators at the error sensors, while the estimated secondary paths are used to adapt the coefficients of the controller.

7.1 asptadjlms

Purpose Sample per sample filtering and coefficient update using the time domain Adjoint-LMS algorithm for single channel active noise and vibration control applications.

Syntax `[w,y,e,p] = asptadjlms(w,x,e,y,s,se,d,p,mu,b)`

Description `asptadjlms()` implements the ADJOINT-LMS algorithm widely used in control applications where a transfer function (the secondary path, s) exists between the filter output and the error signal (see Fig. 7.1). The consequence of this transfer function is twofold. (1) its phase response delays the filter output signal and makes it observable from the error signal after a delay. (2) the filter output signal is colored by the amplitude response of the secondary path s. To correct for those two effects, the ADJOINT-LMS algorithm uses a filtered version of the error signal to update the adaptive filter instead of directly using the error signal as shown in Fig. 7.1. This figure also shows the input and output parameters of `asptadjlms()` which are summarized below.

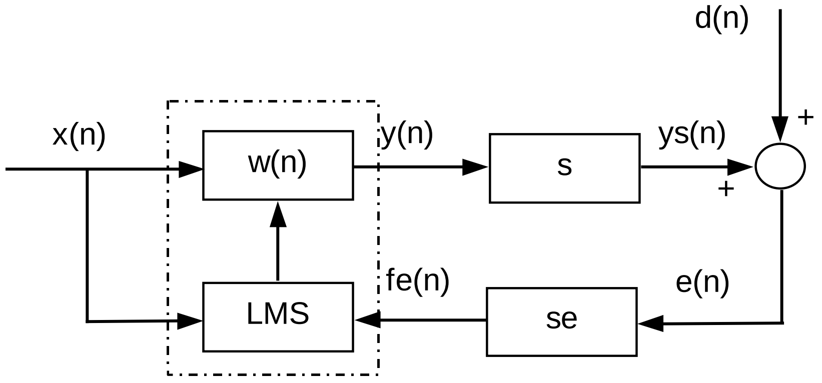

Figure 7.1: Block diagram of the Adjoint-LMS algorithm.

```
Input Parameters ::
    w  : vector of filter coefficients w(n-1) [L x 1]
    x  : vector of input samples  [x(n) x(n-1) .. x(n-(L+M-1))]
    e  : vector of error signal e(n-1) [N x 1]
    y  : vector of filter-output y(n-1) [M x 1]
    s  : accurate FIR model of the secondary path [M x 1]
    se : estimated FIR model of the secondary path [N x 1]
    d  : desired response at sample index n [1 x 1]
    p  : last estimated power of x(n) [1 x 1]
    mu : adaptation constant [1 x 1]
    b  : pole of Autoregressive filter used in estimating p

Output parameters ::
    w  : updated filter coefficients w(n)
    y  : filter output vector [y(n) y(n-1) .. y(n-M-1)]
    e  : error vector [e(n) e(n-1) .. e(n-N-1)]
    p  : updated estimate of input vector power
```

Example

```
iter = 5000;                    % Number of samples to process
ph   = [0;.9;.5;.3;.1];         % Primary path impulse response
sh   = [0.5;0.4;0.1];           % Secondary path impulse response
se   = 0.95*sh;                 % estimation of s
xn   = 2*(rand(iter,1)-0.5);    % Input signal, zero mean random.
dn   = osfilter(ph,xn);         % Primary response at the sensor
sens = zeros(iter,1);           % vector to collect sensor signal

% Initialize ADJLMS algorithm with a controller of 10 coefficients
[w,x,y,d,e,p] = init_adjlms(10,sh,se);

%% Processing Loop
for (m=1:iter)
   % update the input delay line
   x = [xn(m,:); x(1:end-1,:)];

   % call asptadjlms to calculate the controller output
   % and update the coefficients. Below a step size of
   % 0.02 and an AR pole of 0.98 are used.
   [w,y,e,p] = asptadjlms(w,x,e,y,sh,se,dn(m),p, 0.02, 0.98);

   % save the last calculated sensor sample for
   %performance examination
   sens(m) = e(1);
end;

% display the sensor signal before and after the control effort
plot([dn sens]);
```

Running the above script will produce the graph shown in Fig. 7.2. In this figure, the sensor signal with and without control, *sens* and *dn*, respectively, are shown. The sensor signal before applying the controller *dn* results from filtering the random variable xn of zero mean and variance 1 through the primary path *ph*. The adaptive controller adjusts its coefficients to produce a control signal $y(n)$ to drive the secondary actuator that results in reducing the primary noise at the sensor.

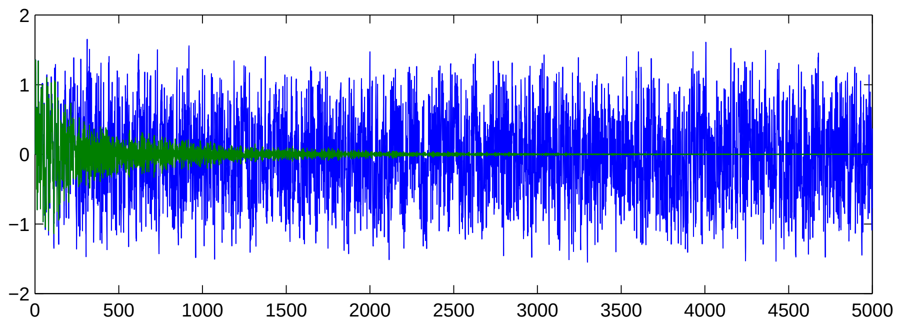

Figure 7.2: Sensor signal before and after applying the adaptive controller in a single channel ANVC system using the adjoint LMS algorithm.

Algorithm `asptadjlms()` performs the following operations.

- filters the input vector $x(n)$ through the adaptive filter coefficients vector $w(n-1)$ to produce the filter output vector $y(n)$
- filters $y(n)$ through the secondary path filter s to produce the secondary actuator response at the sensor $ys(n)$
- evaluates the current error sample $e(n) = d(n) + ys(n)$. Note the error here is formed by adding the signal rather than subtracting them to be compatible with real world sensors such as microphones and accelerometers
- filters the mirrored error vector $e(n)$ through the estimate of the secondary path se to produce the filtered-error signal $fe(n)$
- uses $x(n)$ and $fe(n)$ to calculate the normalized gradient vector and uses this to update the adaptive filter coefficients $w(n)$

Remarks

- Supports both real and complex signals
- The Wiener solution to the above problem is given by $W(\omega) = S(\omega)^{-1}P(\omega)$, where $W(\omega)$ is the controller response at frequency ω, $S(\omega)$ is the response of the secondary path and $P(\omega)$ is the response of the primary path at the same frequency. The adaptive controller will asymptotically approach this Wiener solution provided that $S(\omega)$ is a minimum phase function (does not have zeros outside the unit circle) and the controller length is large enough to accommodate the above convolution. If $S(\omega)$ is not a minimum phase function, the adaptive controller will approach the causal part of the solution. If the controller is too short, the solution will be truncated. In both cases, the noise reduction at the sensor is decreased.
- The adaptive controller will approach the Wiener solution provided that the delay in the primary path is larger than that in the secondary path. This can be quickly checked by using $ph = [.9; .5; .3; .1]$ and $sh = [0; 0.5; 0.4; 0.1]$ in the above example.
- The performance, memory requirements, and processing load of the ADJOINT LMS algorithm are similar to the Filtered-x LMS algorithms for single channel systems. The real advantage of the ADJOINT LMS is in multi-channel applications.

Resources The resources required to implement the ADJOINT LMS algorithm in real time is given in the table below where L is the controller length and N is the estimated secondary path length. The computations given are those required to process one sample.

MEMORY	$2L + 2N + 5$
MULTIPLY	$2L + N + 4$
ADD	$2L + N$
DIVIDE	1

See Also INIT_ ADJLMS, ANVC_ ADJLMS, ASPTMCADJLMS, ASPTFDADJLMS, ASPTMCFDADJLMS.

Reference [3], Chapter 3.

7.2 asptfdadjlms

Purpose Block filtering and coefficient update in frequency domain using the Frequency Domain ADJoint LMS (FDADJLMS) algorithm for single channel active noise and vibration control applications.

Syntax

```
[W,w,x,y,e,p,yF,feF] = asptfdadjlms(NC,W,
                       x,xn,d,yF,feF,S,SE,p,mu,b,c)
```

Description `asptfdadjlms()` is the frequency domain implementation of `asptadjlms()`. The difference between FDADJLMS and its time domain counterpart ADJLMS is that filtering and coefficient update are performed in frequency domain using the overlap-save method as shown in Fig. 7.3. Each call of `asptfdadjlms()` processes NL time samples. The parameters of `asptfdadjlms()` which are summarized below.

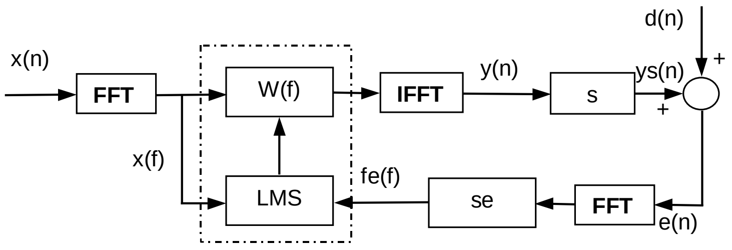

Figure 7.3: Block diagram of the Frequency Domain Adjoint-LMS algorithm.

```
Input Parameters [size] ::
    NC   : controller length in time domain
    W    : frequency domain filter coef. vector [NB x 1]
    x    : previous overlap-save input vector [NB x 1]
    xn   : block of new input samples [NL x 1]
    d    : block of new primary response [NL x 1]
    yF   : previous output buffer [NB x 1]
    feF  : previous filtered error buffer [NB x 1]
    S    : frequency domain secondary path [NB x 1]
    SE   : frequency domain estimated s [NB x 1]
    p    : last estimated power of x(f) [NB x 1]
    mu   : adaptation constant
    b    : pole of AR filter used in estimating p
    c    : if not zero, uses the constrained bfdaf algorithm.
Output parameters ::
    W    : updated frequency domain filter coefficients vector
    w    : updated time domain filter coefficients vector
    x    : updated overlap-save input vector
    y    : controller output block
    e    : new error block
    p    : updated power estimate of x(n)
    yF   : updated output buffer
    feF  : updated filtered error buffer
```

Example

```
iter = 5000;                  % Number of samples to process
ph   = [0;.9;.5;.3;.1];       % Primary path impulse response
sh   = [0.5;0.4;0.1];         % Secondary path impulse response
se   = 0.95*sh;               % estimation of s
xa   = 2*(rand(iter,1)-0.5);  % Input signal, zero mean random.
da   = osfilter(ph,xa);       % Primary response at the sensor
sens = zeros(iter,1);         % vector to collect sensor signal
NC   = 10;                    % controller length
NL   = 6;                     % block length
mu   = 0.02/NC;               % step size for block processing
c    = 1;                     % constrain filter to NC coef.
b    = 0.98;                  % AR pole
% Initialize FDADJLMS algorithm with a controller of NC coef.
% and block length of NL samples
[NB,W,w,x,y,d,e,p,S,SE,yF,feF] = init_fdadjlms(NC,NL,sh,se);

% Processing Loop
for (m=1:NL:iter-NL)
   xn = xa(m:m+NL-1);         % new input block of NL samples
   dn = da(m:m+NL-1);         % new desired block of NL samples
   % call asptfdadjlms to calculate the controller output
   % and update the coefficients.
   [W,w,x,y,e,p,yF,feF] = asptfdadjlms(NC,W,x,xn,dn,yF,...
                          feF,S,SE,p,mu,b,c);
   % save the last calculated sensor block of samples for
   %performance examination
   sens(m:m+NL-1) = e;
end;

% display the sensor signal before and after the control effort
plot([da sens]);
```

Running the above script will produce the graph shown in Fig. 7.4. In this figure, the sensor signal with and without control, *sens* and *da*, respectively, are shown. the sensor signal before applying the controller *da* results from filtering the random variable *xa* of zero mean and variance 1 through the primary path *ph*. The adaptive controller adjusts its coefficients to produce a control signal $y(n)$ to drive the secondary actuator that results in reducing the primary noise at the sensor.

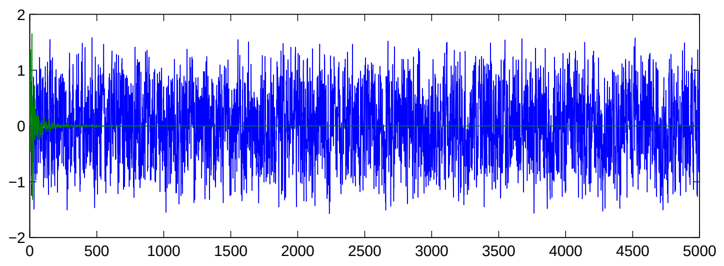

Figure 7.4: Sensor signal before and after applying the adaptive controller in a single channel ANVC system using the frequency domain adjoint LMS algorithm.

Algorithm

`asptfdadjlms()` performs the following operations.

- composes the overlap-save input vector $x(n)$ and computes its FFT, $x(f)$
- filters $x(f)$ through the adaptive filter coefficients vector $W(f)$ in frequency domain to produce the filter-output vector $y(f)$
- filters $y(f)$ through the secondary path filter s to produce the secondary actuator response at the sensor $ys(n)$ (this is also performed in frequency domain)
- evaluates the current error sample $e(n) = d(n)+ys(n); n = 0, 1, \cdots, NL-1$. Note the error here is formed by adding the signals rather than subtracting them to be compatible with real world sensors such as microphones and accelerometers. The error vector is padded with zeros and transformed to frequency domain giving $e(f)$
- filters the error vector $e(f)$ through the estimate of the secondary path se in frequency domain to produce the filtered-error signal $fe(f)$
- uses $x(f)$ and $fe(f)$ to calculate the normalized gradient vector and uses this to update the frequency domain adaptive filter coefficients $W(f)$. Normalization for both input signal and secondary path are performed at each frequency bin which guarantees faster convergence rate than time domain ADJLMS.
- computes the inverse FFT for the filter coefficients vector, output vector, and error vector producing $w(n)$, $y(n)$, and $e(n)$ respectively

Remarks

- Supports both real and complex signals.
- Much more efficient than time domain processing for long controllers.
- The Wiener solution to the above problem is given by $W(\omega) = S(\omega)^{-1}P(\omega)$, where $W(\omega)$ is the controller response at frequency ω, $S(\omega)$ is the response of the secondary path and $P(\omega)$ is the response of the primary path at the same frequency. The adaptive controller will asymptotically approach this Wiener solution provided that $S(\omega)$ is a minimum phase function (does not have zeros outside the unit circle) and the controller length is large enough to accommodate the above convolution. If $S(\omega)$ is not a minimum phase function, the adaptive controller will approach the causal part of the solution. If the controller is too short, the solution will be truncated. In both cases, the noise reduction at the sensor is decreased.
- The adaptive controller will approach the Wiener solution provided that the delay in the primary path is larger than that in the secondary path. This can be quickly checked by using $ph = [.9; .5; .3; .1]$ and $sh = [0; 0.5; 0.4; 0.1]$ in the above example.
- The performance, memory requirements, and processing load of the FDADJLMS algorithm are similar to the FDFXLMS algorithms for single channel systems. The real advantage of the FDADJLMS is in multichannel applications, see ASPTMCFDADJLMS for more details.

Resources The resources required to implement the constrained FDADJLMS algorithm in real time is given in the table below. In this table, N_L is the block length and N_B is the FFT length given by $N_B = 2^{nextpow2(N_L+N_C-1)}$, and N_C is the controller length in time domain. The computations given are those required to process N_L samples. Note that the unconstrained algorithm uses two FFT operations less than the case shown in the table.

MEMORY	$7N_B + 4N_L + 3$
MULTIPLY	$7N_B$
ADD	$2N_B$
DIVIDE	N_B
FFT	7

See Also INIT_ FDADJLMS, ANVC_ FDADJLMS, ASPTMCADJLMS, ASPTMCFDADJLMS, ASPTADJLMS.

Reference [3], Chapter 3 for detailed description of the FDADJLMS, [8] for the overlap-save method, and [9] for frequency domain adaptive filters.

7.3 asptfdfxlms

Purpose Block filtering and coefficient update in frequency domain using the Frequency Domain Filtered-X LMS (FDFXLMS) algorithm for single channel active noise and vibration control applications.

Syntax

```
[W,w,x,y,e,p,yF,fxF] = asptfdfxlms(NC,W,
                         x,xn,d,yF,fxF,S,SE,p,mu,b,c)
```

Description `asptfdfxlms()` is the frequency domain implementation of `asptfxlms()`. The difference between FDFXLMS and its time domain counterpart FXLMS is that filtering and coefficient update are performed in frequency domain using the overlap-save method as shown in Fig. 7.5. The parameters of `asptfdfxlms()` which are summarized below.

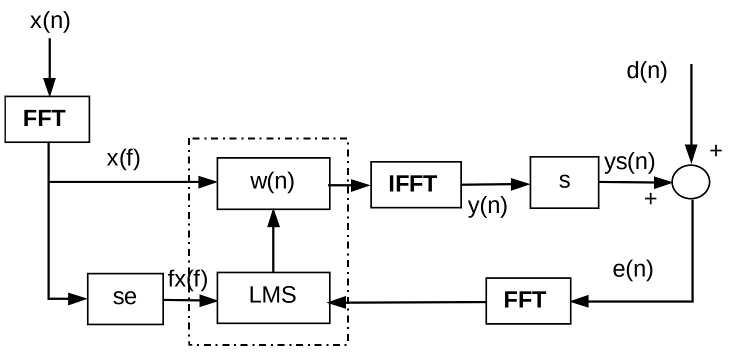

Figure 7.5: Block diagram of the Frequency Domain Filtered-X LMS algorithm.

```
Input Parameters [size] ::
    NC   : controller length in time domain
    W    : frequency domain filter coef. vector [NB x 1]
    x    : previous overlap-save input vector [NB x 1]
    xn   : block of new input samples [NL x 1]
    d    : block of new primary response [NL x 1]
    yF   : previous output buffer [NB x 1]
    fxF  : previous filtered input buffer [NB x 1]
    S    : frequency domain secondary path [NB x 1]
    SE   : frequency domain estimated s [NB x 1]
    p    : last estimated power of fx(f) [NB x 1]
    mu   : adaptation constant
    b    : pole of AR filter used in estimating p
    c    : if not zero, uses the constrained bfdaf algorithm.
Output parameters ::
    W    : updated frequency domain filter coefficients vector
    w    : updated time domain filter coefficients vector
    x    : updated overlap-save input vector
    y    : controller output block
    e    : new error block
    p    : updated power estimate of fx(n)
    yF   : updated output buffer
    fxF  : updated filtered input buffer
```

Example

```
iter = 5000;                    % Number of samples to process
ph   = [0;.9;.5;.3;.1];         % Primary path impulse response
sh   = [0.5;0.4;0.1];           % Secondary path impulse response
se   = 0.95*sh;                 % estimation of s
xa   = 2*(rand(iter,1)-0.5);    % Input signal, zero mean random.
da   = osfilter(ph,xa);         % Primary response at the sensor
sens = zeros(iter,1);           % vector to collect sensor signal
NC   = 10;                      % controller length
NL   = 6;                       % block length
mu   = 0.02/NC;                 % step size for block processing
c    = 1;                       % constrain filter to NC coef.
b    = 0.98;                    % AR pole
% Initialize FDFXLMS algorithm with a controller of NC coef.
% and block length of NL samples
[NB,W,w,x,y,d,e,p,S,SE,yF,fxF] = init_fdfxlms(NC,NL,sh,se);

% Processing Loop
for (m=1:NL:iter-NL)
   xn = xa(m:m+NL-1);           % new input block of NL samples
   dn = da(m:m+NL-1);           % new desired block of NL samples
   % call asptfdfxlms to calculate the controller output
   % and update the coefficients.
   [W,w,x,y,e,p,yF,fxF] = asptfdfxlms(NC,W,x,xn,dn,yF,...
                          fxF,S,SE,p,mu,b,c);
   % save the last calculated sensor block of samples for
   %performance examination
   sens(m:m+NL-1) = e;
end;

% display the sensor signal before and after the control effort
plot([da sens]);
```

Running the above script will produce the graph shown in Fig. 7.6. In this figure, the sensor signal with and without control, *sens* and *da*, respectively, are shown. the sensor signal before applying the controller *da* results from filtering the random variable *xa* of zero mean and variance 1 through the primary path *ph*. The adaptive controller adjusts its coefficients to produce a control signal $y(n)$ to drive the secondary actuator that results in reducing the primary noise at the sensor.

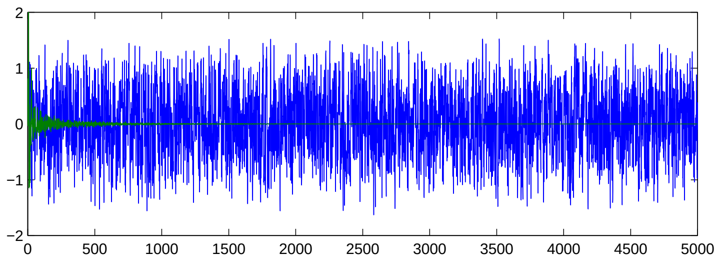

Figure 7.6: Sensor signal before and after applying the adaptive controller in a single channel ANVC system using the frequency domain filtered-x LMS algorithm.

Algorithm `asptfdfxlms()` performs the following operations.

- composes the overlap-save input vector $x(n)$ and computes its FFT, $x(f)$
- filters $x(f)$ through the adaptive filter coefficients vector $W(f)$ in frequency domain to produce the filter output vector $y(f)$
- filters $y(f)$ through the secondary path filter s to produce the secondary actuator response at the sensor $ys(n)$ (this is also performed in frequency domain)
- evaluates the current error sample $e(n) = d(n)+ys(n); n = 0, 1, \cdots, NL-1$. Note the error here is formed by adding the signals rather than subtracting them to be compatible with real world sensors such as microphones and accelerometers. The error vector is padded with zeros and transformed to frequency domain giving $e(f)$
- filters the overlap-save input vector $x(f)$ through the estimate of the secondary path se in frequency domain to produce the filtered input signal $fx(f)$
- uses $e(f)$ and $fx(f)$ to calculate the normalized gradient vector and uses this to update the frequency domain adaptive filter coefficients $W(f)$. Normalization for both input signal and secondary path are performed at each frequency bin which guarantees faster convergence rate than time domain FXLMS.
- computes the inverse FFT for the filter coefficients vector, and output vector producing $w(n)$, and $y(n)$, respectively

Remarks

- Supports both real and complex signals.
- Much more efficient than time domain processing for long controllers.
- The Wiener solution to the above problem is given by $W(\omega) = S(\omega)^{-1}P(\omega)$, where $W(\omega)$ is the controller response at frequency ω, $S(\omega)$ is the response of the secondary path and $P(\omega)$ is the response of the primary path at the same frequency. The adaptive controller will asymptotically approach this Wiener solution provided that $S(\omega)$ is a minimum phase function (does not have zeros outside the unit circle) and the controller length is large enough to accommodate the above convolution. If $S(\omega)$ is not a minimum phase function, the adaptive controller will approach the causal part of the solution. If the controller is too short, the solution will be truncated. In both cases, the noise reduction at the sensor is decreased.
- The adaptive controller will approach the Wiener solution provided that the delay in the primary path is larger than that in the secondary path. This can be quickly checked by using $ph = [.9; .5; .3; .1]$ and $sh = [0; 0.5; 0.4; 0.1]$ in the above example.
- The performance, memory requirements, and processing load of the FD-FXLMS algorithm are similar to the FDADJLMS algorithms for single channel systems. The FDADJLMS, however, is much more efficient in multi-channel applications, see ASPTMCFDADJLMS and ASPTMCFD-FXLMS for more details.

Resources The resources required to implement the constrained FDFXLMS algorithm in real time is given in the table below. In this table, N_L is the block length and N_B is the FFT length given by $N_B = 2^{nextpow2(N_L+N_C-1)}$, and N_C is the controller length in time domain. The computations given are those required to process N_L samples. Note that the unconstrained algorithm uses two FFT operations less than the case shown in the table.

MEMORY	$7N_B + 4N_L + 3$
MULTIPLY	$7N_B$
ADD	$2N_B$
DIVIDE	N_B
FFT	7

See Also INIT_ FDFXLMS, ANVC_ FDFXLMS, ASPTMCFXLMS, ASPTMCFDFXLMS, ASPTFXLMS.

Reference [3], Chapter 3 for detailed description of the FDFXLMS, [8] for the overlap-save method, and [9] for frequency domain adaptive filters.

7.4 asptfxlms

Purpose Sample per sample filtering and coefficient update using the Filtered-x LMS (FXLMS) algorithm for single channel active noise and vibration control applications.

Syntax `[w,y,e,p,fx] = asptfxlms(w,x,y,s,se,d,fx,p,mu,b)`

Description `asptfxlms()` implements the FILTERED-X LMS algorithm widely used in control applications where a transfer function (the secondary path, s) exists between the filter output and the error signal (see Fig. 7.7). The consequence of this transfer function is that (1) its phase response delays the filter output and makes it observable from the error signal after a delay, (2) the filter output is colored by the amplitude response of the secondary path s. To correct for those effects, the FILTERED-X LMS algorithm uses a filtered version of the input signal $fx(n)$ to update the adaptive filter instead of directly using the input signal $x(n)$ as shown in Fig. 7.7. This figure also shows the input and output parameters of `asptfxlms()` which are summarized below.

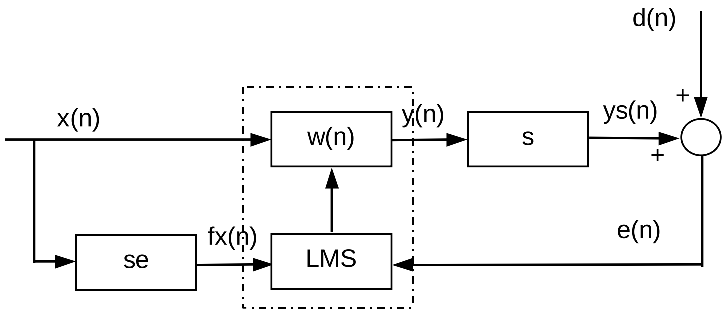

Figure 7.7: Block diagram of the Filtered-x LMS algorithm.

```
Input Parameters ::
    w  : vector of filter coefficients w(n-1) [L x 1]
    x  : vector of input samples
    y  : vector of filter output y(n-1) [M x 1]
    s  : accurate FIR model of the secondary path [M x 1]
    se : estimated FIR model of the secondary path [N x 1]
    d  : desired response at sample index n [1 x 1]
    fx : vector of filtered input signal fx(n-1) [N x 1]
    p  : last estimated power of x(n) [1 x 1]
    mu : adaptation constant [1 x 1]
    b  : pole of Autoregressive filter used in estimating p

Output parameters ::
    w  : updated filter coefficients w(n)
    y  : filter output vector [y(n) y(n-1) .. y(n-M-1)]
    e  : error sample e(n) = d(n) - ys(n)
    p  : updated estimate of input vector power
    fx : updated vector of filtered-x samples fx(n)
```

Example

```
iter = 5000;                  % Number of samples to process
ph   = [0;.9;.5;.3;.1];       % Primary path impulse response
sh   = [0.5;0.4;0.1];         % Secondary path impulse response
se   = 0.95*sh;               % estimation of s
xn   = 2*(rand(iter,1)-0.5);  % Input signal, zero mean random.
dn   = osfilter(ph,xn);       % Primary response at the sensor
sens = zeros(iter,1);         % vector to collect the sensor signal

% Initialize Filtered-x algorithm with a controller of 10 coef.
[w,x,y,d,e,p,fx]  = init_fxlms(10,sh,se);

%% Processing Loop
for (m=1:iter)
   % update the input delay line
   x = [xn(m,:); x(1:end-1,:)];

   % call asptfxlms to calculate the controller output
   % and update the coefficients. Below a step size of
   % 0.02 and an AR pole of 0.98 are used.
   [w,y,e,p,fx]  = asptfxlms(w,x,y,sh,se,dn(m),fx,p, 0.02, 0.98);

   % save the last calculated sensor sample for
   %performance examination
   sens(m) = e(1);
end;

% display the sensor signal signal before and after
% applying the controller
plot([dn sens]);
```

Running the above script will produce the graph shown in Fig. 7.8. In this figure, the sensor signal with and without control, *sens* and *dn*, respectively, are shown. The sensor signal before applying the controller *dn* results from filtering the random variable *xn* of zero mean and variance 1 through the primary path *ph*. The adaptive controller adjusts its coefficients to produce a control signal $y(n)$ to drive the secondary actuator that results in reducing the primary noise at the sensor.

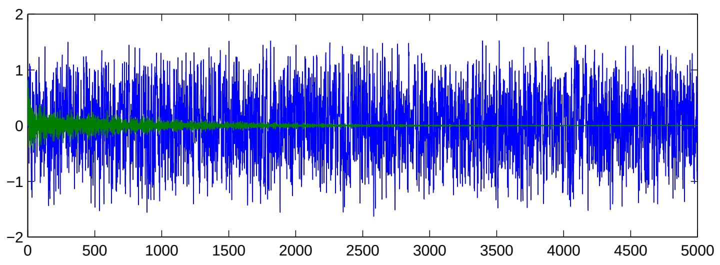

Figure 7.8: Sensor signal before and after applying the adaptive controller in a single channel ANVC system using the filtered-x LMS algorithm.

Algorithm ASPTFXLMS performs the following operations.

- filters the input vector $x(n)$ through the adaptive filter coefficients vector $w(n-1)$ to produce the filter output vector $y(n)$
- filters $y(n)$ through the secondary path filter s to produce the secondary actuator response at the sensor $ys(n)$
- evaluates the current error sample $e(n) = d(n) + ys(n)$. Note the error here is formed by adding the signal rather than subtracting them to be compatible with real world sensors such as microphones and accelerometers
- filters the input signal $x(n)$ through the estimate of the secondary path se to produce the filtered-x signal $fx(n)$
- uses $fx(n)$ and $e(n)$ to calculate the normalized gradient vector and uses this to update the adaptive filter coefficients $w(n)$

Remarks

- Supports both real and complex signals
- The Wiener solution to the above problem is given by $W(\omega) = S(\omega)^{-1}P(\omega)$, where $W(\omega)$ is the controller response at frequency ω, $S(\omega)$ is the response of the secondary path and $P(\omega)$ is the response of the primary path at the same frequency. The adaptive controller will asymptotically approach this Wiener solution provided that $S(\omega)$ is a minimum phase function (does not have zeros outside the unit circle) and the controller length is large enough to accommodate the above convolution. If $S(\omega)$ is not a minimum phase function, the adaptive controller will approach the causal part of the solution. If the controller is too short, the solution will be truncated. In both cases, the noise reduction at the sensor is decreased.
- The adaptive controller will approach the Wiener solution provided that the delay in the primary path is larger than that in the secondary path. This can be quickly checked by using $ph = [.9; .5; .3; .1]$ and $sh = [0; 0.5; 0.4; 0.1]$ in the above example.
- The performance, memory requirements, and processing load of the FILTERED-X LMS algorithm are similar to the ADJOINT LMS algorithms for single channel systems. The ADJOINT LMS is much more efficient, however, in multi-channel applications.

Resources The resources required to implement the FILTERED-X LMS algorithm in real time is given in the table below where L is the controller length and N is the estimated secondary path length. The computations given are those required to process one sample.

MEMORY	$2L + 2N + 5$
MULTIPLY	$2L + N + 4$
ADD	$2L + N$
DIVIDE	1

See Also INIT_ FXLMS, ANVC_ AFXLMS, ASPTFDFXLMS, ASPTMCFDFXLMS.

Reference [3], Chapter 3.

7.5 asptmcadjlms

Purpose Sample per sample filtering and coefficient update using the Multichannel Adjoint-LMS (MCADJLMS) for multichannel active noise and vibration control applications.

Syntax

```
[w,y,e,p] = asptmcadjlms(w,x,e,y,s,se,d,p,mu,b)
```

Description `asptmcadjlms()` implements the Multichannel ADJOINT LMS algorithm widely used in control applications where a transfer function (the matrix of secondary paths, s) exists between the output of the multi input multi output (MIMO) controller $w(n)$ and the error sensors (see Fig. 7.9). The consequence of this matrix of transfer functions is that (1) the phase response of the transfer functions delay each of the controller's outputs and makes it observable from each error signal after a delay, (2) the controller's outputs are colored by the amplitude response of the secondary paths. To correct for those effects, the MCADJLMS algorithm uses filtered versions of the error signals to update the adaptive controller instead of directly using the error signals as shown in Fig. 7.9. The figure also shows the input and output parameters of `asptmcadjlms()` which are summarized below.

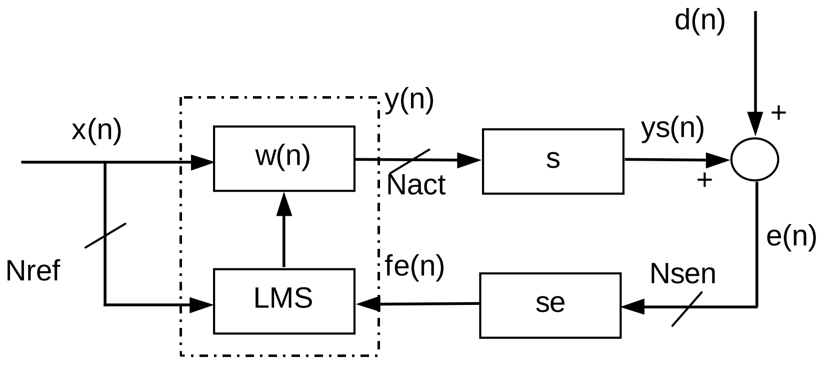

Figure 7.9: Block diagram of the Multichannel Adjoint-LMS algorithm.

```
Input Parameters [size]::
    w  : matrix of filter coefficients [L x Nref x Nact]
    x  : matrix of input samples x(n) [L+M-1 x Nref]
    e  : matrix of error signal e(n-1) [N x Nsens]
    y  : matrix of filter output y(n-1) [M x Nact]
    s  : accurate matrix of secondary paths [M x Nact x Nsens]
    se : estimated matrix of secondary paths [N x Nact x Nsens]
    d  : desired response at sample index n [1 x Nsens]
    p  : last estimated power of x(n) [1 x Nref]
    mu : adaptation constant
    b  : pole of AR filter used to smooth p

Output parameters ::
    w  : updated matrix of filter coefficients
    y  : filter output vector matrix y(n)
    e  : error matrix e(n)
    p  : new estimated input vector power
```

Example

```
% This example simulates a MIMO control system with a single
% primary (reference) signal, two actuators and two sensors.
iter = 5000;                   % Number of samples to process
ph   = [0 .9 .5 .3 .1 ; 0 .8 .5 .2 .5]';
ph   = reshape(ph,5,1,2);      % Primary path impulse response
sh   = zeros(3,2,2);           % Secondary path impulse response
sh(:,1,1) = [0.5;0.4;0.1];
sh(:,2,2) = [0.5;0.4;0.1];
se   = 0.95*sh;                % estimation of sh
xn   = 2*(rand(iter,1)-0.5);   % Input signal, zero mean random.
dn   = mcmixr(ph,xn,0);        % Primary response at the sensor
sens = zeros(iter,2);          % matrix for sensors signal
% Initialize MCADJLMS algorithm with a controller of 10 coef.
[w,x,y,d,e,p] = init_mcadjlms(10,1,2,2,sh,se);
%% Processing Loop
for (m=1:iter)
   % update the input delay line
   x = [xn(m,:); x(1:end-1,:)];
   % call asptmcadjlms to calculate the controller output
   % and update the coefficients. Below a step size of
   % 0.02 and an AR pole of 0.98 are used.
   [w,y,e,p] = asptmcadjlms(w,x,e,y,sh,se,dn(m,:),p,0.02,0.98);
   % save the last calculated sensor vector for
   %performance examination
   sens(m,:) = e(1,:);
end;
% display the sensor signal signal before and after
% applying the controller
subplot(2,2,1); plot([dn(:,1) sens(:,1)]); grid
subplot(2,2,2); plot([dn(:,2) sens(:,2)]); grid
```

Running the above script will produce the graph shown in Fig. 7.10. In this figure, the signals recorded by the sensors before and after applying the MIMO control effort, *dn* and *sens*, respectively, are shown. The adaptive controller adjusts its coefficients to produce N_{act} control signals $y(n)$ that result in reducing the primary noise at the sensors.

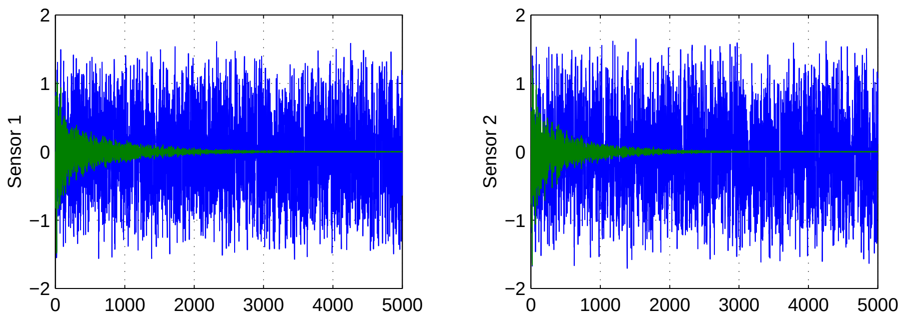

Figure 7.10: Signals recorded by the sensors before and after applying the adaptive controller in a Multichannel ANVC system using the multichannel adjoint LMS algorithm.

Algorithm The MIMO control problem addressed by MCADJLMS is to reduce the noise (or vibration) produced by N_{ref} primary sources at the positions of N_{sens} sensors using a matrix of $[N_{ref}xN_{act}]$ controllers driving N_{act} actuators. To achieve this goal, `asptmcadjlms()` performs the following operations.

- filters the N_{ref} reference signals $x(n)$ through the matrix of adaptive filters $w(n-1)$ to produce the N_{act} signals $y(n)$ used to drive the actuators
- filters $y(n)$ through the matrix of secondary paths s to produce the response of the actuators at the sensors' positions $ys(n)$
- evaluates the current error $e(n) = d(n) + ys(n)$ at all sensors. Note the error here is formed by adding the signal rather than subtracting them to be compatible with real world sensors such as microphones and accelerometers
- filters the mirrored error matrix $e(n)$ through the estimate of the secondary path matrix se to produce the filtered-error signal $fe(n)$
- uses $x(n)$ and $fe(n)$ to calculate the normalized gradient vector and uses this to update the adaptive filter coefficients $w(n)$

Remarks

- Supports both real and complex signals
- The Wiener solution to the above problem is given by $W(\omega) = S(\omega)^{-1}P(\omega)$, where $W(\omega)$ is the controller response at frequency ω, $S(\omega)$ is the response of the secondary path and $P(\omega)$ is the response of the primary path at the same frequency. The adaptive controller will asymptotically approach this Wiener solution provided that the inverse of $S(\omega)$ exists at each frequency and the controller length is large enough.
- The adaptive controller will approach the Wiener solution provided that the delay in the primary paths is larger than that in the secondary paths. This can be quickly checked by removing the leading zero in ph and adding a leading zero in sh in the above example.
- The memory requirements, and processing load of the MCADJLMS algorithm are much less than those of the Multiple Error Filtered-x algorithm, the multichannel counterpart of the Filtered-x LMS.

Resources The resources required for real time implementation of the MCADJLMS algorithm having N_{ref} reference signals, N_{act} actuators, and N_{sens} sensors with each filter of L coefficients and estimated secondary path of N coefficients is given in the table below. The computations given are those required to produce N_{act} control signals.

MEMORY	$N_{ref}(1 + L + LN_{act}) + N_{sens}(1 + N + NN_{act}) + N_{act} + 3$
MULTIPLY	$2LN_{ref}N_{act} + NN_{sens}N_{act} + (L + 3)N_{ref}$
ADD	$2LN_{ref}N_{act} + NN_{sens}N_{act} + (L + 1)N_{ref}$
DIVIDE	$L\ Nref$

See Also INIT_ MCADJLMS, ANVC_ MCADJLMS, ASPTADJLMS, ASPTFDADJLMS, ASPTMCFDADJLMS.

Reference [3], Chapter 3.

7.6 asptmcfdadjlms

Purpose Block filtering and coefficient update in frequency domain using the Multichannel Frequency Domain Adjoint LMS (MCFDADJLMS) for multichannel active noise and vibration control applications.

Syntax

```
[W,w,x,y,e,p,yF,feF] = asptmcfdadjlms(NC,W,x,xn,dn,yF,...
                       feF,S,SE,p,mu,b,c)
```

Description `asptmcfdadjlms()` is the frequency domain implementation of `asptmcadjlms()`. The difference between MCFDADJLMS and its time domain counterpart MCADJLMS is that filtering and coefficient update are performed in frequency domain using the overlap-save method. Fig. 7.11 shows the parameters of `asptmcfdadjlms()` which are summarized below.

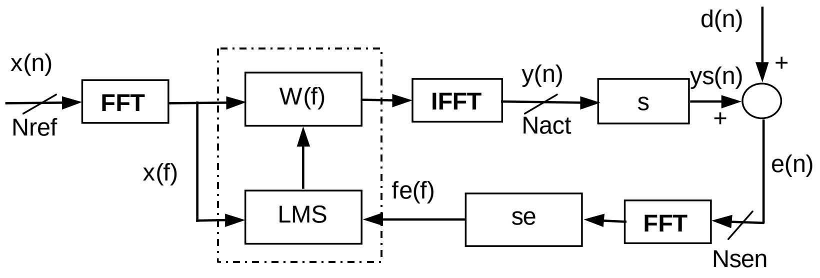

Figure 7.11: Block diagram of the Multi-Channel Frequency Domain Adjoint-LMS algorithm.

```
Input Parameters ::
    NC   : controller length in time domain
    W    : freq. domain filter coef. matrix [NB x Nref x Nact]
    x    : previous overlap-save input matrix [NB x Nref]
    xn   : new input samples block [NL x Nref]
    dn   : new primary samples block [NL x Nsens]
    yF   : previous buffer of y(n) [NB x Nact]
    feF  : previous buffer of fe(n) [NB x Nact]
    S    : FIR model of the secondary paths [NB x Nact x Nsens]
    SE   : estimated FIR model of S [NB x Nact x Nsens]
    p    : last estimated power of x(n) [NB x Nref]
    mu   : adaptation constant
    b    : pole of AR filter used in estimating p
    c    : if not zero, uses the constrained BFDAF algorithm.

Output parameters ::
    W    : updated frequency domain filter coefficients
    w    : updated time domain filter coefficients
    x    : updated overlap-save input matrix
    y    : controller output block
    e    : new error block
    p    : updated estimate of power of x(n)
    yF   : updated output buffer
    feF  : updated filtered error buffer
```

Example

```
% This example simulates a MIMO control system with a single
% primary (reference) signal, two actuators and two sensors.
iter = 5000;                  % Number of samples to process
ph   = [0 .9 .5 .3 .1 ; 0 .8 .5 .2 .5]';
ph   = reshape(ph,5,1,2);     % Primary path impulse response
sh   = zeros(3,2,2);          % Secondary path impulse response
sh(:,1,1) = [0.5;0.4;0.1];
sh(:,2,2) = [0.5;0.4;0.1];
se   = 0.95*sh;               % estimation of sh
xa   = 2*(rand(iter,1)-0.5);  % Input signal, zero mean random.
da   = mcmixr(ph,xa,0);       % Primary response at the sensor
sens = zeros(iter,2);         % matrix for sensors signal
NC   = 10;                    % controller length
NL   = 6;                     % block length
mu   = 0.01/NC;               % step size for block processing
c    = 1;                     % constrain filter to NC coef.
b    = 0.98;                  % AR pole
% Initialize MCFDADJLMS algorithm
[NB,W,w,x,y,d,e,p,S,SE,yF,feF] = init_mcfdadjlms(NC,NL,1,2,2,sh,se);
%% Processing Loop
for (m=1:NL:iter-NL)
   xn = xa(m:m+NL-1,:);         % new input block of NL samples
   dn = da(m:m+NL-1,:);         % new desired block of NL samples
   % call asptmcfdadjlms to calculate the controller output
   % and update the coefficients.
   [W,w,x,y,e,p,yF,feF] = asptmcfdadjlms(NC,W,x,xn,dn,...
                          yF,feF,S,SE,p,mu,b,c);
   sens(m:m+NL-1,:) = e;       % save controlled sensors' signals
end;
% display the sensors' signals before and after the control effort
subplot(2,2,1); plot([da(:,1) sens(:,1)]); grid
subplot(2,2,2); plot([da(:,2) sens(:,2)]); grid
```

Running the above script will produce the graph shown in Fig. 7.12. In this figure, the signals recorded by the sensors before and after applying the MIMO control effort, *dn* and *sens*, respectively, are shown. The adaptive controller adjusts its coefficients to produce N_{act} control signals $y(n)$ that result in reducing the primary noise at the sensors.

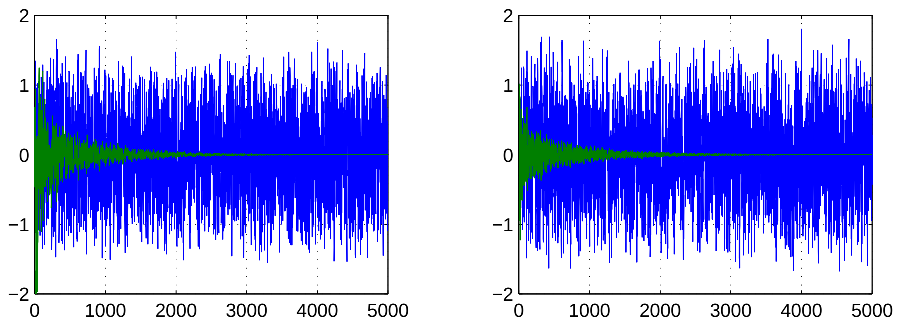

Figure 7.12: Signals recorded by the sensors before and after applying the adaptive controller in a Multichannel ANVC system using the multichannel frequency domain adjoint LMS algorithm.

Algorithm

The MIMO control problem addressed by ASPTMCFDADJLMS is to reduce the noise (or vibration) produced by Nref primary sources at the positions of Nsens sensors using a matrix of [Nref x Nact] controllers driving Nact actuators. To achieve this goal, ASPTMCFDADJLMS performs the following operations.

- composes the NB x Nref overlap-save input matrix $x(n)$ and computes its FFT, $x(f)$
- filters $x(f)$ through the frequency domain matrix of adaptive filters $W(f)$ in frequency domain to produce the Nact signals $y(f)$
- filters $y(f)$ through the matrix of secondary paths s in frequency domain to produce the response of the actuators at the sensors' positions $ys(n)$
- evaluates the current error $e(n) = d(n) + ys(n); n = 0, 1, \cdots, NL - 1$ at all sensors. Note the error here is formed by adding the signal rather than subtracting them to be compatible with real world sensors such as microphones and accelerometers. The error matrix is padded with zeros and transformed to frequency domain giving $e(f)$
- filters the frequency domain error matrix $e(f)$ through the estimate of the secondary path matrix se in frequency domain to produce the filtered-error signals $fe(f)$
- uses $x(f)$ and $fe(f)$ to calculate the normalized gradient vector and uses this to update the frequency domain adaptive filter coefficients $W(f)$. Normalization for both input signals and secondary paths are performed at each frequency bin which guarantees faster convergence rate than time domain MCADJLMS.
- computes the inverse FFT for the filter coefficients matrix, output vector, and error vector producing $w(n)$, $y(n)$, and $e(n)$ respectively.

Remarks

- Supports both real and complex signals
- The required resources to implement the MCFDADJLMS algorithm in real time are much less than those required for the MCFDFXLMS.
- Much more efficient than time domain processing (MCADJLMS) for long controllers.
- The Wiener solution to the above problem is given by $W(\omega) = S(\omega)^{-1}P(\omega)$, where $W(\omega)$ is the controller response at frequency ω, $S(\omega)$ is the response of the secondary path and $P(\omega)$ is the response of the primary path at the same frequency. The adaptive controller will asymptotically approach this Wiener solution provided that the inverse of $S(\omega)$ exists at each frequency and the controller length is large enough.
- The adaptive controller will approach the Wiener solution provided that the delay in the primary paths is larger than that in the secondary paths. This can be quickly checked by removing the leading zero in *ph* and adding a leading zero in *sh* in the above example.

Resources The resources required to implement the constrained MCFDADJLMS algorithm in real time is given in the table below. In this table, N_L is the block length and N_B is the FFT length given by $N_B = 2^{nextpow2(N_L+N_C-1)}$, and N_C is the controller length in time domain. The computations given are those required to process $N_L * N_{ref}$ input samples. Note that the unconstrained algorithm uses $2N_{act}N_{ref}$ FFT operations of length N_B less than the case shown in the table.

MEMORY	$N_B[N_{ref}(N_{act}+3)+N_{act}(N_{sens}+2)]+$ $N_L[2N_{sens}+N_{ref}+N_{act}]$
MULTIPLY	$N_B[3N_{ref}(Nact+1)+N_{act}N_{sens}]$
ADD	$N_B[2N_{ref}(Nact+1)+N_{sens}]$
DIVIDE	$N_BN_{act}N_{ref}$
FFT	$N_{act}[2N_{sens}+3]+N_{ref}+N_{sens}$

See Also INIT_ MCFDADJLMS, ANVC_ MCFDADJLMS, ASPTADJLMS, ASPTFDADJLMS, ASPTMCADJLMS.

Reference [3], Chapter 3 for detailed description of the MCFDADJLMS, [8] for the overlap-save method, and [9] for frequency domain adaptive filters.

7.7 asptmcfdfxlms

Purpose

Block filtering and coefficient update in frequency domain using the Multichannel Frequency Domain Filtered-x LMS (MCFDFXLMS) for multichannel active noise and vibration control applications.

Syntax

```
[W,w,x,y,e,p,yF,fxF] = asptmcfdfxlms(NC,W,x,xn,dn,yF,...
                       fxF,S,SE,p,mu,b,c)
```

Description

`asptmcfdfxlms()` is the frequency domain implementation of `asptmcfxlms()`. The difference between MCFDFXLMS and its time domain counterpart MCFXLMS is that filtering and coefficient update are performed in frequency domain using the overlap-save method. Fig. 7.13 shows the parameters of `asptmcfdfxlms()` which are summarized below.

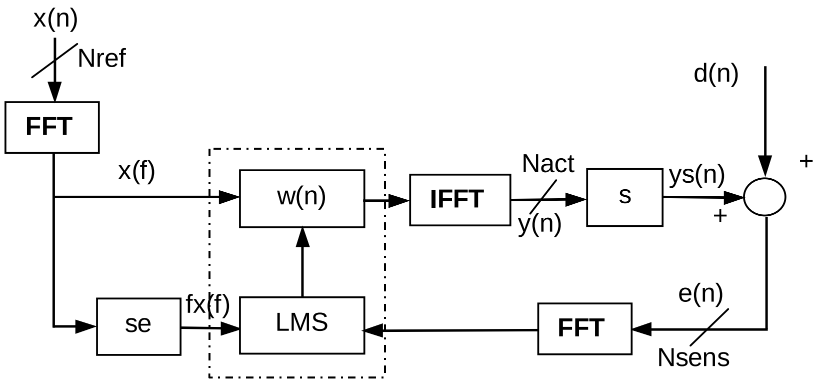

Figure 7.13: Block diagram of the Multichannel Frequency Domain Filtered-X LMS algorithm.

```
Input Parameters ::
    NC   : controller length in time domain
    W    : freq. domain filter coef. matrix [NB x Nref x Nact]
    x    : previous overlap-save input matrix [NB x Nref]
    xn   : new input samples block [NL x Nref]
    dn   : new primary samples block [NL x Nsens]
    yF   : previous buffer of y(n) [NB x Nact]
    fxF  : previous buffer of fx(n) [NB x Nact x Nsens*Nref]
    S    : FIR model of the secondary paths [NB x Nact x Nsens]
    SE   : estimated FIR model S [NB x Nact x Nsens]
    p    : last estimated power of fx(n) [NB x Nref x Nsens*Nref]
    mu   : adaptation constant
    b    : pole of AR filter used in estimating p
    c    : if not zero, uses the constrained BFDAF algorithm.
Output parameters ::
    W    : updated frequency domain filter coefficients
    w    : updated time domain filter coefficients
    x    : updated overlap-save input matrix
    y    : controller output block
    e    : new error block
    p    : updated estimate of power of fx(n)
    yF   : updated output buffer
    fxF  : updated filtered-x buffer
```

Example

```
% This example simulates a MIMO control system with a single
% primary (reference) signal, two actuators and two sensors.
iter = 5000;                    % Number of samples to process
ph   = [0 .9 .5 .3 .1 ; 0 .8 .5 .2 .5]';
ph   = reshape(ph,5,1,2);       % Primary path impulse response
sh   = zeros(3,2,2);            % Secondary path impulse response
sh(:,1,1) = [0.5;0.4;0.1];
sh(:,2,2) = [0.5;0.4;0.1];
se   = 0.95*sh;                 % estimation of sh
xa   = 2*(rand(iter,1)-0.5);    % Input signal, zero mean random.
da   = mcmixr(ph,xa,0);         % Primary response at the sensor
sens = zeros(iter,2);           % matrix for sensors signal
NC   = 10;                      % controller length
NL   = 6;                       % block length
mu   = 0.01/NC;                 % step size for block processing
c    = 1;                       % constrain filter to NC coef.
b    = 0.98;                    % AR pole
% Initialize MCFDFXLMS algorithm
[NB,W,w,x,y,d,e,p,S,SE,yF,fxF] = init_mcfdfxlms(NC,NL,1,2,2,sh,se);
%% Processing Loop
for (m=1:NL:iter-NL)
   xn = xa(m:m+NL-1,:);          % new input block of NL samples
   dn = da(m:m+NL-1,:);          % new desired block of NL samples
   % call asptmcfdfxlms to calculate the controller output
   % and update the coefficients.
   [W,w,x,y,e,p,yF,fxF] = asptmcfdfxlms(NC,W,x,xn,dn,...
                          yF,fxF,S,SE,p,mu,b,c);
   sens(m:m+NL-1,:) = e;         % save controlled sensors' signals
end;
% display the sensors' signals before and after the control effort
subplot(2,2,1); plot([da(:,1) sens(:,1)]); grid
subplot(2,2,2); plot([da(:,2) sens(:,2)]); grid
```

Running the above script will produce the graph shown in Fig. 7.14. In this figure, the signals recorded by the sensors before and after applying the MIMO control effort, *dn* and *sens*, respectively, are shown. The adaptive controller adjusts its coefficients to produce N_{act} control signals $y(n)$ that result in reducing the primary noise at the sensors.

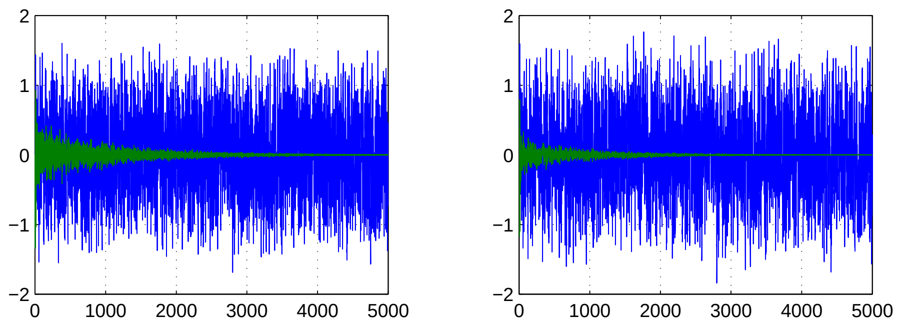

Figure 7.14: Signals recorded by the sensors before and after applying the adaptive controller in a Multichannel ANVC system using the multichannel frequency domain filtered-x LMS algorithm.

Algorithm

The MIMO control problem addressed by MCFDFXLMS is to reduce the noise (or vibration) produced by N_{ref} primary sources at the positions of N_{sens} sensors using a matrix of $[N_{ref} x N_{act}]$ controllers driving N_{act} actuators. To achieve this goal, `asptmcfdfxlms()` performs the following operations.

- composes the [NB x Nref] overlap-save input matrix $x(n)$ and computes its FFT, $x(f)$
- filters $x(f)$ through the frequency domain matrix of adaptive filters $W(f)$ in frequency domain to produce the Nact signals $y(f)$
- filters $y(f)$ through the matrix of secondary paths s in frequency domain to produce the response of the actuators at the sensors' positions $ys(n)$
- evaluates the current error $e(n) = d(n) + ys(n); n = 0, 1, \cdots, NL - 1$ at all sensors. Note the error here is formed by adding the signal rather than subtracting them to be compatible with real world sensors such as microphones and accelerometers. The error matrix is padded with zeros and transformed to frequency domain giving $e(f)$
- filters the frequency domain input matrix $x(f)$ through the estimate of the secondary path matrix se in frequency domain to produce the filtered-input signals $fx(f)$
- uses $fx(f)$ and $e(f)$ to calculate the normalized gradient vector and uses this to update the frequency domain adaptive filter coefficients $W(f)$. Normalization for both input signals and secondary paths are performed at each frequency bin which guarantees faster convergence rate than time domain MCFXLMS.
- computes the inverse FFT for the filter coefficients matrix, and output vector producing $w(n)$, and $y(n)$, respectively.

Remarks

- Supports both real and complex signals
- The required resources to implement the MCFDFXLMS algorithm in real time are usually much larger than those required for the MCFDADJLMS. This is evident from the size of the filtered-input compared to the size of the filtered-error matrixes.
- Much more efficient than time domain processing (MCFXLMS) for long controllers.
- The Wiener solution to the above problem is given by $W(\omega) = S(\omega)^{-1}P(\omega)$, where $W(\omega)$ is the controller response at frequency ω, $S(\omega)$ is the response of the secondary path and $P(\omega)$ is the response of the primary path at the same frequency. The adaptive controller will asymptotically approach this Wiener solution provided that the inverse of $S(\omega)$ exists at each frequency and the controller length is large enough.
- The adaptive controller will approach the Wiener solution provided that the delay in the primary paths is larger than that in the secondary paths. This can be quickly checked by removing the leading zero in *ph* and adding a leading zero in *sh* in the above example.

Resources The resources required to implement the constrained MCFDFXLMS algorithm in real time is given in the table below. In this table, N_L is the block length and N_B is the FFT length given by $N_B = 2^{nextpow2(N_L+N_C-1)}$, and N_C is the controller length in time domain. The computations given are those required to process $N_L * N_{ref}$ input samples. Note that the unconstrained algorithm uses $2N_{act}N_{ref}$ FFT operations of length N_B less than the case shown in the table.

MEMORY	$N_B N_{ref}[2N_{act}N_{sens} + N_{act} + 1)]+$ $N_B[N_{act}N_{sens} + N_{act} + N_{sens}] + N_L[2N_{sens} + N_{ref}] + 4$
MULTIPLY	$N_B N_{ref}(Nact)[5N_{sens} + 2]$
ADD	$N_B N_{ref}(Nact)[3N_{sens}] - N_B N_{act}$
DIVIDE	$N_B N_{act} N_{ref}$
FFT	$N_{act}[2N_{ref}(N_{sens} + 1) + 1] + N_{sens}$

See Also INIT_ MCFDFXLMS, ANVC_ MCFDFXLMS, ASPTFXLMS, ASPTFD-FXLMS, ASPTMCADJLMS.

Reference [3], Chapter 3 for detailed description of the MCFDFXLMS, [8] for the overlap-save method, and [9] for frequency domain adaptive filters.

7.8 asptmcfxlms

Purpose Sample per sample filtering and coefficient update using the Multichannel Filtered-X LMS (MCFXLMS) algorithm for multichannel active noise and vibration control applications.

Syntax `[w,y,e,p,fx] = asptmcfxlms(w,x,y,s,se,d,fx,p,mu,b)`

Description `asptmcfxlms()` implements the Multichannel Filtered-X LMS algorithm widely used in control applications where a transfer function (the matrix of secondary paths, s) exists between the output of the multi input multi output (MIMO) controller $w(n)$ and the error sensors (see Fig. 7.15). The consequence of this matrix of transfer functions is that (1) the phase response of the transfer functions delay each of the controller's outputs and makes it observable from each error signal after a delay, (2) the controller's outputs are colored by the amplitude response of the secondary paths. To correct for those effects, the MCFXLMS algorithm uses filtered version of the input signals to update the adaptive controller instead of directly using the input signals as shown in Fig. 7.15. The figure also shows the input and output parameters of `asptmcfxlms()` which are summarized below.

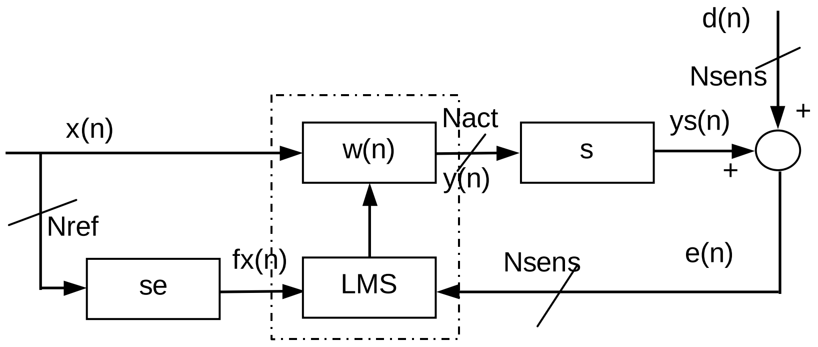

Figure 7.15: Block diagram of the Multichannel Adjoint-LMS algorithm.

```
Input Parameters [size]::
    w  : matrix of filter coefficients w(n-1), [L x Nref x Nact]
    x  : matrix of input samples x(n) [max(L,N) x Nref]
    y  : matrix of filter-outputs y(n-1) [M x Nact]
    s  : accurate matrix of secondary paths [M x Nact x Nsens]
    se : estimated matrix of secondary paths [N x Nact x Nsens]
    d  : desired response at sample index n [1 x Nsens]
    fx : filtered input signals fx(n-1) [L x Nact x Nsens*Nref]
    p  : last estimated power of x(n) [1 x Nref]
    mu : adaptation constant
    b  : pole of AR filter used to smooth p

Output parameters ::
    w  : updated filter coefficients w(n)
    y  : updated filter output vector y(n)
    e  : error vector e(n) = d(n) - ys(n) [1 x Nsens ]
    p  : updated estimate of input vector power
    fx : updated matrix of filtered-x samples fx(n)
```

Example

```
% This example simulates a MIMO control system with a single
% primary (reference) signal, two actuators and two sensors.
iter = 5000;                  % Number of samples to process
ph   = [0 .9 .5 .3 .1 ; 0 .8 .5 .2 .5]';
ph   = reshape(ph,5,1,2);     % Primary path impulse response
sh   = zeros(3,2,2);          % Secondary path impulse response
sh(:,1,1) = [0.5;0.4;0.1];
sh(:,2,2) = [0.5;0.4;0.1];
se   = 0.95*sh;               % estimation of sh
xn   = 2*(rand(iter,1)-0.5);  % Input signal, zero mean random.
dn   = mcmixr(ph,xn,0);       % Primary response at the sensor
sens = zeros(iter,2);         % matrix for sensors signal
% Initialize MCFXLMS algorithm with a controller of 10 coef.
[w,x,y,e,d,p,fx] = init_mcfxlms(10,1,2,2,sh,se);
%% Processing Loop
for (m=1:iter)
   % update the input delay line
   x = [xn(m,:); x(1:end-1,:)];
   % call asptmcfxlms to calculate the controller output
   % and update the coefficients. Below a step size of
   % 0.02 and an AR pole of 0.98 are used.
   [w,y,e,p,fx] = asptmcfxlms(w,x,y,sh,se,dn(m,:),fx,p,0.02,0.98);
   % save the last calculated sensor vector for
   %performance examination
   sens(m,:) = e(1,:);
end;
% display the sensor signal signal before and after
% applying the controller
subplot(2,2,1); plot([dn(:,1) sens(:,1)]); grid
subplot(2,2,2); plot([dn(:,2) sens(:,2)]); grid
```

Running the above script will produce the graph shown in Fig. 7.16. In this figure, the signals recorded by the sensors before and after applying the MIMO control effort, *dn* and *sens*, respectively, are shown. The adaptive controller adjusts its coefficients to produce N_{act} control signals $y(n)$ that result in reducing the primary noise at the sensors.

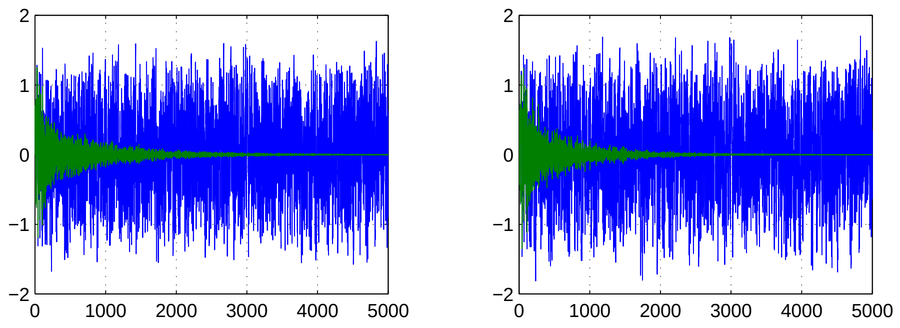

Figure 7.16: Signals recorded by the sensors before and after applying the adaptive controller in a Multichannel ANVC system using the multichannel filtered-x LMS algorithm.

Algorithm The MIMO control problem addressed by MCFXLMS is to reduce the noise (or vibration) produced by N_{ref} primary sources at the positions of N_{sens} sensors using a matrix of $[N_{ref} x N_{act}]$ controllers driving N_{act} actuators. To achieve this goal, `asptmcfxlms()` performs the following operations.

- filters the N_{ref} input (reference) signals $x(n)$ through the matrix of adaptive filters $w(n-1)$ to produce the N_{act} signals $y(n)$ used to drive the actuators
- filters $y(n)$ through the matrix of secondary paths s to produce the response of the actuators at the sensors' positions $ys(n)$
- evaluates the current error $e(n) = d(n) + ys(n)$ at all sensors. Note the error here is formed by adding the signal rather than subtracting them to be compatible with real world sensors such as microphones and accelerometers
- filters the input signals $x(n)$ through the estimate of the secondary path matrix se to produce the filtered-x signals $fx(n)$
- uses $fx(n)$ and $e(n)$ to calculate the normalized gradient vector and uses this to update the adaptive filter coefficients $w(n)$

Remarks

- Supports both real and complex signals
- The Wiener solution to the above problem is given by $W(\omega) = S(\omega)^{-1}P(\omega)$, where $W(\omega)$ is the controller response at frequency ω, $S(\omega)$ is the response of the secondary path and $P(\omega)$ is the response of the primary path at the same frequency. The adaptive controller will asymptotically approach this Wiener solution provided that the inverse of $S(\omega)$ exists at each frequency and the controller length is large enough.
- The adaptive controller will approach the Wiener solution provided that the delay in the primary paths is larger than that in the secondary paths. This can be quickly checked by removing the leading zero in ph and adding a leading zero in sh in the above example.

Resources The resources required for real time implementation of the MCFXLMS algorithm having N_{ref} reference signals, N_{act} actuators, and N_{sens} sensors with each filter of L coefficients and estimated secondary path of N coefficients is given in the table below. The computations given are those required to produce N_{act} control signals.

MEMORY	$LN_{ref}[N_{act}(N_{sens}+1)+1] + N_{sens}[NN_{act}+2]$ $+N_{act} + N_{ref} + 3$
MULTIPLY	$N_{ref}N_{act}[(L+N)N_{sens}+L] + N_{ref}[LN_{act}+3]$
ADD	$(N_{act}N_{ref})[N_{sens}(L+N+1)+2L-1] - LN_{act}$
DIVIDE	$NrefN_{act}$

See Also INIT_ MCFXLMS, ANVC_ MCFXLMS, ASPTFXLMS, ASPTFDFXLMS, ASPTMCFDFXLMS.

Reference [3], Chapter 3.

7.9 init_ adjlms

Purpose Creates and initializes the variables required for the ADJOINT Least Mean Squares (ADJLMS) Adaptive Filter algorithm for use in single channel Active Noise and Vibration Control (ANVC) applications.

Syntax

```
[w,x,y,d,e,p] = init_adjlms(L,s,se)
[w,x,y,d,e,p] = init_adjlms(L,s,se,w0,x0,d0,y0,e0)
```

Description The variables of the ADJOINT LMS are shows in Fig. 7.17 and are summarized below. The length of each variable is given in square brackets, for instance [N x 1] means a column vector of length N.

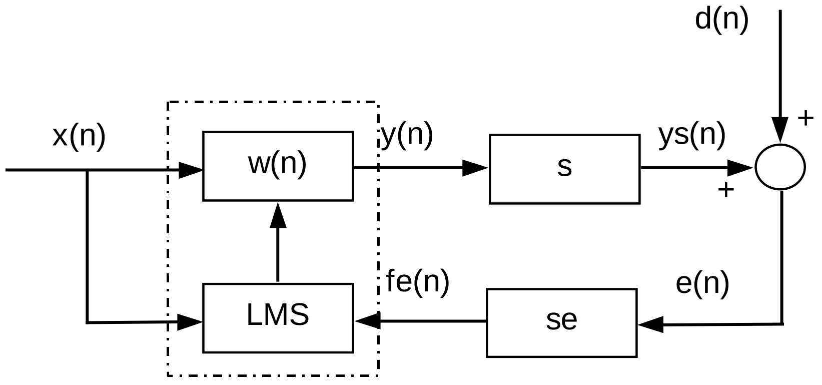

Figure 7.17: Block diagram of the Adjoint-LMS algorithm.

```
Input Parameters::
    L  : Adaptive filter length
    s  : FIR model of the physical secondary path [M x 1]
    se : estimated version of s [N x 1]
    w0 : initial vector of filter coefficients [L x 1]
    x0 : initial vector of input samples [L+M-1]
    d0 : initial desired sample [1 x 1]
    y0 : vector of filter output samples [M x 1]
    e0 : initial error vector [N x 1]

Output parameters [default]::
    w  : Initialized filter coefficients [zeros]
    x  : Initialized input vector [white noise]
    y  : Initial vector of filter output samples
    d  : Initialized desired sample [white noise]
    e  : Initialized error vector
    p  : Initialized input vector power
```

Example

```
ph   = [0;.9;.5;.3;.1];       % Primary path impulse response
sh   = [0.5;0.4;0.1];         % Secondary path impulse response
se   = 0.95*sh;               % estimation of s

% Initialize ADJLMS algorithm with a controller of 10
% coefficients and the accurate and estimated secondary paths
[w,x,y,d,e,p] = init_adjlms(10,sh,se);
```

Remarks

- Supports both real and complex signals
- The ADJOINT LMS algorithm requires an estimate of the secondary path, shown in Fig. 7.17 as se, to calculate the filtered error signal $fe(n)$. This estimate se is usually obtained in an initialization stage or during normal operation using a separate adaptive filter connected between the secondary source and the sensor in a system identification setup. When using $se = s$ in `init_adjlms()` and `asptadjlms()`, which means that a very accurate estimate of the secondary path is available, the effects of secondary path estimation error on the adaptive controller behavior will disappear. To examine those effects use an estimated secondary path se different than s.
- Use input parameters 4 through 8 to initialize the controller storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTADJLMS, ANVC_ ADJLMS, ASPTMCADJLMS, ASPTFDADJLMS, ASPTMCFDADJLMS.

7.10 init_ fdadjlms

Purpose Creates and initializes the variables required for the Frequency Domain AD-Joint Least Mean Squares (FDADJLMS) algorithm for use with single channel Active Noise and Vibration Control (ANVC) applications.

Syntax

```
[NB,W,w,x,y,d,e,p,S,SE,yF,feF]=init_fdadjlms(NC,NL,s,se)
[NB,W,w,x,y,d,e,p,S,SE,yF,feF]=init_fdadjlms(NC,NL,s,se,W0,xn0,d0)
```

Description The FDADJLMS is a block processing algorithm which performs filtering and coefficient update in the frequency domain. The variables of the FDADJLMS are shows in Fig. 7.18 and are summarized below. The size of each variable is given in a square brackets, for instance [N x 1] means a column vector of length N.

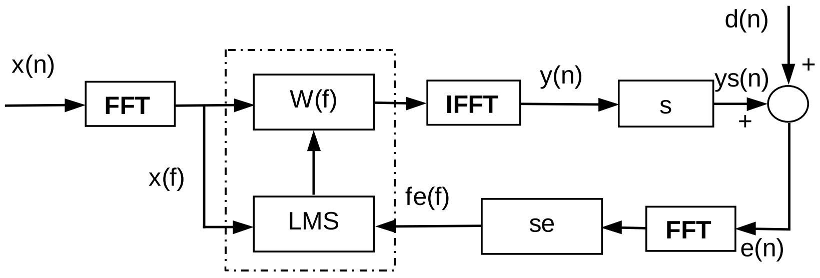

Figure 7.18: Block diagram of the Frequency Domain Adjoint-LMS algorithm.

```
Input Parameters [size] ::
    NC   : Required controller length in time domain
    NL   : new samples per block
    s    : time domain secondary path [M x 1]
    se   : estimate of the secondary path [N x 1]
    w0   : Initial vector of filter coefficients [NC x 1]
    xn0  : Initial input samples block [NL x 1]
    d0   : Initial primary signal block [NL x 1]

Output parameters [default] ::
    NB   : FFT length [2.^nextpow2(NC+NL-1)]
    W    : frequency domain filter coefficients vector [zeros]
    w    : time domain filter coefficients vector [zeros]
    x    : overlap-save input vector [zeros]
    y    : filter output vector in time domain [zeros]
    d    : primary signal block [zeros]
    e    : error signal block [e = d + ys]
    p    : power estimate of x(n) in freq. domain
    S    : frequency domain secondary path [fft(s,NB)]
    SE   : frequency domain estimated secondary path [fft(se,NB)]
    yF   : output vector buffer [zeros]
    feF  : filtered error buffer [zeros]
```

Example

```
ph   = [0;.9;.5;.3;.1];        % Primary path impulse response
sh   = [0.5;0.4;0.1];          % Secondary path impulse response
se   = 0.95*sh;                % estimation of s
NC   = 8 ;                     % Length of each filter
NL   = NC;                     % Block length

% Initialize FDADJLMS algorithm with a controller of NC
% coefficients and a block length NL.
[NB,W,w,x,y,d,e,p,S,SE,yF,feF] = init_fdadjlms(NC,NL,sh,se);
```

Remarks

- Supports both real and complex signals
- You can control the FFT length by choosing the block size (NL) appropriately. Maximum efficiency is achieved when $NL = NC = 2^n$; where n is an integer.
- Processing delay (algorithm latency) equals to NL, since NL new samples have to be collected before an FFT can be performed.
- The FDADJLMS algorithm requires an estimate of the secondary path, shown in Fig. 7.18 as se, to calculate the filtered error signal $fe(f)$. This estimate is usually obtained in an initialization stage or during normal operation using a separate adaptive filter connected between the actuator and the sensor in a Single Input Single Output (SISO) system identification setup. When using $se = s$ in `init_fdadjlms()` and `asptfdadjlms()`, which means that an accurate estimate of the secondary path is available, the effects of the secondary path estimation error on the adaptive controller behavior will disappear. To examine those effects use an estimated secondary path se different than s.
- Use input parameters 5 through 7 to initialize the controller storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTFDADJLMS, ANVC_ FDADJLMS, ASPTMCADJLMS, ASPTADJLMS, ASPTMCFDADJLMS.

7.11 init_ fdfxlms

Purpose Creates and initializes the variables required for the Frequency Domain Filtered-X Least Mean Squares (FDFXLMS) algorithm for use with single channel Active Noise and Vibration Control (ANVC) applications.

Syntax

```
[NB,W,w,x,y,d,e,p,S,SE,yF,fxF]=init_fdfxlms(NC,NL,s,se)
[NB,W,w,x,y,d,e,p,S,SE,yF,fxF]=init_fdfxlms(NC,NL,s,se,w0,xn0,d0)
```

Description The FDFXLMS is a block processing algorithm which performs filtering and coefficient update in the frequency domain.The variables of the FDFXLMS are shows in Fig. 7.19 and are summarized below. The size of each variable is given in a square brackets, for instance [N x 1] means a column vector of length N.

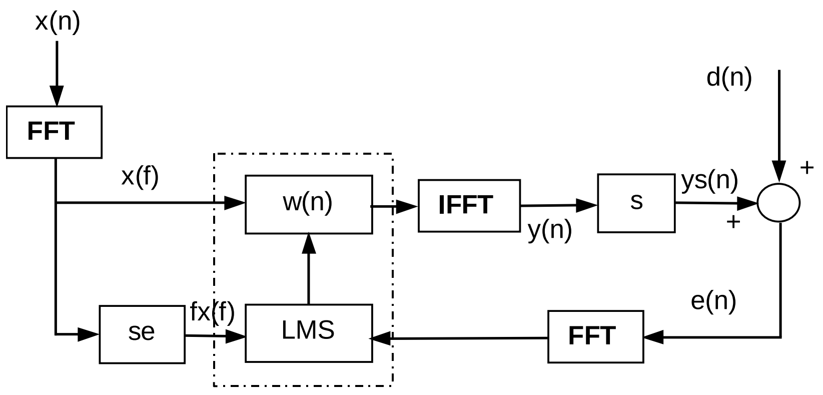

Figure 7.19: Block diagram of the Frequency Domain Filtered-X LMS algorithm.

```
Input Parameters [size] ::
    NC   : Required controller length in time domain
    NL   : new samples per block
    s    : FIR model of the secondary path in time domain [M x 1]
    se   : estimate of secondary path in time domain [N x 1]
    w0   : Initial vector of filter coefficients [NC x 1]
    xn0  : Initial input samples block [NL x 1]
    d0   : Initial primary signal block [NL x 1]

Output parameters [default] ::
    NB   : FFT length [2.^nextpow2(NC+NL-1)]
    W    : frequency domain filter coefficients vector [zeros]
    w    : time domain filter coefficients vector [zeros]
    x    : overlap-save input vector [zeros]
    y    : filter output vector in time domain [zeros]
    d    : primary signal block [zeros]
    e    : error signal block [e = d + ys]
    p    : power estimate of fx(f)
    S    : frequency domain secondary path [fft(s,NB)]
    SE   : frequency domain estimated s [fft(se,NB)]
    yF   : output vector buffer [zeros]
    fxF  : filtered-x buffer [zeros]
```

Example

```
ph   = [0;.9;.5;.3;.1];        % Primary path impulse response
sh   = [0.5;0.4;0.1];          % Secondary path impulse response
se   = 0.95*sh;                % estimation of s
NC   = 8 ;                     % Length of each filter
NL   = NC;                     % Block length

% Initialize FDFXLMS algorithm with a controller of NC
% coefficients and a block length NL.
[NB,W,w,x,y,d,e,p,S,SE,yF,fxF] = init_fdfxlms(NC,NL,sh,se)
```

Remarks

- Supports both real and complex signals
- You can control the FFT length by choosing the block size (NL) appropriately. Maximum efficiency is achieved when $NL = NC = 2^n$; where n is an integer.
- Processing delay (algorithm latency) equals to NL, since NL new samples have to be collected before an FFT can be performed.
- The FDFXLMS algorithm requires an estimate of the secondary path, shown in Fig. 7.19 as se, to calculate the filtered input signal $fx(f)$. This estimate is usually obtained in an initialization stage or during normal operation using a separate adaptive filter connected between the actuator and the sensor in a Single Input Single Output (SISO) system identification setup. When using $se = s$ in `init_fdfxlms()` and `asptfdfxlms()`, which means that an accurate estimate of the secondary path is available, the effects of the secondary path estimation error on the adaptive controller behavior will disappear. To examine those effects use an estimated secondary path se different than s.
- Use input parameters 5 through 7 to initialize the controller storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTFDFXLMS, ANVC_ FDFXLMS, ASPTMCFXLMS, ASPTFXLMS, ASPTMCFDFXLMS.

7.12 init_ fxlms

Purpose Creates and initializes the variables required for the Filtered-x Least Mean Squares (FXLMS) Adaptive Filter algorithm for use in single channel Active Noise and Vibration Control (ANVC) applications.

Syntax

```
[w,x,y,d,e,p,fx] = init_fxlms(L,s,se)
[w,x,y,d,e,p,fx] = init_fxlms(L,s,se,w0,x0,d0,y0)
```

Description The variables of the Filtered-x LMS are shows in Fig. 7.20 and are summarized below. The length of each variable is given in square brackets, for instance [N x 1] means a column vector of length N.

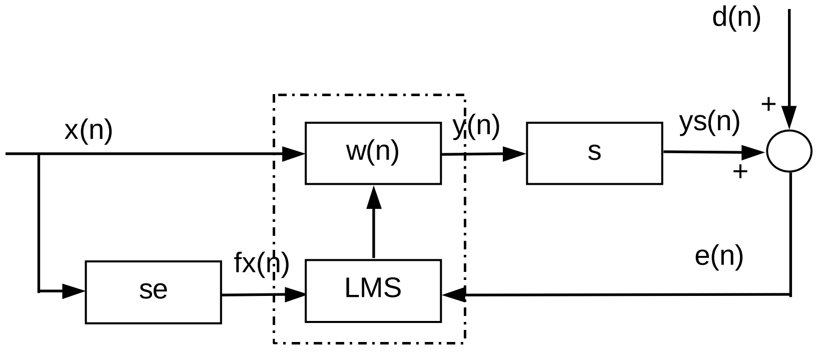

Figure 7.20: Block diagram of the Filtered-x LMS algorithm.

```
Input Parameters [size]::
    L  : Adaptive filter length
    s  : FIR model of the physical secondary path [M x 1]
    se : estimated version of s [N x 1]
    w0 : initial vector of filter coefficients [L x 1]
    x0 : initial vector of input samples [max(L,N) x 1]
    d0 : initial desired sample [1 x 1]
    y0 : vector of filter output samples [M x 1]

Output parameters [default]::
    w  : Initialized filter coefficients [zeros]
    x  : Initialized input delay line [zeros]
    d  : desired sample [white noise]
    y  : filter output samples delay line [zeros]
    e  : error sample
    p  : Initialized input vector power
    fx : filtered-x delay line
```

Example

```
ph   = [0;.9;.5;.3;.1];       % Primary path impulse response
sh   = [0.5;0.4;0.1];         % Secondary path impulse response
se   = 0.95*sh;               % estimation of s

% Initialize the Filtered-x algorithm with a controller of 10
% coefficients and the accurate and estimated secondary paths
[w,x,y,d,e,p] = init_fxlms(10,sh,se);
```

Remarks

- Supports both real and complex signals
- The Filtered-x LMS algorithm requires an estimate of the secondary path, shown in Fig. 7.20 as se, to calculate the filtered input signal $fx(n)$. This estimate se is usually obtained in an initialization stage or during normal operation using a separate adaptive filter connected between the secondary source and the sensor in a system identification setup. When using $se = s$ in `init_fxlms()` and `asptfxlms()`, which means that an accurate estimate of the secondary path is available, the effects of the secondary path estimation error on the adaptive controller behavior will disappear. To examine those effects use an estimated secondary path se different than s.
- Use input parameters 4 through 7 to initialize the controller storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTFXLMS, ANVC_ FXLMS, ASPTFDFXLMS, ASPTMCFDFXLMS.

7.13 init_ mcadjlms

Purpose Creates and initializes the variables required for the Multi-Channel ADJoint Least Mean Squares (MCADJLMS) adaptive filter algorithm for use with multichannel Active Noise and Vibration Control (ANVC) applications.

Syntax

```
[w,x,y,d,e,p]=init_mcadjlms(L,Nref,Nact,Nsens,s,se)
[w,x,y,d,e,p]=init_mcadjlms(L,Nref,Nact,Nsens,s,se,w0,x0,d0,y0,e0)
```

Description The variables of the MCADJLMS are shows in Fig. 7.21 and are summarized below. The size of each variable is given in a square brackets, for instance [N x Nref x Nact] means a matrix of dimension 3 having Nact pages, each page has Nref columns of length N each.

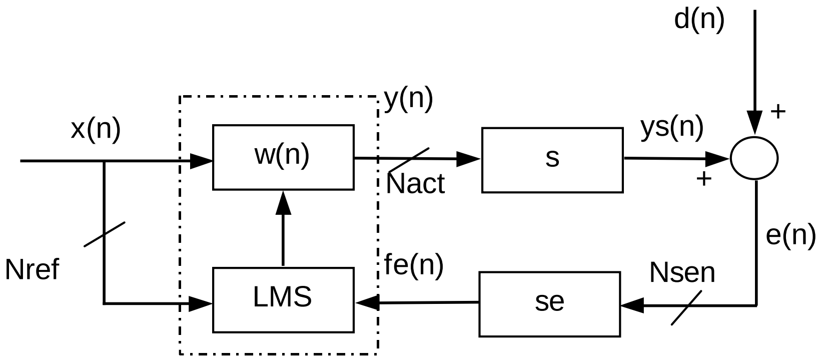

Figure 7.21: Block diagram of the Multichannel Adjoint-LMS algorithm.

```
Input Parameters::
    L     : Adaptive filter length
    Nref  : number of reference signals
    Nact  : number of actuators
    Nsens : number of sensors
    s     : FIR model of the secondary path [M x Nact x Nsens]
    se    : estimate of the secondary path [N x Nact x Nsens]
    w0    : initial vector of filter coef. [L x Nref x Nact]
    x0    : initial vector of input samples [L+N-1 x Nref]
    d0    : initial desired samples [1 x Nsens]
    y0    : vector of filter output samples [M x Nsens]
    e0    : initial error vector [N x Nsens]

Output parameters [default]::
    w     : Initialized filter coefficients [zeros]
    x     : Initialized input vector [white noise]
    y     : Initial vector of filter output samples
    d     : Initialized desired sample [white noise]
    e     : Initialized error vector
    p     : Initialized input vector variance
```

Example

```
load .\data\p22.mat;              % Primary transfer function
load .\data\s22.mat;              % Secondary transfer function
se                = s22;          % accurate estimate of sh22
[Lp,Nref,Nsens] = size(p22);     % Primary TF dimension
[Ls,Nact,Nsens] = size(s22);     % Secondary TF dimension
L                 = Lp + Ls ;     % Length of each filter

% Initialize MCADJLMS algorithm with controllers of L
% coefficients each.
[w,x,y,d,e,p]   = init_mcadjlms(L,Nref,Nact,Nsens,s22,se);
```

Remarks

- Supports both real and complex signals
- The MCADJLMS algorithm requires an estimate of the matrix of secondary paths, shown in Fig. 7.21 as *se*, to calculate the filtered error signals $fe(n)$. This estimate *se* is usually obtained in an initialization stage or during normal operation using a separate matrix of adaptive filters connected between the actuators and the sensors in a Multiple Input Multiple Output (MIMO) system identification setup. When using $se = s$ in `init_mcadjlms()` and `asptmcadjlms()`, which means that an accurate estimate of the secondary paths is available, the effects of the secondary path estimation error on the adaptive controller behavior will disappear. To examine those effects use an estimated secondary path *se* different than *s*.
- Use input parameters 7 through 11 to initialize the controller storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also

ASPTMCADJLMS, ANVC_ MCADJLMS, ASPTADJLMS, ASPTFDADJLMS, ASPTMCFDADJLMS.

7.14 init_ mcfdadjlms

Purpose Creates and initializes the variables required for the Multi-Channel Frequency Domain ADJoint Least Mean Squares (MCFDADJLMS) algorithm for use with multichannel Active Noise and Vibration Control (ANVC) applications.

Syntax

```
[NB,W,w,x,y,d,e,p,S,SE,yF,feF] = init_mcfdadjlms(NC,NL,Nref,Nact,
                                 Nsens,s,se)
[NB,W,w,x,y,d,e,p,S,SE,yF,feF] = init_mcfdadjlms(NC,NL,Nref,Nact,
                                 Nsens,s,se,W0,xn0,d0)
```

Description The MCFDADJLMS is a block processing algorithm which performs filtering and coefficient update in the frequency domain. The variables of the MCFDADJLMS are shows in Fig. 7.22 and are summarized below. The size of each variable is given in a square brackets, for instance [N x Nref x Nact] means a matrix of dimension 3 having Nact pages, each page has Nref columns of length N each.

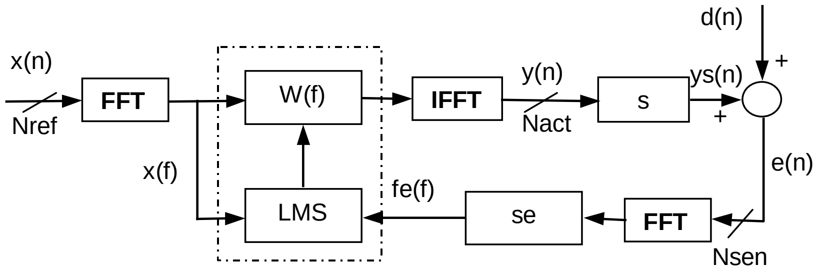

Figure 7.22: Block diagram of the Multichannel Frequency Domain Adjoint-LMS algorithm.

```
Input Parameters [size] ::
   NC    : Required controller length in time domain
   NL    : new samples per block
   Nref  : number of reference signals
   Nact  : number of actuators
   Nsens : number of sensors
   s     : FIR model of the secondary paths [M x Nact x Nsens]
   se    : estimate of secondary paths [N x Nact x Nsens]
   w0    : Initial matrix of filter coef. [NC x Nref x Nact]
   xn0   : Initial input samples block [NL x Nref]
   d0    : Initial primary signals block [NL x Nsens]
Output parameters [default] ::
   NB   : FFT length [2.^nextpow2(NC+NL-1)]
   W    : frequency domain filter coefficients matrix [zeros]
   w    : time domain filter coefficients matrix [zeros]
   x    : overlap-save input matrix [zeros]
   y    : filter output matrix in time domain [zeros]
   d    : primary signals block [zeros]
   e    : error signals block [e = d + ys]
   p    : power estimate of x(f)
   S    : frequency domain secondary paths [fft(s,NB,1)]
   SE   : frequency domain estimated s [fft(se,NB,1)]
   yF   : output matrix buffer [zeros]
   feF  : filtered error matrix buffer [zeros]
```

Example

```
load .\data\p22.mat;                 % Primary transfer function
load .\data\s22.mat;                 % Secondary transfer function
se                = s22;             % accurate estimate of sh22
[Lp,Nref,Nsens]   = size(p22);       % Primary TF dimension
[Ls,Nact,Nsens]   = size(s22);       % Secondary TF dimension
NC                = Lp + Ls ;        % Length of each filter
NL                = NC;              % Block length

% Initialize MCFDADJLMS algorithm with controllers of NC
% coefficients each and a block length NL.
[NB,W,w,x,y,d,e,p,S,SE,yF,feF] = init_mcfdadjlms(NC,NL,...
                                 Nref,Nact,Nsens,s22,se);
```

Remarks

- Supports both real and complex signals
- You can control the FFT length by choosing the block size (NL) appropriately. Maximum efficiency is achieved when $NL = NC = 2^n$; where n is an integer.
- Processing delay (algorithm latency) equals to NL, since NL new samples have to be collected before an FFT can be performed.
- The MCFDADJLMS algorithm requires an estimate of the matrix of secondary paths, shown in Fig. 7.22 as *se*, to calculate the filtered error signals $fe(f)$. This estimate *se* is usually obtained in an initialization stage or during normal operation using a separate matrix of adaptive filters connected between the actuators and the sensors in a Multiple Input Multiple Output (MIMO) system identification setup. When using $se = s$ in `init_mcfdadjlms()` and `asptmcfdadjlms()`, which means that an accurate estimate of the secondary paths is available, the effects of the secondary path estimation error on the adaptive controller behavior will disappear. To examine those effects use an estimated secondary path *se* different than *s*.
- Use input parameters 8 through 10 to initialize the controller storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also

ASPTMCFDADJLMS, ANVC_ MCFDADJLMS, ASPTMCADJLMS, ASPTADJLMS, ASPTFDADJLMS.

7.15 init_ mcfdfxlms

Purpose Creates and initializes the variables required for the Multi-Channel Frequency Domain Filtered-X Least Mean Squares (MCFDFXLMS) algorithm, for use with multichannel Active Noise and Vibration Control (ANVC) applications.

Syntax

```
[NB,W,w,x,y,d,e,p,S,SE,yF,fxF] = init_mcfdfxlms(NC,NL,Nref,Nact,
                                 Nsens,s,se)
[NB,W,w,x,y,d,e,p,S,SE,yF,fxF] = init_mcfdfxlms(NC,NL,Nref,Nact,
                                 Nsens,s,se,w0,xn0,d0)
```

Description The variables of the MCFDFXLMS are shows in Fig. 7.23 and are summarized below. The size of each variable is given in a square brackets.

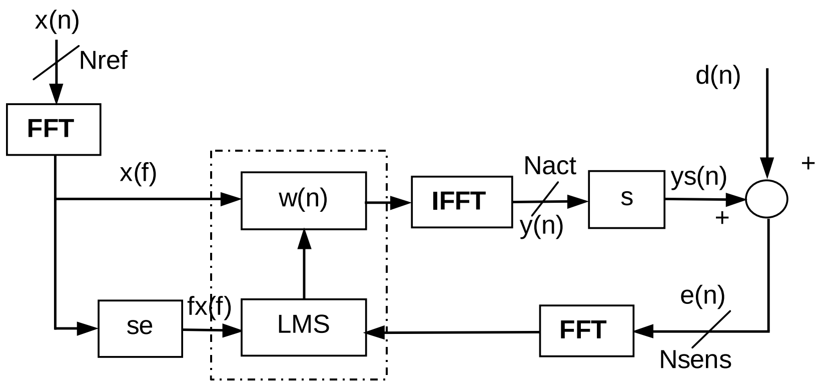

Figure 7.23: Block diagram of the MultChannel Frequency Domain Filtered-X LMS algorithm.

```
Input Parameters [size] ::
    NC    : Required controller length in time domain
    NL    : new samples per block
    Nref  : number of reference signals
    Nact  : number of actuators
    Nsens : number of sensors
    s     : FIR model of the secondary paths [M x Nact x Nsens]
    se    : estimate of secondary paths [N x Nact x Nsens]
    w0    : Initial controller coef. matrix [NC x Nref x Nact]
    xn0   : Initial input samples block [NL x Nref]
    d0    : Initial primary signals block [NL x Nsens]
Output parameters [default] ::
    NB   : FFT length [2.^nextpow2(NC+NL-1)]
    W    : frequency domain filter coefficients matrix [zeros]
    w    : time domain filter coefficients matrix [zeros]
    x    : overlap-save input matrix [zeros]
    y    : filter output matrix in time domain [zeros]
    d    : primary signals block [zeros]
    e    : error signals block [e = d + ys]
    p    : power estimate of fx(n) in freq. domain
    S    : frequency domain secondary paths [fft(s,NB)]
    SE   : frequency domain estimated s [fft(se,NB)]
    yF   : output matrix buffer [zeros]
    fxF  : filtered-x matrix buffer [zeros]
```

Example

```
load .\data\p22.mat;                % Primary transfer function
load .\data\s22.mat;                % Secondary transfer function
se                = s22;            % accurate estimate of sh22
[Lp,Nref,Nsens]   = size(p22);      % Primary TF dimension
[Ls,Nact,Nsens]   = size(s22);      % Secondary TF dimension
NC                = Lp + Ls ;       % Length of each filter
NL                = NC;             % block length

% Initialize MCFDFXLMS algorithm with controllers of NC
% coefficients each and a block length NL.
[NB,W,w,x,y,d,e,p,S,SE,yF,fxF] = init_mcfdfxlms(NC,NL,Nref,
                                  Nact,Nsens,s22,se);
```

Remarks

- Supports both real and complex signals
- You can control the FFT length by choosing the block size (NL) appropriately. Maximum efficiency is achieved when $NL = NC = 2^n$; where n is an integer.
- Processing delay (algorithm latency) equals to NL, since NL new samples have to be collected before an FFT can be performed.
- The MCFDFXLMS algorithm requires an estimate of the matrix of secondary paths, shown in Fig. 7.23 as *se*, to calculate the filtered input signals $fx(f)$. This estimate *se* is usually obtained in an initialization stage or during normal operation using a separate matrix of adaptive filters connected between the actuators and the sensors in a Multiple Input Multiple Output (MIMO) system identification setup. When using $se = s$ in `init_mcfdfxlms()` and `asptmcfdfxlms()`, which means that an accurate estimate of the secondary paths is available, the effects of the secondary path estimation error on the adaptive controller behavior will disappear. To examine those effects use an estimated secondary path *se* different than *s*.
- Use input parameters 8 through 10 to initialize the controller storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTMCFDFXLMS, ANVC_ MCFDFXLMS, ASPTFXLMS, ASPTFDFXLMS, ASPTMCFXLMS.

7.16 init_ mcfxlms

Purpose Creates and initializes the variables required for the Multi-Channel Filtered-X Least Mean Squares (MCFXLMS) adaptive algorithm, also known as Multiple Error Filtered-X LMS (MEFXLMS) for use with multichannel Active Noise and Vibration Control (ANVC) applications.

Syntax

```
[w,x,y,e,d,p,fx] = init_mcfxlms(L,Nref,Nact,Nsens,s,se)
[w,x,y,e,d,p,fx] = init_mcfxlms(L,Nref,Nact,Nsens,s,se,
                                w0,x0,d0,y0,fx0)
```

Description The variables of the MCFXLMS are shows in Fig. 7.24 and are summarized below. The size of each variable is given in a square brackets, for instance [N x Nref x Nact] means a matrix of dimension 3 having Nact pages, each page has Nref columns of length N each.

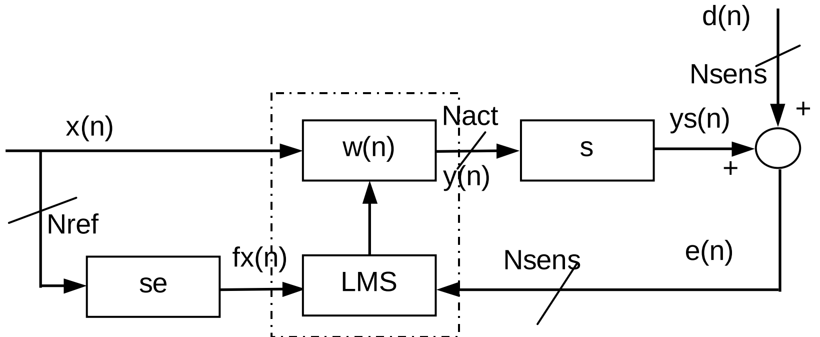

Figure 7.24: Block diagram of the MultChannel Filtered-X LMS algorithm.

```
Input Parameters [size] ::
    L     : Adaptive filter length
    Nref  : number of reference signals
    Nact  : number of actuators
    Nsens : number of sensors
    s     : FIR model of the secondary path [M x Nact x Nsens]
    se    : estimate of the secondary path [N x Nact x Nsens]
    w0    : initial vector of filter coefficients [L x Nref x Nact]
    x0    : initial vector of input samples [max(L,N) x Nref]
    d0    : initial desired samples [1 x Nsens]
    y0    : vector of filter output samples [M x Nsens]
    fx0   : initial filtered input matrix [L x Nact x Nref*Nsens]

Output parameters [default]::
    w     : Initialized filter coefficients [zeros]
    x     : Initialized matrix of input samples [white noise]
    y     : Initial matrix of filter output samples
    d     : Initialized desired samples [white noise]
    e     : Initialized error vector [e=d+y]
    p     : Initialized power of x
    fx    : Initialized filtered input matrix [zeros]
```

Example

```
load .\data\p22.mat;                % Primary transfer function
load .\data\s22.mat;                % Secondary transfer function
se                 = s22;           % accurate estimate of sh22
[Lp,Nref,Nsens]    = size(p22);     % Primary TF dimension
[Ls,Nact,Nsens]    = size(s22);     % Secondary TF dimension
L                  = Lp + Ls ;      % Length of each filter

% Initialize MCFXLMS algorithm with controllers of L
% coefficients each.
[w,x,y,e,d,p,fx] = init_mcfxlms(L,Nref,Nact,Nsens,s22,se);
```

Remarks

- Supports both real and complex signals
- The MCFXLMS algorithm requires an estimate of the matrix of secondary paths, shown in Fig. 7.24 as se, to calculate the filtered input signals $fx(n)$. This estimate se is usually obtained in an initialization stage or during normal operation using a separate matrix of adaptive filter connected between the actuators and the sensors in a Multiple Input Multiple Output (MIMO) system identification setup. When using $se = s$ in `init_mcadjlms()` and `asptmcadjlms()`, which means that an accurate estimate of the secondary paths is available, the effects of the secondary path estimation error on the adaptive controller behavior will disappear. To examine those effects use an estimated secondary path se different than s.
- Use input parameters 7 through 11 to initialize the controller storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTMCFXLMS, ANVC_ MCFXLMS, ASPTFXLMS, ASPTFDFXLMS, ASPTMCFDFXLMS.

Chapter 8

Nonlinear Adaptive Algorithms

This chapter documents the functions used to create, initialize, and update the coefficients of nonlinear adaptive filters (Section 2.2.5). Table 8.1 summarizes the nonlinear adaptive functions currently supported and gives a short description and a pointer to the reference page of each function.

Function Name	Reference	Short Description
asptsovlms	8.1	Second Order Volterra LMS and several of its variants.
asptsovnlms	8.2	Second Order Volterra Normalized LMS algorithm.
asptsovrls	8.3	Second Order Volterra RLS algorithm.
asptsovtdlms	8.4	Second Order Volterra Transform domain LMS algorithm.
asptsovvsslms	8.5	Second Order Volterra Variable Step Size LMS algorithm.
init_ sovlms	8.6	Initialize Second Order Volterra LMS.
init_ sovnlms	8.7	Initialize Second Order Volterra NLMS.
init_ sovrls	8.8	Initialize Second Order Volterra RLS.
init_ sovtdlms	8.9	Initialize Second Order Volterra Transform domain LMS.
init_ sovvsslms	8.10	Initialize Second Order Volterra Variable Step Size LMS.

Table 8.1: Functions for creating, initializing, and updating nonlinear adaptive filters.

Each function is documented in a separate section including the following information related to the function:

- **Purpose:** Short description of the algorithm implemented by this function.
- **Syntax:** Shows the function calling syntax. If the function has optional parameters, this section will have two calling syntaxes. One with only the required formal parameters and one with all the formal parameters.
- **Description:** Detailed description of the function usage with explanation of its input and output parameters.
- **Example:** A short example showing typical use of the function. The examples listed can be found in the ASPT/test directory of the ASPT distribution. The user is encouraged to copy from those examples and paste in her own applications.
- **Algorithm:** A short description of the operations internally performed by the function.
- **Remarks:** Gives more theoretical and practical remarks related to the usage, performance, limitations, and applications of the function.
- **Resources:** Gives a summary of the memory requirements and number of multiplications, addition/subtractions, and division operations required to implement the function in real time. This can be used to roughly calculate the MIPS (Million Instruction Per Second)

required for a specific platform knowing the number of instructions the processor needs to perform each operation.

- **See Also:** Lists other functions that are related to this function.
- **Reference:** Lists literature for more information on the function.

8.1 asptsovlms

Purpose Sample per sample filtering and coefficient update using the Second Order Volterra Least Mean Squares or one of its variants. The LMS variants currently supported are the sign, sign-sign, and signed regressor algorithms.

Syntax

```
[w,y,e,xb] = asptsovlms(xn,xb,w,d,mu,L1,L2)
[w,y,e,xb] = asptsovlms(xn,xb,w,d,mu,L1,L2,alg)
```

Description `asptsovlms()` implements the second order Volterra LMS filter (SOVLMS). The SOVLMS algorithm calculates the filter output $y(n)$ and updates the filter coefficients vector $\underline{\mathbf{w}}(n)$. The filter output is the sum of the outputs of the linear filter part $\underline{\mathbf{w}}_l(n)$ and the nonlinear part $\underline{\mathbf{w}}_n(n)$ as follows.

$$y(n) = \sum_{m_1=0}^{L1-1} w_l(m1)x(n-m_1) + \sum_{m_1=0}^{L1-1} \Sigma_{m_2=m_1}^{L2-1} w_n(m1,m2)x(n-m_1)x(n-m_2) \tag{8.1}$$

The filter coefficients vector $\underline{\mathbf{w}}(n) = [\underline{\mathbf{w}}_l(n); \underline{\mathbf{w}}_n(n)]$ is updated according to the LMS variety given by the 'alg' input parameter. The memory depth of the linear and nonlinear filter parts are governed independently by the $L1$ and $L2$ parameters passed to `init_sovlms()`.
The input and output parameters of `asptsovlms()` for an FIR adaptive filter of linear memory length $L1$ and nonlinear memory length $L2$ are summarized below.

```
Input Parameters [size] ::
  xn  : new input sample [1 x 1]
  xb  : buffer of input samples [L1 + sum(1:L2) x 1]
  w   : vector of filter coefficients w(n-1) [L1 + sum(1:L2) x 1]
  d   : desired output d(n) [1 x 1]
  mu  : adaptation constant [2 x 1]
  L1  : memory length of linear part of w
  L2  : memory length of non-linear part of w
  alg : specifies the variety of the lms to use in the
         update equation. Must be one of the following:
        'lms'     [default]
        'slms'  - sign LMS, uses sign(e)
        'srlms' - signed regressor LMS, uses sign(x)
        'sslms' - sign-sign LMS, uses sign(e) and sign(x)
Output parameters ::
  w   : updated filter coefficients w(n)
  y   : filter output y(n)
  e   : error signal; e(n) = d(n) - y(n)
  xb  : updated vector of input samples
```

Example

```
% SOVLMS used in a simple system identification application.
% By the end of this script the adaptive filter w should have
% the same coefficients as the nonlinear unknown filter h.
iter = 5000;         % Number of samples to process
L1   = 4;
L2   = 4;
% The nonlinear unknown plant transfer function
% y(n) =  x(n) - x(n-1) - .125x(n-2) + .3125x(n-3)
%        +x(n)x(n) -.3x(n)x(n-1) + .2x(n)x(n-2) -.5x(n)x(n-3)
%        +.5x(n-1)x(n-1) -.3x(n-1)x(n-2) -.6x(n-1)x(n-3)
%        -.6x(n-2)x(n-2) +.5x(n-2)x(n-3) -.1x(n-3)x(n-3)
h  =[1;-1;-0;.125;.3125;1;-.3;.2;-.5;.5;-.3;-.6;-.6;.5;-.1];
xn =2*(rand(iter,1)-0.5);      % Input signal, zero mean random.
dn =sovfilt(h,xn,4,4);         % Unknown filter output
en =zeros(iter,1);             % vector to collect the error
[w,xb,d,y,e]=init_sovlms(L1,L2); % Initialize SOVLMS

%% Processing Loop
for (m=1:iter)
   d = dn(m,:) + .001 * rand ;   % additive noise of var = 1e-6
   % call SOVLMS to calculate the filter output, estimation error
   % and update the coefficients.
   [w,y,e,xb]= asptsovlms(xn(m),xb,w,d,[.05,.05],L1,L2,'lms');
   en(m,:) = e;                  % save the last error sample
end;

% display the results
% note that w converges to conj(h) for complex data
subplot(2,2,1);stem([real(w) imag(conj(w))]); grid;
subplot(2,2,2);eb = filter(.1,[1 -.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 8.1. The left side graph of the figure shows the adaptive filter coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the square error in dB versus time during the adaptation process. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB).

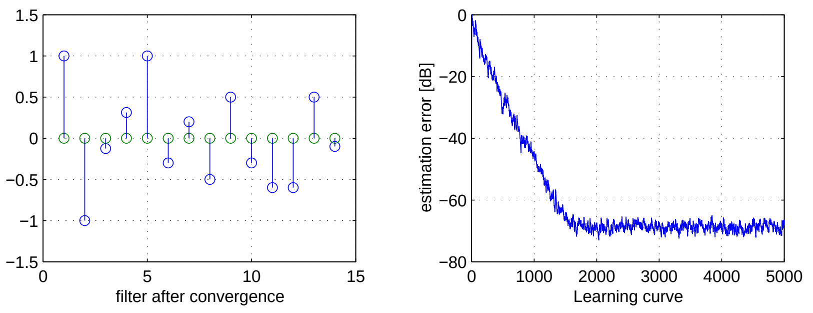

Figure 8.1: The adaptive filter coefficients after convergence and the learning curve for the FIR system identification problem using the SOVLMS algorithm.

Algorithm

The current implementation of `asptsovlms()` performs the following operations

- Constructs the new input samples delay line elements from the previous delay line and new input sample.
- Filters the input delay line $xb(n)$ through the adaptive filter $w(n-1)$ to produce the filter output $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$.
- Updates the adaptive filter coefficients using the error $e(n)$ and the delay line of input samples $xb(n)$ resulting in $w(n)$.

Remarks

SOVLMS uses the regular LMS algorithm to update both the linear ($\underline{\mathbf{w}}_l$) and nonlinear ($\underline{\mathbf{w}}_n$) parts of the adaptive filter. Therefore, the convergence properties of the SOVLMS are similar to those of the LMS. For more control on the convergence speed, the step size is a 2-element vector, the first element of which is used to update the linear filter part and the second is used to update the nonlinear part. `asptsovlms()` also,

- supports both real and complex data and filters. The adaptive filter for the complex SOVLMS algorithm converges to the complex conjugate of the optimum solution.
- internally updates the input delay line $xb(n)$ which includes the past linear and nonlinear samples needed for the next iteration. The past samples kept depend on the L1 and L2 parameters.

Resources

The resources required to implement a SOVLMS filter of linear memory length $L1$ and nonlinear memory length $L2$ are given below. The computations given are those required to process one sample.

MEMORY	$2(L1 + sum(1:L2) + 6$
MULTIPLY	$2(L1 + sum(1:L2) + L2 + 2$
ADD	$2(L1 + sum(1:L2)$
DIVIDE	0

See Also

INIT_ SOVLMS, ASPTSOVNLMS, ASPTLMS.

Reference

[11] and [4] for extensive analysis of the LMS and the steepest-descent search method and [7] for an introduction to the adaptive Volterra filters.

8.2 asptsovnlms

Purpose Sample per sample filtering and coefficients update using the Second Order Volterra Normalized Least Mean Squares Adaptive Filter algorithm.

Syntax

```
[w,y,e,xb,p] = asptsovnlms(xn,xb,w,d,mu,L1,L2,p)
[w,y,e,xb,p] = asptsovnlms(xn,xb,w,d,mu,L1,L2,p,b)
```

Description `asptsovnlms()` implements the second order Volterra NLMS filter (SOVNLMS). The SOVNLMS algorithm calculates the filter output $y(n)$ and updates the filter coefficients vector $\underline{\mathbf{w}}(n)$. The filter output is the sum of the outputs of the linear filter part $\underline{\mathbf{w}}_l(n)$ and the nonlinear part $\underline{\mathbf{w}}_n(n)$ as follows.

$$y(n) = \sum_{m_1=0}^{L1-1} w_l(m1)x(n-m_1) + \sum_{m_1=0}^{L1-1} \Sigma_{m_2=m_1}^{L2-1} w_n(m1,m2)x(n-m_1)x(n-m_2) \tag{8.2}$$

The filter coefficients vector $\underline{\mathbf{w}}(n) = [\underline{\mathbf{w}}_l(n); \underline{\mathbf{w}}_n(n)]$ is updated according to the Normalized LMS algorithm. The memory depth of the linear and nonlinear filter parts are governed independently by the $L1$ and $L2$ parameters passed to `init_sovnlms()`.
The input and output parameters of `asptsovnlms()` for an FIR adaptive filter of linear memory length $L1$ and nonlinear memory length $L2$ are summarized below.

```
Input Parameters [size] ::
  xn : new input sample [1 x 1]
  xb : buffer of input samples [L1 + sum(1:L2) x 1]
  w  : vector of filter coefficients w(n-1) [L1 + sum(1:L2) x 1]
  d  : desired output d(n) [1 x 1]
  mu : adaptation constant [2 x 1]
  L1 : memory length of linear part of w
  L2 : memory length of non-linear part of w
  p  : input signal power [2 x 1]
  b  : low pass filter pole used to estimate p.

Output parameters ::
  w  : updated filter coefficients w(n)
  y  : filter output y(n)
  e  : error signal; e(n) = d(n) - y(n)
  xb : updated vector of input samples
  p  : updated input signal power.
```

Example

```
% SOVNLMS used in a simple system identification application.
% By the end of this script the adaptive filter w should have
% the same coefficients as the nonlinear unknown filter h.
iter = 5000;         % Number of samples to process
L1   = 4;
L2   = 4;
% The nonlinear unknown plant transfer function
% y(n) =  x(n) - x(n-1) - .125x(n-2) + .3125x(n-3)
%        +x(n)x(n) -.3x(n)x(n-1) + .2x(n)x(n-2) -.5x(n)x(n-3)
%        +.5x(n-1)x(n-1) -.3x(n-1)x(n-2) -.6x(n-1)x(n-3)
%        -.6x(n-2)x(n-2) +.5x(n-2)x(n-3) -.1x(n-3)x(n-3)
h  =[1;-1;-0;.125;.3125;1;-.3;.2;-.5;.5;-.3;-.6;-.6;.5;-.1];
xn =2*(rand(iter,1)-0.5);      % Input signal, zero mean random.
dn =sovfilt(h,xn,4,4);         % Unknown filter output
en =zeros(iter,1);             % vector to collect the error
[w,xb,d,y,e,p]=init_sovnlms(L1,L2); % Initialize SOVNLMS

%% Processing Loop
for (m=1:iter)
   d = dn(m,:) + .001 * rand ;   % additive noise of var = 1e-6
   % call SOVNLMS to calculate the filter output, estimation error
   % and update the coefficients.
   [w,y,e,xb,p]= asptsovnlms(xn(m),xb,w,d,[.05,.05],L1,L2,p,.98);
   en(m,:) = e;                  % save the last error sample
end;

% display the results
% note that w converges to conj(h) for complex data
subplot(2,2,1);stem([real(w) imag(conj(w))]); grid;
subplot(2,2,2);eb = filter(.1,[1 -.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 8.2. The left side graph of the figure shows the adaptive filter coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the square error in dB versus time during the adaptation process. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB).

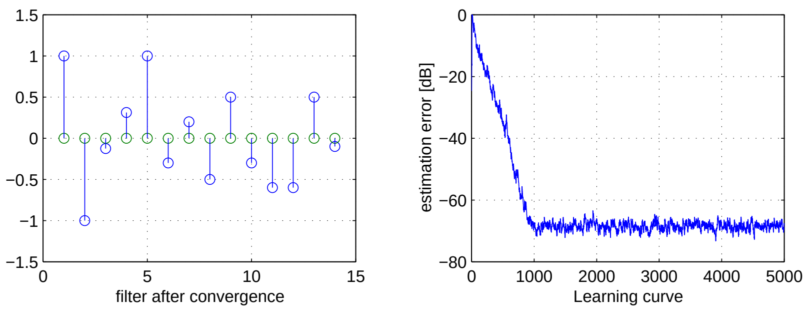

Figure 8.2: The adaptive filter coefficients after convergence and the learning curve for the FIR system identification problem using the SOVNLMS algorithm.

Algorithm

The current implementation of `asptsovnlms()` performs the following operations

- Constructs the new input samples delay line elements from the previous delay line and new input sample.
- Filters the input delay line $xb(n)$ through the adaptive filter $w(n-1)$ to produce the filter output $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$.
- Estimates the power of the input signal and its cross-products to be used in normalizing the update of the linear and nonlinear filter parts, respectively.
- Updates the adaptive filter coefficients using the error $e(n)$ and the delay line of input samples $xb(n)$ resulting in $w(n)$.

Remarks

SOVNLMS uses the regular NLMS algorithm to update both the linear ($\underline{\mathbf{w}}_l$) and nonlinear ($\underline{\mathbf{w}}_n$) parts of the adaptive filter. Therefore, the convergence properties of the SOVNLMS are similar to those of the NLMS. For more control on the convergence speed, the step size is a 2-element vector, the first element of which is used to update the linear filter part and the second is used to update the nonlinear part. A vector of two elements is also used for the power normalization. `asptsovnlms()` also,

- supports both real and complex data and filters. The adaptive filter for the complex SOVNLMS algorithm converges to the complex conjugate of the optimum solution.
- internally updates the input delay line $xb(n)$ which includes the past linear and nonlinear samples needed for the next iteration. The past samples kept depend on the L1 and L2 parameters.

Resources

The resources required to implement a SOVNLMS filter of linear memory length $L1$ and nonlinear memory length $L2$ are given below. The computations given are those required to process one sample.

MEMORY	$2(L1 + sum(1 : L2) + 9$
MULTIPLY	$2(L1 + sum(1 : L2) + 2L2 + 5$
ADD	$2(L1 + sum(1 : L2) + L2 + 1$
DIVIDE	2

See Also

INIT_ SOVNLMS, ASPTSOVLMS, ASPTNLMS.

Reference

[11] and [4] for extensive analysis of the LMS and the steepest-descent search method and [7] for an introduction to the adaptive Volterra filters.

8.3 asptsovrls

Purpose Sample per sample filtering and coefficients update using the Second Order Volterra Recursive Least Squares (SOVRLS) adaptive algorithm.

Syntax

```
[w,y,e,R,xb] = asptsovrls(xn,xb,w,d,R,a,L1,L2)
```

Description `asptsovrls()` implements the second order Volterra RLS filter (SOVRLS). The SOVRLS algorithm calculates the filter output $y(n)$ and updates the filter coefficients vector $\underline{\mathbf{w}}(n)$. The filter output is the sum of the outputs of the linear filter part $\underline{\mathbf{w}}_l(n)$ and the nonlinear part $\underline{\mathbf{w}}_n(n)$ as follows.

$$y(n) = \sum_{m_1=0}^{L1-1} w_l(m1)x(n-m_1) + \sum_{m_1=0}^{L1-1} \Sigma_{m_2=m_1}^{L2-1} w_n(m1,m2)x(n-m_1)x(n-m_2) \tag{8.3}$$

The filter coefficients vector $\underline{\mathbf{w}}(n) = [\underline{\mathbf{w}}_l(n); \underline{\mathbf{w}}_n(n)]$ is updated according to the recursive least squares algorithm equation (4.10). The memory depth of the linear and nonlinear filter parts are governed independently by the $L1$ and $L2$ parameters passed to `init_sovrls()`.
The input and output parameters of `asptsovrls()` for an FIR adaptive filter of linear memory length $L1$ and nonlinear memory length $L2$ are summarized below.

```
Input Parameters [Size]::
  xn : new input sample
  xb : buffer of input samples [L1 + sum(1:L2) x 1]
  w  : vector of filter coefficients w(n-1), [L x 1]
  d  : desired response d(n), [1 x 1]
  R  : last estimate of the inverse of the weighted
       auto correlation matrix of x, [L x L]
  a  : forgetting factor, [1 x 1]
  L1 : memory length of linear part of w
  L2 : memory length of non-linear part of w

Output parameters::
  w  : updated filter coefficients w(n)
  y  : filter output y(n)
  e  : error signal, e(n)=d(n)-y(n)
  R  : updated R
  xb : updated buffer of input samples
```

Example

```
% SOVRLS used in a simple system identification application.
% By the end of this script the adaptive filter w should have
% the same coefficients as the nonlinear unknown filter h.
iter = 5000;         % Number of samples to process
L1   = 4;
L2   = 4;
% The nonlinear unknown plant transfer function
% y(n) =  x(n) - x(n-1) - .125x(n-2) + .3125x(n-3)
%        +x(n)x(n) -.3x(n)x(n-1) + .2x(n)x(n-2) -.5x(n)x(n-3)
%        +.5x(n-1)x(n-1) -.3x(n-1)x(n-2) -.6x(n-1)x(n-3)
%        -.6x(n-2)x(n-2) +.5x(n-2)x(n-3) -.1x(n-3)x(n-3)
h  =[1;-1;-0;.125;.3125;1;-.3;.2;-.5;.5;-.3;-.6;-.6;.5;-.1];
xn =2*(rand(iter,1)-0.5);      % Input signal, zero mean random.
dn =sovfilt(h,xn,4,4);         % Unknown filter output
en =zeros(iter,1);             % vector to collect the error
[w,xb,d,y,e,R]=init_sovrls(L1,L2,.001); % Initialize SOVRLS

%% Processing Loop
for (m=1:iter)
   d = dn(m,:) + .001 * rand ;   % additive noise of var = 1e-6
   % call SOVRLS to calculate the filter output, estimation error
   % and update the coefficients.
   [w,y,e,R,xb]=asptsovrls(xn(m),xb,w,d,R,.92,L1,L2);
   en(m,:) = e;                  % save the last error sample
end;

% display the results
% note that w converges to conj(h) for complex data
subplot(2,2,1);stem([real(w) imag(conj(w))]); grid;
subplot(2,2,2);eb = filter(.1,[1 -.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 8.3. The left side graph of the figure shows the adaptive filter coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the square error in dB versus time during the adaptation process. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB).

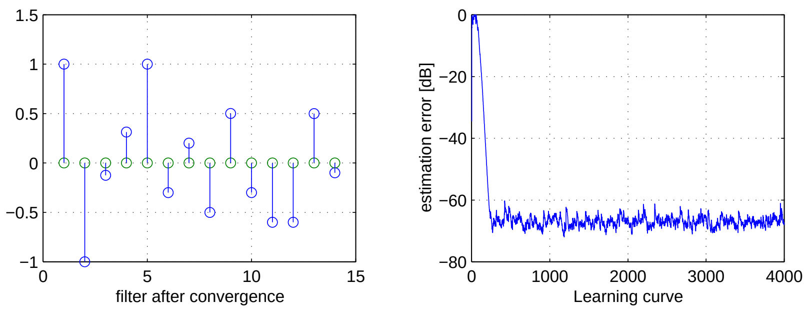

Figure 8.3: The adaptive filter coefficients after convergence and the learning curve for the FIR system identification problem using the SOVRLS algorithm.

Algorithm

The current implementation of `asptsovrls()` performs the following operations

- Constructs the new input samples delay line elements from the previous delay line and new input sample.
- Filters the input delay line $xb(n)$ through the adaptive filter $w(n-1)$ to produce the filter output $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$.
- Recursively updates the gain vector $\underline{\mathbf{K}}(n)$.
- Updates the adaptive filter coefficients according to equation (4.10).

Remarks

SOVRLS uses the regular RLS algorithm to update both the linear ($\underline{\mathbf{w}}_l$) and nonlinear ($\underline{\mathbf{w}}_n$) parts of the adaptive filter. Therefore, the convergence properties of the SOVRLS are similar to those of the RLS. `asptsovrls()` also,

- supports both real and complex data and filters. The adaptive filter for the complex SOVRLS algorithm converges to the complex conjugate of the optimum solution.
- internally updates the input delay line $xb(n)$ which includes the past linear and nonlinear samples needed for the next iteration. The past samples kept depend on the L1 and L2 parameters.

Resources

The resources required to implement a SOVRLS filter of linear memory length $L1$ and nonlinear memory length $L2$ are given below. The computations given are those required to process one sample. In the table below the symbol $M = (L1 + sum(1 : L2)$ is used.

MEMORY	$M^2 + 2M + 5$
MULTIPLY	$4M^2 + 3M + L2$
ADD	$3M^2 + M$
DIVIDE	M+1

See Also

INIT_ SOVRLS, ASPTRLS.

Reference

[2] and [4] for analysis of the RLS algorithm and its variants and [7] for an introduction to the adaptive Volterra filters..

8.4 asptsovtdlms

Purpose Sample per sample filtering and coefficient update using the Second Order Volterra Transform Domain Least Mean Squares Adaptive algorithm. Filtering and coefficients update of both the linear and non-linear coefficients are performed in the transform-domain (T).

Syntax

```
[W,y,e,p,xb,w] = asptsovtdlms(xn,xb,W,d,mu,L1,L2,p,b,T)
```

Description `asptsovtdlms()` implements the second order Volterra transform domain LMS filter (SOVTDLMS). The SOVTDLMS algorithm calculates the filter output $y(n)$ and updates the filter coefficients vector $\underline{\mathbf{W}}(n)$ in the transform domain. The filter output is the sum of the outputs of the linear filter part $\underline{\mathbf{W}}_l(n)$ and the nonlinear part $\underline{\mathbf{W}}_n(n)$ as follows.

$$y(n) = \sum_{m_1=0}^{L1-1} W_l(m1)x(n-m_1) + \sum_{m_1=0}^{L1-1} \Sigma_{m_2=m_1}^{L2-1} W_n(m1,m2)x(n-m_1)x(n-m_2) \tag{8.4}$$

The filter coefficients vector $\underline{\mathbf{W}}(n) = [\underline{\mathbf{W}}_l(n); \underline{\mathbf{W}}_n(n)]$ is updated according to the TDLMS algorithm in the transform domain. The memory depth of the linear and nonlinear filter parts are governed independently by the $L1$ and $L2$ parameters passed to `init_sovtdlms()`.
The input and output parameters of `asptsovtdlms()` for an FIR adaptive filter of linear memory length $L1$ and nonlinear memory length $L2$ are summarized below.

```
Input Parameters [Size]::
  xn  : new input sample [1 x 1]
  xb  : buffer of input samples [L1 + sum(1:L2) x 1]
  W   : previous T-domain coef. vector W(n-1) [L1 + sum(1:L2) x 1]
  d   : desired output d(n) [1 x 1]
  mu  : adaptation constants [1 x 1]
  L1  : memory length of linear part of w
  L2  : memory length of non-linear part of w
  p   : last estimated power of x p(n-1) [L1 + sum(1:L2) x 1]
  b   : AR pole for recursive calculation of p
  T   : The transform to be used {fft|dct|dst|...}
        user defined transforms are also supported.
        use transform T and its inverse iT.

Output parameters::
  W   : updated T-domain coef. vector
  y   : filter output y(n)
  e   : error signal; e(n) = d(n)-y(n)
  p   : new estimated power of x p(n)
  xb  : updated buffer of input samples
  w   : updated t-domain coef. vector w(n), only
        calculated if this output argument is given.
```

Example

```
% SOVTDLMS used in a simple system identification application.
% By the end of this script the adaptive filter w should have
% the same coefficients as the nonlinear unknown filter h.
iter = 5000;         % Number of samples to process
L1   = 4;
L2   = 4;
% The nonlinear unknown plant transfer function
% y(n) =  x(n) - x(n-1) - .125x(n-2) + .3125x(n-3)
%        +x(n)x(n) -.3x(n)x(n-1) + .2x(n)x(n-2) -.5x(n)x(n-3)
%        +.5x(n-1)x(n-1) -.3x(n-1)x(n-2) -.6x(n-1)x(n-3)
%        -.6x(n-2)x(n-2) +.5x(n-2)x(n-3) -.1x(n-3)x(n-3)
h  =[1;-1;-0;.125;.3125;1;-.3;.2;-.5;.5;-.3;-.6;-.6;.5;-.1];
xn =2*(rand(iter,1)-0.5);      % Input signal, zero mean random.
dn =sovfilt(h,xn,4,4);         % Unknown filter output
en =zeros(iter,1);             % vector to collect the error
[W,w,xb,d,y,e,p]=init_sovtdlms(L1,L2); % Initialize SOVTDLMS

%% Processing Loop
for (m=1:iter)
   d = dn(m,:) + .001 * rand ;   % additive noise of var = 1e-6
   % call SOVNLMS to calculate the filter output, estimation error
   % and update the coefficients.
   [W,y,e,p,xb,w] = asptsovtdlms(xn(m),xb,W,d,.05,L1,L2,p,.98,'fft');
   en(m,:) = e;                  % save the last error sample
end;

% display the results
% note that w converges to conj(h) for complex data
subplot(2,2,1);stem([real(w) imag(conj(w))]); grid;
subplot(2,2,2);eb = filter(.1,[1 -.9], en .* conj(en));
plot(10*log10(eb  ));grid
```

Running the above script will produce the graph shown in Fig. 8.4. The left side graph of the figure shows the adaptive filter coefficients after convergence which are almost identical to the unknown filter h. The right side graph shows the square error in dB versus time during the adaptation process. The lower limit of the error signal power in the learning curve is defined here by the additive white noise added at the filter output (-60 dB).

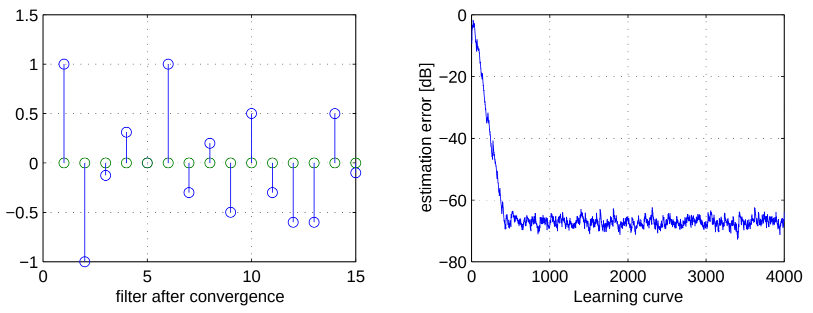

Figure 8.4: The adaptive filter coefficients after convergence and the learning curve for the FIR system identification problem using the SOVTDLMS algorithm.

Algorithm

The current implementation of `asptsovtdlms()` performs the following operations

- Constructs the new input samples delay line elements from the previous delay line and new input sample.
- Calculates the T-transformation of the delay line samples.
- Filters the input delay line through the adaptive filter $W(n-1)$ in the T-domain to produce the filter output $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$.
- Estimates the power of the input delay line in the T-domain.
- Updates the adaptive filter coefficients in the T-domain, and calculates the time domain coefficients if necessary.

Remarks

SOVTDLMS uses the regular TDLMS algorithm to update both the linear ($\underline{\mathbf{W}}_l$) and nonlinear ($\underline{\mathbf{W}}_n$) parts of the adaptive filter. Therefore, the convergence properties of the SOVTDLMS are similar to those of the TDLMS. `asptsovtdlms` also,

- supports both real and complex data and filters. The adaptive filter for the complex SOVTDLMS algorithm converges to the complex conjugate of the optimum solution.
- internally updates the input delay line for $xb(n)$ which includes the past linear and nonlinear samples needed for the next iteration. The past samples kept depend on the L1 and L2 parameters.

Resources

The resources required to implement a SOVTDLMS filter of linear memory length $L1$ and nonlinear memory length $L2$ are given below. The computations given are those required to process one sample. The complexity of the transformation T is indicated as $C(T)$ in the table below.

MEMORY	$4(L1 + sum(1:L2) + 6$
MULTIPLY	$5(L1 + sum(1:L2) + L2 + C(T) + 1$
ADD	$3(L1 + sum(1:L2)$
DIVIDE	(L1+sum(1:L2)

See Also

INIT_ SOVTDLMS, ASPTTDLMS.

Reference

[11] and [4] for extensive analysis of the LMS and the steepest-descent search method and [7] for an introduction to the adaptive Volterra filters.

8.5 asptsovvsslms

Purpose Sample per sample filtering and coefficients update using the Second Order Volterra Variable Step Size Least Mean Squares Adaptive filter algorithm.

Syntax

```
[w,g,mu,y,e,xb] = asptsovvsslms(xn,xb,w,g,d,mu,L1,L2,roh)
[w,g,mu,y,e,xb] = asptsovvsslms(xn,xb,w,g,d,mu,L1,L2,roh,
                                      ssa,mu_min,mu_max)
```

Description `asptsovvsslms()` implements the second order Volterra variable step size LMS adaptive filter (SOVVSSLMS). The SOVVSSLMS algorithm calculates the filter output $y(n)$, updates the filter coefficients vector $\underline{\mathbf{w}}(n)$, and updates the algorithm step size $\mu(n)$. The filter output is the sum of the outputs of the linear filter part $\underline{\mathbf{w}}_l(n)$ and the nonlinear part $\underline{\mathbf{w}}_n(n)$ as follows.

$$y(n) = \sum_{m_1=0}^{L1-1} w_l(m1)x(n-m_1) + \sum_{m_1=0}^{L1-1} \Sigma_{m_2=m_1}^{L2-1} w_n(m1,m2)x(n-m_1)x(n-m_2) \tag{8.5}$$

The filter coefficients vector $\underline{\mathbf{w}}(n) = [\underline{\mathbf{w}}_l(n); \underline{\mathbf{w}}_n(n)]$ is updated according to the VSSLMS algorithm. The memory depth of the linear and nonlinear filter parts are governed independently by the $L1$ and $L2$ parameters passed to `init_sovvsslms()`.
The input and output parameters of `asptsovvsslms()` for an FIR adaptive filter of linear memory length $L1$ and nonlinear memory length $L2$ are summarized below.

```
Input Parameters [size] ::

  xn  : new input sample [1 x 1]
  xb  : buffer of input samples [L1 + sum(1:L2) x 1]
  w   : vector of filter coefficients [L1 + sum(1:L2) x 1]
  g   : gradient vector [L1 + sum(1:L2) x 1]
  d   : desired output [1 x 1]
  mu  : step size vector [L1 + sum(1:L2) x 1]
  L1  : memory length of linear part of w
  L2  : memory length of non-linear part of w
  roh    : mu step size [1 x 1]
  ssa    : if 1, the sign-sign algorithm is used to update mu.
  mu_min : lower bound for mu [1 x 1]
  mu_max : higher bound for mu [1 x 1]

Output parameters ::

  w   : updated filter coefficients w(n)
  g   : updated gradient vector g(n)
  mu  : updated vector of step sizes mu(n)
  y   : filter output y(n)
  e   : error signal; e(n) = d(n) - y(n)
  xb  : updated vector of input samples
```

Algorithm The current implementation of `asptsovvsslms()` performs the following operations

- Constructs the new input samples delay line elements from the previous delay line and new input sample.
- Filters the input delay line $xb(n)$ through the adaptive filter $w(n-1)$ to produce the filter output $y(n)$.
- Calculates the error sample $e(n) = d(n) - y(n)$.
- Updates the step size vector and limits its elements if necessary.
- Updates the adaptive filter coefficients using the error $e(n)$ and the delay line of input samples $xb(n)$ resulting in $w(n)$.

Remarks SOVVSSLMS uses the regular VSSLMS algorithm to update both the linear ($\underline{\mathbf{w}}_l$) and nonlinear ($\underline{\mathbf{w}}_n$) parts of the adaptive filter. Therefore, the convergence properties of the SOVVSSLMS are similar to those of the VSSLMS. `asptsovvsslms()` also,

- supports both real and complex data and filters. The adaptive filter for the complex SOVVSSLMS algorithm converges to the complex conjugate of the optimum solution.
- updates the input delay line $xb(n)$ internally which includes the past linear and nonlinear samples needed for the next iteration. The past samples kept depend on the L1 and L2 parameters.

Resources The resources required to implement a SOVVSSLMS filter of linear memory length $L1$ and nonlinear memory length $L2$ are given below. The computations given are those required to process one real-value input sample.

MEMORY	$4(L1 + sum(1:L2) + 8$
MULTIPLY	$5(L1 + sum(1:L2) + L2$
ADD	$3(L1 + sum(1:L2)$
DIVIDE	0

See Also INIT_ SOVVSSLMS, ASPTVSSLMS, ASPTMVSSLMS, ASPTVFFRLS.

Reference [11] and [4] for extensive analysis of the LMS and the steepest-descent search method and [7] for an introduction to the adaptive Volterra filters.

Example

```
% SOVVSSLMS used in a simple system identification application.
% By the end of this script the adaptive filter w should have
% the same coefficients as the nonlinear unknown filter h.
iter = 5000;         % Number of samples to process
L1   = 4;
L2   = 4;
% The nonlinear unknown plant transfer function
% y(n) =  x(n) - x(n-1) - .125x(n-2) + .3125x(n-3)
%        +x(n)x(n) -.3x(n)x(n-1) + .2x(n)x(n-2) -.5x(n)x(n-3)
%        +.5x(n-1)x(n-1) -.3x(n-1)x(n-2) -.6x(n-1)x(n-3)
%        -.6x(n-2)x(n-2) +.5x(n-2)x(n-3) -.1x(n-3)x(n-3)
h  =[1;-1;-0;.125;.3125;1;-.3;.2;-.5;.5;-.3;-.6;-.6;.5;-.1];
xn  =2*(rand(iter,1)-0.5);      % Input signal, zero mean random.
dn  =sovfilt(h,xn,4,4);         % Unknown filter output
en  =zeros(iter,1);             % vector to collect the error
mu0 =0.05*ones(L1+sum(1:L2),1); % initial step size
muv =zeros(iter,1);             % Evolution of mu(1)
% Initialize the SOVVSSLMS algorithm
[w,xb,d,y,e,g,mu] = init_sovvsslms(L1,L2,[],[],[],mu0);;

%% Processing Loop
for (m=1:iter)
   d = dn(m) + .001 * rand ;      % additive noise of var = 1e-6
   % call SOVVSSLMS to calculate the filter output, estimation
   % error and update the coefficients.
   [w,g,mu,y,e,xb]= asptsovvsslms(xn(m),xb,w,g,...
                               d,mu,L1,L2,.001,1,1e-6,.99);
   en(m,:) = e;                   % save the last error
   muv(m) = mu(1);                % save mu(1) to display
end;

% display the results
subplot(3,3,1);stem(w); grid;
eb = filter(0.1,[1 -0.9], en .* conj(en));
subplot(3,3,2);plot(10*log10(eb ) );grid
subplot(3,3,3);plot(muv); grid;
```

Running the above script will produce the graph shown in Fig. 8.5. The left side graph of the figure shows the adaptive filter coefficients after convergence. The middle graph shows the square error in dB versus time during the adaptation process. The right side graph shows the evolution of the first element of the step size vector during the adaptation process.

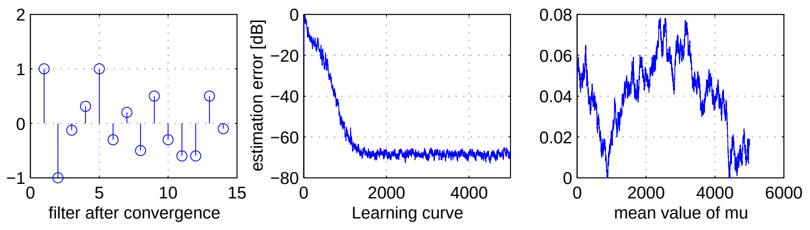

Figure 8.5: The adaptive filter coefficients after convergence, the evolution of the step size, and the learning curve for the FIR system identification problem using the SOVVSSLMS algorithm.

8.6 init_ sovlms

Purpose Creates and initializes the variables required for the Second Order Volterra Least Mean Squares adaptive algorithm.

Syntax

```
[w,x,d,y,e] = init_sovlms(L1,L2)
[w,x,d,y,e] = init_sovlms(L1,L2,w0,x0,d0)
```

Description The second order Volterra LMS filter consists of a linear filter part of length L1 and a nonlinear filter part. The nonlinear part uses the combination of cross-products between samples in the delay line. The number of past samples used in the nonlinear part is defined by the L2 parameter. A value of L2=0 reduces the Volterra filter to a linear LMS filter. The variables of the SOVLMS are summarized below.

```
Input Parameters [Size] ::
  L1 : memory length of the linear part of the filter
  L2 : memory length of the nonlinear part of the filter
  w0 : initial coefficient vector [L1 + sum(1:L2) x 1]
  x0 : initial input samples vector [L1 + sum(1:L2) x 1]
  d0 : initial desired sample [1 x 1]
Output parameters [default] ::
  w  : initialized filter coefficients [zeros]
  x  : initialized input vector [zeros]
  d  : initialized desired sample [white noise]
  y  : Initialized filter output
  e  : initialized error sample [e = d - y]
```

Example

```
L1 = 3;           % Memory of linear filter
L2 = 2;           % Memory of nonlinear filter
w0 = zeros(6,1);  % initial filter coefficients
x0 = rand(6,1);   % initial delay line
d0 = 0;           % desired sample

% Create and initialize a SOVLMS FIR filter
[w,x,d,y,e]=init_sovlms(L1,L2,w0,x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 3 through 5 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTSOVLMS.

8.7 init_ sovnlms

Purpose Creates and initializes the variables required for the Second Order Volterra Normalized Least Mean Squares adaptive algorithm.

Syntax

```
[w,x,d,y,e,p] = init_sovnlms(L1,L2)
[w,x,d,y,e,p] = init_sovnlms(L1,L2,w0,x0,d0)
```

Description The second order Volterra NLMS filter consists of a linear filter part of length L1 and a nonlinear filter part. The nonlinear part uses the combination of cross-products between samples in the delay line. The number of past samples used in the nonlinear part is defined by the L2 parameter. A value of L2=0 reduces the Volterra filter to a linear NLMS filter. The variables of the SOVNLMS are summarized below.

```
Input Parameters [Size] ::
  L1 : memory length of the linear part of the filter
  L2 : memory length of the nonlinear part of the filter
  w0 : initial coefficients vector [L1 + sum(1:L2) x 1]
  x0 : initial input samples vector [L1 + sum(1:L2) x 1]
  d0 : initial desired sample [1 x 1]
Output parameters [default] ::
  w  : initialized filter coefficients [zeros]
  x  : initialized input vector [zeros]
  d  : initialized desired sample [white noise]
  y  : Initialized filter output
  e  : initialized error sample [e = d - y]
  p  : input signal power.
```

Example

```
L1 = 3;           % Memory of linear filter
L2 = 2;           % Memory of nonlinear filter
w0 = zeros(6,1);  % initial filter coefficients
x0 = rand(6,1);   % initial delay line
d0 = 0;           % desired sample

% Create and initialize a SOVNLMS FIR filter
[w,x,d,y,e,p]=init_sovnlms(L1,L2,w0,x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 3 through 5 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTSOVNLMS.

8.8 init_ sovrls

Purpose Creates and initializes the variables required for the Second Order Volterra Recursive Least Squares (RLS) Adaptive Filter.

Syntax

```
[w,x,d,y,e,R] = init_sovrls(L1,L2,b)
[w,x,d,y,e,R] = init_sovrls(L1,L2,b,w0,x0,d0)
```

Description The second order Volterra RLS filter consists of a linear filter part of length L1 and a nonlinear filter part. The nonlinear part uses the combination of cross-products between samples in the delay line. The number of past samples used in the nonlinear part is defined by the L2 parameter. A value of L2=0 reduces the Volterra filter to a linear RLS filter. The variables of the SOVRLS are summarized below.

```
Input Parameters [size]::
  L1 : memory length of the linear part of the filter
  L2 : memory length of the nonlinear part of the filter
  b  : a small +ve constant to initialize R
  w0 : initial coefficients vector [L1 + sum(1:L2) x 1]
  x0 : initial input samples vector [L1 + sum(1:L2) x 1]
  d0 : initial desired sample [1 x 1]
Output parameters [default]::
  w  : Initialized filter coefficients [zeros]
  x  : Initialized input vector [zeros]
  d  : Initialized desired sample [white noise]
  y  : Initialized filter output [y = w' * x]
  e  : Initialized error sample [e = d - y]
  R  : Initialized inverse of the weighted
       auto correlation matrix of x, [R=b*eye(L1 + sum(1:L2))]
```

Example

```
L1 = 3;           % Memory of linear filter
L2 = 2;           % Memory of nonlinear filter
w0 = zeros(6,1);  % initial filter coefficients
x0 = rand(6,1);   % initial delay line
d0 = 0;           % desired sample

% Create and initialize a SOVRLS FIR filter
[w,x,d,y,e,R]=init_sovrls(L1,L2,.01,w0,x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 4 through 6 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTSOVRLS.

8.9 init_ sovtdlms

Purpose Creates and initializes the variables required for the Second Order Volterra Transform Domain LMS Adaptive filter.

Syntax

```
[W,w,x,d,y,e,p] = init_sovtdlms(L1,L2)
[W,w,x,d,y,e,p] = init_sovtdlms(L1,L2,W0,x0,d0)
```

Description The second order Volterra TDLMS filter consists of a linear filter part of length L1 and a nonlinear filter part. The nonlinear part uses the combination of cross-products between samples in the delay line. The number of past samples used in the nonlinear part is defined by the L2 parameter. A value of L2=0 reduces the Volterra filter to a linear TDLMS filter. The variables of the SOVTDLMS are summarized below.

```
Input Parameters [Size]::
  L1  : memory length of the linear part of the  filter
  L2  : memory length of the nonlinear part of the  filter
  w0  : initial T-domain coef. vector [L1 + sum(1:L2) x 1]
  x0  : initial input samples vector [L1 + sum(1:L2) x 1]
  d0  : initial desired sample [1 x 1]
Output parameters [default]::
  W   : initialized T-domain coef. vector [zeros]
  w   : initialized time-domain coef. vector [zeros]
  x   : initialized input vector [white noise]
  d   : initialized desired sample [white noise]
  y   : Initialized filter output
  e   : initialized error sample [e = d - y]
  p   : initialized power estimate
```

Example

```
L1 = 3;           % Memory of linear filter
L2 = 2;           % Memory of nonlinear filter
W0 = zeros(6,1);  % initial filter coefficients
x0 = rand(6,1);   % initial delay line
d0 = 0;           % desired sample

% Create and initialize a SOVTDLMS FIR filter
[W,w,x,d,y,e,p]=init_sovtdlms(L1,L2,W0,x0,d0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 3 through 5 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTSOVTDLMS.

8.10 init_ sovvsslms

Purpose Creates and initializes the variables required for the Second Order Volterra Variable Step Size Least Mean Squares adaptive algorithm.

Syntax

```
[w,x,d,y,e,g,mu] = init_sovvsslms(L1,L2)
[w,x,d,y,e,g,mu] = init_sovvsslms(L1,L2,w0,x0,d0,mu0,g0)
```

Description The second order Volterra VSSLMS filter consists of a linear filter part of length L1 and a nonlinear filter part. The nonlinear part uses the combination of cross-products between samples in the delay line. The number of past samples used in the nonlinear part is defined by the L2 parameter. A value of L2=0 reduces the Volterra filter to a linear VSSLMS filter. The variables of the SOVVSSLMS are summarized below.

```
Input Parameters [Size] ::
  L1  : memory length of the linear part of the filter
  L2  : memory length of the non-linear part of the filter
  w0  : initial coefficient vector [L1 + sum(1:L2) x 1]
  x0  : initial input samples vector [L1 + sum(1:L2) x 1]
  d0  : initial desired sample [1 x 1]
  mu0 : initial step-size vector [L1 + sum(1:L2) x 1]
  g0  : initial gradient vector [L1 + sum(1:L2) x 1]

Output parameters [default] ::
  w   : initialized filter coefficients [zeros]
  x   : initialized input vector [zeros]
  d   : initialized desired sample [white noise]
  y   : Initialized filter output
  e   : initialized error sample [e = d - y]
  g   : initialized gradient vector [zeros]
  mu  : initialized step-size vector [zeros]
```

Example

```
L1 = 3;              % Memory of linear filter
L2 = 2;              % Memory of nonlinear filter
W0 = zeros(6,1);     % initial filter coefficients
x0 = rand(6,1);      % initial delay line
d0 = 0;              % desired sample
mu0= .001*ones(6,1); % initial step sizes

% Create and initialize a SOVTDLMS FIR filter
[w,x,d,y,e,g,mu]=init_sovvsslms(L1,L2,w0,x0,d0,mu0);
```

Remarks

- Supports both real and complex signals and filters.
- Use input parameters 3 through 7 to initialize the algorithm storage. This is helpful when the adaptation process is required to start from a known operation point calculated off-line or from previous simulations.

See Also ASPTSOVVSSLMS.

Chapter 9

Non-adaptive, Visualization and Help Functions

This chapter documents the functions used to generate plots, manage the iteration progress window, and provide other functionalities that are used by the supported applications. Table 9.1 summarizes those functions and gives a short description and a pointer to the reference page of each function. Some of the help functions, such as `ipwin()` and `guifb()` are meant to be used internally by other functions and therefore not documented here.

Function Name	Reference	Short Description
init_ ipwin	9.1	Initializes iteration progress GUI window.
ipwin		Builds the iteration progress GUI window.
getStop		Returns the condition of the stop button in the IPWIN.
guifb		Handles the GUI feedback functions of the IPWIN.
mcmixr	9.2	Calculates the response of N speakers at M microphones.
osfilter	9.3	fast FIR filter using overlap-save.
plot_ ale	9.4	Generates plots for the Adaptive Line Enhancer problems.
plot_ anvc	9.5	Generates plots for Active Noise and Vibration Control.
plot_ beam	9.6	Generates plots for beam forming problems.
plot_ echo	9.7	Generates plots for echo canceling applications.
plot_ invmodel	9.8	Generates plots for inverse modeling problems.
plot_ model	9.9	Generates plots for modeling problems.
plot_ predict	9.10	Generates plots for linear prediction problems.
sovfilt	9.11	Second Order Volterra filter.
update_ ipwin	9.12	Updates the iteration progress GUI window.

Table 9.1: Visualization and help functions.

Each of the help functions is documented in a separate section including the following information related to the function:

- **Purpose:** Short description of the function.
- **Syntax:** Shows the function calling syntax. If the function has optional parameters, this section will have two calling syntaxes. One with only the required formal parameters and one with all the formal parameters. You can leave out an optional parameter when you call the function or use `[]` to use its default value.
- **Description:** Detailed description of the function usage with explanation of its input and output parameters.
- **Example:** A short example showing typical use of the function.
- **Remarks:** Gives more remarks specific to the usage of the function.

- **See Also:** Lists other functions that are related to this function.

9.1 init_ ipwin

Purpose Creates and initializes the iteration progress window.

Syntax

```
[E]=init_ipwin(L)
[E]=init_ipwin(L,ch)
```

Description The function `init_ipwin()` will bring up the iteration progress window (IPWIN) shown in Fig. 9.1. This window has a few widgets that makes it easy to control the progress of iterative applications such as adaptive filters applications. The IPWIN also shows the iteration number which indicates the number of samples processed so far and the mean squared error (MSE) in dB compared to the desired signal at that iteration. The iteration number and MSE are updated by calling `update_ipwin()`. IPWIN has four buttons, the `Stop`, `Plot`, `Break`, and `Quit` buttons. Pressing the `Stop` button while a simulation is inside the processing loop will halt the simulation and the button text will change to `Cont` so that the simulation can be resumed from where it stopped by pressing the same button again. The `Plot` button shows and hides an ASPT plot, allowing examining the signals without interrupting the simulation. The `Break` button breaks out of the processing loop without closing the open figures. The `Quit` button breaks out of the loop and closes all open plot figures. `init_ipwin()` takes two input arguments, the maximum number of samples to be processed and the number of channels in the desired signal, and returns one output argument, namely a vector where the mean square error will be stored. The variables of `init_ipwin()` are summarized below.

```
Input variable
  L  : maximum length of learning curve
  ch : number of channels (for multichannel applications)
Output variables
  E  : Learning curve vector for plotting
```

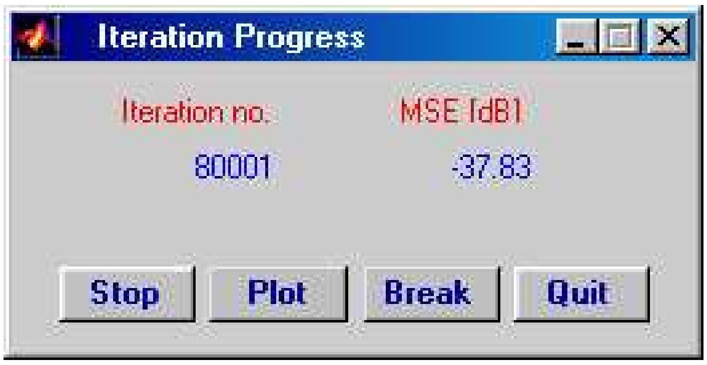

Figure 9.1: The iteration progress window.

Example

```
% Get the size of data to be processed
% L is two-element array [samples,channels]
L = wavread(dfile,'size');
% Initialize the iteration progress window
E = init_ipwin(L);
```

Remarks The iteration progress window can be used with any of the applications supported by the adaptive signal processing toolbox. Note however that `init_ipwin` uses the following global variables `stop`, `k_`, `des_`, `err_`, `pltf`, `brk`, and `ipw` and it is not recommended to use the same variable names in your applications that make use of the iteration progress window.

See Also UPDATE_ IPWIN.

9.2 mcmixr

Purpose Multichannel mixer, calculates the signals measured by M sensors in response to applying N signals at N different actuators.

Syntax `yn = mcmixr(h,xn,scale)`

Description Consider a multichannel systems with N actuators (speakers for instance) and M sensors (microphones) as that shown in Fig. 9.2. A signal applied at one of the actuators usually contribute to the response of all sensors. The response of each sensor is given by

$$y_m(n) = \sum_{n=1}^{N} h_{nm} * x_n(n); \quad m = 1, 2, \cdots, M, \tag{9.1}$$

where h_{nm} in (9.1) is the transfer function between the n^{th} actuator and m^{th} sensor, and $*$ is the convolution operator. `mcmixr()` takes as input the multichannel transfer function h and the actuator signals $x(n)$ and returns the sensors' response $y(n)$. The multichannel transfer function must be stored in a 3D matrix of dimensions $[L \times N \times M]$, where L is the number of coefficients of each transfer function, N is the number of actuators, and M is the number of sensors. If the lengths of the transfer functions are not the same, the shorter ones should be padded with zeros at the end. The input signals to the actuators $x(n)$ can either be a column vector or a matrix of N columns. If the input $x(n)$ is a column vector, the same signal is applied to all actuators. If $x(n)$ is a matrix of N columns, each column is applied to the corresponding actuator in h. The sensors' response is returned as a matrix of M columns. The third input parameter to `mcmixr()` is a flag, if 1, scaling of the calculated response will be performed. The input and output parameters of `mcmixr()` are summarized below.

```
Input Parameters [Size] ::
   h     : multichannel transfer function [L1 x N x M]
   xn    : actuators’ inputs [L2 x N]
   scale : if 1, scaling of output is performed

Output parameters [Size] ::
   y     : sensors’ output [L2 x M]
```

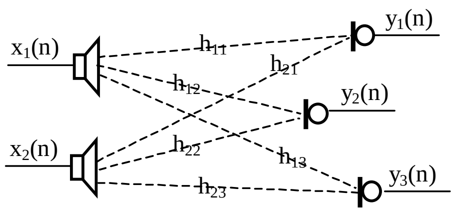

Figure 9.2: A multichannel system with two actuators and three sensors.

Example

```
% Multichannel transfer function between two actuators
% and three sensors
h = zeros(32,2,3);
ip       = [1; zeros(31,1)];  % impulse vector
h(:,1,1) = filter(1,[1 -.9 .9],ip);
h(:,1,2) = filter(1,[1 -.5 .5],ip);
h(:,1,3) = filter(1,[1 -.4],ip);
h(:,2,1) = filter(1,[1 -.85 .85],ip);
h(:,2,2) = filter(1,[1 -.7],ip);
h(:,2,3) = filter(1,[1 -.3 .9],ip);
xn = randn(1000,2);    % Input signal [1000 x 2]
yn = mcmixr(h,xn,0);   % output signal [1000 x 3]
```

Remarks

`mcmixr()` supports both real and complex signals and transfer functions. Although speakers and microphones are used to describe the functionality of `mcmixr()`, its use is not limited to audio applications and acoustic transfer functions. Transfer functions measured between shakers and accelerometers or between voltage sources and voltmeters can also be processed using `mcmixr()`.

9.3 osfilter

Purpose Fast FIR filter implementation in frequency domain using the overlap-save method.

Syntax

```
y = osfilter(h,x)
y = osfilter(h,x,L)
```

Description `osfilter(h,x)` filters the signal x through the filter h in frequency domain using the overlap-save method. If given, the optional third input argument L is used as the number of new input samples per block. If L is not given, it is calculated internally for maximum speed.
x can be either a vector or a matrix. If x is a vector, then h must also be a vector. If x is a matrix, then h can either be a vector in which case each column of x will be filtered through the same filter h, or a matrix of the same number of columns as x, in which case each column of x will be filtered by the corresponding coefficients vector in h. If the filter length is smaller than the number of columns, pad h with zeros so that the number of rows is always larger than the number of columns. The input and output parameters of `osfilter()` are summarized below.

```
Input Parameters [Size] ::
   h : single or multichannel FIR coefficients
   x : single or multichannel input signal.
   L : [optional] number of new input samples per block

Output parameters [Size] ::
   y  : filter output
```

Example

```
% Filter two signals in one go through the
% same filter
x = rand(1000,2);
h = [.5 .3 0 .1];
y = osfilter(h,x);
```

See also MCMIXER

9.4 plot_ ale

Purpose Displays the adaptive filter transfer function and the signals involved in adaptive line enhancer applications.

Syntax `plot_ale(w,x,y,e)`

Description `plot_ale()` displays the adaptive filter transfer function and the signals of interest in adaptive line enhancer applications. `plot_ale()` takes as input parameters the adaptive filter coefficients vector $w(n)$, the filter input signal $x(n)$, output signal $y(n)$, and error signal $e(n)$ and returns after rendering the ALE graph. The variables of `plot_model()` are summarized below.

```
Input Parameters ::
   w  : The adaptive filter response
   x  : filter input signal (narrow-band + wide-band)
   y  : filter output signal (narrow-band)
   e  : error signal (wide-band)
```

Example An example ALE graph generated using `plot_ale()` is shown in Fig. 9.3. The top left and right panels display the impulse response and frequency response of the adaptive filter, respectively. The next three panels display the line enhancer input $x(n)$, output $y(n)$, and error $e(n)$, respectively.

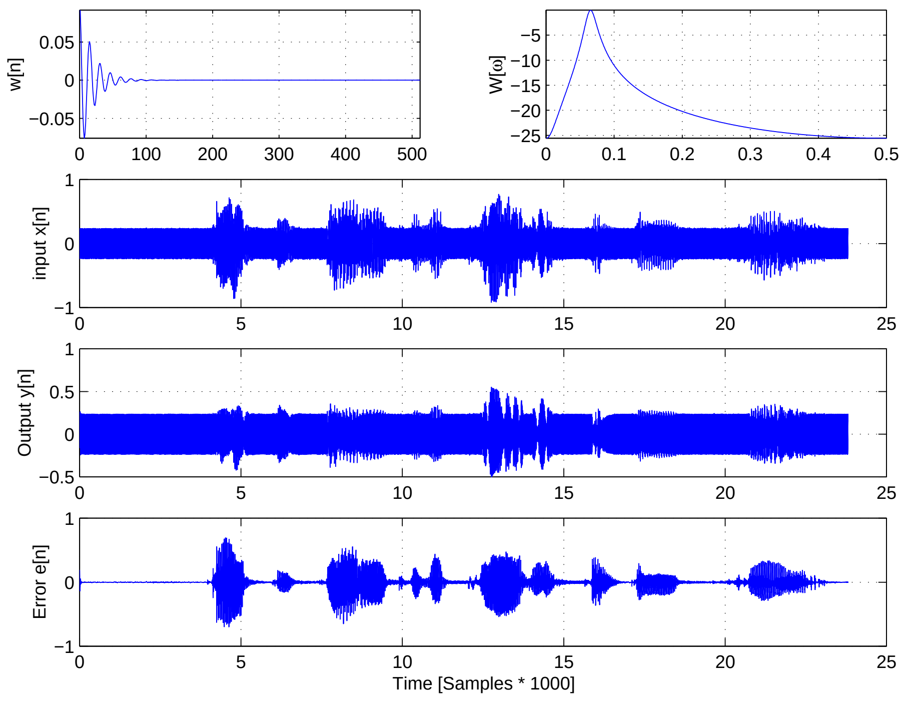

Figure 9.3: The adaptive line enhancer graph window.

9.5 plot_ anvc

Purpose Displays the adaptive controller transfer functions and the signals of interest in evaluating the performance of active noise and vibration control applications.

Syntax

```
plot_anvc(w,p,s,e)
plot_anvc(w,p,s,e,a,b)
```

Description `plot_anvc()` provides a quick access to the performance of an active noise and vibration controller. `plot_anvc()` takes as input the adaptive controller $w(n)$, the primary p and secondary s transfer functions, the mean square error vector $e(n)$ (that is usually returned by `update_ipwin()`), and returns after rendering the ANVC graphs. For multichannel systems, you can specify the index of the controller to be examined using the optional a and b parameters. The variables of `plot_anvc()` are summarized below.

```
Input variables [Size]:
  w   : estimated impulse response [L x Nref x Nact]
  p   : primary impulse response [Lp x Nref x Nsens]
  s   : secondary impulse response [Ls x Nact x Nsens]
  e   : mean square error
  a,b : displays the w(:,a,b) filter (multichannel only),
        if a & b are not given will display w(:,1,1).
```

Example An example ANVC graph window generated using `plot_anvc()` is shown in Fig. 9.4. The top left panel give the impulse response of the optimal solution $w_{opt}(:, a, b)$ calculated internally using the primary and secondary transfer functions. The frequency response of this optimal solution is plotted in the middle left panel. The top right panel show the impulse response of the adaptive controller $w(:, a, b)$ and the middle right panel shows its frequency response. The bottom left panel displays the evolution of the mean square error with time and the bottom right panel displays the difference between the optimum solution coefficients and the adaptive controller coefficients.

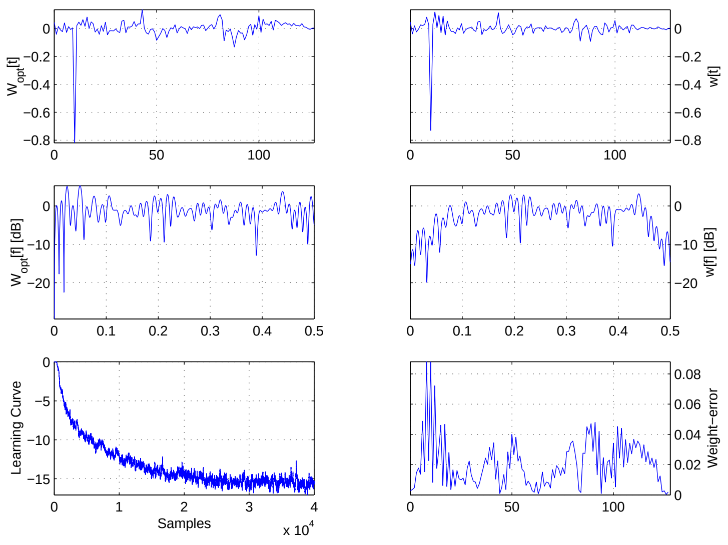

Figure 9.4: The active noise and vibration control graph window.

9.6 plot_ beam

Purpose Displays the directivity pattern and optionally the learning curve for evaluating the performance of beam former applications.

Syntax

```
plot_beam(E, w, L, Wo, cT)
plot_beam(E, w, L, Wo, cT, lc)
```

Description `plot_beam()` displays the learning curve and directivity pattern for evaluating the performance of an adaptive array (beam former). `plot_beam()` takes as input the mean square error vector $E(n)$ (returned by `update_ipwin()`), the adaptive filter coefficients vector $w(n)$, the array distance vector L, the radian frequency W_o, and the product term cT, and returns after rendering the graphs. If the last input argument lc is given and equals to one, only the directivity pattern is plotted. The variables of `plot_beam()` are summarized below.

```
Input variables:
  E  : mean square error
  w  : filter coefficients vector
  L  : vector containing distances between array elements
  Wo : sampled radian frequency
  cT : product of propagation speed * sampling period
  lc : if 1 will plot the directivity pattern only.
```

Example An example mean former graph window generated using `plot_beam()` is shown in Fig. 9.5. The left panel in this graph displays the learning curve of the adaptive array and the right panel displays the directivity pattern.

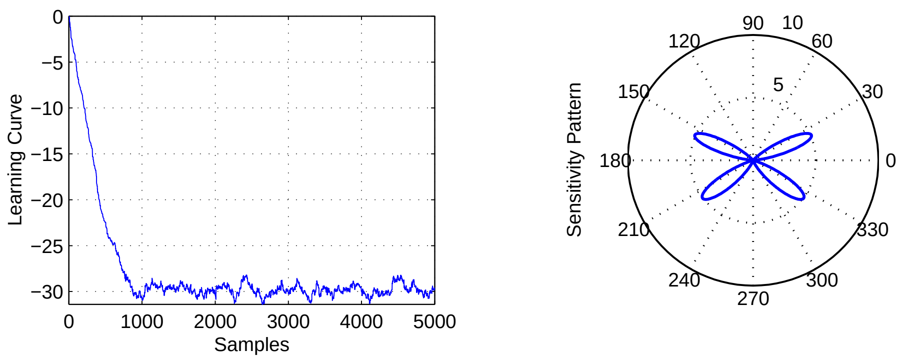

Figure 9.5: The adaptive beam former graph window.

9.7 plot_ echo

Purpose Displays the adaptive filter impulse response and the signals of interest in evaluating the performance of an echo canceler.

Syntax `plot_echo(w, x, r)`

Description Although formal testing of echo cancelers is a complicated process, usually quick performance measures are necessary during the process of developing the echo canceler. The `plot_echo()` function provides this quick performance measures. `plot_echo()` takes as input the adaptive filter coefficient vector $w(n)$, the echo canceler's subtractor input, and the residual signal, and returns after rendering the graph window. The variables of `plot_echo()` are summarized below.

```
Input arguments:
   w  : The adaptive filter coefficient vector
   x  : The echo-contaminated input signal
   r  : The residual signal
   x and r can be either vectors or wave files
```

Example An example graph window generated using `plot_echo()` is shown in Fig. 9.6. The generated graph has three panels. The top panel shows the coefficients of the adaptive filter at the moment of calling `plot_echo()` which is the impulse response of the echo path. The middle panel shows the input and output signals of the echo canceler subtractor. For acoustic echo cancelers, those two signals are the near-end speech S_{in} and residual signals S_{out}, respectively (see Fig. 2.15). For network echo cancelers those are the S_{in} and S_{out} signals in Fig. 2.14. The bottom panel shows the evolution of the echo return loss (ERL) introduced by the echo canceler. The ERL is calculated as the ratio between the mean square values of the input and output of the echo canceler subtractor signals in dB.

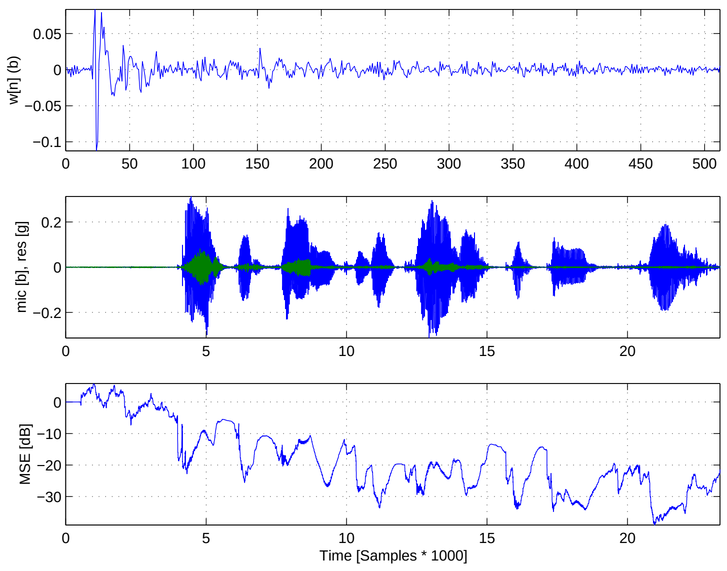

Figure 9.6: The echo canceler graph window.

9.8 plot_ invmodel

Purpose Displays the optimal filter and the solution achieved by the adaptive filter for evaluating the adaptive inverse modeling (equalization) applications.

Syntax `plot_invmodel(w,h,e,D)`

Description `plot_invmodel()` is helpful in the verification stage in adaptive inverse modeling (equalization) applications. `plot_invmodel()` takes as input the adaptive filter coefficients vector $w(n)$, the transfer function to be inverted h, the mean square error vector $e(n)$ (that is usually returned by `update_ipwin()`), and the modeling delay D, and returns after rendering the graph. The variables of `plot_invmodel()` are summarized below.

```
Input variables:
  w : adaptive filter coefficients vector
  h : impulse response of the channel to be equalized
  e : mean square error vector
  D : modeling delay in the desired response path.
```

Example An example graph window generated using `plot_invmodel()` is shown in Fig. 9.7. The top left panel shows the impulse response of the optimal solution w_{opt} which is the inverse of the channel response h in the sense that the convolution $w_{opt} * h$ results in a delayed impulse $\delta(n - D)$. The frequency response of this optimal solution is plotted in the middle left panel. The top two right panels show the impulse response and frequency response of the adaptive model. The bottom panel displays the evolution of the mean square error with time.

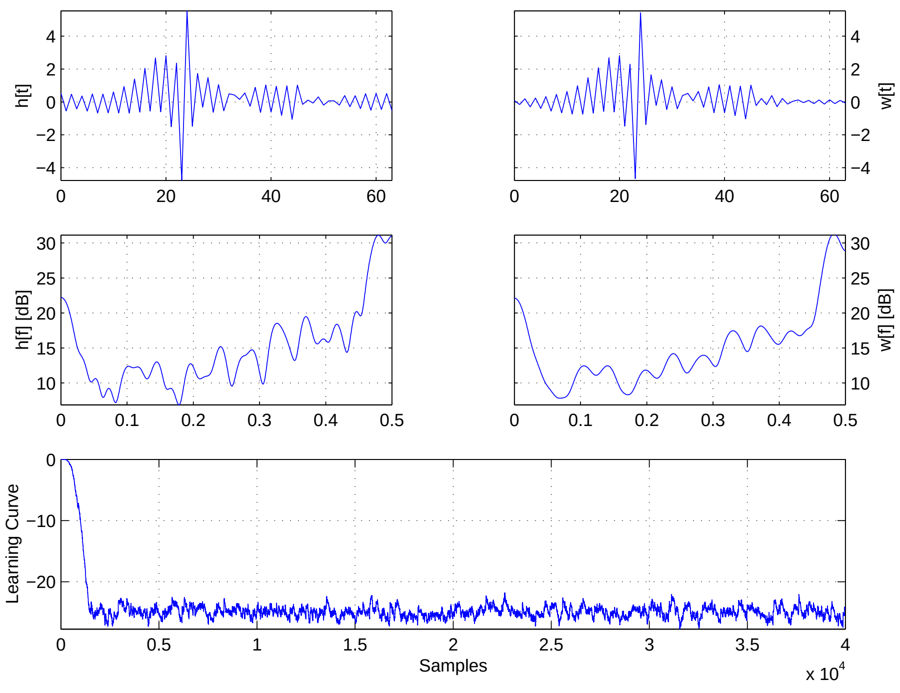

Figure 9.7: The inverse modeling (equalizer) graph window.

9.9 plot_ model

Purpose Displays the optimal filter and the solution achieved by the adaptive filter for evaluating the adaptive system identification applications.

Syntax `plot_model(w,h,e)`

Description `plot_model()` is helpful in the verification stage in adaptive system identification applications. `plot_model()` takes as input the adaptive filter coefficients vector $w(n)$, the impulse response of the system to be modeled h, and the mean square error vector $e(n)$ (that is usually returned by `update_ipwin()`), and returns after rendering the graph window. The variables of `plot_model()` are summarized below.

```
Input variables:
   w : estimated impulse response
   h : actual impulse response
   e : estimation error history
```

Example An example graph window generated using `plot_model()` is shown in Fig. 9.8. The top left panel shows the impulse response of the optimal solution w_{opt} for the system identification problem, which is the impulse response h in this case. The frequency response of this optimal solution is plotted in the middle left panel. The top two right panels show the impulse response and frequency response of the adaptive model. The bottom left panel displays the evolution of the mean square error with time and the bottom right panel displays the difference between the optimum solution coefficients and the adaptive model coefficients.

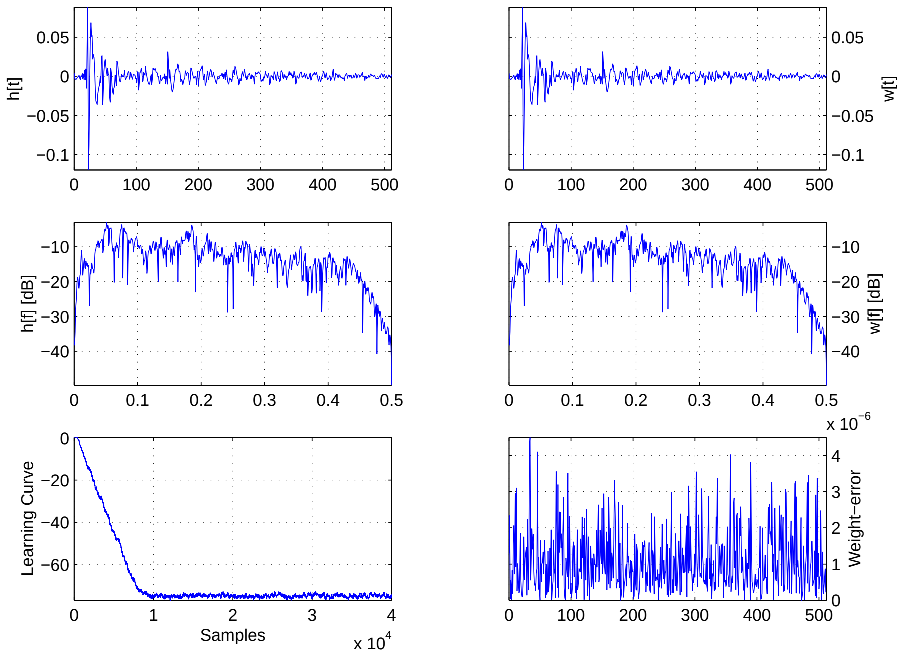

Figure 9.8: The modeling (system identification) graph window.

9.10 plot_ predict

Purpose Displays the learning curve of the adaptive filter and the signals of interest in adaptive prediction applications.

Syntax `plot_predict(x,y,r,e)`

Description `plot_predict()` displays the input, output, and error signals, and the learning curve of a prediction error filter. `plot_predict()` takes as input the prediction error filter input signal $x(n)$, its output $y(n)$, the error signal $r(n), and the mean square error vector e$(n) (that is usually returned by `update_ipwin()`), and returns after rendering the graph window. The variables of `plot_predict()` are summarized below.

```
Input variables:
   x  : predictor input signal
   y  : predictor output signal
   r  : predictor error signal
   e  : mean square error vector
```

Example An example graph window generated using `plot_predict()` is shown in Fig. 9.9. The top left panel shows the predictor input signal, the top right panel displays the prediction error, the bottom left panel displays the predictor output, and the bottom right panel displays the evolution of the mean square error with time.

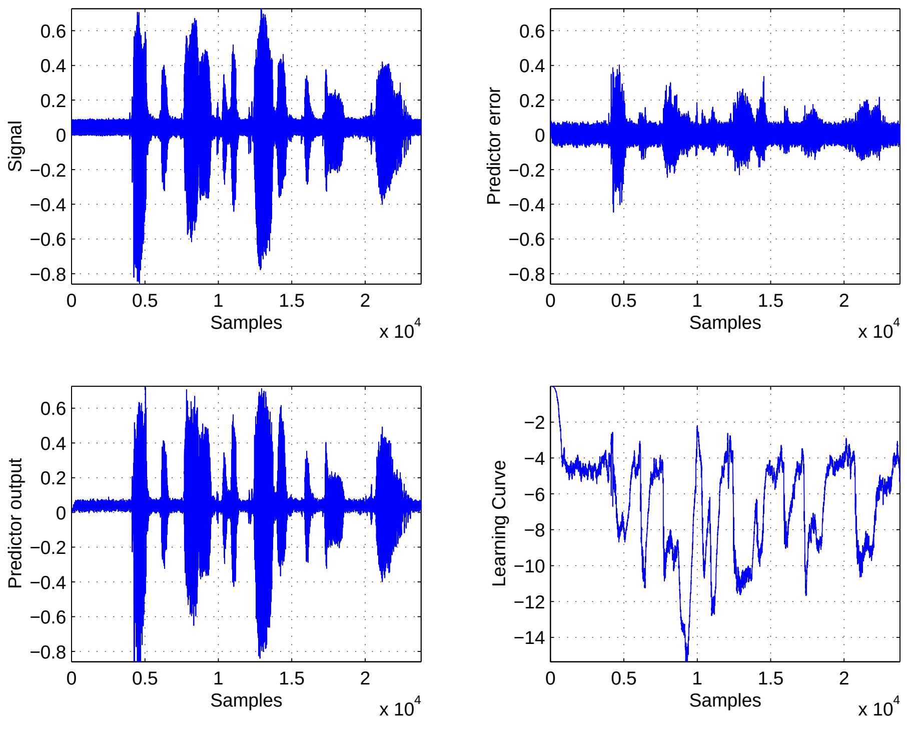

Figure 9.9: The adaptive prediction graph window.

9.11 sovfilt

Purpose Single channel second order Volterra filter.

Syntax

```
y = sovfilt(h,x,L1,L2)
```

Description `sovfilt(h,x,L1,L2)` returns the response of the nonlinear second order Volterra filter h when the signal x is applied at the filter input. $L1$ is the memory length of the linear part of h, and $L2$ is the memory length of the nonlinear part. The memory length is defined as the number of current and previous samples involved in the calculation of the filter output. The total number of filter coefficients in h is $L1 + sum(1 : L2)$. The first $L1$ coefficients are the linear filter part and the last $sum(1 : L2)$ coefficients are the nonlinear part of the filter. The filter output is given by

$$y(n) = \sum_{m_1=0}^{L1-1} h_l(m1)x(n-m_1) + \sum_{m_1=0}^{L1-1} \Sigma_{m_2=m_1}^{L2-1} h_n(m1, m2)x(n-m_1)x(n-m_2) \tag{9.2}$$

Example

```
% Nonlinear filter of L1=4, L2=3.
% y(n) =  x(n) - x(n-1) - .125x(n-2) + .3125x(n-3)
%        +x(n)x(n) -.3x(n)x(n-1) + .2x(n)x(n-2)
%        +.5x(n-1)x(n-1) -.3x(n-1)x(n-2)
%        -.6x(n-2)x(n-2)
h = [1;-1;-0.125;0.3125;1;-0.3;0.2;0.5;-0.3;-0.6];
% input signal is a one second sinusoidal of 100 Hz
% sampled at 1000 Hz.

t = (1:1000)/1000;
x = cos(2*pi*100*t);
y = sovfilt(h,x,4,3);
subplot(2,2,1); plot(abs(fft(x)));
subplot(2,2,2); plot(abs(fft(y)));
```

Running the above script will produce the graph shown in Fig. 9.10. Note that although the input signal has only one frequency component at 100 Hz, the filter output has three components at 0, 100, and 200 Hz. This is a general characteristic of nonlinear filters.

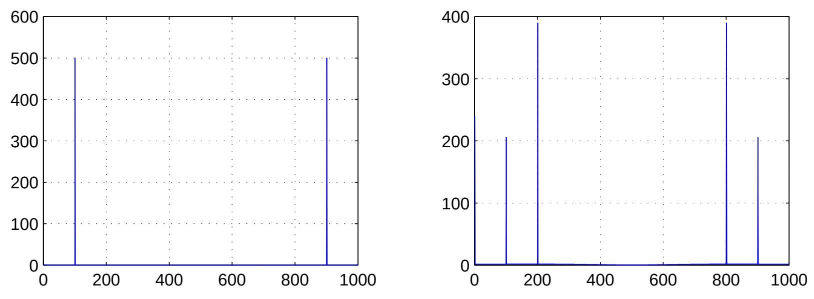

Figure 9.10: The frequency contents of the input and output of a second order Volterra filter.

9.12 update_ ipwin

Purpose Updates the iteration progress window and handles the callback functions of its buttons.

Syntax `[E,stop,brk] = update_ipwin(E,e,d,wp,x1,x2,x3,x4,x5)`

Description `update_ipwin()` updates the iteration number and mean square error (MSE) values on the iteration progress window (IPWIN) shown in Fig. 9.11, and manages the actions to be taken when one of the four buttons is pressed (see `init_ipwin()` for description of the IPWIN widgets). The first three input arguments to `update_ipwin()` are used to calculate the new MSE value that appear on the IPWIN and update the MSE vector. The input and desired signals are considered ergodic processes, therefore time averages are used to calculate the MSE instead of ensample averages. The fourth input parameter tells `update_ipwin()` which plotting function to call when the `Plot` button is pressed. The rest of the input arguments are passed to the plotting function without modification. `update_ipwin` returns the updated MSE vector and two control variables to be used for managing the `Stop` and `Break` buttons. The variables of `update_ipwin()` are summarized below.

```
Input variable
   E  : mean square error vector
   e  : new error sample (column vector for block processing)
   d  : new desired sample (column vector for block processing)
   wp : plot function to call, must be one of the following
      'l' => calls plot_ale
      'a' => calls plot_anvc
      'b' => calls plot_beam
      'e' => calls plot_echo
      'i' => calls plot_invmodel
      'p' => calls plot_predict
      'm' => calls plot_model
   x1-x5: parameters passed to the plot_xyz functions

Output variables
   E    : updated learning curve vector
   stop : control flag [0 = continue, 1 = stop]
   brk  : flag for breaking out of processing loop
```

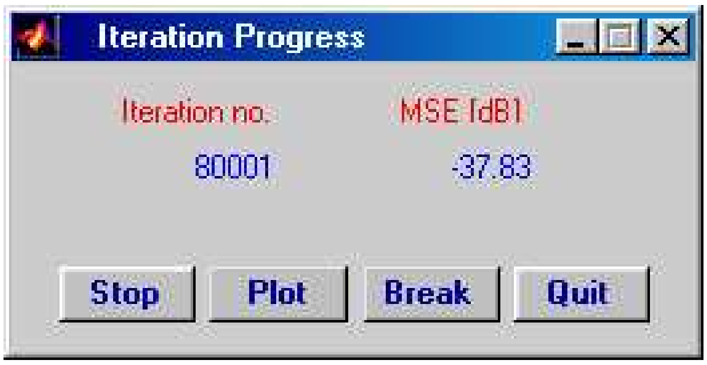

Figure 9.11: The iteration progress window.

Example

```
%% Typical processing loop using IPWIN
for (m=1:inSize)
   % read new input sample (x) and new desired sample
   % (d) and update the adaptive filter (w) here

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,d, 'm', wp, h);
   % handle the Stop button
while (stop ~= 0), stop  = getStop; end;
   % handle the Break button
if (brk), plot_model(w,h,E); break; end;
end;
```

Remarks

The iteration progress window can be used with any of the applications supported by the adaptive signal processing toolbox. Note however that `update_ipwin` uses the following global variables `stop`, `k_`, `des_`, `err_`, `pltf`, `brk`, and `ipw` and it is not recommended to use the same variable names in your applications that make use of the iteration progress window.

See Also

INIT_ IPWIN, PLOT_ ALE, PLOT_ ANVC, PLOT_ BEAM, PLOT_ ECHO, PLOT_ INVMODEL, PLOT_ MODEL, PLOT_ PREDICT.

Chapter 10

Applications and Examples

This chapter documents the applications and examples scripts supplied with the current distribution of the adaptive signal processing toolbox. Table 10.1 summarizes those scripts and gives a short description and a pointer to the reference page of each script. The application scripts documented in this chapter can be found in the `apps` directory. Each script is documented in a separate section which includes the following information related to the application:

- **Purpose:** Short description of the application implemented in this file.
- **Syntax:** Shows how to run the application.
- **Description:** Detailed description of the application.
- **Code:** A listing of the application code.
- **Results:** Presents the output generated by running the application.
- **Audio Files:** Lists the audio signals used in the application if any.
- **See Also:** Lists other related components of the toolbox.
- **Reference:** Lists literature for more information on the application.

Script Name	Reference	Short Description
ale_ csoiir2	10.1	Adaptive Line Enhancer using CSOIIR2.
ale_ soiir1	10.2	Adaptive Line Enhancer using SOIIR1.
ale_ soiir2	10.3	Adaptive Line Enhancer using SOIIR2.
anvc_ adjlms	10.4	Active noise and vibration control using ADJLMS.
anvc_ fdadjlms	10.5	Active noise and vibration control using FDADJLMS.
anvc_ fdfxlms	10.6	Active noise and vibration control using FDFXLMS.
anvc_ fxlms	10.7	Active noise and vibration control using FXLMS.
anvc_ mcadjlms	10.8	Active noise and vibration control using MCADJLMS.
anvc_ mcfdadjlms	10.9	Active noise and vibration control using MCFDADJLMS.
anvc_ mcfdfxlms	10.10	Active noise and vibration control using MCFDFXLMS.
anvc_ mcfxlms	10.11	Active noise and vibration control using MCFXLMS.
beambb_ lclms	10.12	Beam former at base-band frequency using LCLMS.
beamrf_ lms	10.13	Beam former at RF frequency using LMS.
echo_ bfdaf	10.14	Echo canceler using BFDAF.
echo_ leakynlms	10.15	Echo canceler using LEAKYNLMS.
echo_ nlms	10.16	Echo canceler using NLMS.
echo_ pbfdaf	10.17	Echo canceler using PBFDAF.
echo_ rcpbfdaf	10.18	Echo canceler using RCPBFDAF.
equalizer_ nlms	10.19	Inverse modeling using NLMS.
equalizer_ rls	10.20	Inverse modeling using RLS.
model_ arlmsnewt	10.21	Modeling using LMS-NEWTON.
model_ eqerr	10.22	IIR modeling using EQERR.
model_ lmslattice	10.23	Modeling using LMSLATTICE.
model_ mvsslms	10.24	FIR modeling using MVSSLMS.
model_ outerr	10.25	IIR modeling using OUTERR.
model_ rlslattice	10.26	Modeling using RLSLATTICE.
model_ sharf	10.27	IIR modeling using SHARF.
model_ tdlms	10.28	FIR modeling using TDLMS.
model_ vsslms	10.29	FIR modeling using VSSLMS.
predict_ lbpef	10.30	Prediction using LBPEF.
predict_ lfpef	10.31	Prediction using LFPEF.
predict_ rlslbpef	10.32	Prediction using RLSLBPEF.
predict_ rlslfpef	10.33	Prediction using RLSLFPEF.

Table 10.1: Adaptive filters applications.

10.1 ale_ csoiir2

Purpose Simulation of an Adaptive Line Enhancer (ALE) application using a cascade of M second order type-2 recursive adaptive filter.

Syntax `ale_csoiir2`

Description This application demonstrates the capability of the line enhancer to separate a wide-band signal from multiple narrow-band signals at different frequencies even when the narrow-band signals are time-varying. The block diagram of the cascaded adaptive line enhancer is shown in Fig. 10.1. The input signal $u(n)$ (a speech fragment contaminated with three sinusoidal noise signals with time-varying frequencies) is stored in the file infile. The application attempts to separate the speech from the sinusoidal noise and stores the former in the file wbfile and the latter in nbfile. First the variables for the cascade adaptive line enhancer filters are creates and initializes using `init_csoiir2()`, and the input signal is read from file, then a processing loop is started. In each iteration of the loop `asptcsoiir2()` is called with a new input sample to calculate the line enhancer output $y(n)$ (the sum of estimated narrow-band signals), the error sample $e(n)$ (the wide-band signal) and update the filter parameters s and t. The evolution of the adaptive parameters is also tracked for later examination.

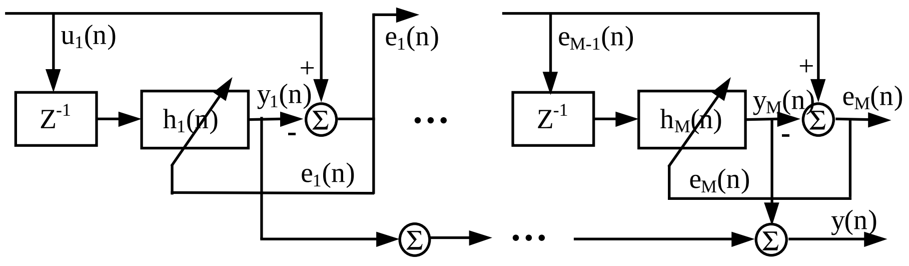

Figure 10.1: Block diagram of a Cascade of M second order adaptive line enhancer sections.

Code

```
clear all;
infile = '.\wavin\hnramp.wav';     % input, speech + sinusoidal
wbfile = '.\wavout\csoiir2wb.wav'; % wide-band signal (speech)
nbfile = '.\wavout\csoiir2nb.wav'; % narrow-band signal (harmonic)

[xn,inFs,inBits] = wavread(infile); % read input
[L,ch]           = size(xn);        % get data size

M     = 3;                  % No. of harmonics.
s0    = 0.25*ones(1,M);     % initial s
t0    = 0.5*ones(1,M);      % initial t
mu_s  = 0.001*ones(1,M);    % s-parameter adaptation constant
mu_t  = 0.05*ones(1,M);     % t-parameter adaptation constant
s_lim = [.1 .9];            % bounds for s
t_lim = [0.05 3.1];         % bounds for t
sv    = zeros(L,M);         % tracking vector for s
tv    = zeros(L,M);         % tracking vector for t
yv    = zeros(L,M);         % filter output
ev    = zeros(L,M);         % filter output

% Initialize the csoiir2 filters
[s,t,u,y,a,b,p]=init_csoiir2(M,s0,t0);

for k=2:L
   % Call CSOIIR2
   [y,a,b,u,t,s,p] = asptcsoiir2(xn(k),u,y,a,b,t,s,p,...
                     mu_t,mu_s,t_lim,s_lim);
   sv(k,:) = s;             % save s-state
   tv(k,:) = t;             % save t-state
   ev(k,:) = u(1,2:end);    % error signals
   yv(k,:) = y(1,:);        % narrow-band components
end

% Show tracking behavior
figure
subplot(2,2,1)
plot([tv]);grid
xlabel('Time [samples]')
ylabel('Center freq. [rad.]')
subplot(2,2,2)
plot(sv);grid
xlabel('Time [samples]')
ylabel('s parameter')

% save the narrow-band and wide-band signals
wavwrite(ev(1,M+1),inFs,inBits,wbfile);
wavwrite(sum(yv,2),inFs,inBits,nbfile);
```

Results

Running the above script will produce the graph shown in Fig. 10.2. The left panel in Fig. 10.2 shows the values taken by the t parameter for each of the three SOIIR2 sections versus time. The right panel shows the values taken by the three s parameters. The first second order section adapts and tracks the strongest sinusoidal component in the input signal. As it approaches its target, its s-parameter saturates to its maximum value as shown in the first vertical line in the right graph in Fig. 10.2. As soon as the first section has converged, the second section starts adapting to the strongest sinusoidal component in its input signal (the error signal of the preceding section). This process continues until each section has converged to one sinusoidal component. The error signal of the last section is the wide-band signal and the sum of the outputs of all sections is the output of the cascade combination and contains the estimated narrow-band signals.

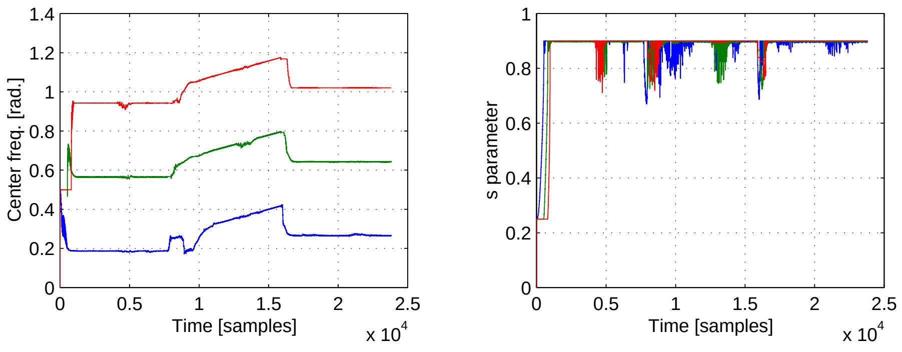

Figure 10.2: Convergence and tracking behavior of the cascade second order type-2 IIR adaptive line enhancer.

Audio Files

The following files demonstrate the performance of the CSOIIR2 algorithm in the adaptive line enhancer application mentioned above.

`wavin\hnramp.wav`	input signal, speech + sinusoidal noise.
`wavout\csoiir2wb.wav`	error signal, separated speech.
`wavout\csoiir2nb.wav`	filter output, narrow-band signals.

See Also

INIT_ CSOIIR2, ASPTCSOIIR2, ASPTSOIIR1, ASPTSOIIR2.

Reference

[2] and [10] for introduction to recursive adaptive filters.

10.2 ale_ soiir1

Purpose Simulation of an Adaptive Line Enhancer (ALE) application using a second order type-1 recursive adaptive filter.

Syntax `ale_soiir1`

Description This application demonstrates the capability of the line enhancer to separate a wide-band signal from a narrow-band signal even when the narrow-band signal is time-varying. The block diagram of the adaptive line enhancer problem is shown in Fig. 10.3. The input signal $u(n)$ (a speech fragment contaminated with a sinusoidal noise with time-varying frequency) is stored in the file infile. The application attempts to separate the speech from the sinusoidal noise and stores the former in the file wbfile and the latter in nbfile. First the variables for the adaptive line enhancer filter $h(n)$ are created and initialized using `init_soiir1()`, and the input signal is read from file, then a processing loop is started. In each iteration of the loop `asptsoiir1()` is called with a new input sample to calculate the filter output $y(n)$ (estimated narrow-band signal), the error sample $e(n)$ (the wide-band signal) and update the filter parameters s and w. The evolution of the adaptive parameters is also tracked for later examination.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graphs, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the residual signal $e(n)$ is written to a wave audio file and a graph presenting the performance of the line enhancer is generated.

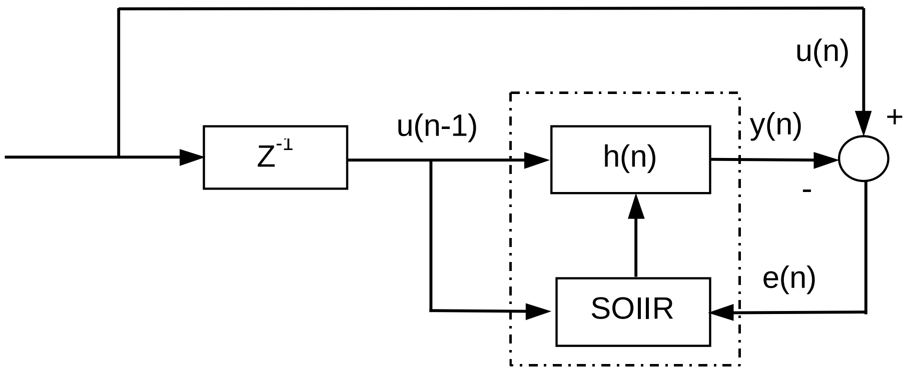

Figure 10.3: Block diagram of an adaptive line enhancer implemented using the second order type-1 IIR adaptive filter.

Code

```
clear all;
infile = '.\wavin\hramp.wav';     % input, speech + sinusoidal
wbfile = '.\wavout\soiir1wb.wav'; % wide-band signal (speech)
nbfile = '.\wavout\soiir1nb.wav'; % narrow-band signal

[xn,inFs,inBits] = wavread(infile); % read input
[L,ch]           = size(xn);        % get data size

w0    = 0.5;                % initial value for w
s0    = 0.3;                % initial value for s
w_lim = [-.999 .999];       % bounds for w
s_lim = [.1 .9];            % bounds for s
mu_w  = 0.5;                % step size for w
mu_s  = 0.01;               % step size for s

% Create and initialize soiir1 filter
[s,w,u,y,a,b,e] = init_soiir1(s0,w0);

sv = zeros(L,1);            % tracking vector for s
wv = sv;                    % tracking vector for w
yv = sv;                    % filter output
ev = sv;                    % filter output
E  = init_ipwin(L,ch);      % Initialize IPWIN
ip = [1; zeros(511,1)];     % Impulse vector
for k=2:L
   u = [xn(k); u(1:2)];
   [y,a,b,e,w,s] = asptsoiir1(u,y,a,b,e,w,s,mu_w,...
                   mu_s,w_lim,s_lim);
   sv(k) = s;
   wv(k) = w;
   ev(k) = e(1);
   yv(k) = y(1);

   % update the iteration progress window
   h = filter([w*(1-s) -(1-s)],[1 -w*(1+s) s],ip);
   [E,stop,brk]=update_ipwin(E,e(1),u(1),'l',h, xn,yv,ev);

   % handle the Stop button
   while (stop ~= 0), stop = getStop; end;

   % handle the Break button
   if (brk), plot_ale(h,xn,yv,ev); break; end;

end

h = filter([w*(1-s) -(1-s)],[1 -w*(1+s) s],ip);
plot_ale(h,xn,yv,ev);
figure; plot([sv cos(wv)]);grid
ylabel('s [blue], cos(w) [green]');
xlabel('Time [samples]')
wavwrite(ev,inFs,inBits,wbfile);
wavwrite(yv,inFs,inBits,nbfile);
```

Results

Running the above script will produce the graph shown in Fig. 10.4. The top two panels in Fig. 10.4 show the time and frequency responses of the adaptive IIR filter by the end of the simulation (end of input file). The second panel shows the input signal, the third shows the filter output (estimated narrow band signal), and the bottom panel shows the error signal (the separated wide-band signal).

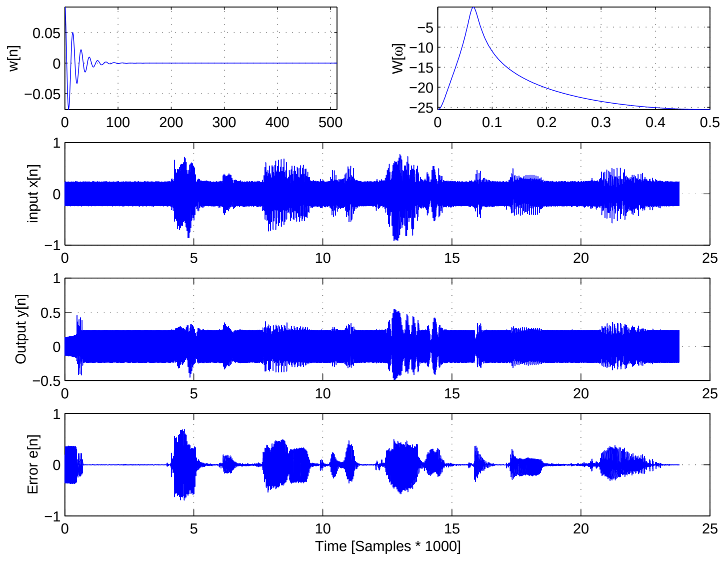

Figure 10.4: Performance of the second order type-1 IIR adaptive line enhancer.

Audio Files

The following files demonstrate the performance of the SOIIR1 algorithm in the adaptive line enhancer application mentioned above.

`wavin\hramp.wav`	input signal, speech + sinusoidal noise.
`wavout\soiir1wb.wav`	error signal, separated speech.
`wavout\soiir1nb.wav`	filter output, narrow-band signal.

See Also

INIT_ SOIIR1, ASPTSOIIR1, ASPTSOIIR2, ASPTCSOIIR2.

Reference

[2] and [10] for introduction to recursive adaptive filters.

10.3 ale_ soiir2

Purpose Simulation of an Adaptive Line Enhancer (ALE) application using a second order type-2 recursive adaptive filter.

Syntax `ale_soiir2`

Description This application demonstrates the capability of the line enhancer to separate a wide-band signal from a narrow-band signal even when the narrow-band signal is time-varying. The block diagram of the adaptive line enhancer problem is shown in Fig. 10.5. The input signal $u(n)$ (a speech fragment contaminated with a sinusoidal noise with time-varying frequency) is stored in the file infile. The application attempts to separate the speech from the sinusoidal noise and stores the former in the file wbfile and the latter in nbfile. First the variables for the adaptive line enhancer filter $h(n)$ are created and initialized using `init_soiir2()`, and the input signal is read from file, then a processing loop is started. In each iteration of the loop `asptsoiir2()` is called with a new input sample to calculate the filter output $y(n)$ (estimated narrow-band signal), the error sample $e(n)$ (the wide-band signal) and update the filter parameters s and t. The evolution of the adaptive parameters is also tracked for later examination.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graphs, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the residual signal $e(n)$ is written to a wave audio file and a graph presenting the performance of the line enhancer is generated.

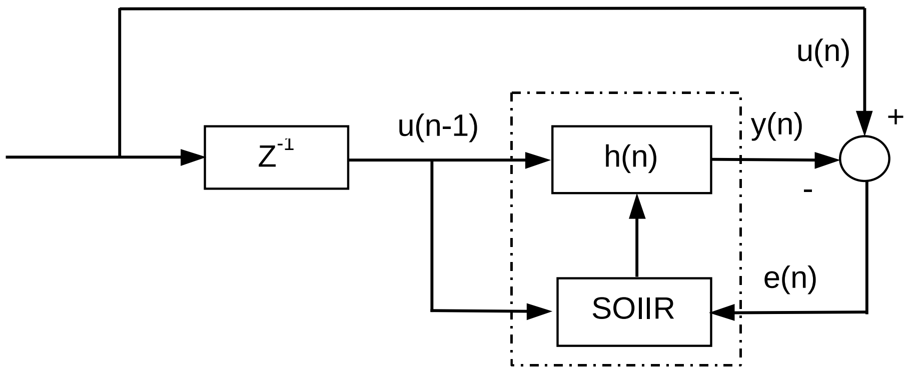

Figure 10.5: Block diagram of an adaptive line enhancer implemented using the second order type-2 IIR adaptive filter.

Code

```
clear all;
infile = '.\wavin\hramp.wav';     % input, speech + sinusoidal
wbfile = '.\wavout\soiir2wb.wav'; % wide-band signal (speech)
nbfile = '.\wavout\soiir2nb.wav'; % narrow-band signal

[xn,inFs,inBits] = wavread(infile); % read input
[L,ch]           = size(xn);        % get data size

t0    = 0.5;               % initial value for t
s0    = 0.3;               % initial value for s
t_lim = [0.05 3.1];        % bounds for t
s_lim = [.1 .9];           % bounds for s
mu_t  = 0.5;               % step size for t
mu_s  = 0.01;              % step size for s

% Create and initialize soiir2 filter
[s,t,u,y,a,b,e] = init_soiir2(s0,t0);

sv = zeros(L,1);           % tracking vector for s
tv = sv;                   % tracking vector for t
yv = sv;                   % filter output
ev = sv;                   % filter output
E  = init_ipwin(L,ch);     % Initialize IPWIN
ip = [1;zeros(511,1)];     % Impulse vector
for k=2:L
   u = [xn(k); u(1:2)];
   [y,a,b,e,t,s] = asptsoiir2(u,y,a,b,e,t,s,mu_t,...
                   mu_s,t_lim,s_lim);
   sv(k) = s;
   tv(k) = t;
   ev(k) = e(1);
   yv(k) = y(1);

   % update the iteration progress window
   h = filter([cos(t)*(1-s) -(1-s)],[1 -cos(t)*(1+s) s], ip);
   [E,stop,brk]=update_ipwin(E,e(1),u(1),'l',h, xn,yv,ev);

   % handle the Stop button
   while (stop ~= 0), stop = getStop; end;

   % handle the Break button
   if (brk), plot_ale(h,xn,yv,ev); break; end;

end

h = filter([cos(t)*(1-s) -(1-s)],[1 -cos(t)*(1+s) s],ip);
plot_ale(h,xn,yv,ev);
figure; plot([sv tv]);grid
ylabel('s [blue], t [green]');
xlabel('Time [samples]')
wavwrite(ev,inFs,inBits,wbfile);
wavwrite(yv,inFs,inBits,nbfile);
```

Results Running the above script will produce the graph shown in Fig. 10.6. The top two panels in Fig. 10.6 show the time and frequency response of the adaptive IIR filter by the end of the simulation (end of input file). The second panel shows the input signal, the third shows the filter output (estimated narrow band signal), and the bottom panel shows the error signal (the separated wide-band signal).

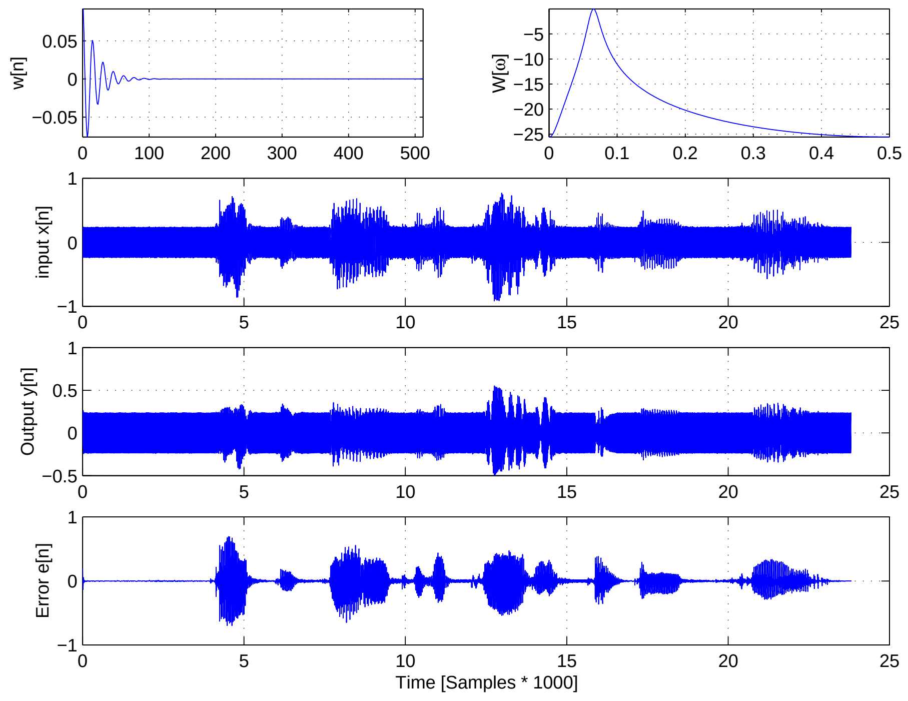

Figure 10.6: Performance of the second order type-2 IIR adaptive line enhancer.

Audio Files The following files demonstrate the performance of the SOIIR2 algorithm in the adaptive line enhancer application mentioned above.

`wavin\hramp.wav`	input signal, speech + sinusoidal noise.
`wavout\soiir2wb.wav`	error signal, separated speech.
`wavout\soiir2nb.wav`	filter output, narrow-band signal.

See Also INIT_ SOIIR2, ASPTSOIIR2, ASPTSOIIR1, ASPTCSOIIR2.

Reference [2] and [10] for introduction to recursive adaptive filters.

10.4 anvc_ adjlms

Purpose Simulation of a single channel Active Noise and Vibration Control (ANVC) application using an adaptive controller updated according to the ADJOINT Least Mean Squares (ADJLMS) algorithm.

Syntax `anvc_adjlms`

Description The block diagram of the single channel ANVC problem using ADJLMS is shown in Fig. 10.7. The primary impulse response p is the impulse response measured between the noise source and the error microphone in a small room. The secondary impulse response s is that between the secondary source and the error microphone. Two sets of transfer functions are provided in the simulation, the simple data set (system 1) and the measured data set (system 2). The description below applies to system 1, with a primary impulse response sampled at 8 kHz and truncated to 32 coefficients, and a simple FIR secondary impulse response so that it can be easily experimented with its minimum phase and delay properties. The primary noise at the microphone is stored in the file dfile and the reference signal $x(n)$ (white noise) is stored in the file infile. First the variables for the controller $w(n)$ are creates and initializes using `init_adjlms()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptadjlms()` is called with a new reference sample and a new primary sample to calculate the controller output (control effort) and update the controller coefficients. The sensor signal $e(n)$ is saved in each iteration for later examination.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the controller performance is generated.

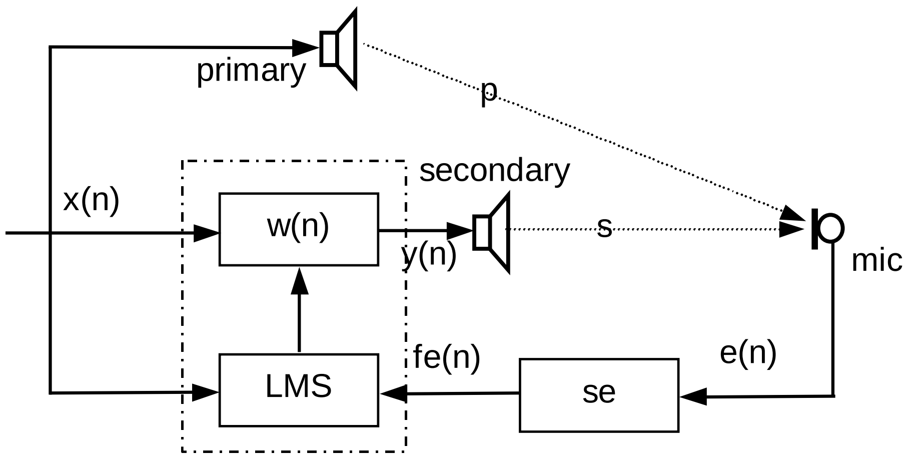

Figure 10.7: Block diagram of a single channel noise cancellation application using the Adjoint-LMS algorithm.

Code

```
clear all;
load .\data\h32;
ph        = h32;                               % Primary IR
sh        = [0; 0.4; 0.3; 0.2; 0.1; 0.05];  % Secondary IR
se        = 0.9*sh;                            % estimated sh
Lph       = length(ph);
Lsh       = length(sh);
L         = 40;                                % controller length
mu        = 0.02;                              % adaptation constant
b         = 0.98;                              % autoregressive pole
infile    = '.\wavin\scinwn.wav';              % reference signal
dfile     = '.\wavin\scdwn32.wav';             % d(n) = conv(x,h32)
outfile   = '.\wavout\adjlms.wav';             % sensor signal

[w,x,y,d,e,p]    = init_adjlms(L,sh,se);    % Init ADJLMS
inSize           = wavread(infile, 'size'); % get input data size
[xn,inFs,inBits] = wavread(infile);         % get sampling details
dSize            = wavread(dfile, 'size');  % data size
[dn,inFs,dBits]  = wavread(dfile);          % get sampling details
inSize           = max(min(inSize,dSize));  % Max. samples index
E                = init_ipwin(inSize);      % Initialize IPWIN
out              = zeros(size(xn));         % sensor signal

%% Processing Loop
for (m=1:inSize)

   % update the delay line
   x = [xn(m,:); x(1:end-1,:)];

   % update the controller
   [w,y,e,p] = asptadjlms(w,x,e,y,sh,se,dn(m),p,mu,b);

   % save the last sensor sample
   out(m) = e(1);

   % IPWIN handling section, see UPDATE\_~IPWIN for more details
   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e(1),dn(m), 'a', w, ph,sh);
   % handle the Stop button
   while (stop  ~= 0), stop  = getStop; end;
   % handle the Break button
   if (brk), plot_anvc(w,ph,sh,E); break; end;

end;

plot_anvc(w,ph,sh,E);                       % generate ANVC plot
wavwrite(out(1:m),inFs,inBits,outfile);    % write to audio file
```

Results

Running the above script will produce the graph shown in Fig. 10.8. This figure shows the optimum (Wiener) solution of the problem at hand and compares that with the solution approached by the adaptive controller. The top left and top right panels show the impulse responses of the optimal solution and the adaptive controller, respectively. The middle left and middle right panels show the frequency responses of the optimal solution and the adaptive controller, respectively. The bottom left panel shows the learning curve for the adaptive controller, and the bottom right shows the difference between the optimal solution coefficients and the adaptive controller coefficients.

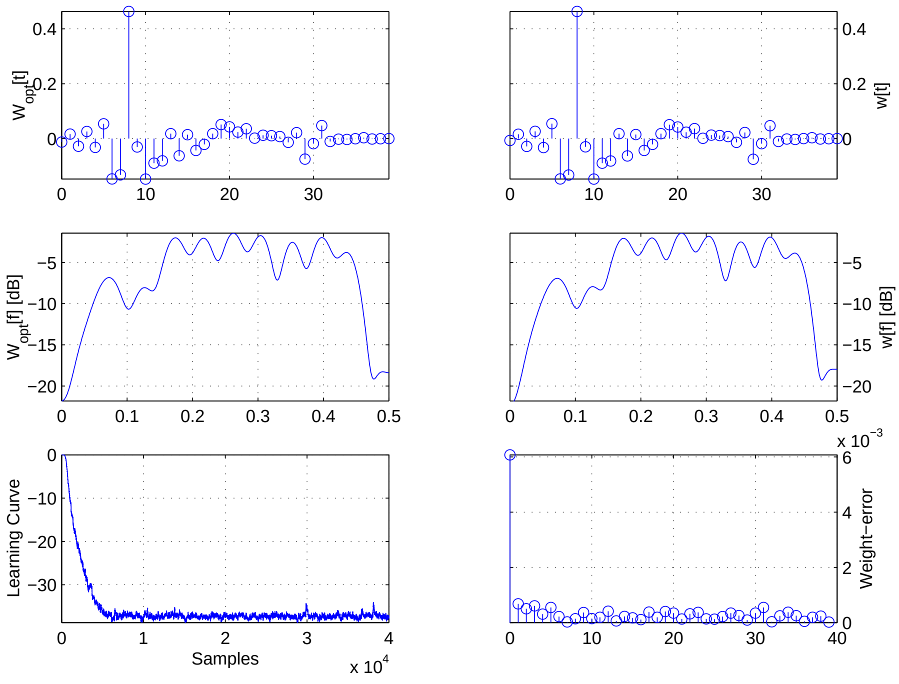

Figure 10.8: Performance of the ADJLMS algorithm.

Audio Files

The following files demonstrate the performance of the ADJLMS algorithm in the application mentioned above.

`wavin\scinwn.wav`	Reference (input) signal
`wavin\scdwn32.wav`	microphone signal with controller OFF.
`wavout\adjlms.wav`	microphone signal with controller ON.

See Also

INIT_ ADJLMS, ASPTADJLMS, ASPTMCADJLMS, ASPTFDADJLMS, ASPTMCFDADJLMS.

Reference

[3], Chapter 3.

10.5 anvc_ fdadjlms

Purpose Simulation of a single channel Active Noise and Vibration Control (ANVC) application using an adaptive controller updated according to the Frequency Domain ADJOINT Least Mean Squares (FDADJLMS) algorithm.

Syntax `anvc_fdadjlms`

Description The block diagram of the single channel ANVC problem using FDADJLMS is shown in Fig. 10.9. The primary impulse response p is the impulse response measured between the noise source and the error microphone in a small room. The secondary impulse response s is that between the secondary source and the error microphone. Two sets of transfer functions are provided in the simulation, the simple data set (system 1) and the measured data set (system 2). The description below applies to system 2, with impulse responses sampled at 8 kHz and truncated to 256 coefficients. The primary noise at the microphone is stored in the file dfile and the reference signal $x(n)$ (white noise) is stored in the file infile. First the variables for the controller $w(n)$ are creates and initializes using `init_fdadjlms()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptfdadjlms()` is called with a new block of reference samples and a new block of primary samples to calculate the a block of the controller output samples (control effort) and update the controller coefficients. The calculated block of sensor signal samples $e(n)$ is saved in each iteration for later examination.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the controller performance is generated.

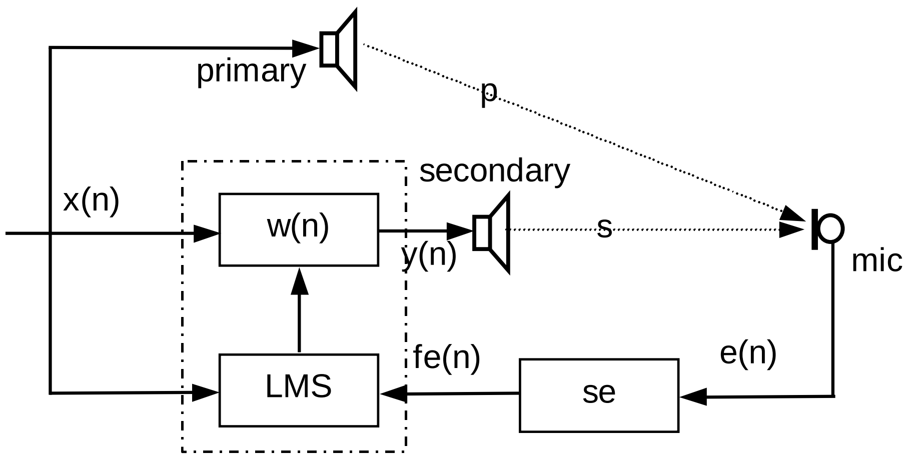

Figure 10.9: Block diagram of a single channel noise cancellation application using the Frequency Domain ADJoint-LMS algorithm.

Code

```
load .\data\ph11_256;              % Primary IR (p11)
load .\data\sh11_256;              % Secondary IR (s11)
Lp        = length(p11);
se        = 0.9*s11;               % estimated s11
NC        = Lp;                    % controller length
NL        = NC;                    % block length
c         = 1;                     % constrained filter
mu        = .02/NC;                % adaptation constant
b         = 0.98;                  % autoregressive pole
infile    = '.\wavin\scinwn.wav';    % x(n), white noise
dfile     = '.\wavin\scdwn256.wav';  % d(n) = conv(x,p11)
outfile   = '.\wavout\fdadjlms.wav'; % microphone output

% Initialize FDADJLMS algorithm and data files
[NB,W,w,x,y,d,e,p,S,SE,yF,feF] = init_fdadjlms(NC,NL,s11,se);
inSize           = wavread(infile, 'size'); % input data size
[xn,inFs,inBits] = wavread(infile,NL);      % input properties
dSize            = wavread(dfile, 'size');  % primary data size
[dn,inFs,dBits]  = wavread(dfile,NL);       % primary properties
inSize           = max(min(inSize,dSize));  % samples to process
E                = init_ipwin(inSize);      % Initialize IPWIN
out              = zeros(size(xn));         % microphone signal

%% Processing Loop
for (m=1:NL:inSize-NL)
   % read NL samples from input and primary and scale the block
   xn = 2^(inBits-1) * wavread(infile,[m,m+NL-1]);
   dn = 2^(inBits-1) * wavread(dfile,[m,m+NL-1]);
   % Call ASPTFDADJLMS to calculate the output and update coef.
   [W,w,x,y,e,p,yF,feF] = asptfdadjlms(NC,W,x,xn,dn,...
                          yF,feF,S,SE,p,mu,b,c);
   out(m:m+NL-1) = 2^-(dBits-1) * e;         % save error block
   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,dn, 'a', w, p11, s11);
   % handle Stop button
while (stop~=0), stop = getStop; end;
   % handle Break button
if (brk), plot_anvc(w,p11,s11,E); break; end;
end;
plot_anvc(w,p11,s11,E);                   % performance plots
wavwrite(out(1:m),inFs,inBits,outfile);   % save mic signal
```

Results Running the above script will produce the graph shown in Fig. 10.10. This figure shows the optimum (Wiener) solution of the problem at hand and compares that with the solution approached by the adaptive controller. The top left and top right panels show the impulse responses of the optimal solution and the adaptive controller, respectively. The middle left and middle right panels show the frequency responses of the optimal solution and the adaptive controller, respectively. The bottom left panel shows the learning curve for the adaptive controller, and the bottom right shows the difference between the optimal solution coefficients and the adaptive controller coefficients.

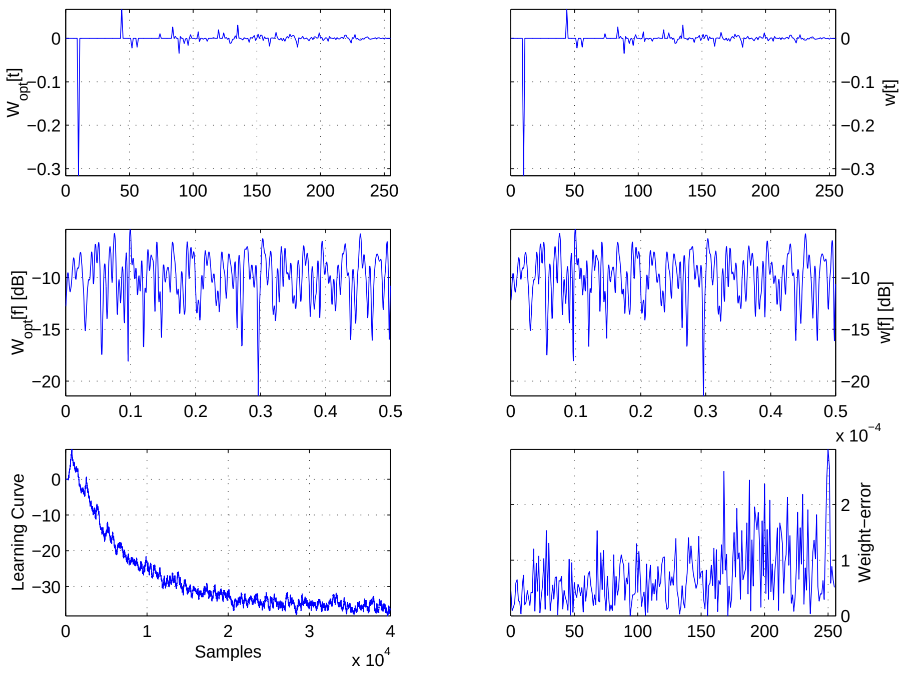

Figure 10.10: Performance of the FDADJLMS algorithm.

Audio Files The following files demonstrate the performance of the ADJLMS algorithm.

`wavin\scinwn.wav`	Reference (input) signal
`wavin\scdwn256.wav`	microphone signal with controller OFF.
`wavout\fdadjlms.wav`	microphone signal with controller ON.

See Also INIT_ FDADJLMS, ASPTFDADJLMS, ASPTMCFDADJLMS, ASPTADJLMS, ASPTMCADJLMS.

Reference [3], Chapter 3 for detailed description of the FDADJLMS, [8] for the overlap-save method, and [9] for frequency domain adaptive filters.

10.6 anvc_ fdfxlms

Purpose Simulation of a single channel Active Noise and Vibration Control (ANVC) application using an adaptive controller updated according to the Frequency Domain Filtered-X Least Mean Squares (FDFXLMS) algorithm.

Syntax `anvc_fdfxlms`

Description The block diagram of the single channel ANVC problem using FDFXLMS is shown in Fig. 10.11. The primary impulse response p is the impulse response measured between the noise source and the error microphone in a small room. The secondary impulse response s is that between the secondary source and the error microphone. Two sets of transfer functions are provided in the simulation, the simple data set (system 1) and the measured data set (system 2). The description below applies to system 2, with impulse responses sampled at 8 kHz and truncated to 256 coefficients. The primary noise at the microphone is stored in the file dfile and the reference signal $x(n)$ (white noise) is stored in the file infile. First the variables for the controller $w(n)$ are creates and initializes using `init_fdfxlms()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptfdfxlms()` is called with a new block of reference samples and a new block of primary samples to calculate the a block of the controller output samples (control effort) and update the controller coefficients. The calculated block of sensor signal samples $e(n)$ is saved in each iteration for later examination.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the controller performance is generated.

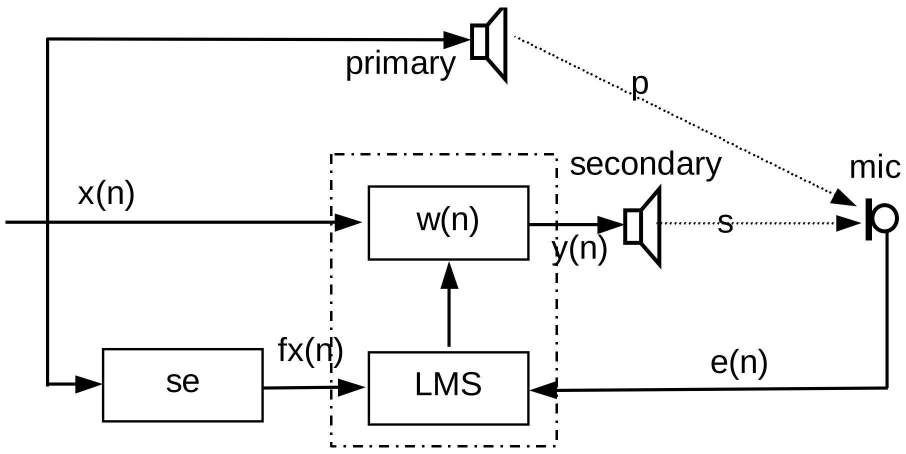

Figure 10.11: Block diagram of a single channel noise cancellation application using the Frequency Domain Filtered-X LMS algorithm.

Code

```
load .\data\ph11_256;               % Primary IR (p11)
load .\data\sh11_256;               % Secondary IR (s11)
Lp        = length(p11);
se        = 0.9*s11;                % estimated s11
NC        = Lp;                     % controller length
NL        = NC;                     % block length
c         = 1;                      % constrained filter
mu        = .02/NC;                 % adaptation constant
b         = 0.98;                   % autoregressive pole
infile    = '.\wavin\scinwn.wav';    % x(n), white noise
dfile     = '.\wavin\scdwn256.wav';  % d(n) = conv(x,p11)
outfile   = '.\wavout\fdfxlms.wav';  % microphone output

% Initialize FDFXLMS algorithm and data files
[NB,W,w,x,y,d,e,p,S,SE,yF,fxF] = init_fdfxlms(NC,NL,s11,se);
inSize           = wavread(infile, 'size'); % input data size
[xn,inFs,inBits] = wavread(infile,NL);      % input properties
dSize            = wavread(dfile, 'size');  % primary data size
[dn,inFs,dBits]  = wavread(dfile,NL);       % primary properties
inSize           = max(min(inSize,dSize));  % samples to process
E                = init_ipwin(inSize);      % Initialize IPWIN
out              = zeros(size(xn));         % microphone signal

%% Processing Loop
for (m=1:NL:inSize-NL)
   % read NL samples from input and primary and scale the block
   xn = 2^(inBits-1) * wavread(infile,[m,m+NL-1]);
   dn = 2^(inBits-1) * wavread(dfile,[m,m+NL-1]);
   % Call ASPTFDFXLMS to calculate the output and update coef.
   [W,w,x,y,e,p,yF,fxF] = asptfdfxlms(NC,W,x,xn,dn,...
                          yF,fxF,S,SE,p,mu,b,c);
   out(m:m+NL-1) = 2^-(dBits-1) * e;         % save error block
   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,dn, 'a', w, p11, s11);
   % handle Stop button
while (stop~=0), stop = getStop; end;
   % handle Break button
if (brk), plot_anvc(w,p11,s11,E); break; end;
end;
plot_anvc(w,p11,s11,E);                   % performance plots
wavwrite(out(1:m),inFs,inBits,outfile);   % save mic signal
```

Results

Running the above script will produce the graph shown in Fig. 10.12. This figure shows the optimum (Wiener) solution of the problem at hand and compares that with the solution approached by the adaptive controller. The top left and top right panels show the impulse responses of the optimal solution and the adaptive controller, respectively. The middle left and middle right panels show the frequency responses of the optimal solution and the adaptive controller, respectively. The bottom left panel shows the learning curve for the adaptive controller, and the bottom right shows the difference between the optimal solution coefficients and the adaptive controller coefficients.

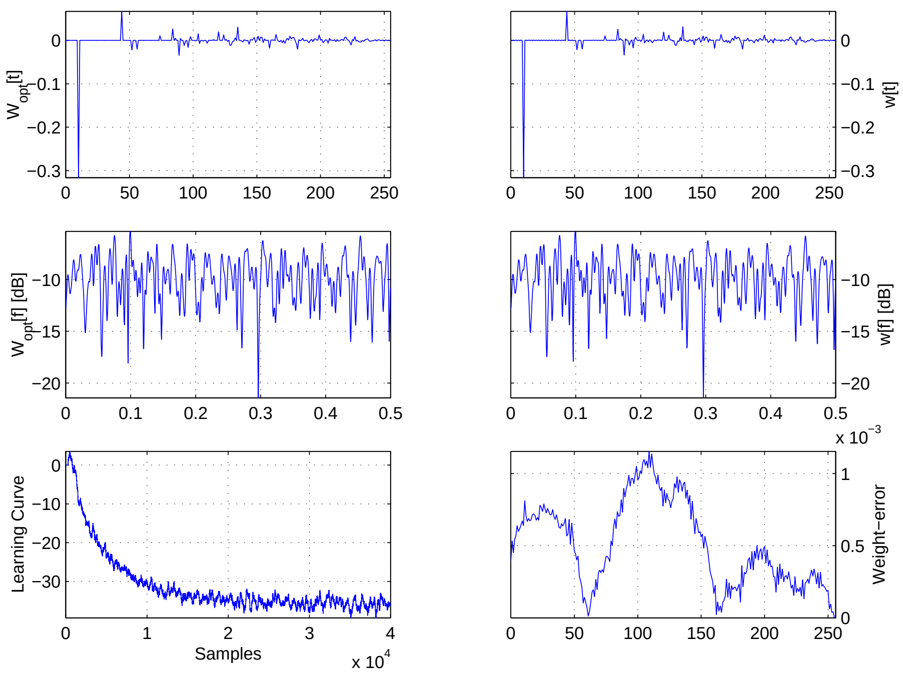

Figure 10.12: Performance of the FDFXLMS algorithm.

Audio Files

The following files demonstrate the performance of the ADJLMS algorithm.

`wavin\scinwn.wav`	Reference (input) signal
`wavin\scdwn256.wav`	microphone signal with controller OFF.
`wavout\fdfxlms.wav`	microphone signal with controller ON.

See Also

INIT_ FDFXLMS, ASPTFDFXLMS, ASPTMCFDFXLMS, ASPTFXLMS, ASPTMCFXLMS.

Reference

[3], Chapter 3 for detailed description of the FDFXLMS, [8] for the overlap-save method, and [9] for frequency domain adaptive filters.

10.7 anvc_ fxlms

Purpose Simulation of a single channel Active Noise and Vibration Control (ANVC) application using an adaptive controller updated according to the FILTERED-X Least Mean Squares (FXLMS) algorithm.

Syntax `anvc_fxlms`

Description The block diagram of the single channel ANVC problem using FXLMS is shown in Fig. 10.13. The primary impulse response p is the impulse response measured between the noise source and the error microphone in a small room. The secondary impulse response s is that between the secondary source and the error microphone. Two sets of transfer functions are provided in the simulation, the simple data set (system 1) and the measured data set (system 2). The description below applies to system 1, with primary impulse response sampled at 8 kHz and truncated to 32 coefficients, and a simple FIR secondary impulse response so that it can be easily experimented with its minimum phase and delay properties. The primary noise at the microphone is stored in the file dfile and the reference signal $x(n)$ (white noise) is stored in the file infile. First the variables for the controller $w(n)$ are creates and initializes using `init_fxlms()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptfxlms()` is called with a new reference sample and a new primary sample to calculate the controller output (control effort) and update the controller coefficients. The sensor signal $e(n)$ is saved in each iteration for later examination.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the controller performance is generated.

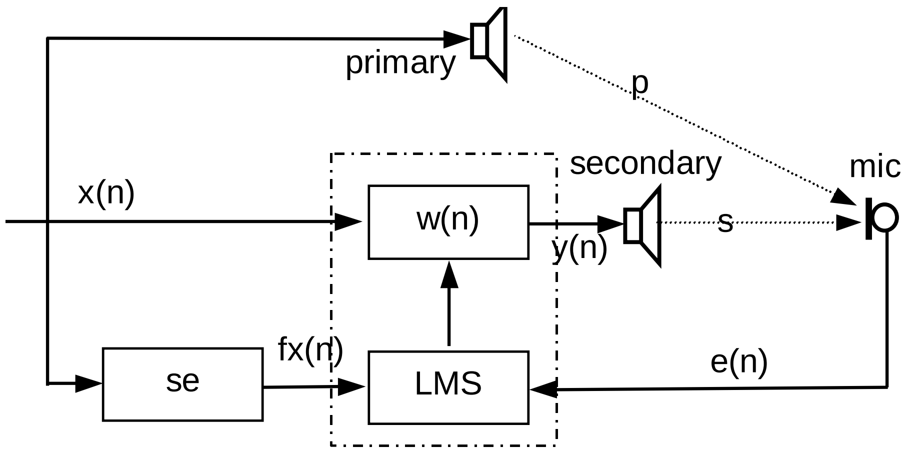

Figure 10.13: Block diagram of a single channel noise cancellation application using the Filtered-x LMS algorithm.

Code

```
clear all;
load .\data\h32;
ph        = h32;                             % Primary IR
sh        = [0; 0.4; 0.3; 0.2; 0.1; 0.05];  % Secondary IR
se        = 0.9*sh;                          % estimated sh
Lph       = length(ph);
Lsh       = length(sh);
L         = 40;                              % controller length
mu        = 0.02;                            % adaptation constant
b         = 0.98;                            % autoregressive pole
infile    = '.\wavin\scinwn.wav';            % reference signal
dfile     = '.\wavin\scdwn32.wav';           % d(n) = conv(x,h32)
outfile   = '.\wavout\fxlms.wav';            % sensor signal

[w,x,y,d,e,p,fx] = init_fxlms(L,sh,se);     % Init FXLMS
inSize           = wavread(infile, 'size'); % get input data size
[xn,inFs,inBits] = wavread(infile);         % get sampling details
dSize            = wavread(dfile, 'size');  % data size
[dn,inFs,dBits]  = wavread(dfile);          % get sampling details
inSize           = max(min(inSize,dSize));  % Max. samples index
E                = init_ipwin(inSize);      % Initialize IPWIN
out              = zeros(size(xn));         % sensor signal

%% Processing Loop
for (m=1:inSize)

   % update the delay line
   x = [xn(m,:); x(1:end-1,:)];

   % update the controller
   [w,y,e,p,fx]  = asptfxlms(w,x,y,sh,se,dn(m),fx,p, mu,b);

   % save the last sensor sample
   out(m) = e(1);

   % IPWIN handling section, see UPDATE\_~IPWIN for more details
   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e(1),dn(m), 'a', w, ph,sh);
   % handle the Stop button
   while (stop  ~= 0), stop  = getStop; end;
   % handle the Break button
   if (brk), plot_anvc(w,ph,sh,E); break; end;

end;

plot_anvc(w,ph,sh,E);                       % generate ANVC plot
wavwrite(out(1:m),inFs,inBits,outfile);    % write to audio file
```

Results Running the above script will produce the graph shown in Fig. 10.14. This figure shows the optimum (Wiener) solution of the problem at hand and compares that with the solution approached by the adaptive controller. The top left and top right panels show the impulse responses of the optimal solution and the adaptive controller, respectively. The middle left and middle right panels show the frequency responses of the optimal solution and the adaptive controller, respectively. The bottom left panel shows the learning curve for the adaptive controller, and the bottom right shows the difference between the optimal solution coefficients and the adaptive controller coefficients.

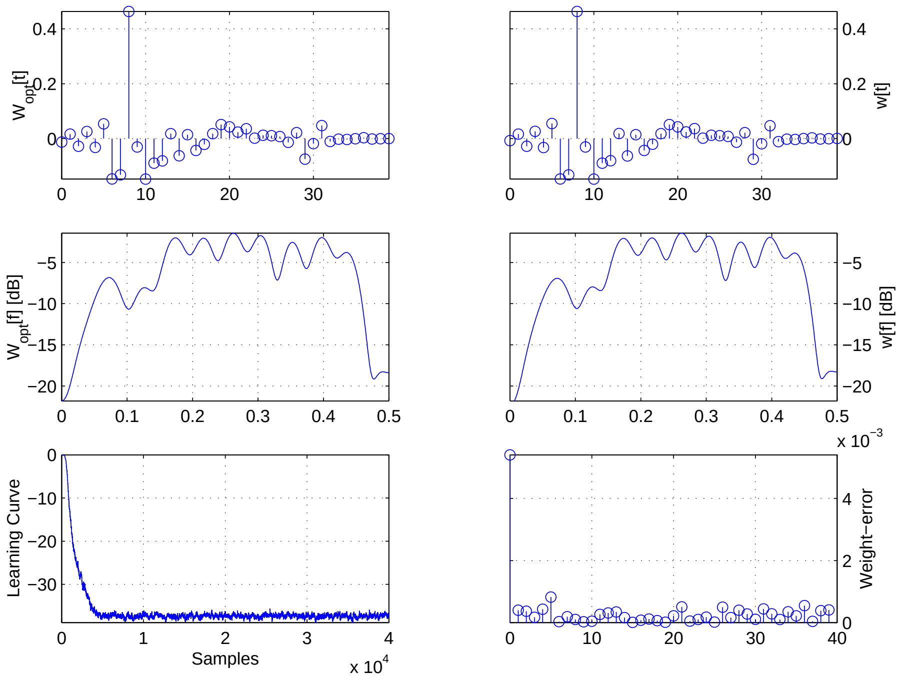

Figure 10.14: Performance of the FXLMS algorithm.

Audio Files The following files demonstrate the performance of the FXLMS algorithm.

`wavin\scinwn.wav`	Reference (input) signal
`wavin\scdwn32.wav`	microphone signal with controller OFF.
`wavout\fxlms.wav`	microphone signal with controller ON.

See Also INIT_ FXLMS, ASPTFXLMS, ASPTFDFXLMS, ASPTMCFDFXLMS.

Reference [3], Chapter 3.

10.8 anvc_ mcadjlms

Purpose Simulation of a multichannel Active Noise and Vibration Control (ANVC) application using a matrix of adaptive controllers updated using the Multi-Channel ADJoint Least Mean Squares (MCADJLMS) algorithm.

Syntax `anvc_mcadjlms`

Description The block diagram of a multichannel ANVC problem using MCADJLMS is shown in Fig. 10.15. The matrix of primary impulse responses p is the impulse responses measured between the noise sources and the error microphones in a small room. The matrix of secondary impulse responses s is that between the secondary sources and the error microphones. Two sets of transfer functions are provided in the simulation, the simple data set (system 1) and the measured data set (system 2). The description below applies to system 1, with two primary sources, two secondary sources, and two error microphones. The primary and secondary impulse responses are sampled at 8 kHz and truncated to 32 coefficients. The noise from the noise sources measured at the error microphones is stored in the file dfile and the reference signals $x(n)$ (white noise) is stored in the file infile. First the variables for the controller $w(n)$ are creates and initializes using `init_mcadjlms`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptmcadjlms` is called with a new reference vector and a new primary vector to calculate the controllers' outputs (control effort) and update the controllers' coefficients. The sensor signals $e(n)$ are also saved in each iteration for later examination. This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the controller performance is generated.

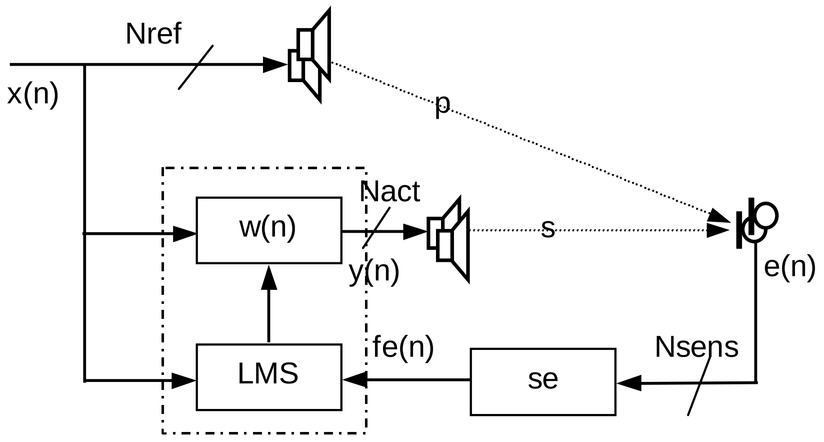

Figure 10.15: Block diagram of a multichannel noise cancellation application using the Multichannel Adjoint-LMS algorithm.

Code

```
% System transfer functions, Nref=2, Nact=2, Nsens=2.
load .\data\ph22_32.mat;      % Primary impulse responses
load .\data\sh22_32.mat;      % Secondary impulse responses
sh   = sh22_32;
ph   = ph22_32;
se   = 0.98 * sh;             %estimated sh22

%% Simulation parameters
[Lph,Nref,Nsens]= size(ph);   % Primary TF dimension
[Lsh,Nact,Ns2]  = size(sh);   % Secondary TF dimension
if (Nsens ~= Ns2), error('Dimension mismatch'); end
L    = Lph;                   % length of controller w
mu   = .1/L;                  % adaptation constant
b    = 0.98;                  % autoregressive pole

% Reference and primary signals
infile  = '.\wavin\mcin222wn.wav';  % x(n), white noise, 2-ch
dfile   = '.\wavin\mcd222wn.wav';   % d(n), 2-ch channels
outfile = '.\wavout\mcadjlms.wav';  % noise level at microphone

% initialize MCADJLMS
[w,x,y,d,e,p]     = init_mcadjlms(L,Nref,Nact,Nsens,sh,se);
inSize            = wavread(infile, 'size'); % get input data size
[xn,inFs,inBits] = wavread(infile);         % get sampling details
dSize             = wavread(dfile, 'size');  % data size
[dn,inFs,dBits]  = wavread(dfile);          % get sampling details
inSize            = max(min(inSize,dSize));  % Max. samples index
E                 = init_ipwin(inSize,Nsens);% Initialize IPWIN
out               = zeros(size(xn));         % sensor signal

%% Processing Loop
for (m=1:inSize)
   x = [xn(m,:);x(1:end-1,:) ];   % update the delay line

   % call MCADJLMS to calculate the output and update the filters
   [w,y,e,p] = asptmcadjlms(w,x,e,y,sh,se,dn(m,:),p,mu,b);
   out(m,:)  = e(1,:);            % save last error

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e(1,:),dn(m,:),'a',w,ph,sh);
   % handle the Stop button
   while (stop ~= 0), stop  = getStop; end;
   % handle the Break button
   if (brk), plot_anvc(w,ph,sh,E,1,1); break; end;
end;

plot_anvc(w,ph,sh,E,1,1);               % generate ANVC plot
wavwrite(out(1:m),inFs,inBits,outfile); % write to audio file
```

Results

Running the above script will produce the graph shown in Fig. 10.16. This figure shows the optimum (Wiener) solution of the problem at hand and compares that with the solution approached by the adaptive controller. Only the controller between the first reference and first actuator is shown in the graph. You can control which of the [2 × 2] filters to view by using the last two input arguments of `plot_anvc()`. The top left and top right panels in Fig. 10.16 show the impulse responses of the optimal solution and the adaptive controller, respectively. The middle left and middle right panels show the frequency responses of the optimal solution and the adaptive controller, respectively. The bottom left panel shows the learning curve for the adaptive controller, and the bottom right shows the difference between the optimal solution coefficients and the adaptive controller coefficients.

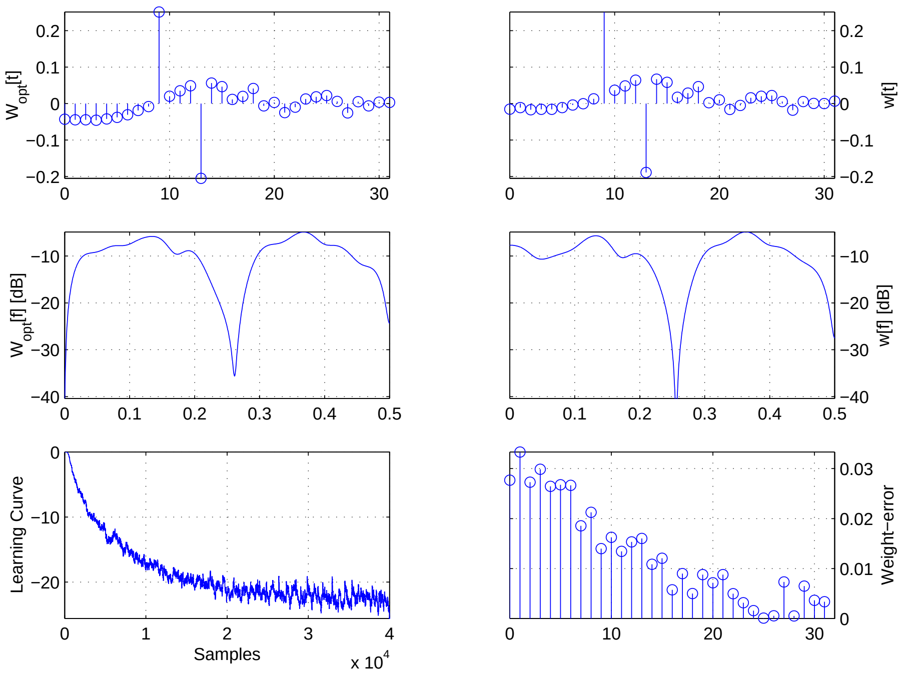

Figure 10.16: Performance of the MCADJLMS algorithm.

Audio Files

The following files demonstrate the performance of the MCADJLMS algorithm in the application mentioned above.

`wavin\mcin222wn.wav`	Reference (input) signals
`wavin\mcd222wn.wav`	microphones' signals with controller OFF.
`wavout\mcadjlms.wav`	microphones' signals with controller ON.

See Also

INIT_ MCADJLMS, ASPTMCADJLMS, ASPTADJLMS, ASPTFDADJLMS, ASPTMCFDADJLMS.

Reference

[3], Chapter 3.

10.9 anvc_ mcfdadjlms

Purpose Simulation of a Multichannel Active Noise and Vibration Control (ANVC) application using an adaptive controller updated according to the Multi Channel Frequency Domain ADJoint Least Mean Squares (MCFDADJLMS) algorithm.

Syntax `anvc_mcfdadjlms`

Description The block diagram of the multichannel ANVC problem using MCFDADJLMS is shown in Fig. 10.17. The matrix of primary impulse responses p is the impulse responses measured between the noise sources and the error microphones in a small room. The matrix of secondary impulse responses s is that between the secondary sources and the error microphones. Two sets of transfer functions are provided in the simulation, the simple data set (system 1) and the measured data set (system 2). The description below applies to system 2, with two primary sources, three secondary sources, and two error microphones. The primary and secondary impulse responses are sampled at 8 kHz and truncated to 128 coefficients. The noise from the noise sources measured at the error microphones is stored in the file dfile and the reference signals $x(n)$ (white noise) is stored in the file infile. First the variables for the controller $w(n)$ are creates and initializes using `init_mcfdadjlms`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptmcfdadjlms` is called with a new block of reference samples and a new block of primary noise samples to calculate the controllers' outputs (control effort) and update the controllers' coefficients. The sensor signals $e(n)$ are also saved in each iteration for later examination.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the controller performance is generated.

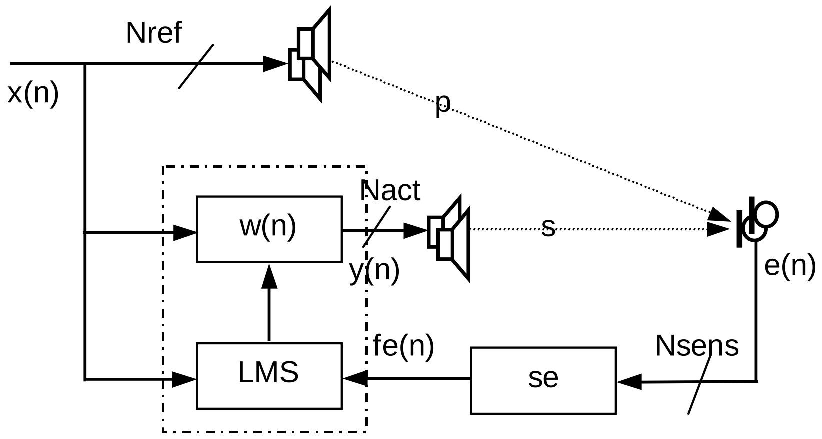

Figure 10.17: Block diagram of a multichannel noise cancellation application using the Multi Channel Frequency Domain Adjoint LMS algorithm.

Code

```
% System transfer functions, Nref=2, Nact=3, Nsens=2.
load .\data\p232_128;               % Primary impulse responses
load .\data\s232_128;               % Secondary impulse responses
sh   = sh232;
ph   = ph232;
se   = 0.9*sh;                      % estimated sh

%% Simulation parameters
[Lph,Nref,Nsens] = size(ph);        % Primary TF dimension
[Lsh,Nact,Nsens] = size(sh);        % Secondary TF dimension
NC  = Lph;                          % length of controller w
NL  = NC;                           % block length
c   = 1;                            % constrain controller
mu  = .01/NC;                       % adaptation constant
b   = 0.99;                         % autoregressive pole

infile  = '.\wavin\mcin232wn.wav';   % x(n), white noise, 2 channels
dfile   = '.\wavin\mcd232wn.wav';    % d(n), Nsens channels
outfile = '.\wavout\mcfdfxlms.wav';  % noise level at microphone

% Initialize MCFDFXLMS algorithm and data files
[NB,W,w,x,y,d,e,p,S,SE,yF,fxF] = init_mcfdfxlms(NC,NL,Nref,...
                                 Nact,Nsens,sh,se);
inSize          = wavread(infile, 'size'); % input data size
[xn,inFs,inBits] = wavread(infile,NL);      % input properties
dSize           = wavread(dfile, 'size');  % primary data size
[dn,inFs,dBits]  = wavread(dfile,NL);       % primary properties
inSize          = max(min(inSize,dSize));  % samples to process
E               = init_ipwin(inSize,Nsens);% Initialize IPWIN
out             = zeros(size(dn));         % microphone signal

%% Processing Loop
for (m=1:NL:inSize-NL)
   % read NL samples from input and primary and scale the block
   xn = wavread(infile,[m,m+NL-1]);
   dn = wavread(dfile, [m,m+NL-1]);

   % Call ASPTMCFDFXLMS to calculate the output and update coef.
   [W,w,x,y,e,p,yF,fxF] = asptmcfdfxlms(NC,W,x,xn,dn,...
                          yF,fxF,S,SE,p,mu,b,c);
   out(m:m+NL-1,:) = e;   % save error block

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,dn, 'a', w, ph, sh);
   % handle Stop button
   while (stop~=0), stop = getStop; end;
   % handle Break button
if (brk), plot_anvc(w,ph,sh,E,2,1); break; end;
end;

plot_anvc(w,ph,sh,E,2,1);          % performance plots
wavwrite(out(1:m,:),inFs,inBits,outfile); % save mic signal
```

Results

Running the above script will produce the graph shown in Fig. 10.18. This figure shows the optimum (Wiener) solution of the problem at hand and compares that with the solution approached by the adaptive controller. Only the controller between the second reference and first actuator is shown in the graph. You can control which of the $[2 \times 3]$ filters to view by using the last two input arguments of `plot_anvc()`. The top left and top right panels in Fig. 10.18 show the impulse responses of the optimal solution and the adaptive controller, respectively. The middle left and middle right panels show the frequency responses of the optimal solution and the adaptive controller, respectively. The bottom left panel shows the learning curve for the adaptive controller, and the bottom right shows the difference between the optimal solution coefficients and the adaptive controller coefficients.

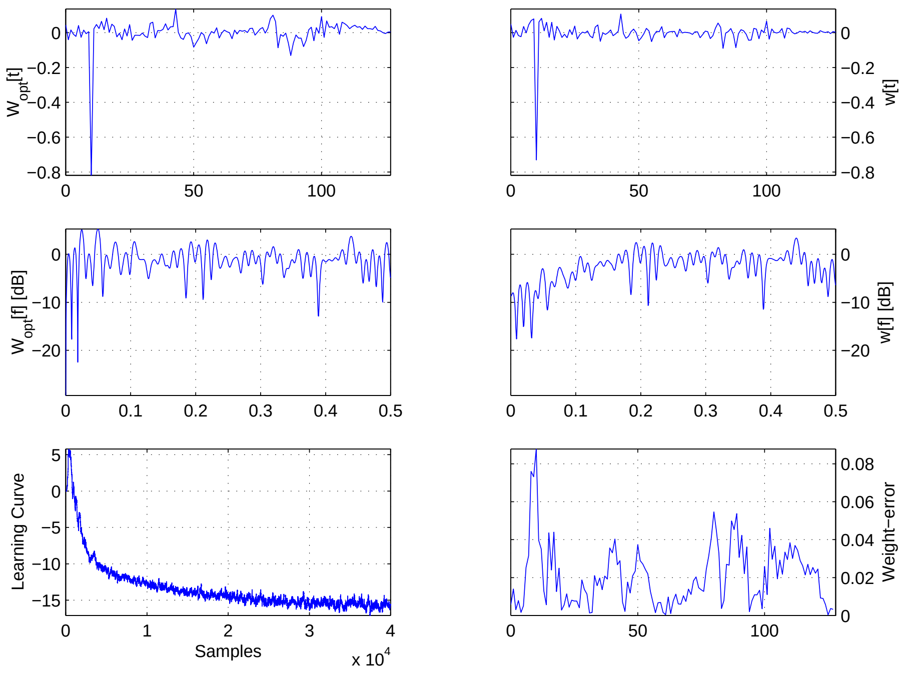

Figure 10.18: Performance of the MCFDADJLMS algorithm.

Audio Files

The following files demonstrate the performance of the MCFDADJLMS algorithm in the application mentioned above.

`wavin\mcin232wn.wav`	Reference (input) signal
`wavin\mcd232wn.wav`	microphone signal with controller OFF.
`wavout\mcfdadjlms.wav`	microphone signal with controller ON.

See Also

INIT_ MCFDADJLMS, ASPTMCFDFXLMS, ASPTFDADJLMS, ASPTMCFXLMS.

Reference

[3], Chapter 3 for detailed description of the MCFDFXLMS, [8] for the overlap-save method, and [9] for frequency domain adaptive filters.

10.10 anvc_ mcfdfxlms

Purpose Simulation of a multichannel Active Noise and Vibration Control (ANVC) application using an adaptive controller updated according to the Multi Channel Frequency Domain Filtered-X Least Mean Squares (MCFDFXLMS) algorithm.

Syntax `anvc_mcfdfxlms`

Description The block diagram of the multichannel ANVC problem using MCFDFXLMS is shown in Fig. 10.19. The matrix of primary impulse responses p is the impulse responses measured between the noise sources and the error microphones in a small room. The matrix of secondary impulse responses s is that between the secondary sources and the error microphones. Two sets of transfer functions are provided in the simulation, the simple data set (system 1) and the measured data set (system 2). The description below applies to system 2, with two primary sources, three secondary sources, and two error microphones. The primary and secondary impulse responses are sampled at 8 kHz and truncated to 128 coefficients. The noise from the noise sources measured at the error microphones is stored in the file dfile and the reference signals $x(n)$ (white noise) is stored in the file infile. First the variables for the controller $w(n)$ are creates and initializes using `init_mcfdfxlms`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptmcfdfxlms` is called with a new block of reference samples and a new block of primary noise samples to calculate the controllers' outputs (control effort) and update the controllers' coefficients. The sensor signals $e(n)$ are also saved in each iteration for later examination.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the controller performance is generated.

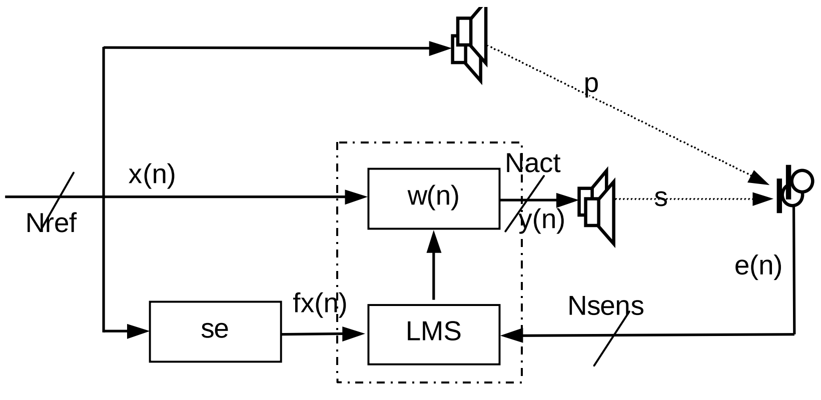

Figure 10.19: Block diagram of a multichannel noise cancellation application using the Multi Channel Frequency Domain Filtered-X LMS algorithm.

Code

```
% System transfer functions, Nref=2, Nact=3, Nsens=2.
load .\data\p232_128;               % Primary impulse responses
load .\data\s232_128;               % Secondary impulse responses
sh   = sh232;
ph   = ph232;
se   = 0.9*sh;                      % estimated sh

%% Simulation parameters
[Lph,Nref,Nsens] = size(ph);        % Primary TF dimension
[Lsh,Nact,Nsens] = size(sh);        % Secondary TF dimension
NC  = Lph;                          % length of controller w
NL  = NC;                           % block length
c   = 1;                            % constrain controller
mu  = .01/NC;                       % adaptation constant
b   = 0.99;                         % autoregressive pole

infile  = '.\wavin\mcin232wn.wav';   % x(n), white noise, 2 channels
dfile   = '.\wavin\mcd232wn.wav';    % d(n), Nsens channels
outfile = '.\wavout\mcfdfxlms.wav';  % noise level at microphone

% Initialize MCFDFXLMS algorithm and data files
[NB,W,w,x,y,d,e,p,S,SE,yF,fxF] = init_mcfdfxlms(NC,NL,Nref,...
                                  Nact,Nsens,sh,se);
inSize           = wavread(infile, 'size'); % input data size
[xn,inFs,inBits] = wavread(infile,NL);      % input properties
dSize            = wavread(dfile, 'size');  % primary data size
[dn,inFs,dBits]  = wavread(dfile,NL);       % primary properties
inSize           = max(min(inSize,dSize));  % samples to process
E                = init_ipwin(inSize,Nsens);% Initialize IPWIN
out              = zeros(size(dn));         % microphone signal

%% Processing Loop
for (m=1:NL:inSize-NL)
   % read NL samples from input and primary and scale the block
   xn = wavread(infile,[m,m+NL-1]);
   dn = wavread(dfile,[m,m+NL-1]);

   % Call ASPTFDFXLMS to calculate the output and update coef.
   [W,w,x,y,e,p,yF,fxF] = asptmcfdfxlms(NC,W,x,xn,dn,...
                           yF,fxF,S,SE,p,mu,b,c);
   out(m:m+NL-1,:) = 2^-(dBits-1) * e;    % save error block

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,dn, 'a', w, ph, sh);
   % handle Stop button
while (stop~=0), stop = getStop; end;
   % handle Break button
if (brk), plot_anvc(w,ph,sh,E,1,2); break; end;
end;

plot_anvc(w,ph,sh,E,2,2);                 % performance plots
wavwrite(out(1:m,:),inFs,inBits,outfile); % save mic signal
```

Results

Running the above script will produce the graph shown in Fig. 10.20. This figure shows the optimum (Wiener) solution of the problem at hand and compares that with the solution approached by the adaptive controller. Only the controller between the second reference and first actuator is shown in the graph. You can control which of the [2 × 3] filters to view by using the last two input arguments of `plot_anvc()`. The top left and top right panels in Fig. 10.20 show the impulse responses of the optimal solution and the adaptive controller, respectively. The middle left and middle right panels show the frequency responses of the optimal solution and the adaptive controller, respectively. The bottom left panel shows the learning curve for the adaptive controller, and the bottom right shows the difference between the optimal solution coefficients and the adaptive controller coefficients.

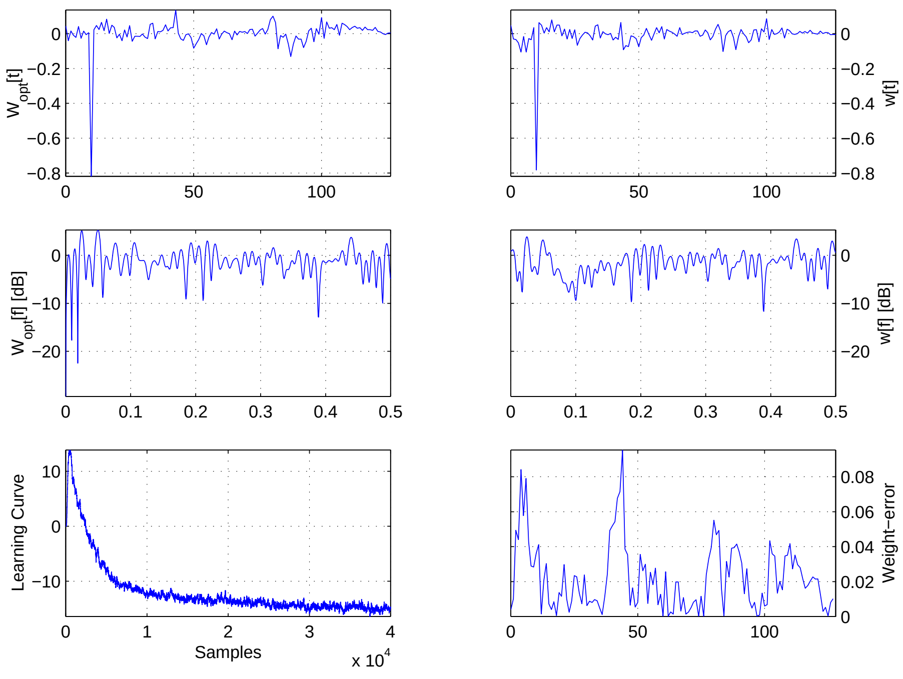

Figure 10.20: Performance of the MCFDFXLMS algorithm.

Audio Files

The following files demonstrate the performance of the MCFDFXLMS algorithm in the application mentioned above.

`wavin\mcin232wn.wav`	Reference (input) signal
`wavin\mcd232wn.wav`	microphone signal with controller OFF.
`wavout\mcfdfxlms.wav`	microphone signal with controller ON.

See Also

INIT_ MCFDFXLMS, ASPTMCFDFXLMS, ASPTFDFXLMS, ASPTFXLMS, ASPTMCFXLMS.

Reference

[3], Chapter 3 for detailed description of the MCFDFXLMS, [8] for the overlap-save method, and [9] for frequency domain adaptive filters.

10.11 anvc_ mcfxlms

Purpose Simulation of a multichannel Active Noise and Vibration Control (ANVC) application using a matrix of adaptive controllers updated using the MultChannel Filtered-X Least Mean Squares (MCFXLMS) algorithm, also known as the Multiple Error Filtered-X LMS (MEFXLMS).

Syntax `anvc_mcfxlms`

Description The block diagram of a multichannel ANVC problem using MCFXLMS is shown in Fig. 10.21. The matrix of primary impulse responses p is the impulse responses measured between the noise sources and the error microphones in a small room. The matrix of secondary impulse responses s is that between the secondary sources and the error microphones. Two sets of transfer functions are provided in the simulation, the simple data set (system 1) and the measured data set (system 2). The description below applies to system 1, with two primary sources, two secondary sources, and two error microphones. The primary and secondary impulse responses are sampled at 8 kHz and truncated to 32 coefficients. The noise from the noise sources measured at the error microphones is stored in the file dfile and the reference signals $x(n)$ (white noise) is stored in the file infile. First the variables for the controller $w(n)$ are creates and initializes using `init_mcfxlms`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptmcfxlms` is called with a new reference vector and a new primary vector to calculate the controllers' outputs (control effort) and update the controllers' coefficients. The sensor signals $e(n)$ are also saved in each iteration for later examination. This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the array performance is generated.

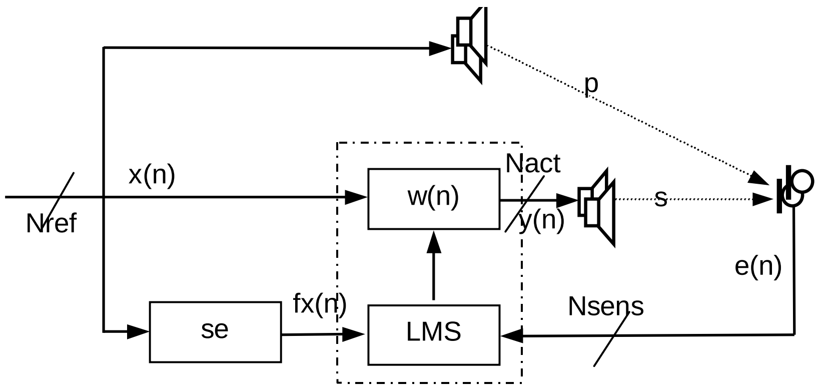

Figure 10.21: Block diagram of a multichannel noise cancellation application using the Multichannel Filtered-X LMS algorithm.

Code

```
% System transfer functions, Nref=2, Nact=2, Nsens=2.
load .\data\ph22_32.mat;      % Primary impulse responses
load .\data\sh22_32.mat;      % Secondary impulse responses
sh   = sh22_32;
ph   = ph22_32;
se   = 0.98 * sh;             %estimated sh22

%% Simulation parameters
[Lph,Nref,Nsens]= size(ph);   % Primary TF dimension
[Lsh,Nact,Ns2]  = size(sh);   % Secondary TF dimension
if (Nsens ~= Ns2), error('Dimension mismatch'); end
L    = Lph;                   % length of controller w
mu   = .1/L;                  % adaptation constant
b    = 0.98;                  % autoregressive pole

% Reference and primary signals
infile   = '.\wavin\mcin22wn.wav';  % x(n), white noise, 2-ch
dfile    = '.\wavin\mcd22wn.wav';   % d(n), 2-ch channels
outfile  = '.\wavout\mcfxlms.wav';  % noise level at microphone

% initialize MCFXLMS
[w,x,y,d,e,p,fx] = init_mcfxlms(L,Nref,Nact,Nsens,sh22,se);
inSize           = wavread(infile, 'size'); % get input data size
[xn,inFs,inBits] = wavread(infile);         % get sampling details
dSize            = wavread(dfile, 'size');  % data size
[dn,inFs,dBits]  = wavread(dfile);          % get sampling details
inSize           = max(min(inSize,dSize));  % Max. samples index
E                = init_ipwin(inSize,Nsens);% Initialize IPWIN
out              = zeros(size(xn));         % sensor signal

%% Processing Loop
for (m=1:inSize)
   x = [xn(m,:);x(1:end-1,:) ];   % update the delay line

   % call MCFXLMS to calculate the output and update the filters
   [w,y,e,p,fx] = asptmcfxlms(w,x,y,sh22,se,dn(m,:),fx,p,mu,b);
   out(m,:) = e(1,:);              % save last error

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e(1,:),dn(m,:),'a',w,ph22,sh22);
   % handle the Stop button
   while (stop ~= 0), stop  = getStop; end;
   % handle the Break button
   if (brk), plot_anvc(w,ph,sh,E,1,1); break; end;
end;

plot_anvc(w,ph,sh,E,1,1);                % generate ANVC plot
wavwrite(out(1:m),inFs,inBits,outfile);   % write to audio file
```

Results

Running the above script will produce the graph shown in Fig. 10.22. This figure shows the optimum (Wiener) solution of the problem at hand and compares that with the solution approached by the adaptive controller. Only the controller between the first reference and first actuator is shown in the graph. You can control which of the $[2 \times 2]$ filters to view by using the last two input arguments of `plot_anvc()`. The top left and top right panels in Fig. 10.22 show the impulse responses of the optimal solution and the adaptive controller, respectively. The middle left and middle right panels show the frequency responses of the optimal solution and the adaptive controller, respectively. The bottom left panel shows the learning curve for the adaptive controller, and the bottom right shows the difference between the optimal solution coefficients and the adaptive controller coefficients.

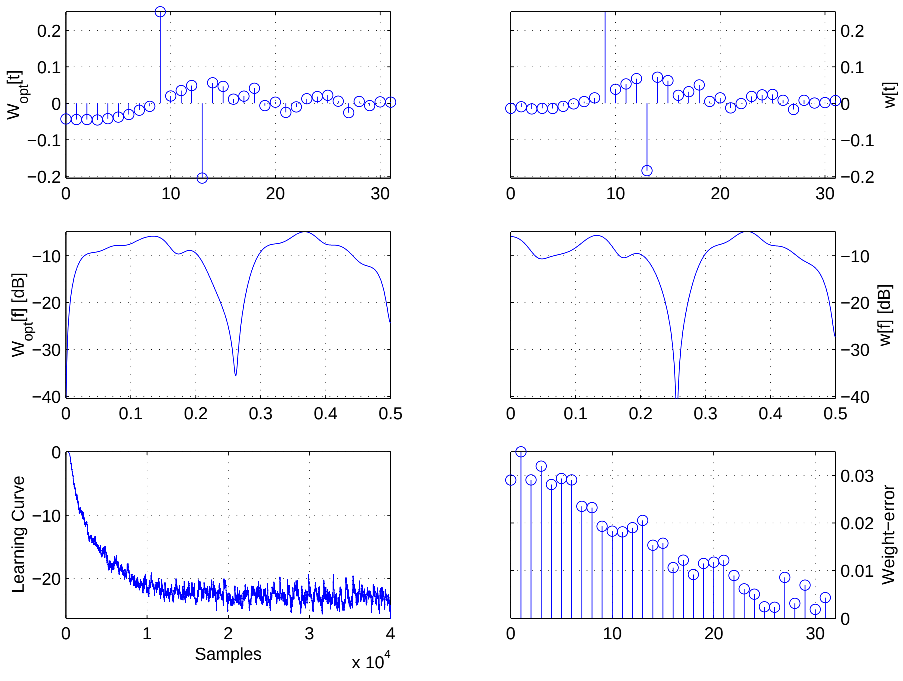

Figure 10.22: Performance of the MCFXLMS algorithm.

Audio Files

The following files demonstrate the performance of the MCFXLMS algorithm in the application mentioned above.

`wavin\mcin22wn.wav`	Reference (input) signals
`wavin\mcd22wn.wav`	microphones' signals with controller OFF.
`wavout\mcfxlms.wav`	microphones' signals with controller ON.

See Also

INIT_ MCFXLMS, ASPTMCFXLMS, ASPTFXLMS, ASPTFDFXLMS, ASPTMCFDFXLMS.

Reference

[3], Chapter 3.

10.12 beambb_ lclms

Purpose Simulation of a beam former application using an adaptive array with the adaptive coefficients adjusted using the LCLMS algorithms. Signals are assumed to be complex and at the baseband frequency (modulated carrier gone through a phase quadrature demodulator for instance).

Syntax `beambb_lclms`

Description The block diagram of the adaptive array simulated in this application is shown in Fig. 10.23. The array is composed of M omnidirectional (equally sensitive in all directions) sensors. All array elements receive narrow-band incident signals that include one or more jammer at the same center frequency ω_c. Assuming that the received signals has been demodulated to baseband frequencies, complex signal processing can be used at this baseband frequency, which allows using much lower sampling frequency and therefore reduces the computation complexity, compared to processing at the RF or IF frequency. At baseband frequency, complete control can be achieved using only one complex coefficient in each branch to control the amplitude and the phase. Since no desired signal is used in the array shown in Fig. 10.23, the filter will converge to the trivial solution $\underline{\mathbf{w}} = \underline{\mathbf{0}}$, unless another constraint is imposed on the adaptation process. The Linearly Constrained LMS is used in such applications to adapt the filter coefficients under the constraint $\underline{\mathbf{c}}^H \underline{\mathbf{w}} = a$. The vector $\underline{\mathbf{c}}$ is used to define the look direction of the array; the direction at which the main lobe is obtained which should coincide with the incident direction of the useful signal.
`beambb_lclms` first sets the array parameters, and then creates and initializes an adaptive filter of length M, where M is the number of array elements, by calling `init_lclms()`. The adaptive filter here is an adaptive linear combiner with complex input signals equal to those received by the array sensors after demodulation to the base band frequency. A processing loop is then started, in each iteration in this loop, `asptlclms()` is called with a new set of sensor samples to calculate the filter output, error signal, and update the constrained filter coefficients. No desired signal is used here, and the desired is always set to zero.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the controller performance is generated.

Code

```
iter   = 5000;                  % samples to process
c      = 3450;                  % propagation speed
fc     = 40000;                 % Carrier frequency
Wc     = 2*pi*fc;               % Carrier radian freq
lambda = 2*pi*c/Wc;             % Carrier wave length
Fs     = 100000;                % sampling frequency
T      = 1/Fs;                  % sampling period
Wo     = Wc * T;                % sampled carrier freq
```

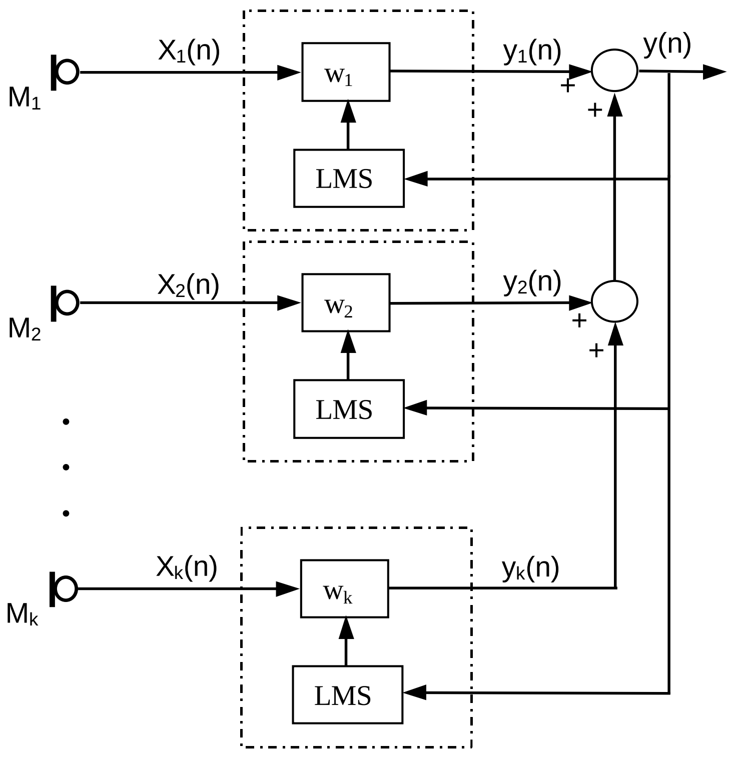

Figure 10.23: Block diagram of an adaptive array using the Linearly Constrained LMS algorithm.

```
Po     = [80 0 -45];              % arrival angles [deg]
Avar   = [1 1 1];                 % variance of signals
M      = length(Po);              % # array elements
L      = lambda/2 * (1:M-1);      % distances bet. sensors
Po     = pi*Po/180;               % arrival angles [rad]
Do     = (Wc/c)*(L' * sin(Po));   % phase shifts [rad]

ph     = 2*pi*rand(iter,M);       %random phase
A      = repmat(sqrt(Avar),[iter 1]).*randn(iter,M);
mu     = 0.01;                    % step size
E      = init_ipwin(iter,1);      % Initialize IPWIN

% need M filters each has 1 complex coefficient. Desired
% signal here is always d(n) = 0 and the filter is updated
% so that sum(w) = 1 exp(j*0) and will produce a main lobe
% at zero degree, the incident direction of the useful signal.

[w,x,d,y,e] = init_lclms(M);
d = 0;
v = ones(M,1); % constraint vector.
a = 1; % constraint scalar.

for n=1:iter
   % signal at the primary branch + receiver noise
   x(1) = sum(A(n,:) .* exp(j*ph(n,:) ),2) + 1e-3*rand;
```

```
      % signals at the reference branches
      for i=2:M
         r    = 1e-3*rand;     % receiver noise
         x(i) = sum(A(n,:).*exp(j*ph(n,:)-j*Do(i-1,:)),2)+r;
      end

      % calculate output, error, and update coefficients
      [w,y,e]  = asptlclms(x,w,d,mu,v, a);

      % update the Iteration Progress Window
      [E, stop,brk] = update_ipwin(E,e,d, 'b', w, L, Wo, c*T, 1);
   % handle the Stop button
   while (stop ~= 0), stop  = getStop; end;
   % handle the Break button
   if (brk), plot_beam(E, w, L, Wo, c*T,1); break; end;
   end
   plot_beam(E, w, L, Wo, c*T,1);
```

Results Running the above script will produce the sensitivity pattern shown in Fig. 10.24. The sensitivity pattern shows that the array main lobe is indeed at (0°), where the array gain is almost unity. The array, therefore, passes signals received from this direction through while attenuates interferences incident with other angles.

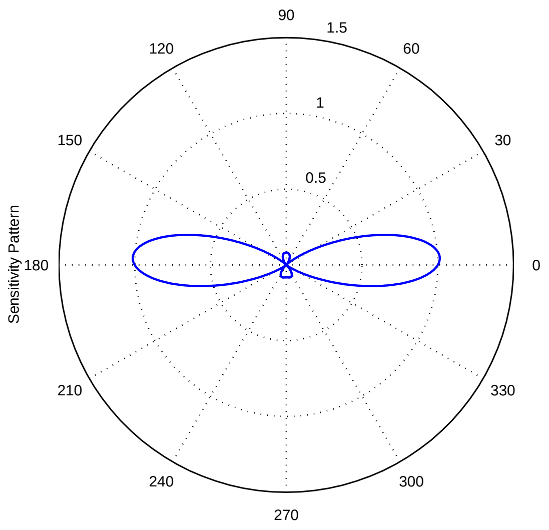

Figure 10.24: Sensitivity pattern of an adaptive array adapted at the base-band frequency using the Linearly Constrained LMS algorithms.

See Also INIT_ LCLMS, ASPTLCLMS, BEAMRF_ LMS.

Reference [11] for an introduction to adaptive array signal processing.

10.13 beamrf_ lms

Purpose Simulation of a sidelobe canceler application using an adaptive array with the array coefficients adjusted using the LMS algorithms. The array will produce a spatial notch in the directions of the strong signals (interferences), where all signals are assumed real and narrow-band.

Syntax `beamrf_lms`

Description The block diagram of an adaptive array functioning as a sidelobe canceler is shown in Fig. 10.25. The array is composed of two omnidirectional (equally sensitive in all directions) sensors. The two array elements receive narrow-band (modulated signals at RF frequency for instance) incident signals that include one signal and one jammer at the same center frequency ω_c. At this frequency, complete control is achieved using only two coefficients one to control the amplitude and the other to control the phase. In Fig. 10.25 this is implemented by two coefficients with input signals at 90° phase difference. In this arrangement, the adaptive filter weights are completely controlled by the signal incident with higher power [11]. Assuming that the jammer has higher power than the useful signal, after convergence, the adaptive filter output will contain a component close to the jammer component in the primary signal, and the array output $e(n)$ will contain only the useful signal. This will cause the sensitivity pattern (the array output power divided by the power of an incident signal at angle spanning the range $0 < \theta < 2\pi$) to have a spatial notch in the direction of arrival of the jammer. This kind of sidelobe canceler rely on the phase difference between the signals received at the different array elements. For two array elements spaced L meters apart, an incident signal with an angle of arrival θ will be received by the elements with phase difference equals to $D = (L \sin(\theta)\omega_c/c)$ rad., where c is the wave propagation speed [m/s] and ω_c is the (analog) center frequency. In general, M array elements are required to cancel $(M-1)$ jammers by duplicating the reference branch in Fig. 10.25.
`beamrf_lms` first sets the array parameters, and then creates and initializes an adaptive filter of length $2(M-1)$, where $(M-1)$ is the number of reference branches, by calling `init_lms()`. The adaptive filter in this application has an adaptive linear combiner structure with input signals equal to those received by the reference sensors and their 90○ phase shifted versions. A processing loop is then started, in each iteration of this loop `asptlms()` is called with a new primary sample and a new set of reference samples to calculate the filter output, the error, and update the filter coefficients.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the array performance is generated.

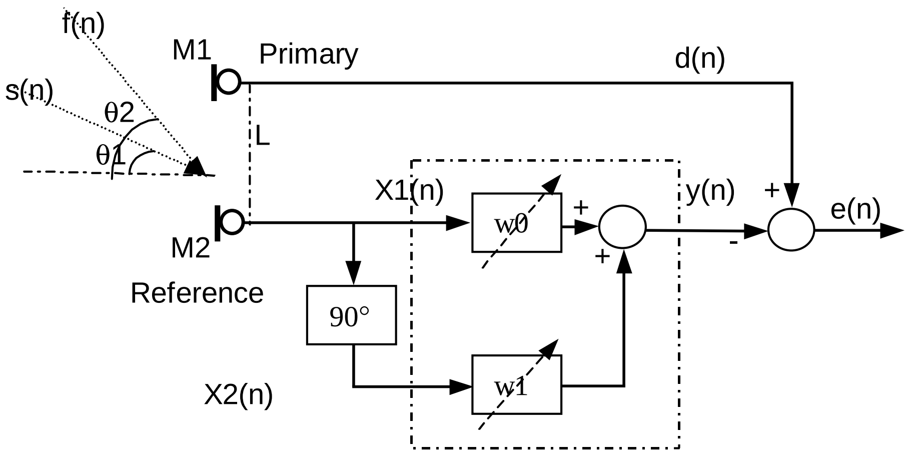

Figure 10.25: Block diagram of an adaptive array functioning as a side-lobe canceler.

Code

```
clear all
rand('seed',12)
iter   = 5000;                     % samples to process
c      = 3450;                     % propagation speed
fc     = 40000;                    % Carrier frequency
Wc     = 2*pi*fc;                  % Carrier radian freq
lambda = 2*pi*c/Wc;                % Carrier wave length
Fs     = 100000;                   % sampling frequency
T      = 1/Fs;                     % sampling period
Wo     = Wc * T;                   % sampled carrier freq

Po     = [ 90 45 0];               % arrival angles [deg]
Avar   = [1 .001 2];               % variance of each sig.
M      = length(Po);               % # array elements
L      = lambda/2 * (1:M-1);       % distances vector
Po     = pi*Po/180;                % arrival angles [rad]
Do     = (Wc/c)*(L' * sin(Po));    % phase shifts [rad]

ph     = 2*pi*rand(iter,M);        %random phase
A      = repmat(sqrt(Avar),[iter 1]).*randn(iter,M);
mu     = 0.05;                     % step size
E      = init_ipwin(iter,1);       % Initialize IPWIN

% need M-1 filters each of 2 coefficients
[w,x,d,y,e] = init_lms(2*(M-1));

for n=1:iter
   % signal at the primary branch + receiver noise
   d = sum(A(n,:) .* cos(n*Wo + ph(n,:) ),2) + 1e-3*rand;;

   % signals at the reference branches
   for i=1:M-1
      r            = 1e-3*rand;     % receiver noise
      x(2*(i-1)+1) = sum(A(n,:).*cos(n*Wo + ph(n,:)-Do(i,:)),2)+r;
      x(2*i)       = sum(A(n,:).*sin(n*Wo+ph(n,:)-Do(i,:)),2) +r;
   end
   % calculate output, error, and update coefficients
   [w,y,e] = asptlms(x,w,d,mu);

   % update the Iteration Progress Window
   [E, stop,brk] = update_ipwin(E,e,d, 'b', w, L, Wo, c*T);
% handle the Stop button
while (stop ~= 0), stop  = getStop; end;
% handle the Break button
if (brk), plot_beam(E, w, L, Wo, c*T); break; end;
end

plot_beam(E, w, L, Wo, c*T);
```

Results Running the above script will produce the graph shown in Fig. 10.26. The left panel of this figure shows the learning curve for the adaptive coefficients and the right panel shows the directivity pattern of the array. Since the two strong signals used in the script are arriving at angles 0∘ and 90∘, the adaptive coefficients have been adjusted to make spatial dips in those two directions.

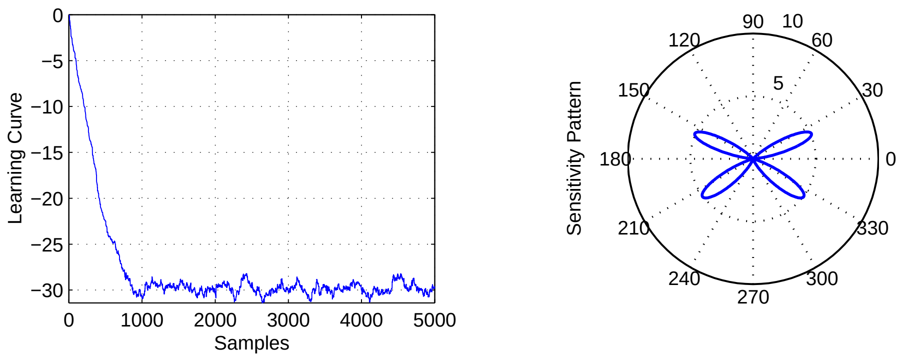

Figure 10.26: Performance of an adaptive sidelobe canceler implemented using the LMS algorithm.

See Also INIT_ LMS, ASPTLMS, ASPTLCLMS, BEAMBB_ LCLMS.

Reference [11] for an introduction to adaptive array signal processing.

10.14 echo_ bfdaf

Purpose Simulation of an acoustic echo canceler application using a transversal adaptive filter updated according to the Block Frequency Domain Adaptive Filter (BFDAF) algorithm.

Syntax `echo_bfdaf`

Description The block diagram of the acoustic echo cancellation problem is shown in Fig. 10.27. The simulation considered here uses a transversal FIR filter for the adjustable filter and the coefficients of the filter are updated in the frequency domain using the BFDAF algorithm. The far-end speech signal $x(n)$ (the speech from the remote speaker) is stored in the file infile. The local speaker is assumed to be silent (listening to the remote speaker and not interrupting). The echo picked by the microphone when playing $x(n)$ through the local loudspeaker is stored in the file dfile. First the variables for the echo canceler $W(f)$ are created and initialized using `init_bfdaf()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptbfdaf()` is called with a new block of samples from the FES and a new block of samples from the NES signals to calculate the filter output block (estimated echo) and update the filter coefficients. The residual signal $e(n)$ is saved in each iteration for later examination.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

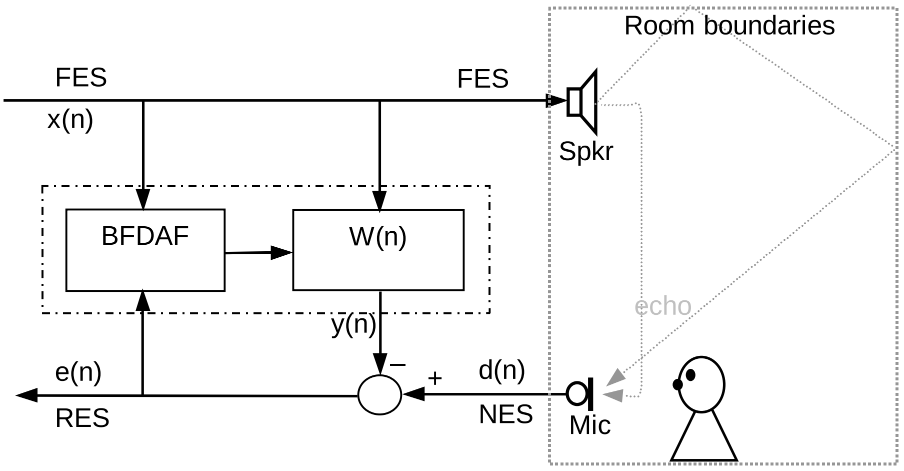

Figure 10.27: Block diagram of an acoustic echo canceler implemented using the block frequency domain adaptive filter (BFDAF).

Code

```
clear all;
infile = '.\wavin\aecfes.wav';          % Far-end speech (FES)
dfile  = '.\wavin\aecnes.wav';          % Near-end speech (NES)
rfile  = '.\wavout\resbfdaf.wav';       % residual signal
M      = 512;                           % adaptive filter length
L      = M;                             % block length
mu     = 0.02/L;                        % adaptation constant
b      = 0.99;                          % autoregressive pole

[W,x,dn,e,y,Px,w]=init_bfdaf(L,M);      % Init BFDAF
[xt,inFs,inBits] = wavread(infile);     % read FES
[dt,inFs,dBits]  = wavread(dfile);      % read NES
inSize  = max(length(xt),length(dt));   % Samples to process
res     = dt;                           % Residual array
E       = init_ipwin(inSize);           % Initialize IPWIN

%% Processing Loop
for (m=1:L:inSize-L)
   % Read and scale a block from FES
   xn = 2^(inBits-1) * xt(m+1:m+L,:);

   % Read and scale a block from NES
   dn = 2^(dBits-1) * dt(m+1:m+L,:);

   % Update the adaptive filter
   [W,x,y,e,Px,w]=asptbfdaf(M,x,xn,dn,W,mu,1,1,b,Px);

   % Scale and store the error (residual)
   res(m+1:m+L) = 2^-(dBits-1)* e;

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,dn,'e',w,dt,res);

   % handle the Stop button
   while (stop  ~= 0), stop  = getStop; end;

   % handle the Break button
   if (brk), plot_echo(w,dt,res); break; end;
end;

wavwrite(res,inFs,inBits,rfile);    % Save the residual
plot_echo(w, dfile, rfile);         % Show results
```

Results

Running the above script will produce the graph shown in Fig. 10.28. The top panel in Fig. 10.28 shows the values taken by the filter coefficients by the end of the simulation (end of input files). The middle panel show the waveforms of the near-end speech signal $d(n)$ and the residual signal $e(n)$ for visual comparison between the echo before and after applying the echo canceler. The bottom panel shows the echo energy decrease in dB achieved by the echo canceler versus time, usually known as the Echo Return Loss Enhancement (ERLE). Note that the ERLE is meaningful only in the time periods where there is echo to be canceled.

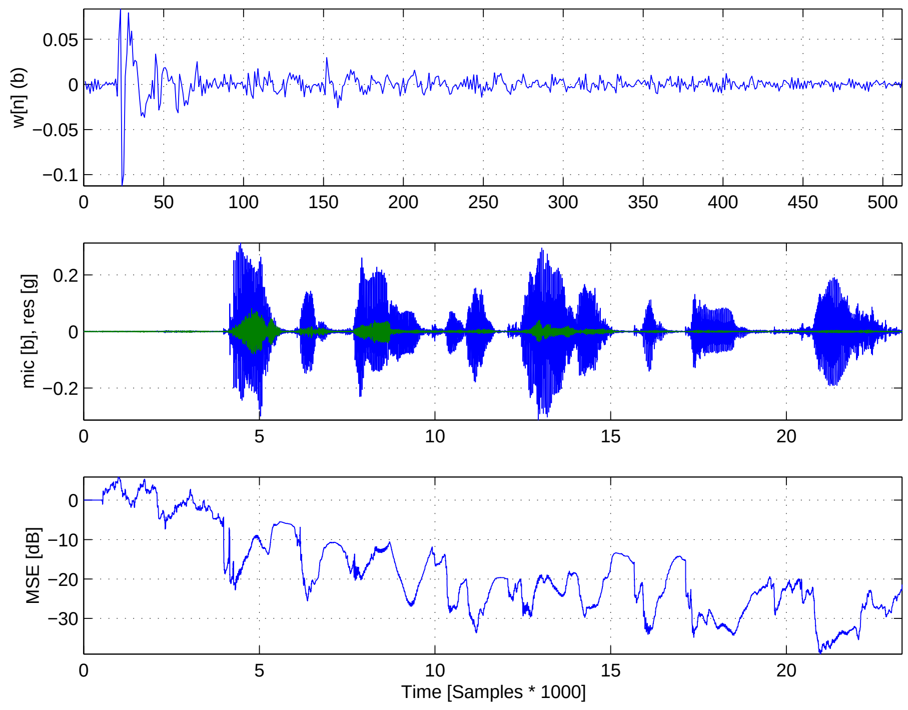

Figure 10.28: Performance of an Acoustic Echo Canceler implemented using the BFDAF algorithm.

Audio Files

The following files demonstrate the performance of the BFDAF algorithm in the echo canceler application mentioned above.

`wavin\aecfes.wav`	far-end speech (input) signal.
`wavin\nesaec.wav`	near-end speech (microphone) signal.
`wavout\resbfdaf.wav`	residual signal (echo canceler output).

See Also

INIT_ BFDAF, ASPTBFDAF, ECHO_ NLMS, ECHO_ PBFDAF.

Reference

[3], Chapter 3 for detailed description of BFDAF, [8] for the overlap-save method, and [9] for frequency domain adaptive filters.

10.15 echo_ leakynlms

Purpose Simulation of an acoustic echo canceler application using a transversal adaptive filter updated according to the Leaky Normalized Least Mean Squares (LEAKYNLMS) algorithm.

Syntax `echo_leakynlms`

Description The block diagram of the acoustic echo cancellation problem is shown in Fig. 10.29. The simulation considered here uses a transversal FIR filter for the adjustable filter and the coefficients of the filter are updated in time domain using the Leaky NLMS algorithm. The far-end speech signal $x(n)$ (the speech from the remote speaker) is stored in the file infile. The local speaker is assumed to be silent (listening to the remote speaker and not interrupting). The echo picked by the microphone when playing $x(n)$ through the local loudspeaker is stored in the file dfile. First, the variables for the echo canceler $w(n)$ are created and initialized using `init_leakynlms()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptleakynlms()` is called with a new sample from the FES and a new sample from the NES signals to calculate the filter output $y(n)$ (estimated echo), filter error $e(n)$, and update the filter coefficients. The residual signal $e(n)$ is saved in each iteration for later examination.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

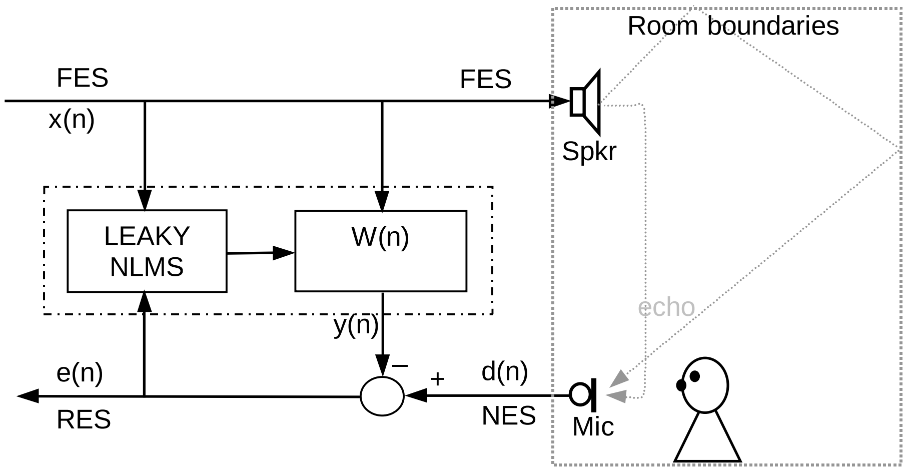

Figure 10.29: Block diagram of an acoustic echo canceler implemented using the Leaky NLMS adaptive filter.

Code

```
clear all;
infile = '.\wavin\aecfes.wav';          % Far-end speech (FES)
dfile  = '.\wavin\aecnes.wav';          % Near-end speech (NES)
rfile  = '.\wavout\resleaky.wav';       % residual signal

M      = 512;                           % adaptive filter length
mu     = 0.2/M;                         % adaptation constant
b      = 0.99;                          % autoregressive pole
a      = 1 - 1e-5;                      % leak factor

[w,x,d,y,e,p]    = init_leakynlms(M);   % Init NLMS
[xn,inFs,inBits] = wavread(infile);     % read FES
[dn,inFs,dBits]  = wavread(dfile);      % read NES
inSize  = max(length(xn),length(dn));   % Samples to process
res     = dn;                           % Residual array
E       = init_ipwin(inSize);           % Initialize IPWIN

fprintf('equivalent noise variance = %f\n', (1 - a)/ (2*mu));

%% Processing Loop
for (m=1:inSize)
   % update the delay line
   x  = [2^(inBits-1) * xn(m); x(1:M-1)];

   % scale the Mic signal
   d  = 2^(inBits-1) *dn(m);

   % call asptleakynlms to update the filter
   [w,y,e,p]= asptleakynlms(x,w,d,mu,a,p,b);

   % save the last residual sample
   res(m) = 2^-(inBits-1)*e;

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,d, 'e', w, dn, res);

   % handle the Stop button
   while (stop  ~= 0), stop  = getStop; end;

   % handle the Break button
   if (brk), plot_echo(w,dn,res); break; end;
end;

wavwrite(res,inFs,inBits,rfile);    % Save the residual
plot_echo(w, dfile, rfile);         % Show results
```

Results Running the above script will produce the graph shown in Fig. 10.30. The top panel in Fig. 10.30 shows the values taken by the filter coefficients by the end of the simulation (end of input files). The middle panel show the waveforms of the near-end speech signal $d(n)$ and the residual signal $e(n)$ for visual comparison between the echo before and after applying the echo canceler. The bottom panel shows the echo energy decrease in dB achieved by the echo canceler versus time, usually known as the Echo Return Loss Enhancement (ERLE). Note that the ERLE is meaningful only in the time periods where there is echo to be canceled.

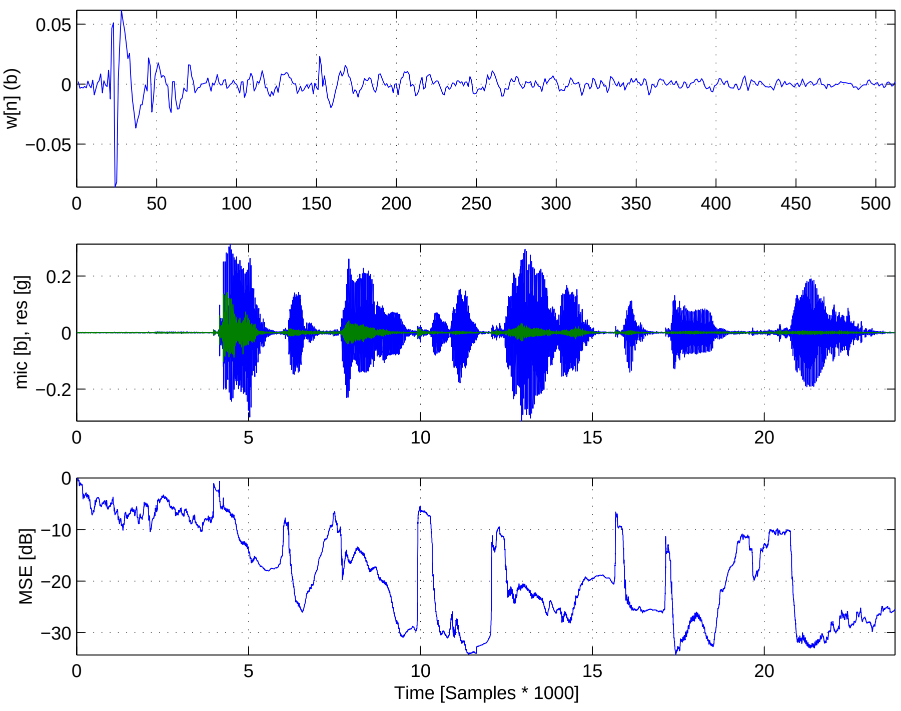

Figure 10.30: Performance of an Acoustic Echo Canceler implemented using the Leaky NLMS adaptive filter.

Audio Files The following files demonstrate the performance of the Leaky NLMS algorithm in the echo canceler application mentioned above.

`wavin\aecfes.wav`	far-end speech (input) signal.
`wavin\nesaec.wav`	near-end speech (microphone) signal.
`wavout\resleaky.wav`	residual signal (echo canceler output).

See Also INIT_ LEAKYNLMS, ASPTLEAKYNLMS.

Reference [11] and [4] for extensive analysis of the NLMS and the steepest-descent search method.

10.16 echo_ nlms

Purpose Simulation of an acoustic echo canceler application using a transversal adaptive filter updated according to the Normalized Least Mean Squares (NLMS) algorithm.

Syntax `echo_nlms`

Description The block diagram of the acoustic echo cancellation problem is shown in Fig. 10.31. The simulation considered here uses a transversal FIR filter for the adjustable filter and the coefficients of the filter are updated in time domain using the NLMS algorithm. The far-end speech signal $x(n)$ (the speech from the remote speaker) is stored in the file infile. The local speaker is assumed to be silent (listening to the remote speaker and not interrupting). The echo picked by the microphone when playing $x(n)$ through the local loudspeaker is stored in the file dfile. First, the variables for the echo canceler $w(n)$ are created and initialized using `init_nlms()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptnlms()` is called with a new sample from the FES and a new sample from the NES signals to calculate the filter output $y(n)$ (estimated echo), filter error $e(n)$, and update the filter coefficients. The residual signal $e(n)$ is saved in each iteration for later examination.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

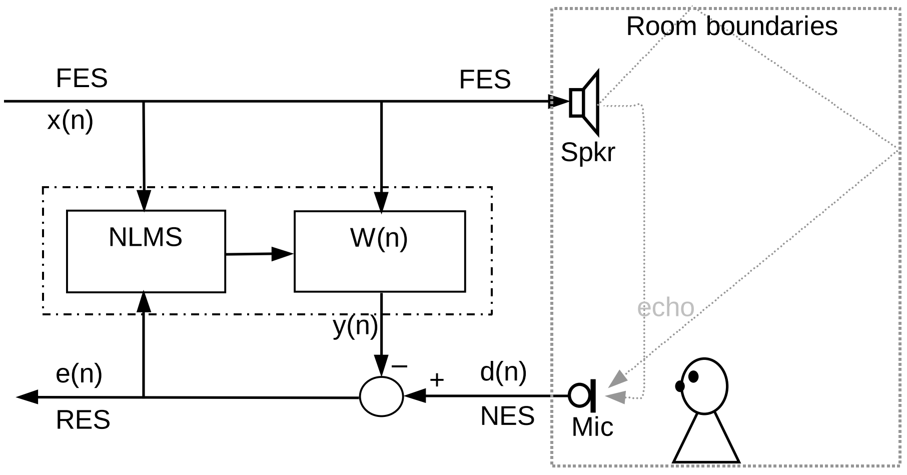

Figure 10.31: Block diagram of an acoustic echo canceler implemented using the NLMS algorithm.

Code

```
clear all;
infile = '.\wavin\aecfes.wav';          % Far-end speech (FES)
dfile  = '.\wavin\aecnes.wav';          % Near-end speech (NES)
rfile  = '.\wavout\resnlms.wav';        % residual signal

M      = 512;                           % adaptive filter length
mu     = 0.2/M;                         % adaptation constant
b      = 0.99;                          % autoregressive pole

[w,x,d,y,e,p]    = init_nlms(M);        % Init NLMS
[xn,inFs,inBits] = wavread(infile);     % read FES
[dn,inFs,dBits]  = wavread(dfile);      % read NES
inSize   = max(length(xn),length(dn));  % Samples to process
res      = dn;                          % Residual array
E        = init_ipwin(inSize);          % Initialize IPWIN

%% Processing Loop
for (m=1:inSize)
   % update the delay line
   x  = [2^(inBits-1) * xn(m); x(1:M-1)];

   % scale the Mic signal
   d  = 2^(inBits-1) *dn(m);

   % call asptnlms to update the filter
   [w,y,e,p]= asptnlms(x,w,d,mu,p,b);

   % save the last residual sample
   res(m) = 2^-(inBits-1)*e;

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,d, 'e', w, dn, res);

   % handle the Stop button
   while (stop  ~= 0), stop  = getStop; end;

   % handle the Break button
   if (brk), plot_echo(w,dn,res); break; end;
end;

wavwrite(res,inFs,inBits,rfile);    % Save the residual
plot_echo(w, dfile, rfile);         % Show results
```

Results

Running the above script will produce the graph shown in Fig. 10.32. The top panel in Fig. 10.32 shows the values taken by the filter coefficients by the end of the simulation (end of input files). The middle panel show the waveforms of the near-end speech signal $d(n)$ and the residual signal $e(n)$ for visual comparison between the echo before and after applying the echo canceler. The bottom panel shows the echo energy decrease in dB achieved by the echo canceler versus time, usually known as the Echo Return Loss Enhancement (ERLE). Note that the ERLE is meaningful only in the time periods where there is echo to be canceled.

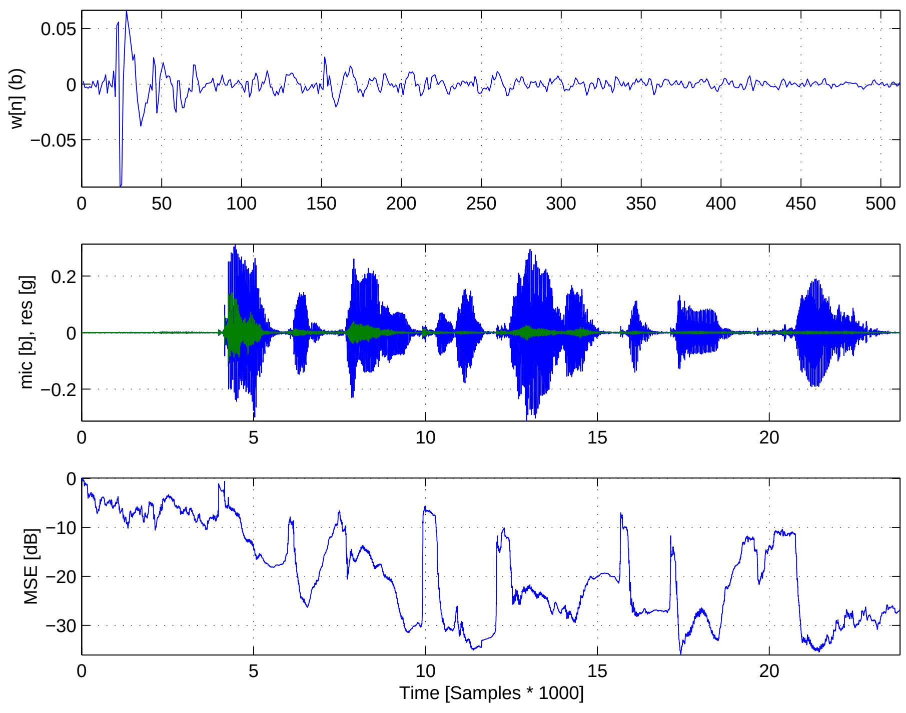

Figure 10.32: Performance of an Acoustic Echo Canceler implemented using the NLMS algorithm.

Audio Files

The following files demonstrate the performance of the NLMS algorithm in the echo canceler application mentioned above.

`wavin\aecfes.wav`	far-end speech (input) signal.
`wavin\nesaec.wav`	near-end speech (microphone) signal.
`wavout\resnlms512.wav`	residual signal (echo canceler output).

See Also

INIT_ NLMS, ASPTNLMS.

Reference

[11] and [4] for extensive analysis of the NLMS and the steepest-descent search method.

10.17 echo_ pbfdaf

Purpose Simulation of an acoustic echo canceler application using a transversal adaptive filter updated according to the Partitioned Block Frequency Domain Adaptive Filter (PBFDAF).

Syntax `echo_pbfdaf`

Description The block diagram of the acoustic echo cancellation problem is shown in Fig. 10.33. The simulation considered here uses a transversal FIR filter for the adjustable filter and the coefficients of the filter are updated in the frequency domain using the PBFDAF algorithm. The filter is divided into P partitions, each of M coefficients, which results in a filter of length PM coefficients in total. The block length is chosen to be $L = M$ so that the processing delay is P times shorter than that introduced by a BFDAF filter of the same length. The far-end speech signal $x(n)$ (the speech from the remote speaker) is stored in the file infile. The local speaker is assumed to be silent (listening to the remote speaker and not interrupting). The echo picked by the microphone when playing $x(n)$ through the local loudspeaker is stored in the file dfile. First the variables for the echo canceler $W(f)$ are created and initialized using `init_pbfdaf()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptpbfdaf()` is called with a new block of samples from the FES and a new block of samples from the NES signals to calculate the filter output block (estimated echo) and update the filter coefficients. The residual signal $e(n)$ is saved in each iteration for later examination.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

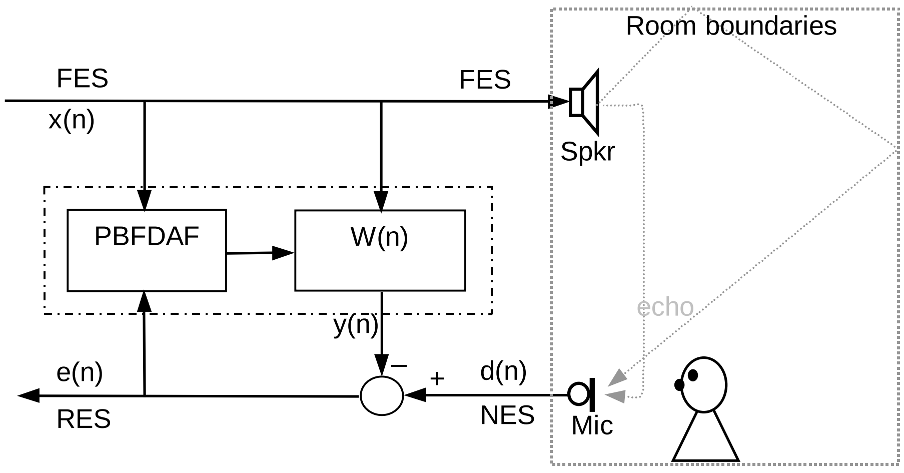

Figure 10.33: Block diagram of an acoustic echo canceler implemented using the Partitioned Block Frequency Domain Adaptive Filter (PBFDAF).

Code

```
clear all;
infile = '.\wavin\aecfes.wav';           % Far-end speech (FES)
dfile  = '.\wavin\aecnes.wav';           % Near-end speech (NES)
rfile  = '.\wavout\respbfdaf.wav';       % residual signal
P      = 4;                              % number of partitions
M      = 128;                            % adaptive filter length
L      = M;                              % block length
mu     = 0.01/L;                         % adaptation constant
b      = 0.99;                           % autoregressive pole

[W,x,d,e,y,Px,X,w] = init_pbfdaf(L,M,P); % Init PBFDAF
[xt,inFs,inBits]   = wavread(infile);    % read FES
[dt,inFs,dBits]    = wavread(dfile);     % read NES
inSize     = max(length(xt),length(dt)); % Samples to process
res                = dt;                 % Residual array
E                  = init_ipwin(inSize); % Initialize IPWIN

%% Processing Loop
for (m=1:L:inSize-L)
   % Read and scale a block from FES
   xn = 2^(inBits-1) * xt(m+1:m+L,:);

   % Read and scale a block from NES
   dn = 2^(dBits-1) * dt(m+1:m+L,:);

   % Update the adaptive filter
   [W,X,x,y,e,Px,w]=asptpbfdaf(M,x,xn,dn,X,W,mu,1,1,b,Px);

   % Scale and store the error (residual)
   res(m+1:m+L) = 2^-(dBits-1)* e;

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,dn,'e',w,dt,res);

   % handle the Stop button
   while (stop  ~= 0), stop  = getStop; end;

   % handle the Break button
   if (brk), plot_echo(w,dt,res); break; end;
end;

wavwrite(res,inFs,inBits,rfile);    % Save the residual
plot_echo(w, dfile, rfile);         % Show results
```

Results

Running the above script will produce the graph shown in Fig. 10.34. The top panel in Fig. 10.34 shows the values taken by the filter coefficients by the end of the simulation (end of input files). The middle panel show the waveforms of the near-end speech signal $d(n)$ and the residual signal $e(n)$ for visual comparison between the echo before and after applying the echo canceler. The bottom panel shows the echo energy decrease in dB achieved by the echo canceler versus time, usually known as the Echo Return Loss Enhancement (ERLE). Note that the ERLE is meaningful only in the time periods where there is echo to be canceled.

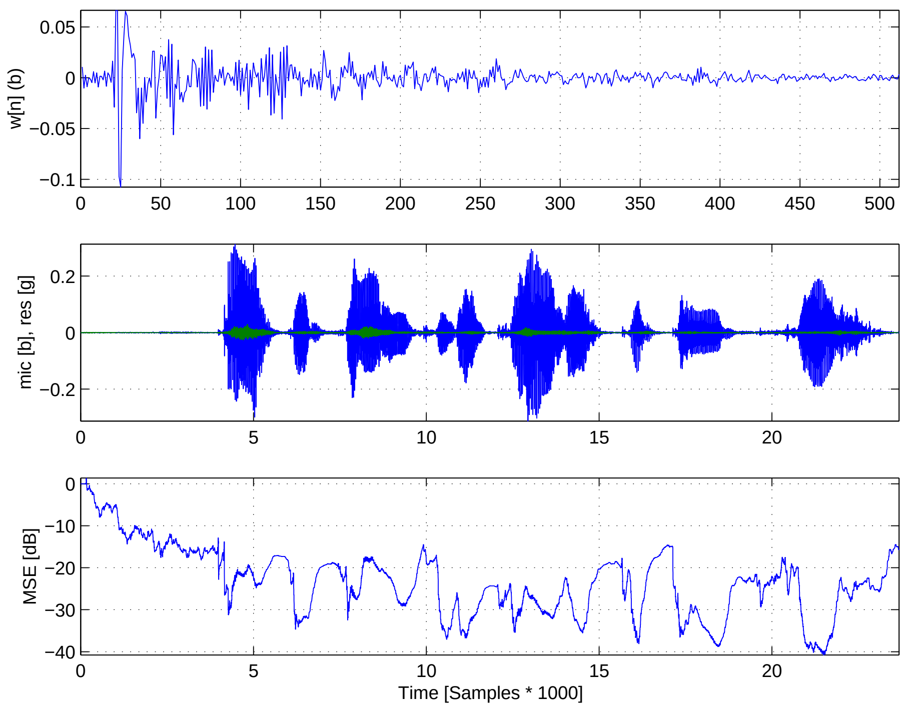

Figure 10.34: Performance of an Acoustic Echo Canceler implemented using the PBFDAF algorithm.

Audio Files

The following files demonstrate the performance of the PBFDAF algorithm in the echo canceler application mentioned above.

`wavin\aecfes.wav`	far-end speech (input) signal.
`wavin\nesaec.wav`	near-end speech (microphone) signal.
`wavout\respbfdaf.wav`	residual signal (echo canceler output).

See Also

INIT_ PBFDAF, ASPTPBFDAF, ASPTRCPBFDAF, ECHO_ NLMS, ECHO_ BFDAF.

Reference

[1] and [9] for detailed description of frequency domain adaptive filters.

10.18 echo_ rcpbfdaf

Purpose Simulation of an acoustic echo canceler application using a transversal adaptive filter updated according to the Reduced Complexity Partitioned Block Frequency Domain Adaptive Filter (RCPBFDAF).

Syntax `echo_rcpbfdaf`

Description The block diagram of the acoustic echo cancellation problem is shown in Fig. 10.35. The simulation considered here uses a transversal FIR filter for the adjustable filter and the coefficients of the filter are updated in the frequency domain using the RCPBFDAF algorithm. The filter is divided into P partitions, each of M coefficients, which results in a filter of length PM coefficients in total. The block length is chosen to be $L = M/2$ to further decrease the processing delay compared to BFDAF and PBFDAF filters of the same total number of coefficients. The far-end speech signal $x(n)$ (the speech from the remote speaker) is stored in the file infile. The local speaker is assumed to be silent (listening to the remote speaker and not interrupting). The echo picked by the microphone when playing $x(n)$ through the local loudspeaker is stored in the file dfile. First the variables for the echo canceler $W(f)$ are created and initialized using `init_rcpbfdaf()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptrcpbfdaf()` is called with a new block of samples from the FES and a new block of samples from the NES signals to calculate the filter output block (estimated echo) and update the filter coefficients. The residual signal $e(n)$ is saved in each iteration for later examination.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

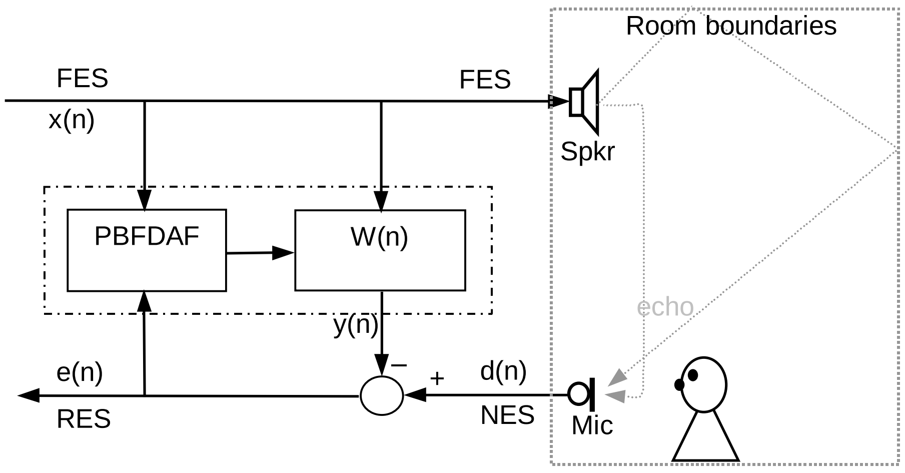

Figure 10.35: Block diagram of an acoustic echo canceler implemented using the (Reduced Complexity) partitioned block frequency domain adaptive filter (RCPBFDAF).

Code

```
clear all;
infile = '.\wavin\aecfes.wav';          % Far-end speech (FES)
dfile  = '.\wavin\aecnes.wav';          % Near-end speech (NES)
rfile  = '.\wavout\resrcpbfdaf.wav';    % residual signal
P      = 8;                             % number of partitions
M      = 64;                            % adaptive filter length
L      = M/2;                           % block length
mu     = 0.04/L;                        % adaptation constant
b      = 0.99;                          % autoregressive pole

[W,x,d,e,y,Px,X,ci,w]=init_rcpbfdaf(L,M,P);    % Init RCPBFDAF
[xt,inFs,inBits]   = wavread(infile);    % read FES
[dt,inFs,dBits]    = wavread(dfile);     % read NES
inSize     = max(length(xt),length(dt)); % Samples to process
res                = dt;                 % Residual array
E                  = init_ipwin(inSize); % Initialize IPWIN

%% Processing Loop
for (m=1:L:inSize-L)
   % Read and scale a block from FES
   xn = 2^(inBits-1) * xt(m+1:m+L,:);

   % Read and scale a block from NES
   dn = 2^(dBits-1) * dt(m+1:m+L,:);

   % Update the adaptive filter
   % only 2 partitions are constrained each call
   [W,X,x,y,e,Px,ci,w]=asptrcpbfdaf(M,x,xn,dn,X,W,mu,1,2,b,Px,ci);

   % Scale and store the error (residual)
   res(m+1:m+L) = 2^-(dBits-1)* e;

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,dn,'e',w,dt,res);

   % handle the Stop button
   while (stop  ~= 0), stop  = getStop; end;

   % handle the Break button
   if (brk), plot_echo(w,dt,res); break; end;
end;

wavwrite(res,inFs,inBits,rfile);    % Save the residual
plot_echo(w, dfile, rfile);         % Show results
```

Results Running the above script will produce the graph shown in Fig. 10.36. The top panel in Fig. 10.36 shows the values taken by the filter coefficients by the end of the simulation (end of input files). The middle panel show the waveforms of the near-end speech signal $d(n)$ and the residual signal $e(n)$ for visual comparison between the echo before and after applying the echo canceler. The bottom panel shows the echo energy decrease in dB achieved by the echo canceler versus time, usually known as the Echo Return Loss Enhancement (ERLE). Note that the ERLE is meaningful only in the time periods where there is echo to be canceled.

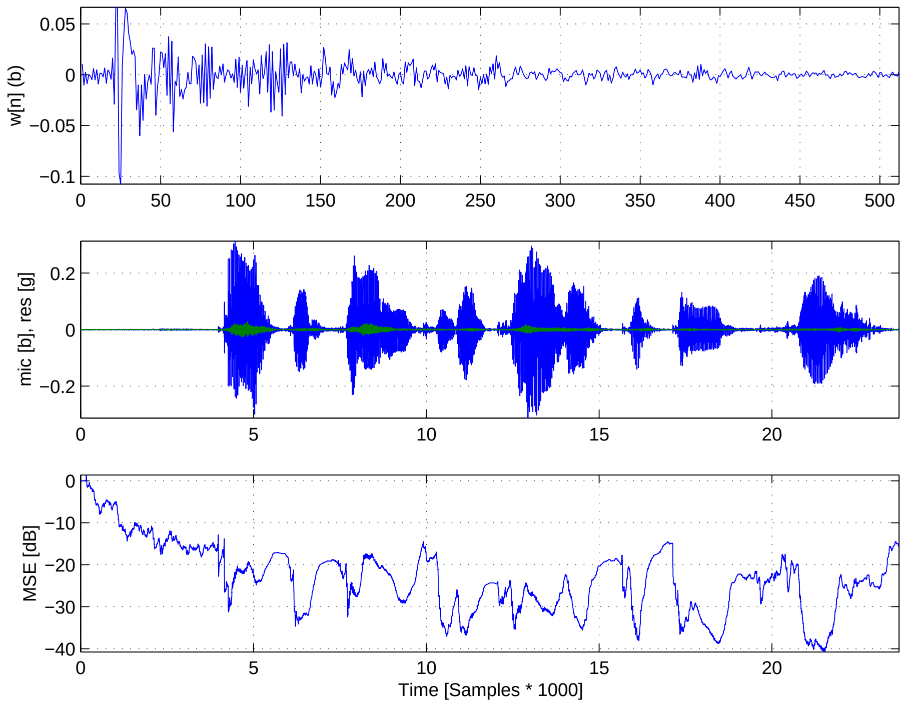

Figure 10.36: Performance of an Acoustic Echo Canceler implemented using the RCPBFDAF algorithm with two partitions out of eight are constrained each block and a block length equals to half the partition length.

Audio Files The following files demonstrate the performance of the RCPBFDAF algorithm in the echo canceler application mentioned above.

`wavin\aecfes.wav`	far-end speech (input) signal.
`wavin\nesaec.wav`	near-end speech (microphone) signal.
`wavout\resrcpbfdaf.wav`	residual signal (echo canceler output).

See Also INIT_ RCPBFDAF, ASPTRCPBFDAF, ASPTPBFDAF, ECHO_ PBFDAF, ECHO_ BFDAF.

Reference [1] and [9] for detailed description of frequency domain adaptive filters.

10.19 equalizer_ nlms

Purpose Simulation of an adaptive inverse modeling application using an adaptive filter updated according to the Normalized Least Mean Squares (NLMS) algorithm.

Syntax `equalizer_nlms`

Description The block diagram of the equalization (inverse modeling) problem is shown in Fig. 10.37. The simulation considered here uses a transversal FIR filter for the adjustable filter and the coefficients of the filter are updated using the NLMS algorithm. The input signal $u(n)$ (measured signal at the physical system input) is stored in the file ufile. The physical system output $x(n)$ (the signal measured at the system output in response to applying $u(n)$ at its input) is stored in the file xfile. The desired signal for the equalization problem is a delayed version of the system input signal. In practice, only the system output $x(n)$ is available during normal operation and the system input $u(n)$ should be provided in a training session. First the variables for the adaptive model $w(n)$ are created and initialized using `init_nlms()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptnlms()` is called with a new input sample and a new desired sample to calculate the filter output $y(n)$ and update the filter coefficients.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

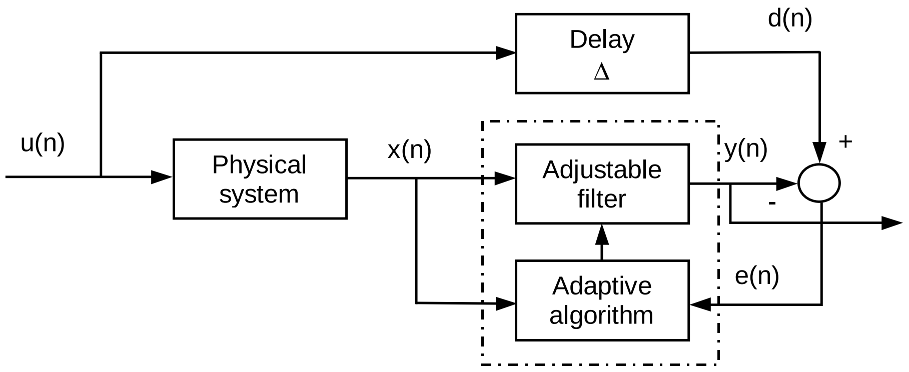

Figure 10.37: Block diagram of the inverse modeling application.

Code

```
clear all;
load .\data\h32;                  % for verification

ufile  = '.\wavin\scinwn.wav';    % input signal
xfile  = '.\wavin\scdwn32.wav';   % system output

h      = h32;
D      = 32;                      % delay in desired path
M      = 64;                      % adaptive model length
mu     = .2/M;                    % Step size
b      = 0.98;                    % smoothing pole

%% Initialize storage
[w,x,d,y,e,p]    = init_nlms(M);               % Init NLMS algorithm
[un,x1Fs,x1Bits] = wavread(ufile);             % Get system input
[xn,x2Fs,x2Bits] = wavread(xfile);             % Get system output
inSize           = min(length(un),length(xn)); % Samples to process
E                = init_ipwin(inSize);         % Initialize IPWIN

%% Processing Loop
% The desired signal is the delayed un(n) and the adaptive filter
% input is xn(n) which is the output of the system to be equalized.

for (m=D+1:inSize)
   x  = [xn(m,:);x(1:M-1,:) ];              % update the delay line
   d  = un(m-D,:);                          % desired sample

   % Update the adaptive filter and calculate the output and error
   [w,y,e,p]= asptnlms(x,w,d,mu,p,b);

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,d, 'i', w, h, D);

   % handle the Stop button
   while (stop ~= 0), stop = getStop; end;

   % handle the Break button
if (brk), plot_invmodel(w,h,E,D); break; end;
end;

plot_invmodel(w,h,E,D);
```

Results

Running the above script will produce the graph shown in Fig. 10.38. The two top-left panels in Fig. 10.38 show the time and frequency responses of the optimum solution for the inverse modeling problem at hand. The time and frequency responses for the model obtained by the adaptive filter are shown in the two top-right panels. The bottom panel shows the learning curve for the adaptive filter. Note that the input signal is colored by the physical system which might increase the eigenvalue spread in the adaptive filter input signal $x(n)$. This might result in slow convergence and large final misadjustment at frequencies not well excited when the LMS or one of its derivatives is used to adapt the filter.

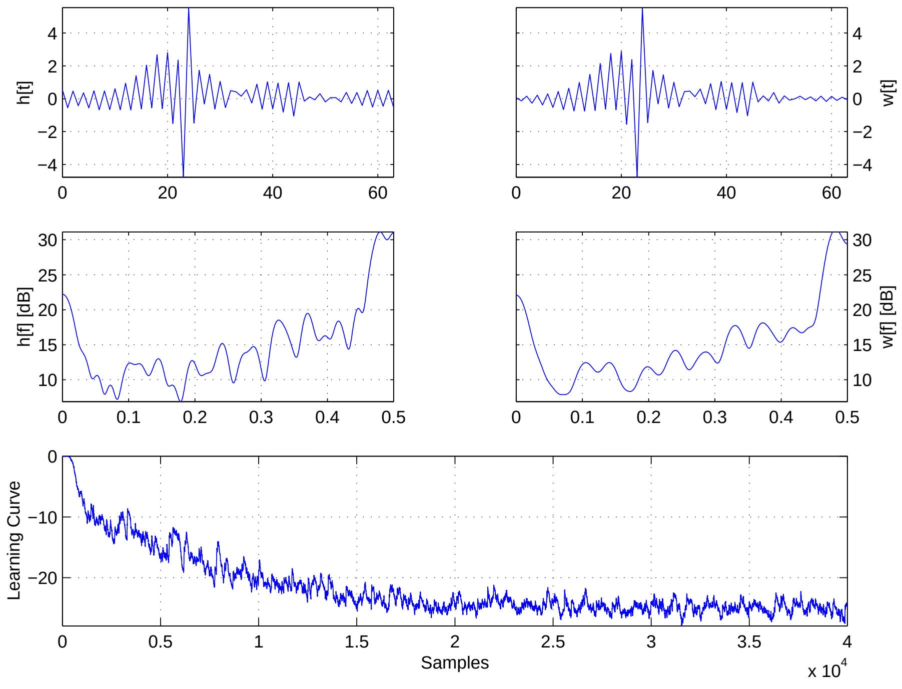

Figure 10.38: Performance of the NLMS adaptive algorithm in an inverse modeling application.

See Also

INIT_ NLMS, ASPTNLMS, MODEL_ NLMS.

Reference

[11] and [4] for extensive analysis of the NLMS and the steepest-descent search method.

10.20 equalizer_ rls

Purpose Simulation of an adaptive inverse modeling application using an adaptive filter updated according to the Recursive Least Squares (RLS) algorithm.

Syntax `equalizer_rls`

Description The block diagram of the equalization (inverse modeling) problem is shown in Fig. 10.39. The simulation considered here uses a transversal FIR filter for the adjustable filter and the coefficients of the filter are updated using the RLS algorithm. The input signal $u(n)$ (measured signal at the physical system input) is stored in the file ufile. The physical system output $x(n)$ (the signal measured at the system output in response to applying $u(n)$ at its input) is stored in the file xfile. The desired signal for the equalization problem is a delayed version of the system input signal. In practice, only the system output $x(n)$ is available during normal operation and the system input $u(n)$ should be provided in a training session. First the variables for the adaptive model $w(n)$ are created and initialized using `init_rls()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptrls()` is called with a new input sample and a new desired sample to calculate the filter output $y(n)$ and update the filter coefficients.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

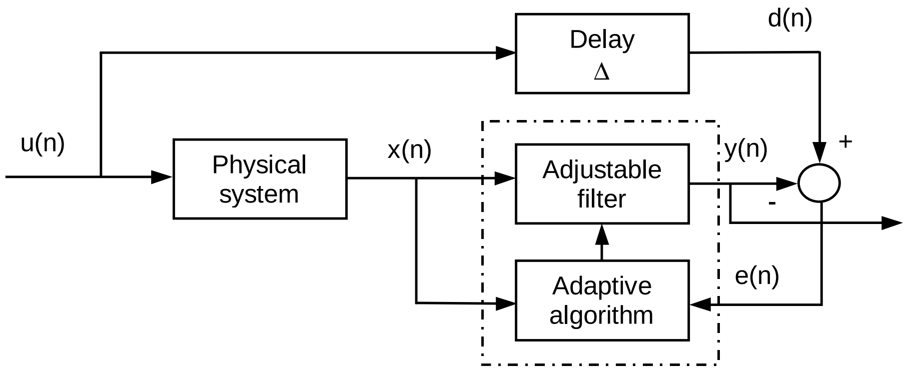

Figure 10.39: Block diagram of the inverse modeling application.

Code

```
clear all;
load .\data\h32;                  % for verification

ufile  = '.\wavin\scinwn.wav';    % input signal
xfile  = '.\wavin\scdwn32.wav';   % system output

h      = h32;
D      = 32;                      % delay in desired path
M      = 64;                      % adaptive model length
b      = 0.01;                    % initial diagonal of R
a      = 0.99;                    % forgetting factor

%% Initialize storage
[w,x,d,y,e,R]    = init_rls(M,b);               % Init RLS algorithm
[un,x1Fs,x1Bits] = wavread(ufile);              % Get system input
[xn,x2Fs,x2Bits] = wavread(xfile);              % Get system output
inSize           = min(length(un),length(xn));  % Samples to process
E                = init_ipwin(inSize);          % Initialize IPWIN

%% Processing Loop
% The desired signal is the delayed un(n) and the adaptive filter
% input is xn(n) which is the output of the system to be equalized.

for (m=D+1:inSize)
   x  = [xn(m,:);x(1:M-1,:) ];             % update the delay line
   d  = un(m-D,:);                         % desired sample

   % Update the adaptive filter and calculate the output and error
   [w,y,e,R] = asptrls(x,w,d,R,a);

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,d, 'i', w, h, D);

   % handle the Stop button
   while (stop ~= 0), stop = getStop; end;

   % handle the Break button
if (brk), plot_invmodel(w,h,E,D); break; end;
end;

plot_invmodel(w,h,E,D);
```

Results

Running the above script will produce the graph shown in Fig. 10.40. The two top-left panels in Fig. 10.40 show the time and frequency responses of the optimum solution for the inverse modeling problem at hand. The time and frequency responses for the model obtained by the adaptive filter are shown in the two top-right panels. The bottom panel shows the learning curve for the adaptive algorithm.

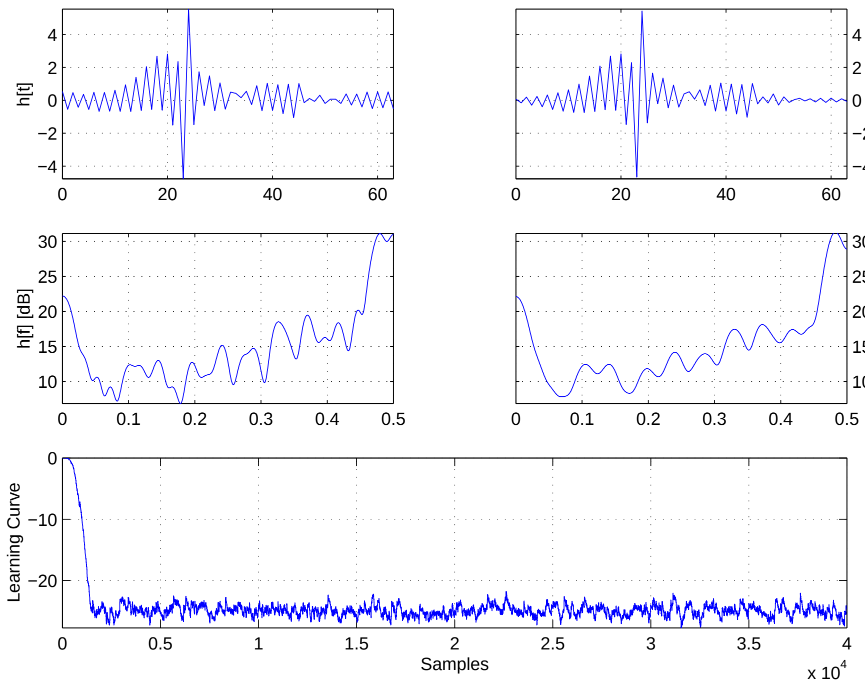

Figure 10.40: Performance of the RLS algorithm in a a channel equalization application.

See Also

INIT_ RLS, ASPTRLS.

Reference

[2] and [4] for analysis of the RLS algorithm and its variants.

10.21 model_ arlmsnewt

Purpose Simulation of an adaptive forward modeling application using an adaptive transversal filter updated with the autoregressive modeling version of the LMS-Newton algorithm.

Syntax `model_arlmsnewt`

Description The block diagram of the system identification (forward modeling) problem using the autoregressive LMS-Newton adaptive algorithm is shown in Fig. 10.41, (see Section 4.1 for more details on the ARLMSNEWT algorithm). The input signal $x(n)$ (measured signal at the input of the system to be modeled) is stored in the file infile. The desired signal $d(n)$ (the signal measured at the system output in response to applying $x(n)$ at its input) is stored in the file dfile. First the variables for the LMS-Newton algorithm are creates and initializes using `init_arlmsnewt()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptarlmsnewt()` is called with a new input sample and a new desired sample to calculate the filter output (estimated desired signal) and update the adaptive model coefficients.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

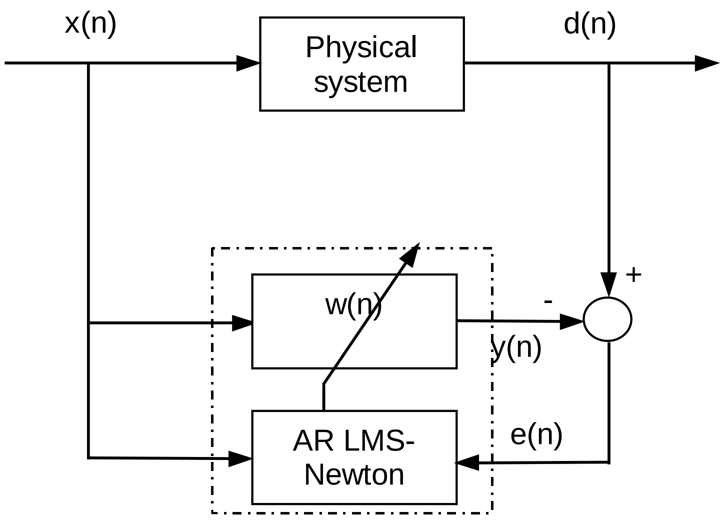

Figure 10.41: Block diagram of a forward modeling application using the autoregressive LMS-Newton algorithm.

Code

```
clear all;
load .\data\h512;                  % for verification

% Data files
infile = '.\wavin\scinwn.wav';   % input signal, white noise
dfile  = '.\wavin\scdwn512.wav'; % system output

% Simulation parameters
L      = 512;                      % adaptive model length
M      = 3;                        % AR model coef.
mu_w   = .4/L;                     % FIR filter step size
mu_p   = 1e-6;                     % lattice predictor step size
maxk   = .99;                      % maximum value for PARCOR

%% Initialize storage
[k,w,x,b,u,P,d,y,e] = init_arlmsnewt(L,M); % Init LMS Newton
[xn,inFs,inBits] = wavread(infile);    % read input signal
[dn,inFs,dBits]  = wavread(dfile);     % read desired signal
inSize  = min(length(dn),length(xn));  % samples to process
E       = init_ipwin(inSize);          % Initialize IPWIN

%% Processing Loop
for (m=1:inSize)
   x  = [xn(m,:);x(1:end-1,:) ];     % update the delay line
   d  = dn(m);                       % new desired sample

   % update the adaptive model
   [k,w,b,u,P,y,e] = asptarlmsnewt(k,w,x,b,u,P,d,mu_p,mu_w,maxk);

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,d,'m',w,h512);

   % handle the Stop button
   while (stop ~= 0), stop = getStop; end;

   % handle the Break button
   if (brk), plot_model(w,h512,E); break; end;
end;

plot_model(w,h512,E);
```

Results

Running the above script will produce the graph shown in Fig. 10.42. The two top-left panels in Fig. 10.42 show the time and frequency responses of the unknown system for which this application is intended to provide a FIR model. The time and frequency responses for the model obtained by the adaptive filter are shown in the two top-right panels. The bottom-left panel shows the learning curve and the bottom-right panel shows the error in the filter coefficients by the end of the simulation.

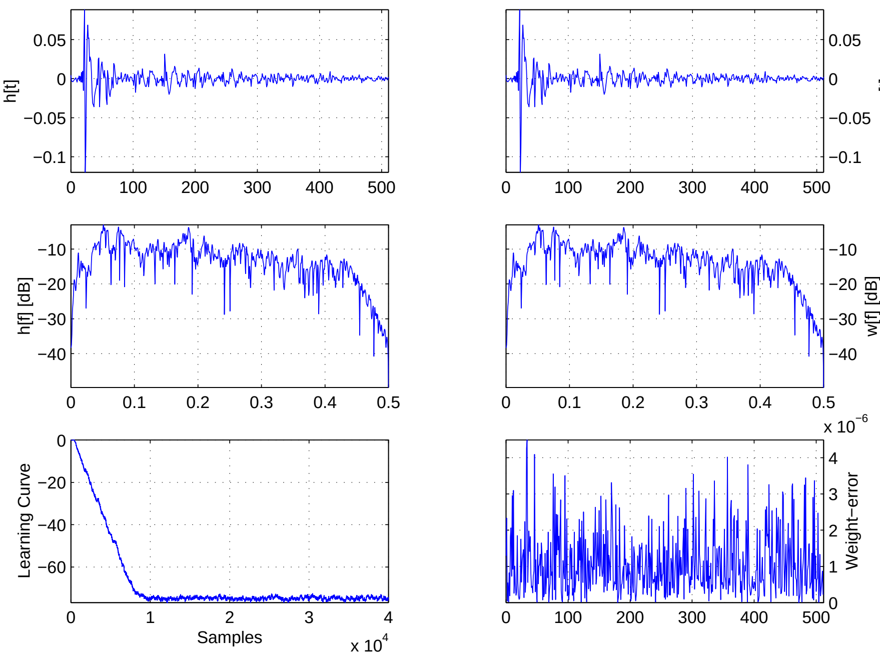

Figure 10.42: Performance of the autoregressive LMS-Newton adaptive filter in a system identification application.

See Also

INIT_ ARLMSNEWT, ASPTARLMSNEWT.

Reference

[2] and [4] for analysis of the adaptive Lattice filters, [2] and [11] for analysis of the LMS-Newton algorithm.

10.22 model_ eqerr

Purpose Simulation of an adaptive forward modeling application using a recursive adaptive filter updated according to the Equation Error algorithm.

Syntax `model_eqerr`

Description The block diagram of the system identification (forward modeling) problem using the Equation Error adaptive algorithm is shown in Fig. 10.43, (see Section 6.2 for more details on the Equation Error algorithm). The input signal $x(n)$ (measured signal at the input of the system to be modeled) is stored in the file infile. The desired signal $d(n)$ (the signal measured at the system output in response to applying $x(n)$ at its input) is stored in the file dfile. First the variables for the adaptive IIR model $w(n)$ are created and initialized using `init_eqerr()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `aspteqerr()` is called with a new input sample and a new desired sample to calculate the filter output (estimated desired signal) and update the filter coefficients.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

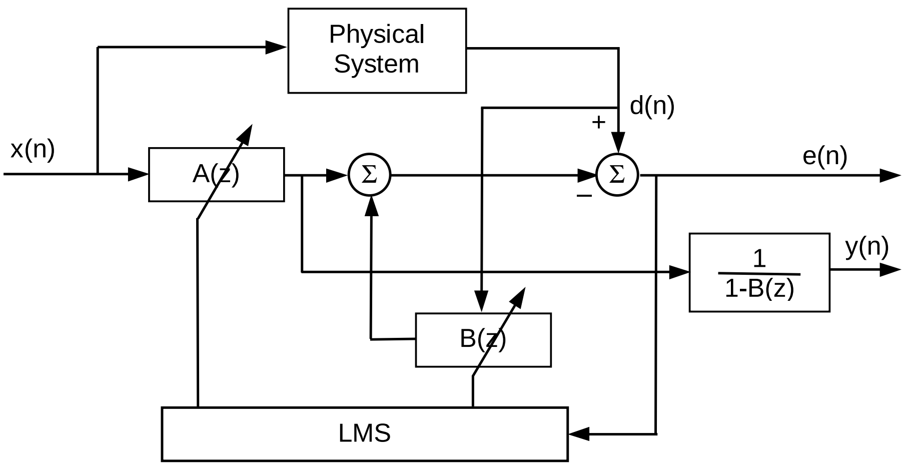

Figure 10.43: Block diagram of the forward modeling application using the Equation Error recursive adaptive filter.

Code

```
clear all;
infile = '.\wavin\scinwn.wav';    % input signal
dfile  = '.\wavin\scdar22.wav';   % desired signal

N  = 2;                           % number of zeros
M  = 2;                           % number of poles
H  = 50;                          % response length
p1 = .2 + j* .85;                 % unknown filter poles
p2 = .2 - j* .85;                 % for verification
ip = [1; zeros(H-1,1)];           % impulse vector
h  = filter([0.6 -.01],[1 -(p1+p2) (p1*p2)],ip);

% Initial parameters
u0  = zeros(N+M,1);               % composite input vector
w0  = u0;                         % initial filter vector
y0  = zeros(M,1);                 % initial output delay line
d0  = randn(1,1);                 % initial desired sample
mu  = [.02;0.02;.01;0.01] ;       % Step size vector

% Create and initialize EQERR IIR filter
[u,w,y,e,mu,Px,Pd]=init_eqerr(N,M,u0,w0,y0,d0,mu);
[xn,inFs,inBits] = wavread(infile);   % read input
[dn,inFs,dBits]  = wavread(dfile);    % read desired
inSize = max(length(dn),length(xn));  % samples to process
E      = init_ipwin(inSize);          % Initialize IPWIN

%% Processing Loop
for (m=1:inSize)
   x = 2^(inBits-1) * xn(m);      % input sample
   d = 2^(inBits-1) * dn(m);      % desired sample

   % update the filter
   [u,w,y,e,Px,Pd]=aspteqerr(N,M,u,w,y,x,d,mu,Px,Pd);

   % impulse response for verification
   wp = filter(w(1:N),[1 ; -w(N+1:N+M)],ip);

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,d, 'm', wp, h);

   % handle the Stop button
while (stop ~= 0), stop  = getStop; end;
   % handle the Break button
if (brk), plot_model(wp,h,E); break; end;
end;

plot_model(wp,h,E);
```

Results

Running the above script will produce the graph shown in Fig. 10.44. The two top-left panels in Fig. 10.44 show the time and frequency responses of the unknown system for which this application is intended to provide an IIR model. The time and frequency responses for the model obtained by the adaptive filter are shown in the two top-right panels. The bottom-left panel shows the learning curve and the bottom-right panel shows the estimation error in the impulse response.

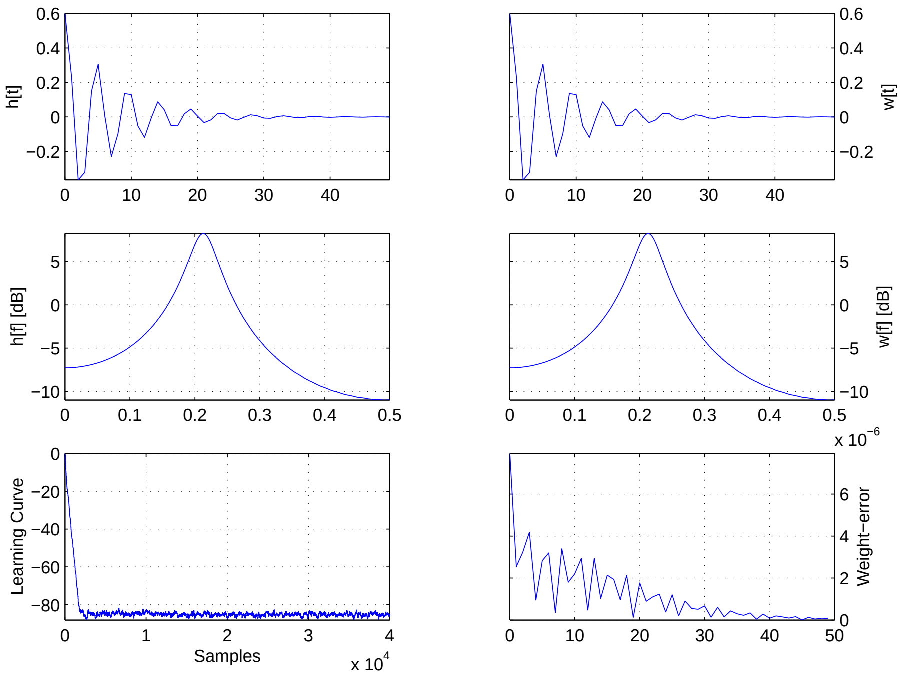

Figure 10.44: Performance of the equation error adaptive filter in a system identification application.

See Also

INIT_ EQERR, ASPTEQERR.

Reference

[2] and [10] for introduction to recursive adaptive filters.

10.23 model_ lmslattice

Purpose Simulation of an adaptive forward modeling application using an adaptive joint process estimator updated according to the LMS Lattice algorithm.

Syntax `model_lmslattice`

Description The block diagram of the system identification (forward modeling) problem using the LMS Lattice adaptive algorithm is shown in Fig. 10.45. The LMS Lattice algorithm adjusts the PARCOR coefficients of the lattice predictor and the linear combiner coefficients simultaneously to minimize the mean square of the forward and backward prediction errors as well as the modeling error $e(n)$ (see Section 5.4 for more information on the LMSLATTICE algorithm). The input signal $x(n)$ (measured signal at the input of the system to be modeled) is stored in the file infile. The desired signal $d(n)$ (the signal measured at the system output in response to applying $x(n)$ at its input) is stored in the file dfile. First the variables for the LMS-Lattice filter are creates and initializes using `init_lmslattice()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptlmslattice()` is called with a new input sample and a new desired sample to calculate the filter output (estimated desired signal) and update the adaptive model coefficients. This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

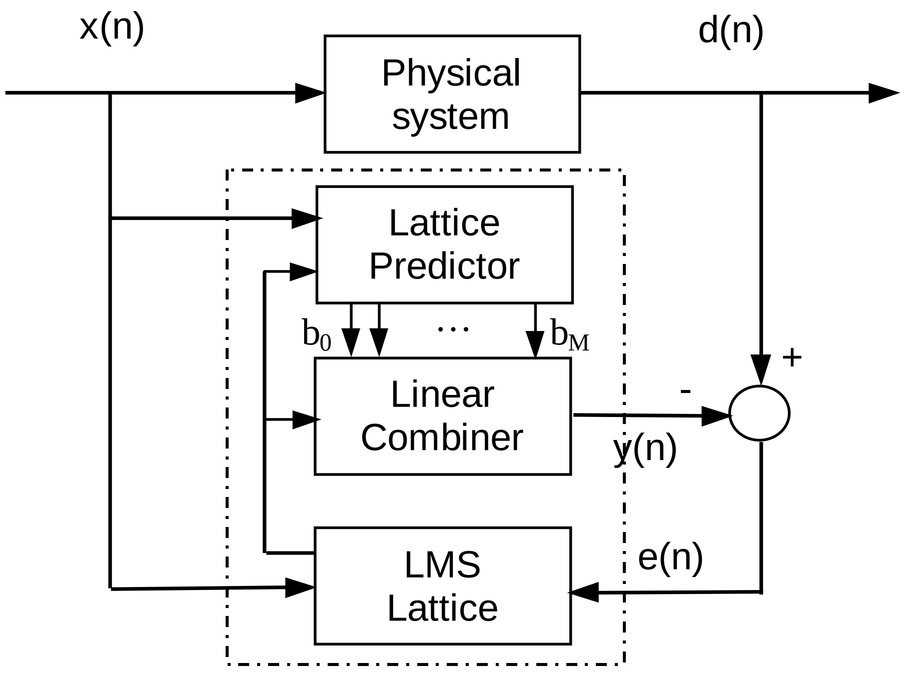

Figure 10.45: Block diagram of the Lattice joint process estimator in a forward modeling application.

Code

```
clear all;
load .\data\h32;                    % for verification

% Data files
infile = '.\wavin\scinwn.wav';   % input signal, white noise
dfile  = '.\wavin\scdwn32.wav';  % system output

% Simulation parameters
L      = 32;                        % adaptive model length
mu_c   = .1/L;                      % linear combiner step size
mu_p   = 1e-6;                      % linear predictor step size

%% Initialize storage
[k,w,b,P,d,y,e]  = init_lmslattice(L); % Init LMS Lattice
[xn,inFs,inBits] = wavread(infile);    % read input signal
[dn,inFs,dBits]  = wavread(dfile);     % read desired signal
inSize = min(length(dn),length(xn));   % samples to process
E      = init_ipwin(inSize);           % Initialize IPWIN
uk     = 1;                            % PARCOR update flag

%% Processing Loop
for (m=1:inSize)
   % stop updating k after 2000 samples
   if (m == 2000), uk=0;end
   x  = xn(m);       % new input sample
   d  = dn(m);       % new desired sample

   % update the adaptive model
   [k,w,b,P,y,e] = asptlmslattice(k,w,b,P,x,d,mu_p,mu_c,uk);

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,d,'m',w,h32);

   % handle the Stop button
   while (stop ~= 0), stop = getStop; end;

   % handle the Break button
   if (brk), plot_model(w,h32,E); break; end;
end;

plot_model(w,h32,E);
```

Results Running the above script will produce the graph shown in Fig. 10.46. The two top-left panels in Fig. 10.46 show the time and frequency responses of the unknown system for which this application is intended to provide a FIR model. The time and frequency responses for the model obtained by the adaptive filter are shown in the two top-right panels. The bottom-left panel shows the learning curve and the bottom-right panel shows the error in the filter coefficients by the end of the simulation.

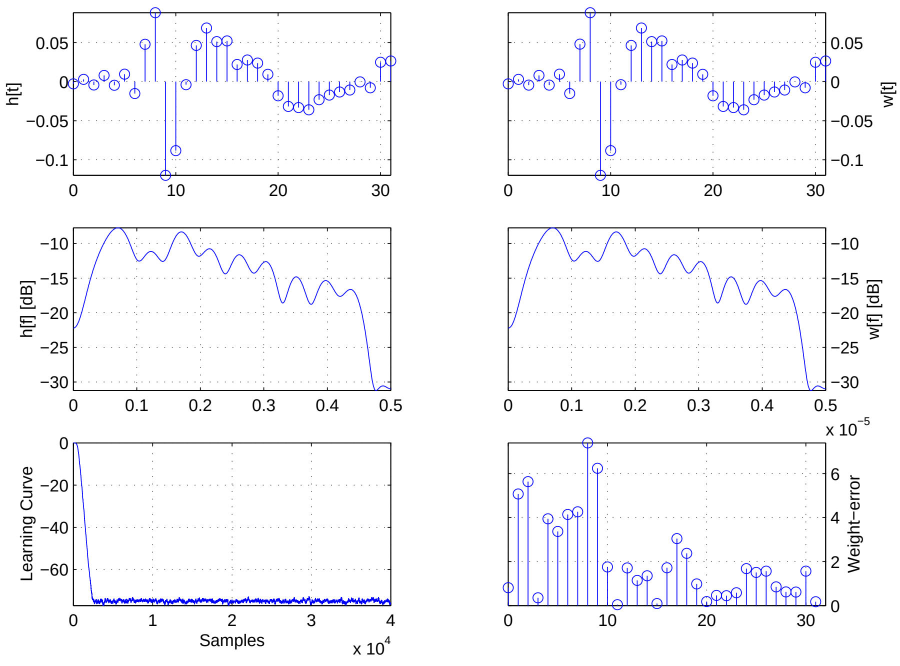

Figure 10.46: Performance of the LMS Lattice adaptive filter in a system identification application.

See Also INIT_ LMSLATTICE, ASPTLMSLATTICE.

Reference [2] and [4] for analysis of the adaptive Lattice filters.

10.24 model_ mvsslms

Purpose Simulation of an adaptive forward modeling application using a transversal adaptive filter updated according to the Modified Variable Step Size LMS (MVSSLMS) algorithm.

Syntax `model_mvsslms`

Description The block diagram of the system identification (forward modeling) problem using the MVSSLMS adaptive algorithm is shown in Fig. 10.47 (see Section 4.10 for more details on the modified variable step size algorithm). The simulation considered here uses a transversal FIR filter for the adjustable filter and the coefficients of the filter are updated using the MVSSLMS algorithm. The input signal $x(n)$ (measured signal at the input of the system to be modeled) is stored in the file infile. The desired signal $d(n)$ (the signal measured at the system output in response to applying $x(n)$ at its input) is stored in the file dfile. First the variables for the adaptive model $w(n)$ are created and initialized using `init_mvsslms()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptmvsslms()` is called with a new input sample and a new desired sample to calculate the filter output (estimated desired signal) and update the filter coefficients.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

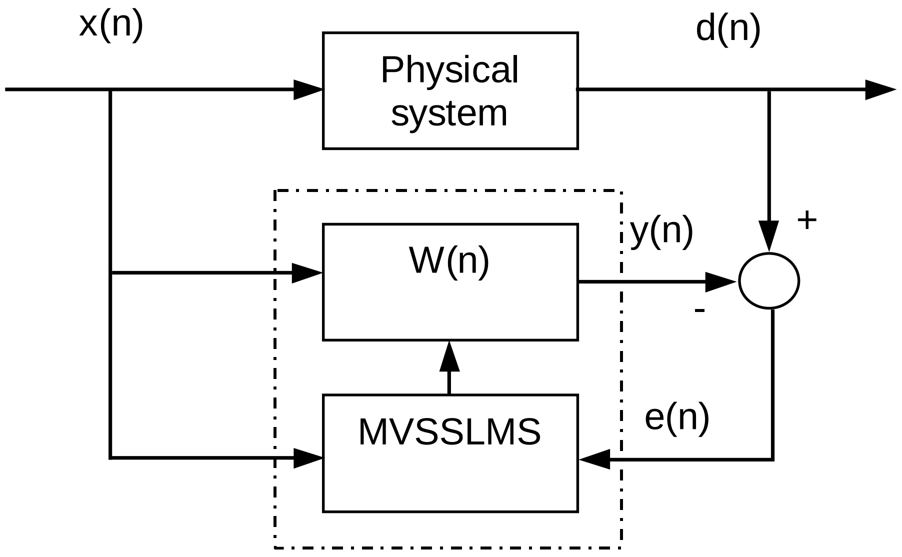

Figure 10.47: Block diagram of an FIR forward modeling using the MVSSLMS adaptive algorithm.

Code

```
clear all;
load .\data\h32;                       % for verification
infile = '.\wavin\scinwn.wav';         % input signal, white noise
dfile  = '.\wavin\scdwn32.wav';        % system output

L      = 32;                           % adaptive model length
roh    = 1e-3;                         % adaptation constant of mu
mu_min = 1e-6;                         % lower bound for mu
mu_max = 0.99;                         % higher bound for mu

%% Initialize storage
[w,x,d,y,e,g,mu] = init_mvsslms(L);  % Initialize MVSSLMS
[xn,inFs,inBits] = wavread(infile);  % read input signal
[dn,inFs,dBits]  = wavread(dfile);   % read desired signal
inSize = min(length(dn),length(xn)); % samples to process
E      = init_ipwin(inSize);         % Initialize IPWIN
muv    = zeros(inSize,1);            % time evolution of mu

%% Processing Loop
for (m=1:inSize)
   x  = [xn(m); x(1:L-1,:) ];   % update the input delay line
   d  = dn(m);                  % get the new desired sample

   % Update the adaptive filter
   [w,g,mu,y,e] = asptmvsslms(x,w,g,d,mu,roh,mu_min,mu_max);
   muv(m) = mu;                 % save mu to display later

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,d,'m',w,h32);

   % handle the Stop button
   while (stop ~= 0), stop = getStop; end;

   % handle the Break button
   if (brk), plot_model(w,h32,E); break; end;
end;

plot_model(w,h32,E);
subplot(3,2,6);
plot(muv(1:m));grid
```

Results Running the above script will produce the graph shown in Fig. 10.48. The two top-left panels in Fig. 10.48 show the time and frequency responses of the unknown system for which this application is intended to provide a FIR model. The time and frequency responses for the model obtained by the adaptive filter are shown in the two top-right panels. The bottom-left panel shows the learning curve and the bottom-right panel shows the evolution of the step size variable with time during the simulation.

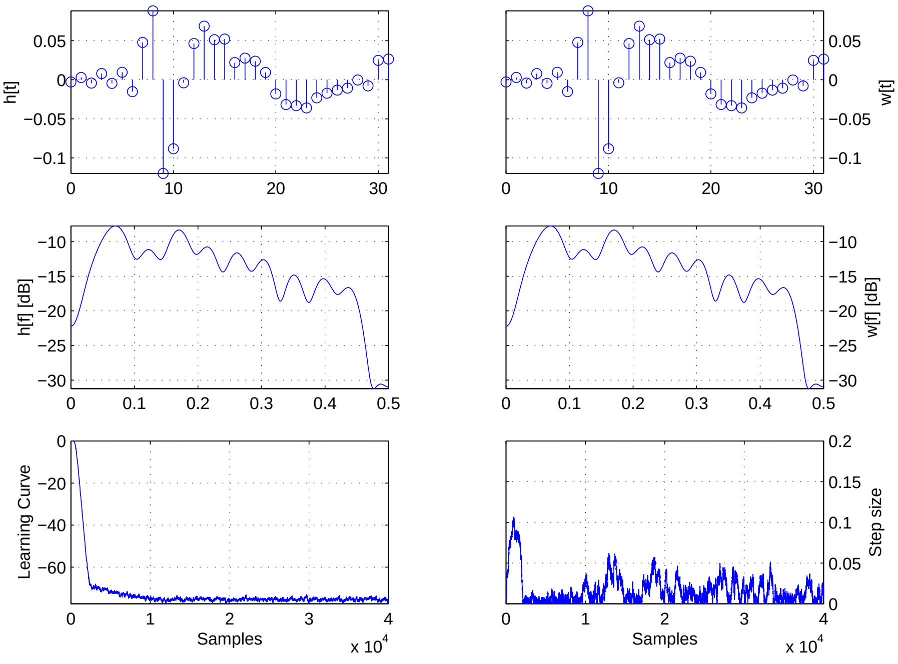

Figure 10.48: Performance of the Modified Variable Step Size LMS (MVSSLMS) adaptive filter in a system identification application.

See Also INIT_ MVSSLMS, ASPTMVSSLMS, ASPTVSSLMS.

Reference [11] and [4] for extensive analysis of the LMS and the steepest-descent search method.

10.25 model_ outerr

Purpose Simulation of an adaptive forward modeling application using a recursive adaptive filter updated according to the Output Error algorithm.

Syntax `model_outerr`

Description The block diagram of the system identification (forward modeling) problem using the Output Error adaptive algorithm is shown in Fig. 10.49 (see Section 6.3 for more details on the Output Error algorithm). The simulation considered here uses a recursive filter for the adjustable filter and the coefficients of the filter are updated using the Output Error algorithm. The input signal $x(n)$ (measured signal at the input of the system to be modeled) is stored in the file infile. The desired signal $d(n)$ (the signal measured at the system output in response to applying $x(n)$ at its input) is stored in the file dfile. First the variables for the adaptive IIR model $w(n)$ are created and initialized using `init_outerr()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptouterr()` is called with a new input sample and a new desired sample to calculate the filter output (estimated desired signal) and update the filter coefficients.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

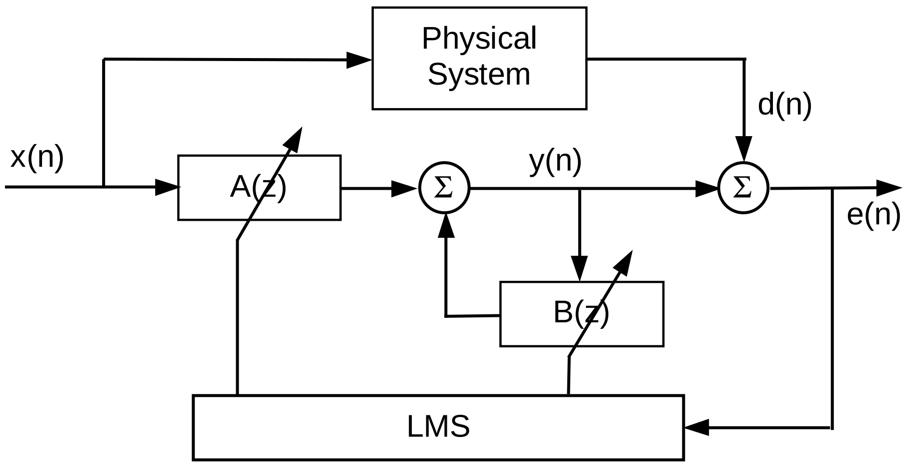

Figure 10.49: Block diagram of the forward modeling application using the Output Error recursive adaptive filter.

Code

```
clear all;
infile = '.\wavin\scinwn.wav';    % input signal
dfile  = '.\wavin\scdar22.wav';   % desired signal

N  = 2;                           % number of zeros
M  = 2;                           % number of poles
H  = 50;                          % response length
p1 = .2 + j* .85;                 % unknown filter poles
p2 = .2 - j* .85;                 % for verification
ip = [1; zeros(H-1,1)];           % impulse vector
h  = filter([0.6 -.01],[1 -(p1+p2) (p1*p2)],ip);

% Initial parameters
u0  = zeros(N+M,1);               % composite input vector
w0  = u0;                         % initial filter vector
c0  = u0;                         % initial delay line
d0  = randn(1,1);                 % initial desired sample
mu  = [.01;0.01;.001;0.001] ;     % Step size vector

% Create and initialize OUTERR IIR filter
[u,w,c,y,d,e,mu,Px,Py] = init_outerr(N,M,u0,w0,c0,d0,mu);
[xn,inFs,inBits] = wavread(infile);   % read input
[dn,inFs,dBits]  = wavread(dfile);    % read desired
inSize = max(length(dn),length(xn));  % samples to process
E      = init_ipwin(inSize);          % Initialize IPWIN

%% Processing Loop
for (m=1:inSize)
   x = 2^(inBits-1) * xn(m);      % input sample
   d = 2^(inBits-1) * dn(m);      % desired sample

   % update the filter
   [u,w,c,y,e,Px,Py] = asptouterr(N,M,u,w,c,x,d,mu,Px,Py);

   % impulse response for verification
   wp = filter(w(1:N),[1 ; -w(N+1:N+M)],ip);

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,d, 'm', wp, h);

   % handle the Stop button
   while (stop ~= 0), stop  = getStop; end;
   % handle the Break button
if (brk), plot_model(wp,h,E); break; end;
end;

plot_model(wp,h,E);
```

Results Running the above script will produce the graph shown in Fig. 10.50. The two top-left panels in Fig. 10.50 show the time and frequency responses of the unknown system for which this application is intended to provide an IIR model. The time and frequency responses for the model obtained by the adaptive filter are shown in the two top-right panels. The bottom-left panel shows the learning curve and the bottom-right panel shows the estimation error in the impulse response.

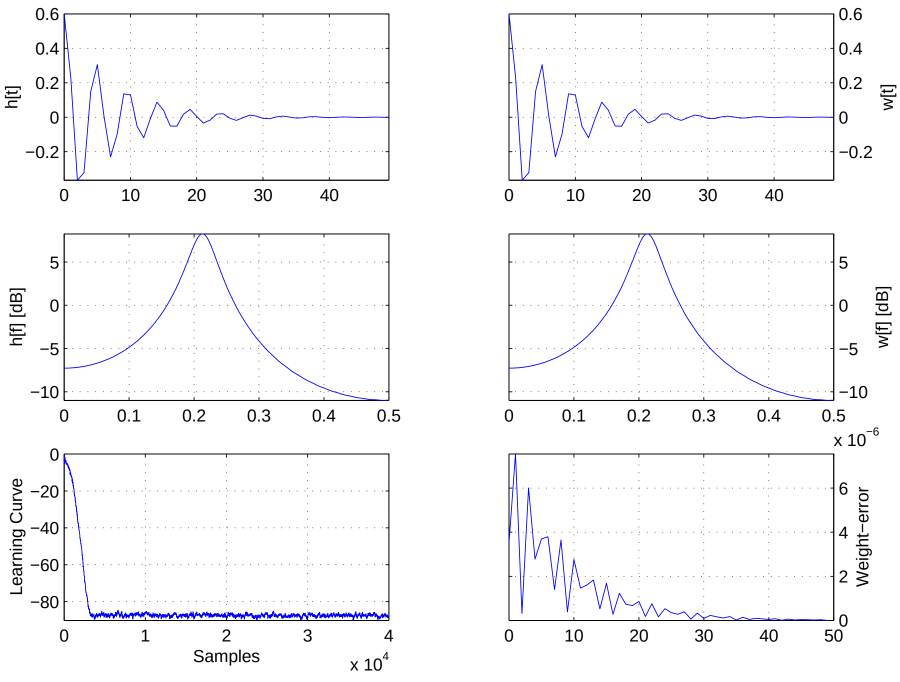

Figure 10.50: Performance of the output error algorithm in a system identification application.

See Also INIT_ OUTERR, ASPTOUTERR.

Reference [2] and [10] for introduction to recursive adaptive filters.

10.26 model_ rlslattice

Purpose Simulation of an adaptive forward modeling application using a lattice joint process estimator updated according to the RLS Lattice adaptive algorithm.

Syntax `model_rlslattice`

Description The block diagram of the system identification (forward modeling) problem using the RLS Lattice adaptive algorithm is shown in Fig. 10.51 (see Section 5.5 for more information on the RLS-Lattice algorithm). The RLS Lattice algorithm adjusts the PARCOR coefficients of the lattice predictor and the linear combiner coefficients simultaneously to minimize the forward and backward prediction errors as well as the modeling error $e(n)$ in the least squares sense. The input signal $x(n)$ (measured signal at the input of the system to be modeled) is stored in the file infile. The desired signal $d(n)$ (the signal measured at the system output in response to applying $x(n)$ at its input) is stored in the file dfile. First the variables for the RLS lattice are created and initialized using `init_rlslattice()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptrlslattice()` is called with a new input sample and a new desired sample to calculate the filter output (estimated desired signal) and update the adaptive model coefficients. This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

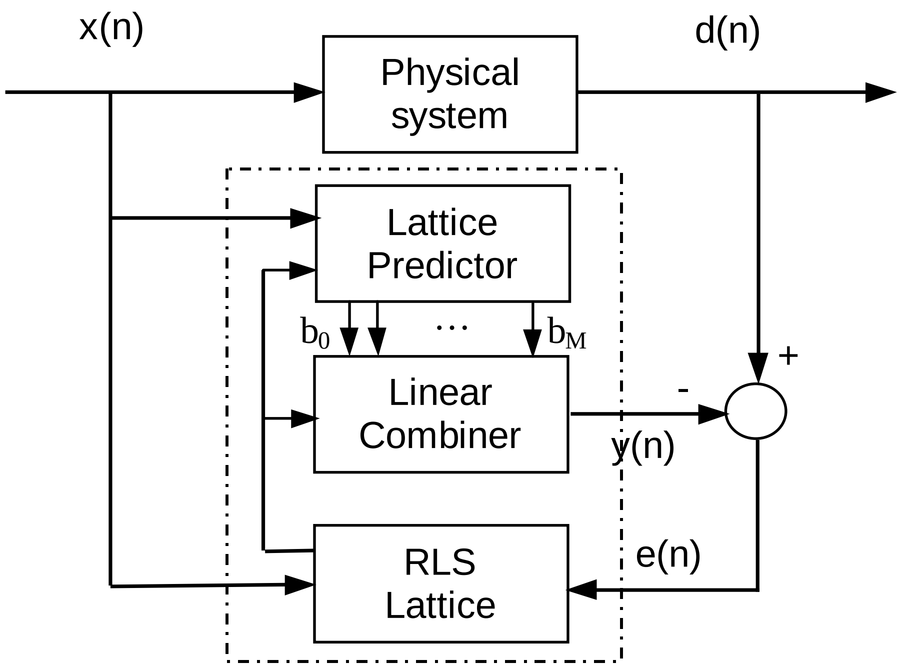

Figure 10.51: Block diagram of the RLS Lattice joint process estimator in a forward modeling application.

Code

```
clear all;
load .\data\h512;                  % for verification
infile = '.\wavin\scinwn.wav';   % input signal, white noise
dfile  = '.\wavin\scdwn512.wav'; % system output

M      = 512;                      % adaptive model length
a      = 0.999                     % forgetting factor

%% Initialize storage
% Init RLS Lattice algorithm
[ff,bb,fb,be,cf,b,d,y,e,kf,kb,w] = init_rlslattice(M);
[xn,inFs,inBits] = wavread(infile);  % read input signal
[dn,inFs,dBits]  = wavread(dfile);   % read desired signal
inSize = min(length(dn),length(xn)); % samples to process
E      = init_ipwin(inSize);         % Initialize IPWIN

%% Processing Loop
for (m=1:inSize)

   x = xn(m,:);  % new input sample
   d = dn(m,:);  % new desired output

   % Update the adaptive filter
   [ff,bb,fb,be,cf,b,y,e,kf,kb,w] = asptrlslattice(ff,...
                                    bb,fb,be,cf,b,a,x,d);
   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,d,'m',w,h512);

   % handle the Stop button
   while (stop ~= 0), stop = getStop; end;

   % handle the Break button
   if (brk), plot_model(w,h512,E); break; end;
end;

plot_model(w,h512,E);
```

Results

Running the above script will produce the graph shown in Fig. 10.52. The two top-left panels in Fig. 10.52 show the time and frequency responses of the unknown system for which this application is intended to provide a FIR model. The time and frequency responses for the model obtained by the adaptive filter are shown in the two top-right panels. The bottom-left panel shows the learning curve and the bottom-right panel shows the error in the filter coefficients by the end of the simulation.

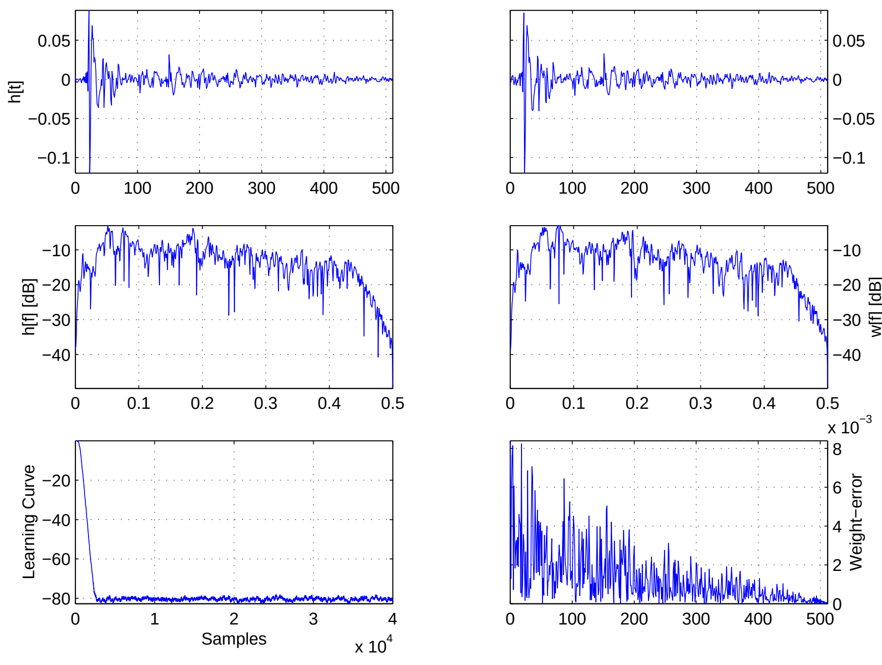

Figure 10.52: Performance of the RLS-Lattice algorithm in a system identification application.

See Also

INIT_ RLSLATTICE, ASPTRLSLATTICE.

Reference

[2] and [4] for analysis of the adaptive Lattice filters.

10.27 model_ sharf

Purpose Simulation of an adaptive forward modeling application using an recursive adaptive filter updated according to the Simple Hyperstable Adaptive Recursive Filter (SHARF) algorithm.

Syntax `model_sharf`

Description The block diagram of the system identification (forward modeling) problem using the SHARF adaptive algorithm is shown in Fig. 10.53 (see Section 6.4 for more information on the SHARF algorithm). The simulation considered here uses a recursive filter for the adjustable filter and the coefficients of the filter are updated using the SHARF algorithm. The input signal $x(n)$ (measured signal at the input of the system to be modeled) is stored in the file infile. The desired signal $d(n)$ (the signal measured at the system output in response to applying $x(n)$ at its input) is stored in the file dfile. First the variables for the adaptive IIR model $w(n)$ are created and initialized using `init_sharf()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptsharf()` is called with a new input sample and a new desired sample to calculate the filter output (estimated desired signal) and update the filter coefficients.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

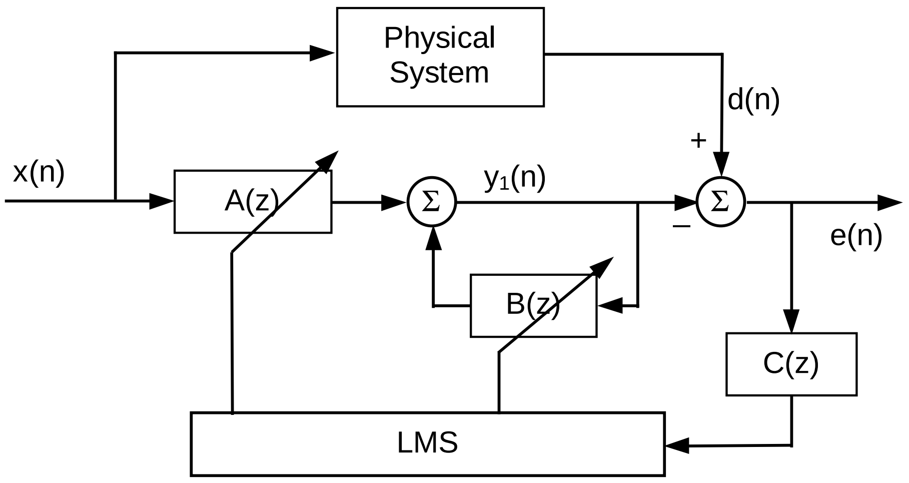

Figure 10.53: Block diagram of the forward modeling application using the SHARF algorithm.

Code

```
clear all;
infile = '.\wavin\scinwn.wav';    % input signal
dfile  = '.\wavin\scdar22.wav';   % desired signal

N  = 2;                           % number of zeros
M  = 2;                           % number of poles
L  = 5;                           % smoothing FIR length
H  = 50;                          % response length
p1 = .2 + j* .85;                 % unknown filter poles
p2 = .2 - j* .85;                 % for verification
ip = [1; zeros(H-1,1)];           % impulse vector
h  = filter([0.6 -.01],[1 -(p1+p2) (p1*p2)],ip);

% Initial parameters
u  = zeros(N+M,1);                % composite input vector
w  = u;                           % initial filter vector
c  = filter(.01,[1 -.99],ip(1:L));% error smoothing filter
e  = randn(L,1);                  % initial error vector
d  = randn(1,1);                  % initial desired sample
mu = [.01;0.01;.003;0.003] ;      % Step size vector

% Create and initialize EQERR IIR filter
[u,w,e,c,d,mu,Px,Py] = init_sharf(N,M,L,u,w,e,c,d,mu);
[xn,inFs,inBits] = wavread(infile);   % read input
[dn,inFs,dBits]  = wavread(dfile);    % read desired
inSize = max(length(dn),length(xn));  % samples to process
E      = init_ipwin(inSize);          % Initialize IPWIN

%% Processing Loop
for (m=1:inSize)
   x = 2^(inBits-1) * xn(m);      % input sample
   d = 2^(inBits-1) * dn(m);      % desired sample

   % update the adaptive coefficients
   [w,u,y,e,Px,Py]=asptsharf(N,M,w,u,x,d,e,c,mu,Px,Py);

   % impulse response for verification
   wp = filter(w(1:N),[1 ; -w(N+1:N+M)],ip);

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e(1),d, 'm', wp, h);

   % handle the Stop button
   while (stop ~= 0), stop  = getStop; end;

   % handle the Break button
if (brk), plot_model(wp,h,E); break; end;
end;

plot_model(wp,h,E);
```

Results

Running the above script will produce the graph shown in Fig. 10.54. The two top-left panels in Fig. 10.54 show the time and frequency responses of the unknown system for which this application is intended to provide an IIR model. The time and frequency responses for the model obtained by the adaptive filter are shown in the two top-right panels. The bottom-left panel shows the learning curve and the bottom-right panel shows the estimation error in the impulse response.

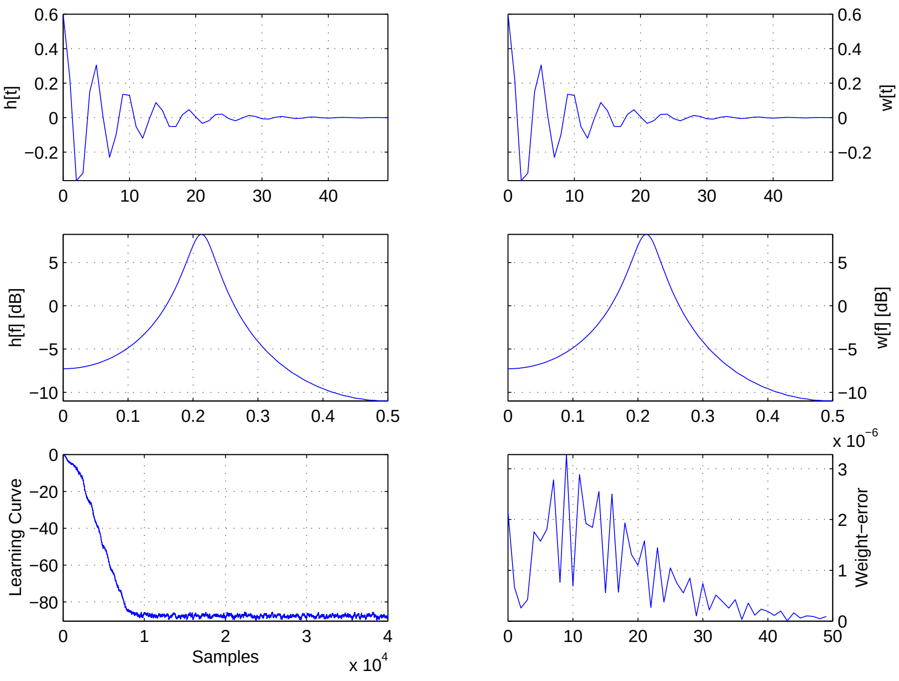

Figure 10.54: Performance of the SHARF IIR adaptive filter in a system identification application.

See Also

INIT_ SHARF, ASPTSHARF.

Reference

[2] and [10] for introduction to recursive adaptive filters.

10.28 model_ tdlms

Purpose Simulation of an adaptive forward modeling application using a transversal adaptive filter updated according to the Transform Domain LMS adaptive algorithm.

Syntax `model_tdlms`

Description The block diagram of the system identification (forward modeling) problem using the TDLMS adaptive algorithm is shown in Fig. 10.55. The simulation uses a transversal FIR filter for the adjustable filter and the coefficients of the filter are updated using the TDLMS algorithm. The input signal $x(n)$ (measured signal at the input of the system to be modeled) is stored in the file infile. The desired signal $d(n)$ (the signal measured at the system output in response to applying $x(n)$ at its input) is stored in the file dfile. First the variables for the adaptive model $\underline{\mathbf{W}}(n)$ are created and initialized using `init_tdlms()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptdlms()` is called with a new input sample and a new desired sample to calculate the filter output (estimated desired signal) and update the filter coefficients.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

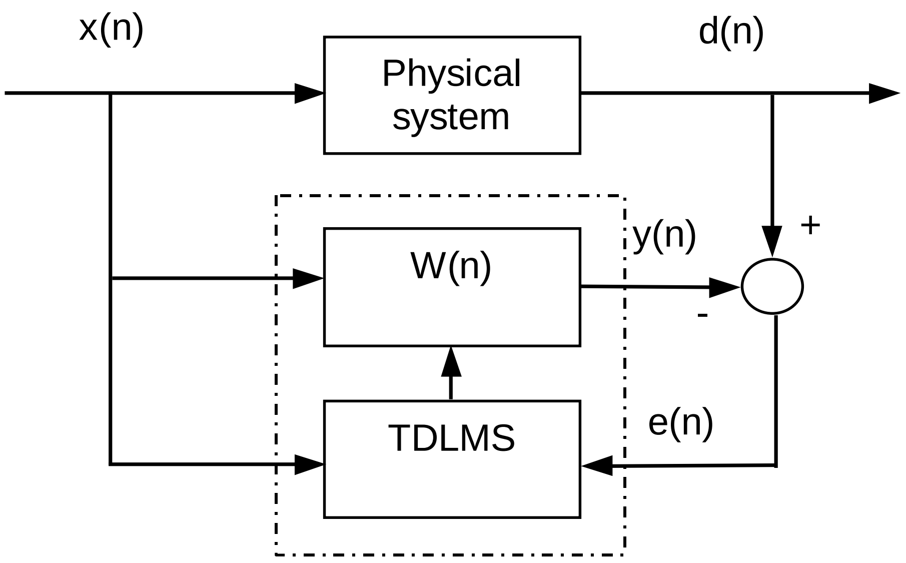

Figure 10.55: Block diagram of an FIR forward modeling using the TDLMS adaptive algorithm.

Code

```
clear all;
load .\data\h512;                    % for verification
infile = '.\wavin\scinwn.wav';       % input signal, white noise
dfile  = '.\wavin\scdwn512.wav';     % system output

L      = 512;                        % adaptive model length
mu     = 0.5/L;                      % Step size
b      = 0.98;                       % pole for power estimation
T      = 'fft';                      % Transform type

%% Initialize storage
[W,w,x,d,y,e,p]  = init_tdlms(L);    % Initialize TDLMS algorithm
[xn,inFs,inBits] = wavread(infile); % read input signal
[dn,inFs,dBits]  = wavread(dfile);  % read desired signal
inSize = min(length(dn),length(xn));% samples to process
E      = init_ipwin(inSize);         % Initialize IPWIN

%% Processing Loop
for (m=1:inSize)
   x  = [xn(m); x(1:L-1,:) ]; % update the input delay line
   d  = dn(m);                % get the new desired sample
   [W,y,e,p,w] = asptdlms(x,W,d,mu,p,b,T);

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,d,'m',w,h512);

   % handle the Stop button
   while (stop ~= 0), stop = getStop; end;

   % handle the Break button
   if (brk), plot_model(w,h512,E); break; end;
end;

plot_model(w,h512,E);
```

Results

Running the above script will produce the graph shown in Fig. 10.56. The two top-left panels in Fig. 10.56 show the time and frequency responses of the unknown system for which this application is intended to provide a FIR model. The time and frequency responses for the model obtained by the adaptive filter are shown in the two top-right panels. The bottom-left panel shows the learning curve and the bottom-right panel shows the error in the filter coefficients by the end of the simulation.

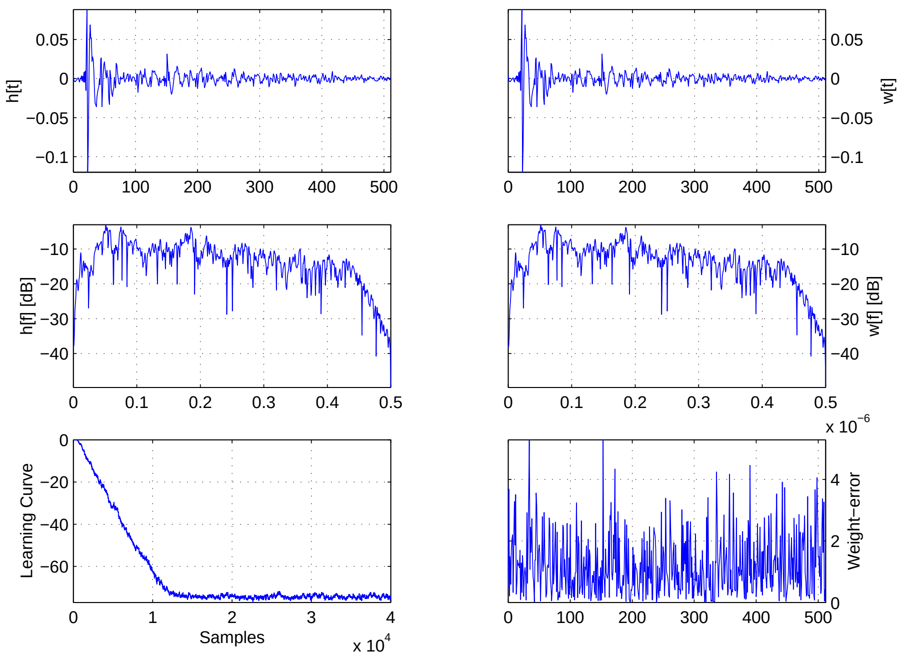

Figure 10.56: Performance of the Transform Domain LMS (TDLMS) adaptive filter in a system identification application.

See Also

INIT_ TDLMS, ASPTTDLMS.

Reference

[11], [2], and [4] for extensive analysis of the LMS and the steepest-descent search method.

10.29 model_ vsslms

Purpose Simulation of an adaptive forward modeling application using a transversal adaptive filter updated according to the Variable Step Size LMS adaptive algorithm.

Syntax `model_vsslms`

Description The block diagram of the system identification (forward modeling) problem using the VSSLMS adaptive algorithm is shown in Fig. 10.57 (see Section 4.20 for more information on the VSSLMS algorithm). The simulation considered here uses a transversal FIR filter for the adjustable filter and the coefficients of the filter are updated using the VSSLMS algorithm. The input signal $x(n)$ (measured signal at the input of the system to be modeled) is stored in the file infile. The desired signal $d(n)$ (the signal measured at the system output in response to applying $x(n)$ at its input) is stored in the file dfile. First the variables for the adaptive model $w(n)$ are created and initialized using `init_vsslms()`, and the input signals are read from files, then a processing loop is started. In each iteration of the loop `asptvsslms()` is called with a new input sample and a new desired sample to calculate the filter output (estimated desired signal) and update the filter coefficients.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

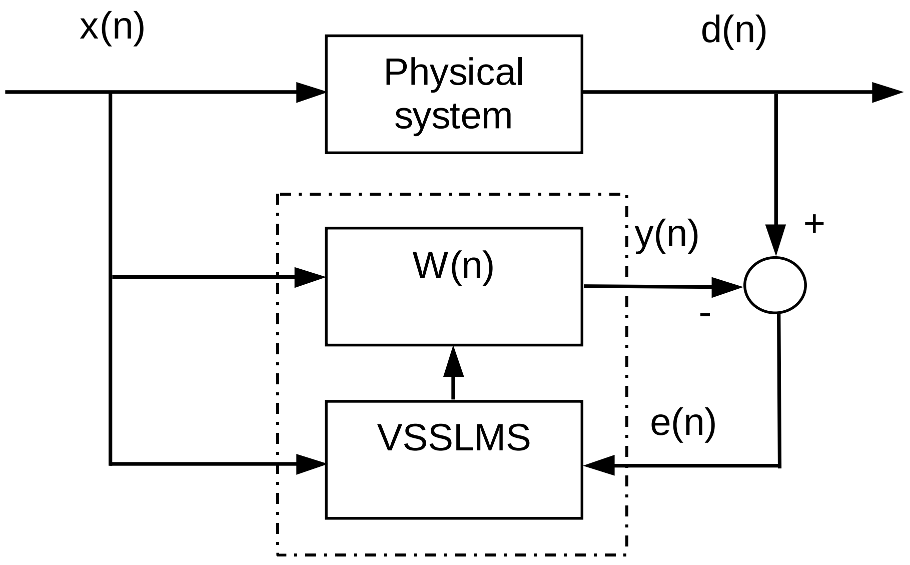

Figure 10.57: Block diagram of an FIR forward modeling using the VSSLMS adaptive algorithm.

Code

```
clear all;
load .\data\h32;                     % for verification
infile = '.\wavin\scinwn.wav';       % input signal, white noise
dfile  = '.\wavin\scdwn32.wav';      % system output

L      = 32;                         % adaptive model length
roh    = 1e-3;                       % adaptation constant of mu
mu_min = 1e-6;                       % lower bound for mu
mu_max = 0.99;                       % higher bound for mu

%% Initialize storage
[w,x,d,y,e,g,mu] = init_vsslms(L);  % Initialize VSSLMS
[xn,inFs,inBits] = wavread(infile); % read input signal
[dn,inFs,dBits]  = wavread(dfile);  % read desired signal
inSize = min(length(dn),length(xn));% samples to process
E      = init_ipwin(inSize);         % Initialize IPWIN
muv    = zeros(inSize,1);            % time evolution of mu

%% Processing Loop
for (m=1:inSize)
   x  = [xn(m); x(1:L-1,:) ];  % update the input delay line
   d  = dn(m);                 % get the new desired sample

   % Update the adaptive filter
   [w,g,mu,y,e] = asptvsslms(x,w,g,d,mu,roh,mu_min,mu_max);
   muv(m) = mean(mu);          % save average of mu

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,d,'m',w,h32);

   % handle the Stop button
   while (stop ~= 0), stop = getStop; end;

   % handle the Break button
   if (brk), plot_model(w,h32,E); break; end;
end;

plot_model(w,h32,E);
subplot(3,2,6);
plot(muv(1:m));grid
```

Results Running the above script will produce the graph shown in Fig. 10.58. The two top-left panels in Fig. 10.58 show the time and frequency responses of the unknown system for which this application is intended to provide a FIR model. The time and frequency responses for the model obtained by the adaptive filter are shown in the two top-right panels. The bottom-left panel shows the learning curve and the bottom-right panel shows the evolution of the average value of the step size vector during the simulation.

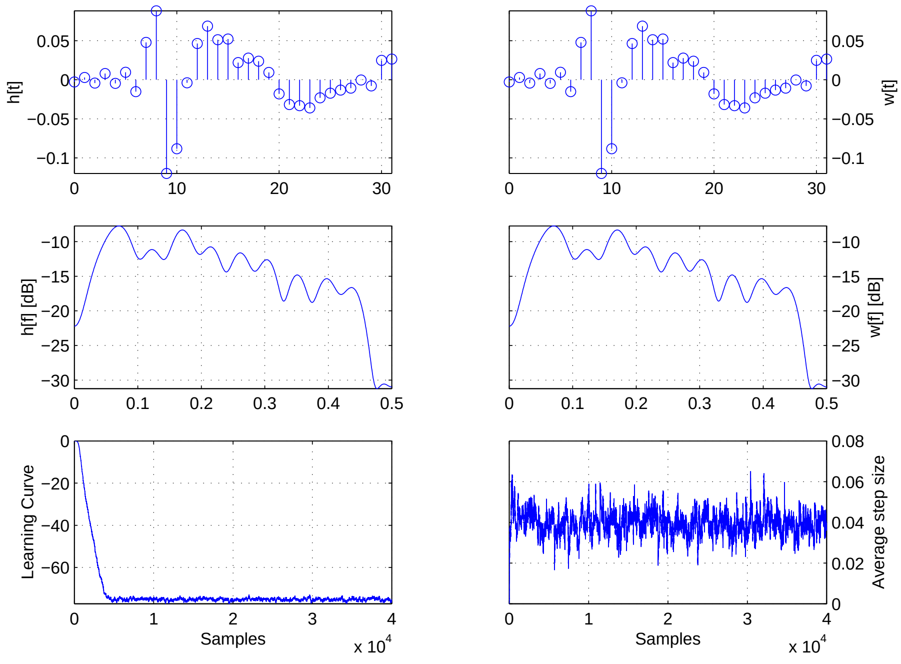

Figure 10.58: Performance of the variable step size LMS (VSSLMS) adaptive filter in a system identification application.

See Also INIT_ VSSLMS, ASPTVSSLMS, ASPTMVSSLMS.

Reference [11] and [4] for extensive analysis of the LMS and the steepest-descent search method.

10.30 predict_ lbpef

Purpose Simulation of an adaptive prediction application using the Lattice Backward Prediction Error Filter.

Syntax `predict_lbpef`

Description The input signal in this application is a speech fragment contaminated with white noise. The predictor will be able to estimate the speech only, which makes the predictor output containing less noise than its input. The error signal will contain the noise rejected by the predictor and any speech components that could not be estimated.
The block diagram of the lattice predictor used in this application is shown in Fig. 10.59. The input signal $x(n)$ (a speech fragment contaminated with white noise) is stored in the file infile. The application attempts to separate the speech from the noise and stores the former in the file outfile and the latter in errfile. First the variables for the adaptive lattice backward prediction error filter are created and initialized using `init_lbpef()`, and the input signal is read from file, then a processing loop is started. In each iteration of the loop `asptlbpef()` is called with a new input sample to calculate the predictor output $y(n)$ (estimated speech), the error sample $e(n)$ (the noise and residual unestimated speech) and update the PARCOR coefficients of the lattice predictor.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

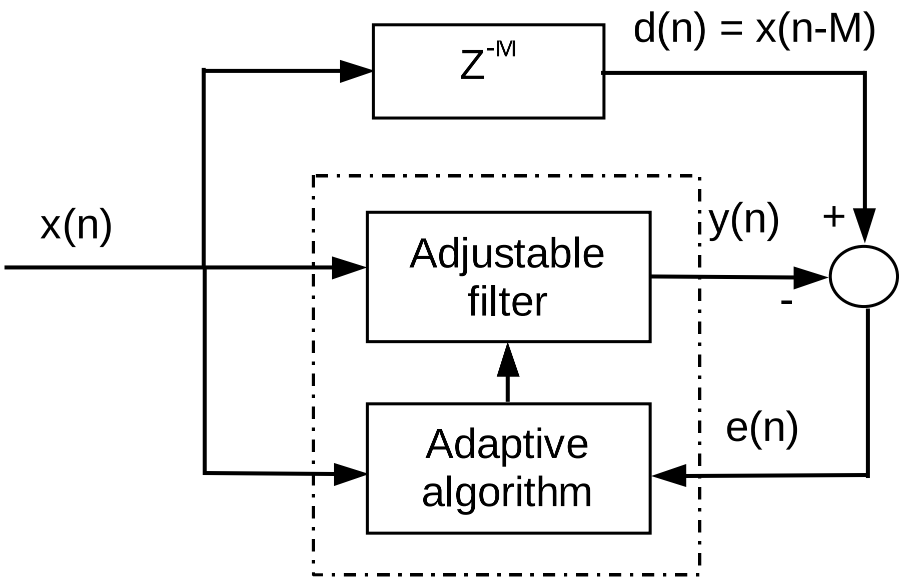

Figure 10.59: Block diagram of a prediction application using the lattice backward prediction error filter.

Code

```
clear all;
infile  = '.\wavin\wnaecfes.wav';    % input signal, speech
outfile = '.\wavout\lbpef_out.wav';  % predictor output
errfile = '.\wavout\lbpef_err.wav';  % predictor error
M       = 3;                         % filter length
mu_p    = 0.01;                      % Step size

%% Initialize storage
[k,b,P,e,y,x,c]  = init_lbpef(M);            % Init LFPEF
inSize           = wavread(infile, 'size');  % input data size
[xn,inFs,inBits] = wavread(infile);          % Read input signal
E                = init_ipwin(max(inSize));  % initialize IPWIN
out              = zeros(size(xn));          % estimated signal
err              = zeros(size(xn));          % prediction error

%% Processing Loop
for (m=1:inSize)
   % update input delay line
   x = [xn(m); x(1:end-1)];

   % update the PARCOR coefficients
   [k,b,P,e,y,c] = asptlbpef(k,b,P,x,mu_p);

   out(m) = y;    % save predictor output
   err(m) = e;    % save prediction error

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,xn(m),'p',xn,out,err);

   % handle the Stop button
while (stop ~= 0), stop = getStop; end;

   % handle the Break button
if (brk), plot_predict(xn,out,err,E); break; end;
end;

plot_predict(xn,out,err,E);

% save the predicted speech to file
wavwrite(out(1:m,:),inFs,inBits,outfile);

% save the prediction error to file
wavwrite(err(1:m,:),inFs,inBits,errfile);
```

Results

Running the above script will produce the graph shown in Fig. 10.60. In this graph, the top left panel shows the PEF input signal, the top right panel shows the prediction error, the bottom left panel shows the predictor output and the bottom right shows the ratio in dB between the power of the prediction error $e(n)$ and the power of the input signal $x(n)$.

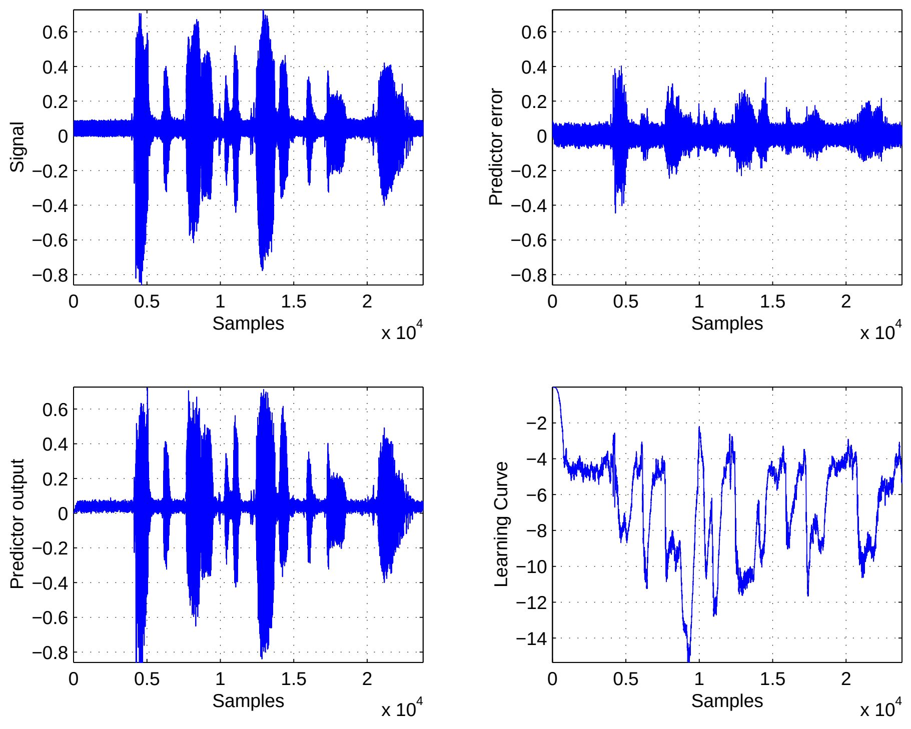

Figure 10.60: Performance of the Lattice Backward Prediction Error Filter in a prediction application.

Audio Files

The following files demonstrate the performance of the LBPEF in the application mentioned above.

`wavin\wnaecfes.wav`	input signal, speech + white noise.
`wavout\lbpef_out.wav`	predictor output, estimated speech.
`wavout\lbpef_err.wav`	prediction error, noise.

See Also

INIT_ LBPEF, ASPTLBPEF.

Reference

[2] and [4] for analysis of the adaptive Lattice filters.

10.31 predict_ lfpef

Purpose Simulation of a prediction application using the Lattice Forward Prediction Error Filter.

Syntax `predict_lfpef`

Description The input signal in this application is a speech fragment contaminated with white noise. The predictor will be able to estimate the speech only, which makes the predictor output containing less noise than its input. The error signal will contain the noise rejected by the predictor and any speech components that could not be estimated.
The block diagram of the lattice predictor used in this application is shown in Fig. 10.61. The input signal $x(n)$ (a speech fragment contaminated with white noise) is stored in the file infile. The application attempts to separate the speech from the noise and stores the former in the file outfile and the latter in errfile. First the variables for the adaptive lattice forward prediction error filter are creates and initializes using `init_lfpef()`, and the input signal is read from file, then a processing loop is started. In each iteration of the loop `asptlfpef()` is called with a new input sample to calculate the predictor output $y(n)$ (estimated speech), the error sample $e(n)$ (the noise and residual unestimated speech) and update the PARCOR coefficients of the lattice predictor.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

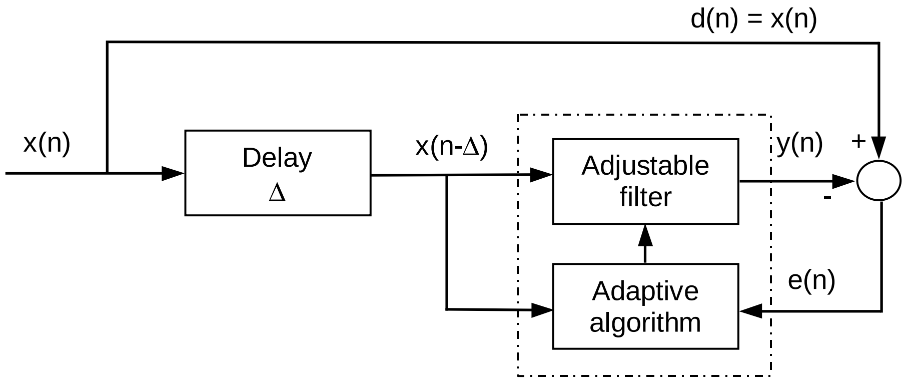

Figure 10.61: Block diagram of a prediction application using the lattice forward prediction error filter.

Code

```
clear all;
infile  = '.\wavin\wnaecfes.wav';    % input signal, speech
outfile = '.\wavout\lfpef_out.wav';  % predictor output
errfile = '.\wavout\lfpef_err.wav';  % predictor error
M       = 3;                         % filter length
mu_p    = 0.01;                      % Step size

%% Initialize storage
[k,b,P,e,y,c]    = init_lfpef(M);            % Init LFPEF
inSize           = wavread(infile, 'size');  % input data size
[xn,inFs,inBits] = wavread(infile);          % Read input signal
E                = init_ipwin(max(inSize));  % initialize IPWIN
out              = zeros(size(xn));          % estimated signal
err              = zeros(size(xn));          % prediction error

%% Processing Loop
for (m=1:inSize)

   % update the PARCOR coefficients
   [k,b,P,e,y] = asptlfpef(k,b,P,xn(m),mu_p);

   out(m) = y;    % save predictor output
   err(m) = e;    % save prediction error

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,xn(m),'p',xn,out,err);

   % handle the Stop button
   while (stop ~= 0), stop = getStop; end;

   % handle the Break button
if (brk), plot_predict(xn,out,err,E); break; end;
end;

plot_predict(xn,out,err,E);

% save the predicted speech to file
wavwrite(out(1:m,:),inFs,inBits,outfile);

% save the prediction error to file
wavwrite(err(1:m,:),inFs,inBits,errfile);
```

Results Running the above script will produce the graph shown in Fig. 10.62. In this graph, the top left panel shows the PEF input signal, the top right panel shows the prediction error, the bottom left panel shows the predictor output and the bottom right shows the ratio in dB between the power of the prediction error $e(n)$ and the power of the input signal $x(n)$.

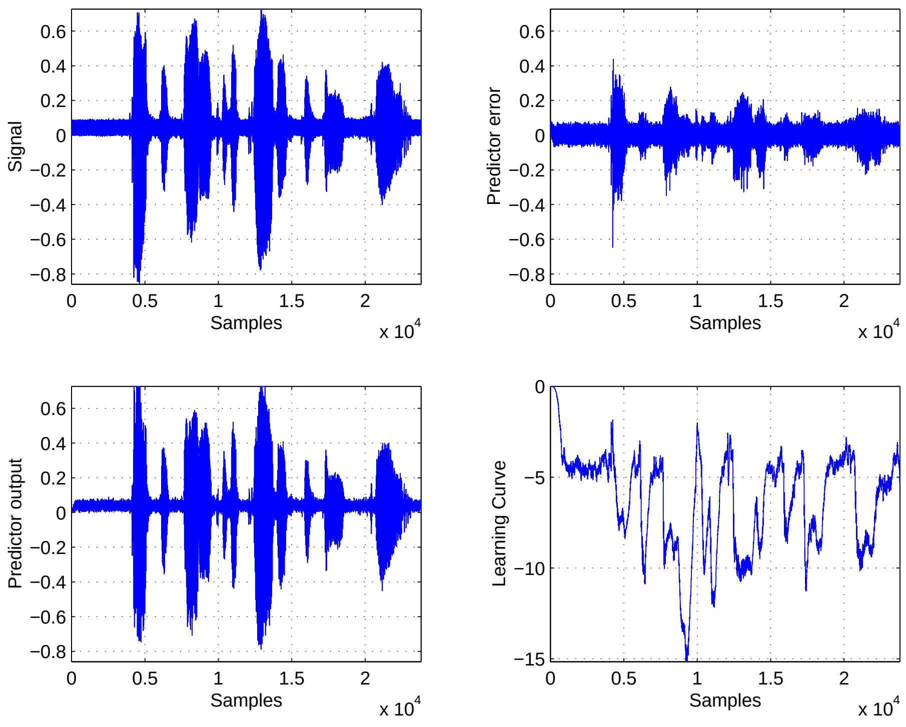

Figure 10.62: Performance of the Lattice Forward Prediction Error Filter in a prediction application.

Audio Files The following files demonstrate the performance of the LFPEF in the application mentioned above.

`wavin\wnaecfes.wav`	input signal, speech + white noise.
`wavout\lfpef_out.wav`	predictor output, estimated speech.
`wavout\lfpef_err.wav`	prediction error, noise.

See Also INIT_ LFPEF, ASPTLFPEF.

Reference [2] and [4] for analysis of the adaptive Lattice filters.

10.32 predict_ rlslbpef

Purpose Simulation of an adaptive prediction application using the RLS Lattice Backward Prediction Error Filter.

Syntax `predict_rlslbpef`

Description The input signal in this application is a speech fragment contaminated with white noise. The predictor will be able to estimate the speech only, which makes the predictor output containing less noise than its input. The error signal will contain the noise rejected by the predictor and any speech components that could not be estimated.
The block diagram of the lattice predictor used in this application is shown in Fig. 10.63. The input signal $x(n)$ (a speech fragment contaminated with white noise) is stored in the file infile. The application attempts to separate the speech from the noise and stores the former in the file outfile and the latter in errfile. First the variables for the adaptive lattice backward prediction error filter are created and initialized using `init_rlslbpef()`, and the input signal is read from file, then a processing loop is started. In each iteration of the loop `asptrlslbpef()` is called with a new input sample to calculate the predictor output $y(n)$ (estimated speech), the error sample $e(n)$ (the noise and residual unestimated speech) and update the PARCOR coefficients of the lattice predictor.
This simulation script uses the standard ASPT iteration progress window (IPWIN). The IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on pressing the break or stop buttons, the sensor signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

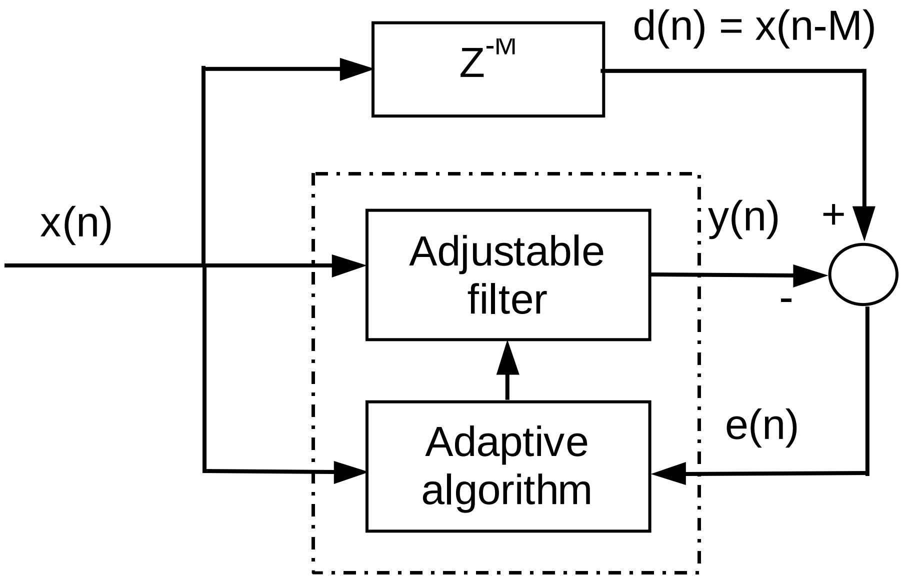

Figure 10.63: Block diagram of a prediction application using the RLS lattice backward prediction error filter.

Code

```
clear all;
infile  = '.\wavin\wnaecfes.wav';       % input signal, speech
outfile = '.\wavout\rlslbpef_out.wav';  % predictor output
errfile = '.\wavout\rlslbpef_err.wav';  % predictor error
M       = 3;                            % filter length
a       = 0.99;                         % forgetting factor

%% Initialize storage
[ff,bb,fb,cf,b,y,e,kf,kb,x] = init_rlslbpef(M);
inSize           = wavread(infile, 'size');  % input data size
[xn,inFs,inBits] = wavread(infile);          % Read input signal
E                = init_ipwin(max(inSize));  % initialize IPWIN
out              = zeros(size(xn));          % estimated signal
err              = zeros(size(xn));          % prediction error

%% Processing Loop
for (m=1:inSize)
   x = [xn(m); x(1:end-1)];

   % update the PARCOR coefficients
   [ff,bb,fb,cf,b,y,e,kf,kb,c]=asptrlslbpef(ff,bb,fb,cf,b,a,x);
   out(m) = y;    % save predictor output
   err(m) = e;    % save prediction error

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,xn(m),'p',xn,out,err);

   % handle the Stop button
   while (stop ~= 0), stop = getStop; end;

   % handle the Break button
if (brk), plot_predict(xn,out,err,E); break; end;
end;

plot_predict(xn,out,err,E);

% save the predicted speech to file
wavwrite(out(1:m,:),inFs,inBits,outfile);

% save the prediction error to file
wavwrite(err(1:m,:),inFs,inBits,errfile);
```

Results

Running the above script will produce the graph shown in Fig. 10.64. In this graph, the top left panel shows the PEF input signal, the top right panel shows the prediction error, the bottom left panel shows the predictor output and the bottom right shows the ratio in dB between the power of the prediction error $e(n)$ and the power of the input signal $x(n)$.

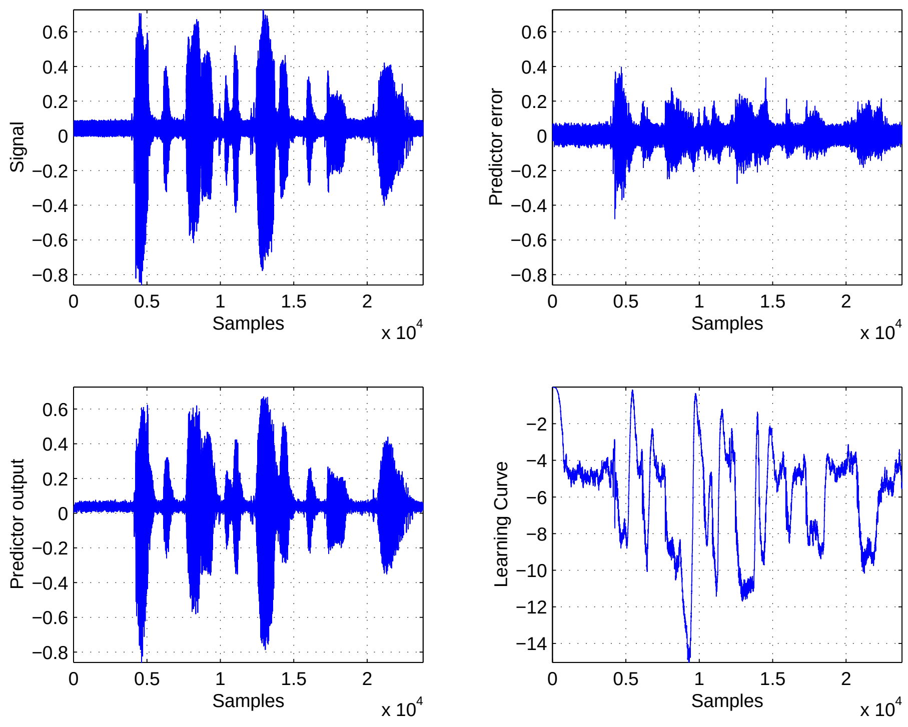

Figure 10.64: Performance of the RLS Lattice Backward Prediction Error Filter in a prediction application.

Audio Files

The following files demonstrate the performance of the LBPEF in the application mentioned above.

`wavin\wnaecfes.wav`	input signal, speech + white noise.
`wavout\rlslbpef_out.wav`	predictor output, estimated speech.
`wavout\rlslbpef_err.wav`	prediction error, noise.

See Also

INIT_ RLSLBPEF, ASPTRLSLBPEF.

Reference

[2] and [4] for analysis of the adaptive Lattice filters.

10.33 predict_ rlslfpef

Purpose Simulation of a prediction application using the RLS Lattice Forward Prediction Error Filter.

Syntax `predict_rlslfpef`

Description The input signal in this application is a speech fragment contaminated with white noise. The predictor will be able to estimate the speech only, which makes the predictor output containing less noise than its input. The error signal will contain the noise rejected by the predictor and any speech components that could not be estimated.
The block diagram of the lattice predictor used in this application is shown in Fig. 10.65. The input signal $x(n)$ (a speech fragment contaminated with white noise) is stored in the file infile. The application attempts to separate the speech from the noise and stores the former in the file outfile and the latter in errfile. First the variables for the adaptive lattice forward prediction error filter are created and initialized using `init_rlslfpef()`, and the input signal is read from file, then a processing loop is started. In each iteration of the loop `asptrlslfpef()` is called with a new input sample to calculate the predictor output $y(n)$ (estimated speech), the error sample $e(n)$ (the noise and residual unestimated speech) and update the PARCOR coefficients of the lattice predictor.
This simulation script uses the standard ASPT iteration progress window (IPWIN). An IPWIN has four buttons which allow you to stop and continue the simulation, show or hide the simulation graph window, break out of the processing loop, and quit the simulation. After processing all the samples, or on breaking out of the processing loop, the residual signal $e(n)$ is written to a wave audio file and a graph presenting the echo canceler performance is generated.

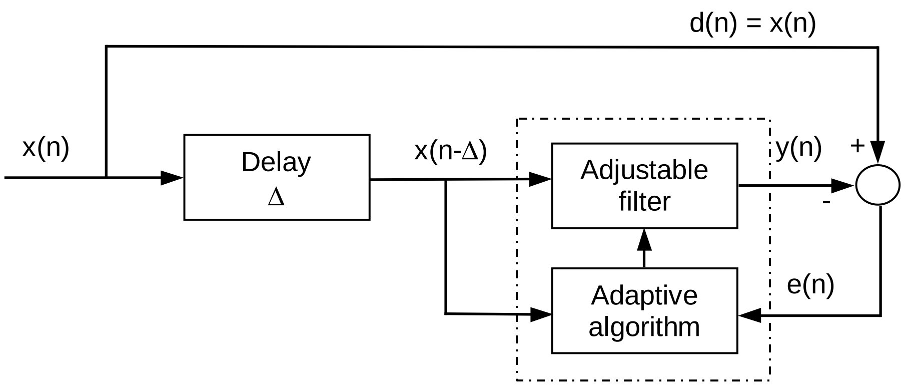

Figure 10.65: Block diagram of a prediction application using the RLS lattice forward prediction error filter.

Code

```
clear all;
infile  = '.\wavin\wnaecfes.wav';       % input signal, speech
outfile = '.\wavout\rlslfpef_out.wav';  % predictor output
errfile = '.\wavout\rlslfpef_err.wav';  % predictor error
M       = 3;                            % filter length
a       = 0.99;                         % forgetting factor

%% Initialize storage
[ff,bb,fb,cf,b,y,e,kf,kb] = init_rlslfpef(M);
inSize           = wavread(infile, 'size');  % input data size
[xn,inFs,inBits] = wavread(infile);          % Read input signal
E                = init_ipwin(max(inSize));  % initialize IPWIN
out              = zeros(size(xn));          % estimated signal
err              = zeros(size(xn));          % prediction error

%% Processing Loop
for (m=1:inSize)

   % update the PARCOR coefficients
   [ff,bb,fb,cf,b,y,e,kf,kb]=asptrlslfpef(ff,bb,fb,cf,b,a,xn(m));
   out(m) = y;    % save predictor output
   err(m) = e;    % save prediction error

   % update the iteration progress window
   [E, stop,brk] = update_ipwin(E,e,xn(m),'p',xn,out,err);

   % handle the Stop button
   while (stop ~= 0), stop = getStop; end;

   % handle the Break button
if (brk), plot_predict(xn,out,err,E); break; end;
end;

plot_predict(xn,out,err,E);

% save the predicted speech to file
wavwrite(out(1:m,:),inFs,inBits,outfile);

% save the prediction error to file
wavwrite(err(1:m,:),inFs,inBits,errfile);
```

Results Running the above script will produce the graph shown in Fig. 10.66. In this graph, the top left panel shows the PEF input signal, the top right panel shows the prediction error, the bottom left panel shows the predictor output and the bottom right shows the ratio in dB between the power of the prediction error $e(n)$ and the power of the input signal $x(n)$.

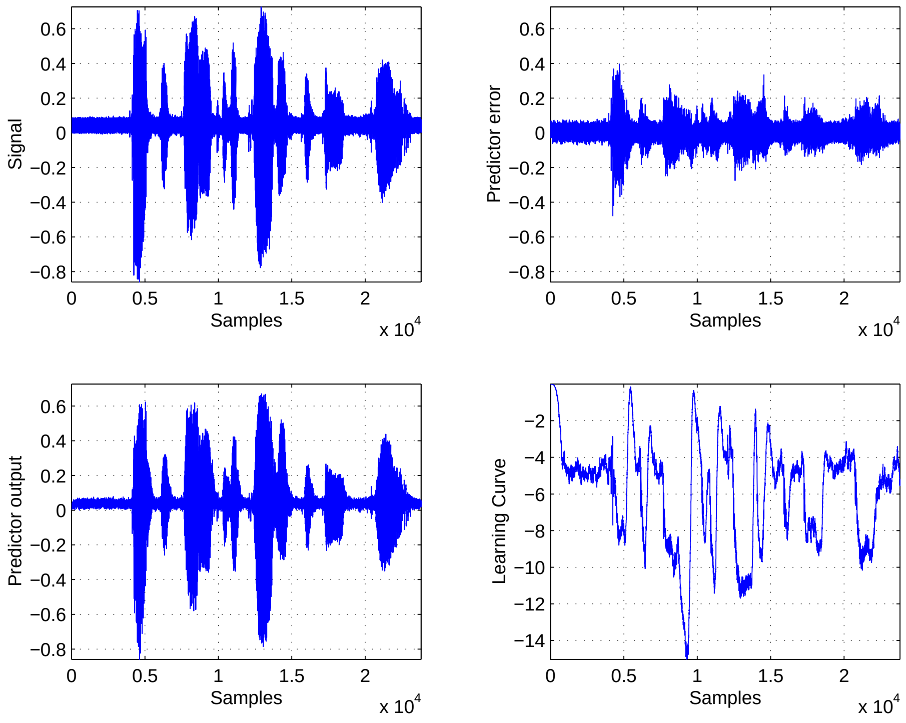

Figure 10.66: Performance of the RLS Lattice Forward Prediction Error Filter in a prediction application.

Audio Files The following files demonstrate the performance of the RLSLFPEF in the application mentioned above.

`wavin\wnaecfes.wav`	input signal, speech + white noise.
`wavout\rlslfpef_out.wav`	predictor output, estimated speech.
`wavout\rlslfpef_err.wav`	prediction error, noise.

See Also INIT_ RLSLFPEF, ASPTRLSLFPEF.

Reference [2] and [4] for analysis of the adaptive Lattice filters.

Bibliography

[1] G.P.M. Egelmeers, *Real Time Realization Concepts of Large Adaptive Filters*, PhD dissertation, Eindhoven University of Technology, Nov. 1995, ISBN 90-386-0456-4.

[2] B. Farhang-Boroujeny, *Adaptive Filters, Theory and Applications*, John wiley & sons Ltd, England, ISBN 0-471-98337-3.

[3] J. Garas, *Adaptive 3D Sound Systems*, Kluwer Academic Publishers, Boston, 2000, ISBN 0-7923-7907-1.

[4] S. Haykin, *Adaptive Filter Theory*, Printice Hall, London, 3rd edition, 1996.

[5] J.R. Treichler, C.R. Johnson Jr., and M.G. Larimore, *Theory and Design of Adaptive Filters*, John Wiley and Sons, 1996, ISBN 0-471-13424-4.

[6] S.M. Kuo and D.R. Morgan, *Active Noise Control Systems, Algorithms and DSP Implementations*, John Wiley and Sons, 1996, ISBN 0-471-13424-4.

[7] W. K. Jenkins, A.W. Hull, J.C. Strait, B.A. Schnaufer and Xiaohui Li, *Advanced Concepts in Adaptive Signal Processing*, Kluwer Academic Publishers, Boston, 1996, ISBN 0-7923-9740-1.

[8] A.V. Oppenheim and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, 1989, ISBN 0-13-216-771-9.

[9] P.C.W. Sommen, *Adaptive Filtering Methods*, PhD dissertation, Eindhoven University of Technology, The Netherlands, June 1992, ISBN 90-9005143-0.

[10] Victor Solo, and Xuan Kong, *Adaptive Signal Processing Algorithms*, Prentice-Hall Inc., Englewood Cliffs, 1995, ISBN 0-13-501263-5.

[11] B. Widrow and S.D. Stearns, *Adaptive Signal Processing*, Prentice-Hall Inc., Englewood Cliffs, 1985, ISBN 0-13-004029-0.