Appendix A

Introduction to MATLAB and Simulink

This appendix provides a quick reference for using MATLAB with its toolboxes and Simulink with its blocksets for DSP applications. These tools are used extensively in the experiments and examples in this book. This appendix covers useful topics related to DSP in MATLAB, the Signal Processing Toolbox, the Filter Design Toolbox, Simulink, the DSP Blockset, and the Fixed-Point Blockset. More detailed descriptions are documented in the MATLAB, toolbox, and Simulink user's guides listed in references at the end of this appendix.

A.1 USING MATLAB

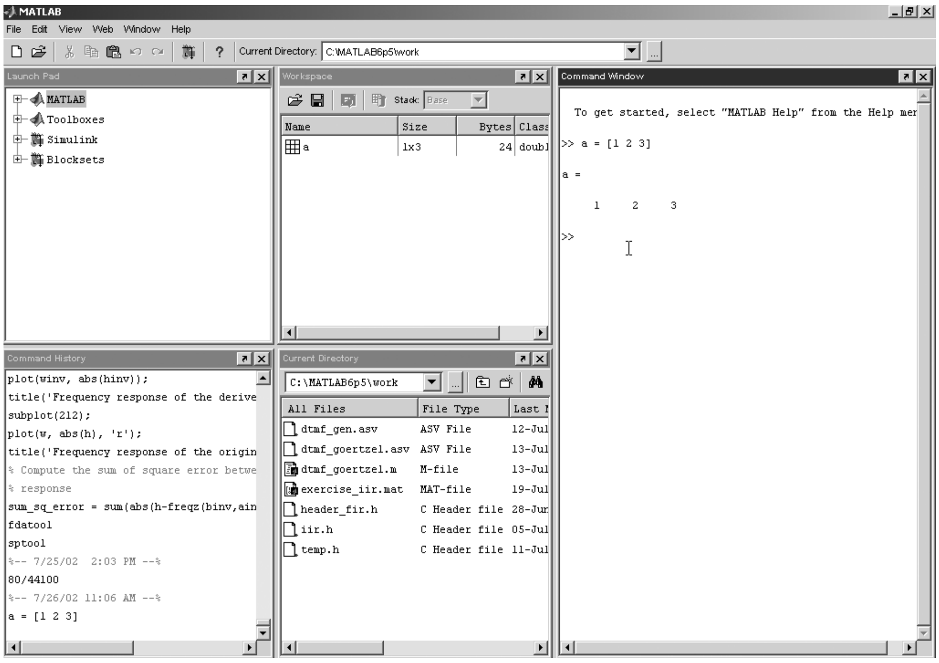
MATLAB stands for "MATrix LABoratory" and is a technical-computing language that allows the user to perform numerical computation, simulation, acquisition, and visualization of data, algorithm design, analysis, development, and implementation. Unlike other high-level programming languages such as C, MATLAB provides a comprehensive suite of mathematical, statistical, and engineering functions. The functionality is extended with interactive graphical capabilities for creating plots. Furthermore, extensive toolboxes are available for working under the MATLAB environment. Toolboxes are collections of algorithms, written by experts in their fields, that provide application-specific capabilities. These toolboxes enhance MATLAB's functionality in signal and image processing, data analysis and statistics, mathematical modeling, control system design, etc.

In addition to MATLAB and its toolboxes, there is another software package called Simulink for modeling, simulating, and analyzing dynamic systems. Simulink is integrated closely with the MATLAB environment. Variables and results derived from Simulink can be put in the MATLAB workspace for postprocessing and visualization. Like the toolboxes that extend MATLAB functions, many blocksets that add additional blocks to the Simulink environment are available. These blocksets include the DSP Blockset, Communication Blockset, and Fixed-Point Blockset.

A.1.1 Startup

To start MATLAB, double-click on the icon on the desktop. A MATLAB window is displayed, as shown in Fig. A.1. This window provides an integrated environment for developing MATLAB code. The command window is the main window in which the user keys MATLAB commands after the prompt >>. Some MATLAB commands are listed as follows:

1. The `help` command is used to show a list of programs installed in the MATLAB environment.
2. The `help topic` command is used to display the usage of a particular MATLAB `topic` or syntax.
3. The `demo` command is used to bring up a demo window.



Name	Size	Bytes	Class
a	1x3	24	doubl

```
 To get started, select "MATLAB Help" from the Help mer
>> a = [1 2 3]
a =
    1    2    3
>>
```

```
plot(winv, abs(hinv));
title('Frequency response of the derive
subplot(212);
plot(w, abs(h), 'r');
title('Frequency response of the origin
% Compute the sum of square error betwe
% response
sum_sq_error = sum(abs(h-freqz(binv,ain
fdatool
sptool
%-- 7/25/02  2:03 PM --%
80/44100
%-- 7/26/02 11:06 AM --%
a = [1 2 3]
```

All Files	File Type	Last I
dtmf_gen.asv	ASV File	12-Jul
dtmf_goertzel.asv	ASV File	13-Jul
dtmf_goertzel.m	M-file	13-Jul
exercise_iir.mat	MAT-file	19-Jul
header_fir.h	C Header file	28-Jur
iir.h	C Header file	05-Jul
temp.h	C Header file	11-Jul

Figure A.1 MATLAB environment (version 6.5) using a five-panel desktop layout

4. The `whos` command is used to display a list of variables currently loaded into the workspace.
5. The `what dir` command is used to display the files in the directory `dir`.
6. The `clear all` command is used to clear all of the variables.
7. The `diary ('record.out')` command is used create a file called `record.out` that contains all of the functions entered in the command window. The command `diary('off')` turns off the record mode.
8. The `quit` command is used to exit MATLAB.

In addition to the **Command Window** shown in the right side of Fig. A.1, there are **Command History** and **Current Directory** windows located in the bottom left of the screen. The **Command History** window records all of the executed commands, as well as the date and time when these commands were executed. This feature is useful in recalling the commands that have been executed previously. The **Current Directory** window keeps track of the files inside the current directory. In the upper-left screen are two default windows: **Launch Pad** and **Workspace**. The **Launch Pad** window contains all of the installed toolboxes that can be easily launched by double-clicking on the desired toolbox. The **Workspace** window is used to organize the loaded variables and also displays the size of variables, bytes, and class.

A set of toolbars at the top of the screen performs the following functions:

New file, open file

Cut, copy, paste

Undo last action, redo last action

Simulink library browser (to be discussed in Section A.3)

Open help window

The MATLAB environment also includes the following features under the **File** pull-down menu:

1. Import data from a file directory into the MATLAB workspace.
2. Save the MATLAB workspace to a file.
3. Set path, which allows commonly used files in the set directory to be searched.
4. Preference, which allows the user to specify the window's color and font size, number display formats, editor used, print options, figure copy options, etc.

A.1.2 Useful Syntax

MATLAB uses very simple programming notations to execute mathematical statements. MATLAB syntax is expressed in a matrix-style operation. The user can represent and manipulate scalars, vectors, and matrices in the MATLAB workspace, as shown by the examples summarized in Table A.1.

A.1.3 Plots

MATLAB is known for its outstanding graphics capabilities. Both 2-D and 3-D graphics can be plotted and manipulated using some commonly used commands summarized in Table A.2.

MATLAB version 6 (or higher) provides a set of powerful editing tools for editing graphics. It supports a point-and-click editing mode to modify the plots.

TABLE A.1 Some Useful MATLAB Syntaxes and Examples

Input	Comment
`g = [1 2 3]`	`g` is a row (1×3) vector
`p = [1 2 3]'`	`p` is a column (3×1) vector
`G = [1 2 3; 4 5 6; 7 8 9]`	`G` is a 3×3 matrix
`g*g'`	Performs an inner-product operation
`g.*g`	Performs element-by-element multiplication
`g+g`	Performs element-by-element addition
`G*g`	Performs matrix-vector multiplication
`r = 0:2:10`	`r` is a six-element vector = [0 2 4 6 8 10] is used to separate the start and end
`G(1,:)`	Selects the first row of `G`
`G(:,1)`	Selects the first column of `G`
`G(1:2,1:2)`	Selects the upper 2×2 submatrix of `G`
`G^2`	Performs G^2 (square of the matrix)
`G.^2`	Performs an element-by-element square
`[g, g]`	Concatenates the row vector
`[g; g]`	Places the next vector in the next row
`ones (1,3)`	Creates a 1×3 vector of all elements = 1
`zeros (2,3)`	Creates a 2×3 matrix of all elements = 0
`save result`	Saves all variables to file `result.mat`
`save result1 x,y,z`	Saves variables `x`, `y`, and `z` to the file `result1.mat`
`load result.mat`	Loads all variables from the file `result.mat`
`clear all`	Clears all of the workspace variables

TABLE A.2 MATLAB Commands and Examples for 2-D and 3-D Graphics

Input	Comment

`t = 0:0.001:1;`
`x = exp(−10.*t);`
	Sets up the time axis
	Creates a vector `x`

`plot(t,x);`
`plot(t,x,'ro')`
	Line plot of the x-axis (`t`) vs. the y-axis (`x`)
	Line plot using a red circle

`figure`
`stem(t,x)`
`stem(t,x, 'filled')`
	Creates a new figure plot
	Discrete sequence plot terminated with an empty circle or full circle (`'filled'`)

`subplot(1,2,1);`
`plot(t,x);`
`subplot(1,2,2);`
`plot(x,t);`
`close all`
	Display window divided into one row and two columns
	Note: (x,y,z) = (number of rows, number of columns, index)
	Closes all of the figures

`x = 0:1:50;`
`y1 = sin(2*pi*x/8);`
`y2 = cos(2*pi*x/8);`
`plot (x,y1,x,y2)`
`plot (x,y1, 'r'); hold on;`
`plot (x,y2, 'g'); hold off;`
	Time index from 0 to 50, step = 1
	Generates two sinewaves
	
	Plots two sinewaves in the same plot
	Plots two sinewaves, one at a time in the same plot

`axis([0 10 −1 1])`
`grid on`
	Sets up the x-axis and y-axis limit
	Shows grid lines

`xlabel('time')`
`ylabel('amplitude')`
`title('Two sine waves')`
	Specifies the name of the x-label
	Specifies the name of the y-label
	Specifies the title of the plot

`t = 0:pi/50:5*pi`
`plot3(sin(t), cos(t),t)`
 	
`axis square; grid on`
	Prepares the coordinates
	Performs a 3-D plot whose coordinates are elements of x, y, and z
	Changes to a square axis plot with a grid

`[X,Y] = meshgrid ([−2:0.1:2]);`
`Z = X.*exp(−X.^2 − Y.^2);`
`mesh (X, Y, Z);`
`surf (X, Y, Z);`
`meshc (X, Y, Z);`
`meshz (X, Y, Z);`
`pcolor (X, Y, Z);`
`surfl (X, Y, Z);`
	Transforms the vector into an array `X,Y`
	Generates another array `Z`
	Mesh surface plot
	Surface plot
	Surface plot with a contour beneath
	Surface plot with a curtain
	Pseudo-color plot
	3-D-shaded surface with lighting

Graphical objects can be selected by clicking on the object, and multiple objects can be selected by shift-clicking. Several editing features are highlighted in Fig. A.2.

Besides selecting, moving, resizing, changing color, cutting, copying, and pasting graphic objects, MATLAB also provides tools for zoom-in, zoom-out, camera

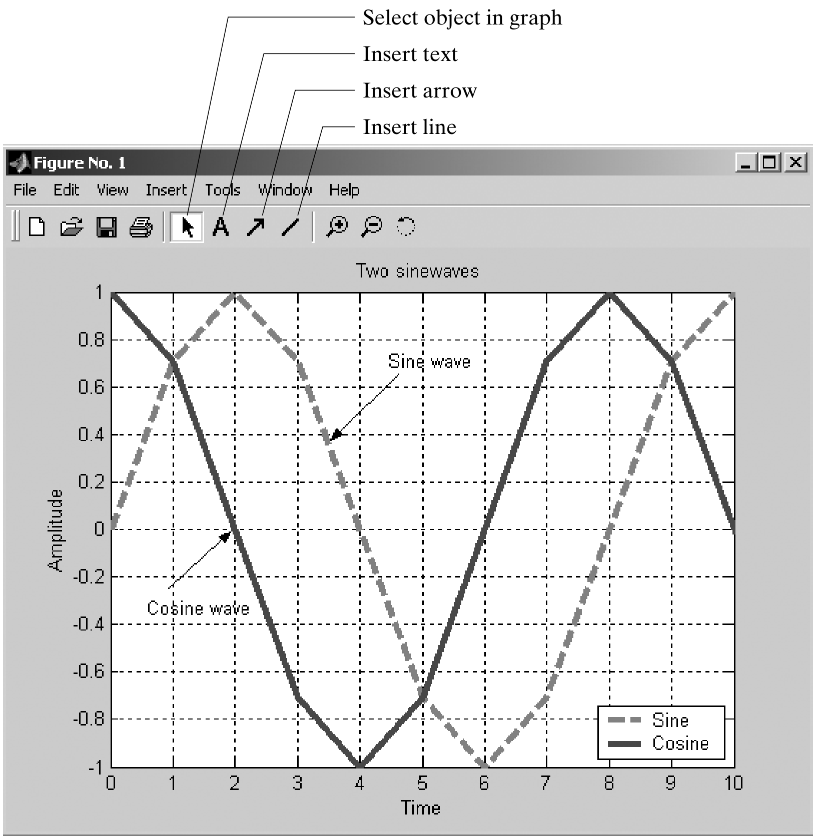


Figure A.2 Example of editing graphics

settings, basic data fitting, and displaying data statistics. These tools can be executed from the **Tools** pull-down menu. Once the diagram has been edited, the user can export the diagram into different graphical file types such as `.bmp`, `.jpeg`, etc., or save the diagram as a `.fig` file. A more detailed explanation can be found in the manual *Using MATLAB Graphics* [2].

A.1.4 Programming

We can enter and execute MATLAB commands directly in the command window. Since we may need to reuse these MATLAB commands and use several commands in sequence for a specific task, a better way is to use the MATLAB M-file editor (or another text editor) to write the MATLAB code (consisting of a sequence of commands) and save it into an M-file with an extension `.m`. The M-file editor can be activated by clicking on either the **New M-file** icon or the **Open File** icon.

There are two types of M-file: scripts and functions. Comments that start with a percent sign (%) can be inserted anywhere inside a function or script file. A script file has no input and output arguments and contains a sequence of MATLAB statements that make use of workspace data globally. To execute the M-file, simply type the name of the M-file in the command window and press **Enter** or **Return**. The user can interrupt a running program by pressing **Ctrl**−**C** or **Crtl**+**Break** at any time.

A function file must contain the word `function` in the first line of the program. For example,

```
function [mean,stdev] = stat(x)
% STAT - statistics of mean and standard deviation
n = length(x);
mean = sum(x)/n;
stdev = sqrt(sum((x-mean).^2)/n);
```

As shown in the example, a function can contain multiple output variables, which are enclosed in square brackets. Input arguments are enclosed in parentheses. All variables inside the function file are local and are not shared with the calling workspace.

Only the M-file function can be converted to a C (or C++) file. A script M-file cannot be compiled into a C file and thus needs to be converted to a function M-file first. Other M-files that cannot be compiled include an M-file that uses objects, an M-file that uses `input` and `eval` to manipulate a workspace, and MATLAB built-in functions. MATLAB supports C and C++ compilers such as (1) a C compiler (LCC C version 2.4), (2) Watcom C/C++ versions 10.6 & 11.0, (3) Borland C++ versions 5.0 onward, and (4) Microsoft Visual C++ versions 5.0 and 6.0. The compiler can be set up by issuing the following statement: `mex -setup`.

The compiler for building external interface files [MATLAB executable (MEX)] is chosen from the list of installed C compilers. The user can perform the following conversions:

- To convert an M-file to a C-file and create a C MEX-file, use

  ```
  mcc –x <M-filename>
  ```

- To convert an M-file to a C-file and create a Simulink S-file, use

  ```
  mcc –S <M-filename>
  ```

- To convert an M-file to a C-file and create a stand-alone C program, use

  ```
  mcc –m <M-filename>
  ```

- To convert an M-file to a C-file and create a stand-alone C++ application, use

  ```
  mcc –p <M-filename>
  ```

A.1.5 Data Types

It is important to understand the data types used in MATLAB since they affect the precision, the dynamic range, and quantization errors in the computation. By default, all MATLAB computations are carried out in double-precision, floating-point format

TABLE A.3 Data Types Used in MATLAB (adapted from [1])

Data type	Description
`single`	Single precision, floating-point (32 bits)
`double`	Double precision, floating-point (64 bits)
`int8, uint8`	Signed, unsigned integer (8 bits)
`int16, uint16`	Signed, unsigned integer (16 bits)
`int32, uint32`	Signed, unsigned integer (32 bits)
`char`	Character array (or string)
`cell`	Cell array has elements that contain other arrays
`structure`	Structure array has field arrays and contains other arrays
`user class`	User-specified class

(64 bits). However, data can be stored in single precision (32 bits) or integer (8 bits, 16 bits, and 32 bits) format to reduce memory requirements. Table A.3 lists some fundamental data types.

Using the data type as a prefix for numbers or variables can specify the precision. For example,

```
a = [1 2 3]       % a double precision array (default)
b = single(a)     % convert to single precision array
c = uint8(a)      % convert to unsigned 8-bit integer
```

Note that we can find the number of bytes used in representing the array by looking at the workspace window. Besides representing numbers in the required format, we can also display the numeric format without converting the data by specifying the display format in **File → Preference → Command Window → Text display → Numeric format**.

MATLAB allows data to be grouped under a single variable known as the cell array. For example, we can group three different variables as an array `X` and specify the cell array using curly braces {} as follows:

```
X = {[1 2 3], 'hello', eye(3)}    % define a cell array
X{1}                              % extract [1 2 3]
X{2}                              % extract 'hello'
```

An alternate form for specifying the preceding cell array by name is to use the `structure` data type as follows:

```
X.num = [1 2 3]
X.char = 'hello'
X.matrix = eye(3)
```

A.1.6 Useful Commands

Many useful MATLAB functions are commonly used for matrix and linear-algebra operations, as summarized in Table A.4. MATLAB also provides several Fourier analysis commands for signal processing and data analysis. These functions are listed in Table A.5.

More signal-processing functions are discussed in the next section on MATLAB toolboxes.

TABLE A.4 MATLAB Functions for Matrix and Linear Algebra (adapted from [1])

Function	Comment
`norm`	Matrix or vector norm
`rank`	Matrix rank
`det`	Determinant of matrix
`trace`	Sum of diagonal elements
`null`	Null space
`orth`	Orthogonalization

TABLE A.5 Fourier Analysis Function (adapted from [1])

Function	Comment
`fft`	FFT
`fft2`	Two-dimensional FFT
`fftn`	N-dimensional FFT
`ifft`	IFFT
`ifft2`	Two-dimensional IFFT
`ifftn`	N-dimensional IFFT
`abs`	Magnitude
`angle`	Phase angle
`unwrap`	Unwraps a phase angle in radian
`fftshift`	Moves a zeroth lag to the center of the spectrum
`cplxpair`	Sorts numbers into complex-conjugate pairs
`nextpow2`	Next higher power-of-two number

A.2 USING DIGITAL SIGNAL PROCESSING TOOLBOXES AND INTERACTIVE TOOLS

As mentioned earlier, a MATLAB toolbox contains many application-specific M-files to solve a particular problem. Commonly used DSP toolboxes include the Signal Processing Toolbox [3], Filter Design Toolbox [4], Communication Toolbox [5], Image Processing Toolbox [6], Wavelet Toolbox [7], etc. This section summarizes the signal processing functions and tools in the Signal Processing Toolbox and Filter Design Toolbox. The reader can refer to the user guides for more details.

A.2.1 Signal Processing Toolbox

The Signal Processing Toolbox contains many functions that perform signal-processing algorithms, such as filter design and implementation, spectral analysis, windowing, statistical signal processing, transforms, multirate signal processing, waveform generation, and other operations. In addition to these DSP functions, the toolbox also contains two useful interactive tools: (1) the Signal Processing Tool, which provides interactive tools for analyzing and filtering signals, and (2) the Filter Design and Analysis Tool, which provides advanced filter-design tools for designing digital filters, quantizing filter coefficients, and analyzing quantization effects. These tools are explained in the following sections.

Table A.6 lists some important signal-processing functions in the Signal Processing Toolbox. This list is grouped under different categories. To learn about the detailed usage of these functions, simply type

```
help function_name
```

to display the help menu for that particular function.

The signal-processing functions summarized in Table A.6, together with the powerful graphical tools, provide a comprehensive tool for signal processing. The Signal Processing Toolbox further provides an interactive tool that integrates the functions with the GUI, a topic which is discussed in the next section.

A.2.2 Signal Processing Tool

SPTool provides several tools for use in analyzing signals, designing and analyzing filters, filtering signals, and analyzing the spectrum of signals. The user can open this tool by typing

```
sptool
```

in the MATLAB command window. The **SPTool** main window appears, as shown in Fig. A.3.

Four windows can be accessed within SPTool:

1. The **Signal Browser** is used to view input signals. Signals from the workspace or file can be loaded into SPTool by clicking on **File → Import**. An **Import to SP-Tool** window appears, which allows the user to select data from the file or

TABLE A.6 Signal-Processing Functions (adapted from [3])

Filter analysis	Description
`filternorm`	Two-norm or inf-norm of a digital filter
`freqs`	Frequency response of an analog filter
`freqspace`	Frequency spacing for a frequency response
`freqz`	Frequency response of a digital filter
`freqzplot`	Plots frequency-response data
`grpdelay`	Group delay of a filter
`impz`	Impulse response of a digital filter
`unwrap`	Unwraps a phase angle
`zplane`	Zero-pole plot
Filter implementation	**Description**
`conv`	Convolution and polynomial multiplication
`conv2`	2-D convolution
`deconv`	Deconvolution and polynomial division
`fftfilt`	FFT-based FIR filtering using overlap-add
`filter`	FIR or IIR filtering
`filter2`	2-D digital filtering
`filtfilt`	Zero-phase digital filtering
`filtic`	Initial condition of a transposed form-II IIR filter
`latcfilt`	Lattice and lattice-ladder filtering
`medfilt1`	1-D median filtering
`sgolayfilt`	Savitzky–Golay filtering
`sosfilt`	Second-order (biquad) IIR filtering
`upfirdn`	Upsample, FIR filtering, and downsample
FIR filter design	**Description**
`convmtx`	Convolution matrix
`cremez`	Complex and nonlinear-phase equiripple FIR
`fir1`	Window-based FIR-filter design
`fir2`	Frequency sampling-based FIR-filter design
`fircls`	Constrained least-square FIR-filter design (multiband),
`fircls1`	(lowpass and highpass)
`firls`	Least-square linear-phase FIR-filter design
`firrcos`	Raised-cosine FIR-filter design
`intfilt`	Interpolation FIR-filter design
`kaiserord`	FIR-filter design using a Kaiser window
`remez`	Computes a Parks–McClellan optimal
`remezord`	FIR-filter design and filter-order estimation
`sgolay`	Savitzky–Golay filter design
IIR filter design	**Description**
`bilinear`	Bilinear transformation
`butter`	Butterworth filter design
`buttord`	Order and cutoff frequency for a Butterworth filter

(*Continued*)

TABLE A.6 (Continued)

IIR filter design	Description
`cheby1`	Chebyshev Type I filter design
`cheb1ord`	Order for a Chebyshev Type I filter
`cheby2`	Chebyshev Type II filter design
`cheb2ord`	Order for a Chebyshev Type II filter
`ellip`	Elliptic filter design
`ellipord`	Minimum order for an elliptic filter
`impinvar`	Impulse-invariance method
`maxflat`	Generalized-digital Butterworth filter design
`prony`	Prony's method for IIR-filter design
`stmcb`	Linear model using a Steiglitz-McBride iteration
`yulewalk`	Recursive digital-filter design (least-square)
Linear system transform	**Description**
`latc2tf`	Converts lattice parameters to a transfer function
`polystab`	Stabilizes a polynomial
`polyscale`	Scales the roots of a polynomial
`residuez`	z-transform partial-fraction expansion
`sos2ss`	Converts a second-order section to a state-space form
`sos2tf`	Converts a second-order section to a transfer function
`sos2zp`	Converts a second-order section to a zero-pole-gain form
`ss2sos`	Convert state-space parameters to a second-order form
`ss2tf`	Converts a state-space to a transfer function
`ss2zp`	Convert state-space parameters to a zero-pole-gain
`tf2latc`	Converts a transfer function to a lattice-filter form
`tf2sos`	Converts a transfer function to a second-order section
`tf2ss`	Converts a transfer function to a state-space
`tf2zp`	Converts a transfer function to a zero-pole-gain
`zp2sos`	Converts a zero-pole-gain to a second-order form
`zp2ss`	Converts a zero-pole-gain to a state-space form
`zp2tf`	Converts a zero-pole-gain to a transfer function
Windows	**Description**
`bartlett`	Bartlett window
`barthannwin`	Modified Bartlett–Hanning window
`blackman`	Blackman window
`blackmanharris`	Minimum four-term Blackman–Harris window
`bohmanwin`	Bohman window
`boxcar`	Rectangular window
`chebwin`	Chebyshev window
`gausswin`	Gaussian window
`hamming`	Hamming window
`hann`	Hann (Hanning) window
`Kaiser`	Kaiser window
`nuttallwin`	Nuttall-defined four-term Blackman–Harris

(*Continued*)

TABLE A.6 (Continued)

Windows	Description
`triang`	Triangular window
`tukeywin`	Tukey window
`window`	Window-function gateway
Transforms	**Description**
`bitrevorder`	Permutes input into a bit-reversed order
`czt`	Chirp *z*-transform
`dct`	Discrete cosine transform
`dftmtx`	DFT matrix
`fft`	1-D FFT
`fft2`	2-D FFT
`fftshift`	Moves zero-th lag to the center of the spectrum
`goertzel`	Second-order Goertzel algorithm
`hilbert`	Computes an analytic signal using a Hilbert transform
`idct`	Inverse-discrete cosine transform
`ifft`	1-D IFFT
`ifft2`	2-D IFFT
Cepstral analysis	**Description**
`cceps`	Complex cepstral analysis
`icceps`	Inverse-complex cepstrum
`rceps`	Real cepstrum, minimum phase reconstruction
Statistical and spectrum analysis	**Description**
`cohere`	Estimates the magnitude-square coherence function
`corrcoef`	Correlation coefficients
`corrmtx`	Autocorrelation matrix
`cov`	Covariance matrix
`csd`	Cross-spectral density
`pburg`	Power-spectrum density (PSD) estimate using Burg's method
`pcov`	PSD estimate using a covariance method
`peig`	PSD estimate using an eigenvector method
`periodogram`	PSD estimate using a periodogram method
`pmcov`	PSD estimate using a modified covariance method
`pmtm`	PSD estimate using a Thomson multitaper method
`pmusic`	PSD estimate using the MUSIC method
`psdplot`	Plots PSD data
`pwelch`	PSD estimate using Welch's method
`pyulear`	PSD estimate using the Yule–Walker autoregressive method
`rooteig`	Frequency and power estimation using the eigenvector algorithm
`rootmusic`	Frequency and power estimation using the MUSIC algorithm
`tfe`	Transfer function estimate
`xcorr`	Crosscorrelation function
`xcorr2`	2-D crosscorrelation
`xcov`	Covariance function

(*Continued*)

TABLE A.6 (Continued)

Linear prediction	Description
ac2rc	Autocorrelation sequence to reflection coefficients
ac2poly	Autocorrelation sequence to a prediction polynomial
is2rc	Inverse sine parameters to reflection coefficients
lar2rc	Log area ratios to reflection coefficients conversion
levinson	Levinson–Durbin recursion
lpc	Linear-predictive coefficients using autocorrelation
lsf2poly	Line-spectral frequencies to prediction polynomial
poly2ac	Prediction polynomial to an autocorrelation sequence
poly2lsf	Prediction polynomial to line-spectral frequencies
poly2rc	Prediction polynomial to reflection coefficients
rc2ac	Reflection coefficients to an autocorrelation sequence
rc2is	Reflection coefficients to inverse-sine parameters
rc2lar	Reflection coefficients to log-area ratios
rc2poly	Reflection coefficients to a prediction polynomial
rlevinson	Reverse Levinson–Durbin recursion
schurrc	Schur algorithm
Multirate signal processing	**Description**
decimate	Resamples at a lower sampling rate (decimation)
downsample	Downsamples an input signal
interp	Resamples data at a higher sample rate (interpolation)
interp1	General 1-D interpolation
resample	Changes the sampling rate by any rational factor
spline	Cubic spline interpolation
upfirdn	Upsample, FIR filtering, down sample
upsample	Upsample input signal
Waveform generation	**Description**
chirp	Swept-frequency cosine
diric	Dirichlet or periodic-sinc function
gauspuls	Gaussian-modulated sinusoidal pulse
gmonopuls	Gaussian monopulse
pulstran	Pulse train
rectpuls	Sampled aperiodic rectangle
sawtooth	Sawtooth (triangle) wave
sinc	Since function
square	Square wave
tripuls	Sampled aperiodic triangle
vco	Voltage-controlled oscillator

workspace. To view the signal, simply highlight it and click on **View**. The **Signal Browser** window, shown in Fig. A.4., allows the user to zoom-in and zoom-out from the signal, read the data values via markers, display the format, and even play the selected signal using the computer's speakers.

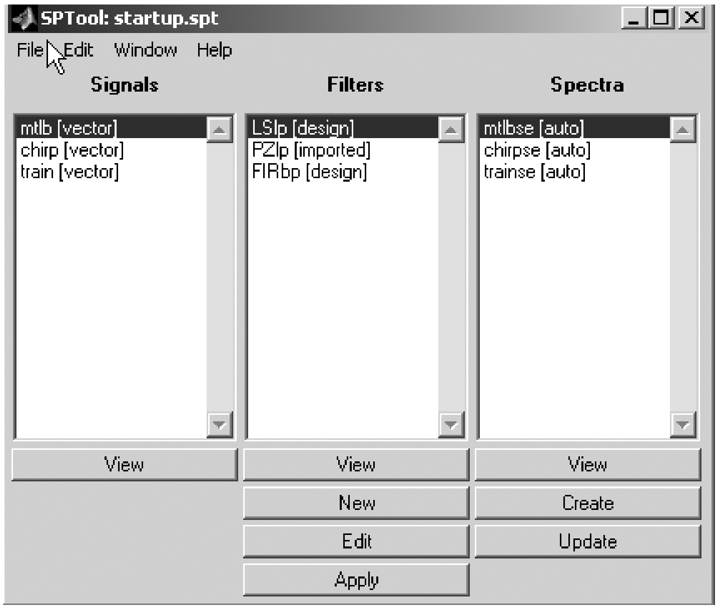
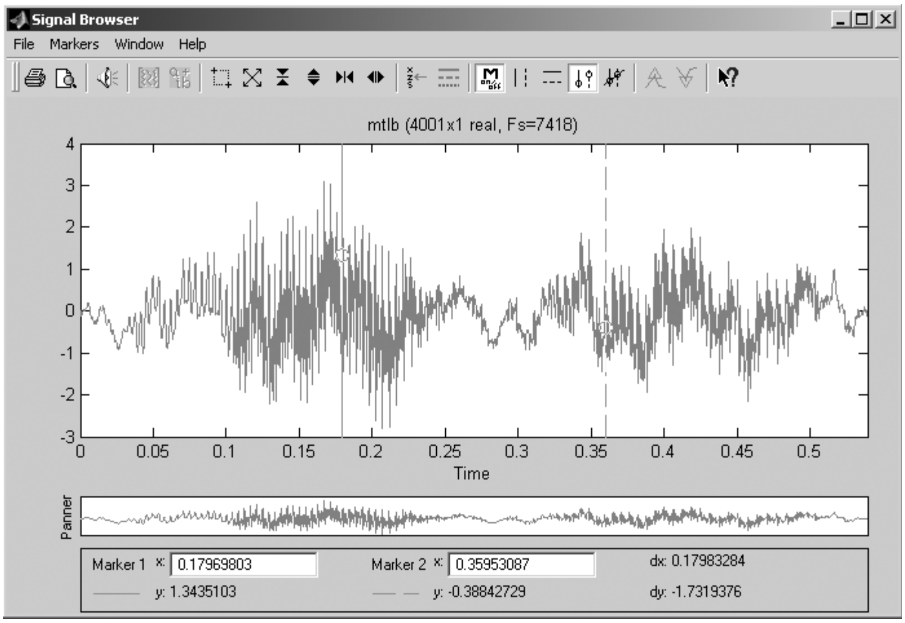


Figure A.3 **SPTool** window



Marker 1 x: 0.17969803 y: 1.3435103

Marker 2 x: 0.35953087 y: -0.38842729

dx: 0.17983284 dy: -1.7319376

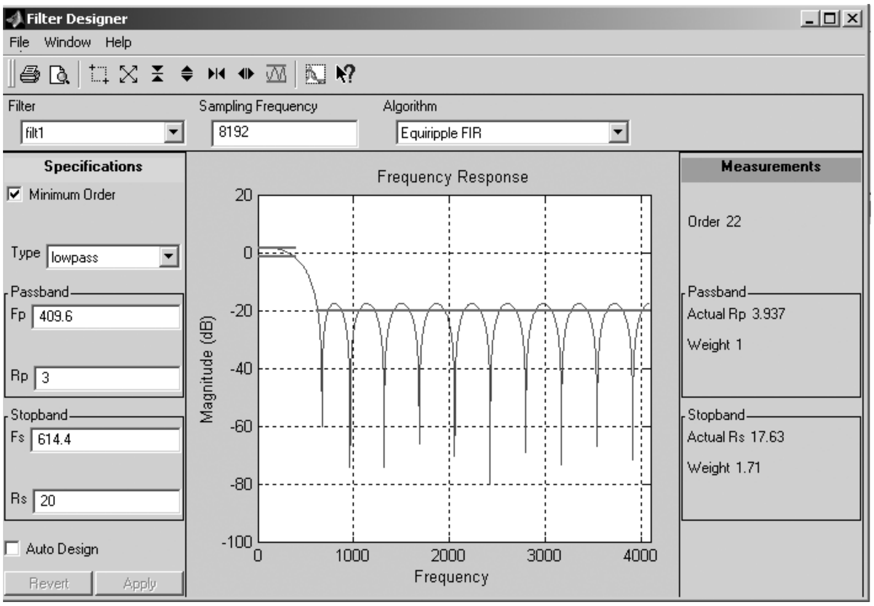
Figure A.4 **Signal Browser** window

2. The **Filter Designer** is used for designing digital FIR and IIR filters. The user simply clicks on the **New** icon for a new filter or the **Edit** icon for an existing filter under the **Filter** column in SPTool to open the **Filter Designer** window, as shown in Fig. A.5. The user can design lowpass, highpass, bandpass, and bandstop filters using different filter-design algorithms. In addition, the user can also design a filter using the **Pole/Zero Editor** to graphically place poles and zeros in the *z*-plane. A useful feature is the ability to overlay the input spectrum onto the frequency response of the filter by clicking on the **Frequency Magnitude/Phase** icon .

3. Once the filter has been designed, frequency specification and other filter characteristics can be verified by using the **Filter Viewer**. Select the name of the designed filter and click on the **View** icon under the **Filter** column in SPTool to open the **Filter Viewer** window, as shown in Fig. A.6. The user can analyze the filter in terms of its magnitude response, phase response, group delay, zero-pole plot, impulse response, and step response.

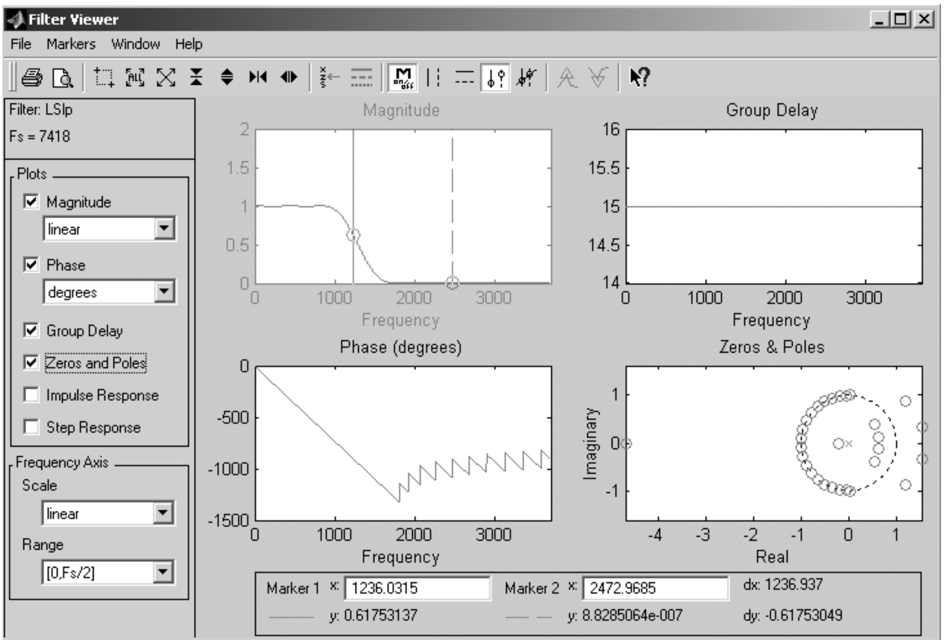
   When the filter characteristics have been confirmed, the user can then select the input signal and the designed filter. Click on the **Apply** button to perform filtering and generate the output signal. The **Apply Filter** window appears, as shown in Fig. A.7, and allows the user to specify the file name of the output signal. The **Algorithm** list provides a choice of several filter structures.

4. The final GUI window is the **Spectrum Viewer**, as shown in Fig. A.8. The user can view existing spectra by clicking on file names and then on the **View** button. Select the signal and click on the **Create** button to view the **Spectrum Viewer** window. The user can select one of the many spectral-estimation methods, such as Burg, covariance, FFT, modified covariance, MUSIC, Welch, Yule-Walker autoregressive (AR), etc., to implement the spectrum estimation. In



Filter: filt1
Sampling Frequency: 8192
Algorithm: Equiripple FIR
Minimum Order: ✓
Type: lowpass
Fp: 409.6
Rp: 3
Fs: 614.4
Rs: 20
Order 22
Actual Rp 3.937
Weight 1
Actual Rs 17.63
Weight 1.71

Figure A.5 **Filter Designer** window



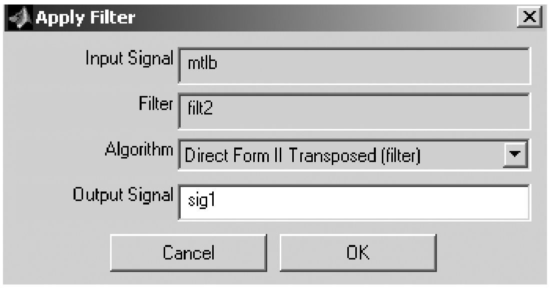
Filter: LSlp
Fs = 7418

Plots
☑ Magnitude — linear
☑ Phase — degrees
☑ Group Delay
☑ Zeros and Poles
☐ Impulse Response
☐ Step Response

Frequency Axis
Scale: linear
Range: [0,Fs/2]

Marker 1 x: 1236.0315 y: 0.61753137
Marker 2 x: 2472.9685 y: 8.8285064e-007
dx: 1236.937
dy: -0.61753049

Figure A.6 **Filter Viewer** window


Apply Filter

Input Signal: mtlb
Filter: filt2
Algorithm: Direct Form II Transposed (filter)
Output Signal: sig1

Cancel OK

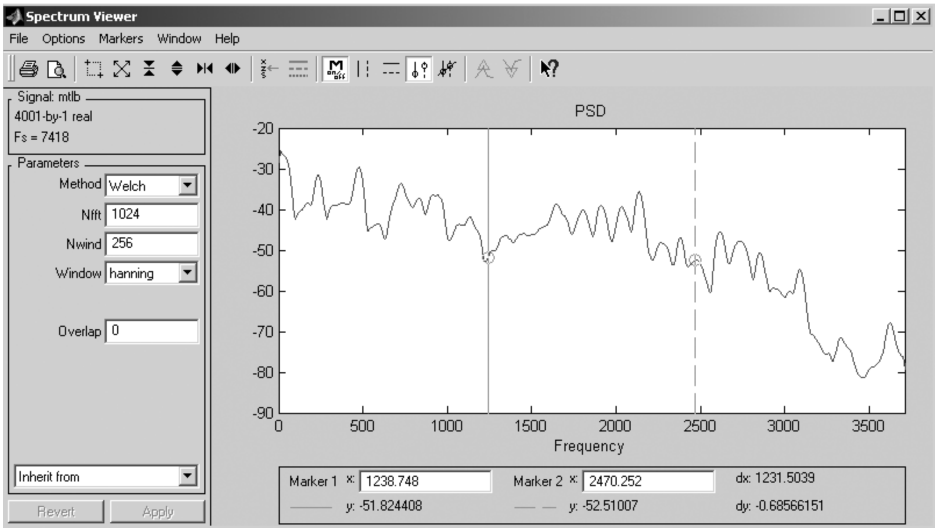
Figure A.7 **Apply Filter** window

addition, the size of the FFT, window functions, and overlapping samples can be selected to complete the power-spectrum density (PSD) estimation.

SPTool also provides a useful tool for exporting signals, filter parameters, and spectra to the MATLAB workspace or files. These saved parameters are represented in MATLAB as a structure of signals, filters, and spectra. More information can be found in the *Signal Processing Toolbox User's Guide* [3]. A step-by-step example of using SPTool in designing an IIR filter is given in Chapter 2.

A.2.3 Filter Design Toolbox

The Filter Design Toolbox is a collection of tools that provides advanced techniques for designing, simulating, and analyzing digital filters. It extends the capabilities of



Signal: mtlb
4001-by-1 real
Fs = 7418

Parameters
Method Welch
Nfft 1024
Nwind 256
Window hanning
Overlap 0
Inherit from

Marker 1 x: 1238.748 y: -51.824408
Marker 2 x: 2470.252 y: -52.51007
dx: 1231.5039
dy: -0.68566151

Figure A.8 **Spectrum Viewer** window

the Signal Processing Toolbox by adding filter structures and design methods for complex real-time DSP applications. This toolbox also provides functions that simplify the design of fixed-point filters and the analysis of quantization effects.

The toolbox provides the following key features for digital-filter designs:

1. *Advanced FIR filter-design methods*: The advanced equiripple FIR design automatically determines the minimum filter order required. It also provides constrained-ripple, minimum-phase, extra-ripple, and maximal-ripple designs. In addition, the least *P*-th norm FIR design allows the user to adjust the tradeoff between minimum-stopband energy and minimum order equiripple characteristics.
2. *Advanced IIR filter-design methods*: Allpass IIR filter design with arbitrary group delay enables the equalization of nonlinear group delays of other IIR filters to obtain an overall approximate linear-phase passband response. Least-*P*th-norm IIR design creates optimal IIR filters with arbitrary magnitude, and constrained least-*P*-th-norm IIR design constrains the maximum radius of the filter poles to improve the robustness of the quantization.
3. *Quantization*: The toolbox provides quantization functions for signals, filters, and FFTs. It also supports quantization of filter coefficients, including coefficients created using the Signal Processing Toolbox.
4. *Analysis of quantized filters*: The toolbox supports analysis of the frequency response, zero-pole plot, impulse response, group delay, step response, and phase response of quantized filters. In addition, it supports limit-cycle analysis for fixed-point IIR filters.

It is important to emphasize that the Filter Design Toolbox supports designing, simulating, and analyzing fixed-point filters for a wide precision range. It also allows the user to compute quantized FFTs and IFFTs. These functions ease the task of determining the effects of quantization on real-world designs. Quantization tools allow the user to model the behavior of fixed-point filters and FFT algorithms precisely. The quantized algorithms in the toolbox exactly match the output of the algorithms implemented on a fixed-point processor because the simulation is bit-true.

The Filter Design Toolbox includes a new GUI tool called FDATool. This tool allows the user to design optimal FIR and IIR filters from scratch, import previously designed filters, quantize floating-point filters, and analyze quantization effects. This tool is introduced in the next section.

A.2.4 Filter Design and Analysis Tool

This interactive tool provides several advanced techniques that support designing, simulating, and analyzing fixed-point and floating-point filters for a wide range of precision. This tool performs the following functions:

1. Designs filters
2. Converts filters between different structures
3. Quantizes filters
4. Quantizes data
5. Quantizes FFT and IFFT
6. Designs adaptive filters

In addition to the existing functions listed in Table A.6, FDATool contains the additional filter-design functions listed in Table A.7.

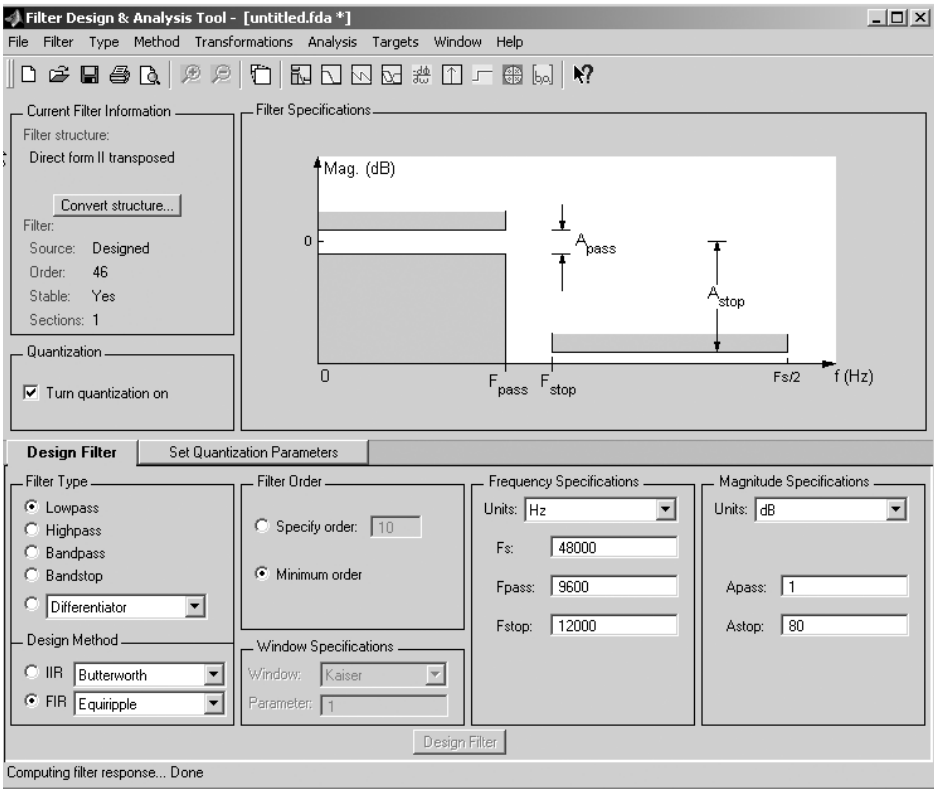
FDATool can be activated by typing

```
fdatool
```

in the MATLAB command window. The **Filter Design & Analysis Tool** window is shown in Fig. A.9. This window includes tools similar to those shown in the **Filter**

TABLE A.7 Additional Filter Design Functions in FDATool

Filter design function	Description
`firlpnorm`	Designs minimax FIR filters using the least-*P*th algorithm
`gremez`	Designs optimal FIR filters (Remez exchange)
`iirgrpdelay`	Designs IIR filters (specifies group delay in the passband)
`iirlpnorm`	Designs minimax IIR filters using the least-*P*th algorithm
`iirlpnormc`	Designs minimax IIR filters using the least-*P*th algorithm, which restricts filter poles and zeros within a fixed radius


Filter Design & Analysis Tool - [untitled.fda *]
File Filter Type Method Transformations Analysis Targets Window Help
Mag. (dB)
0
Apass
Astop
0
Fpass
Fstop
Fs/2
f (Hz)
Design Filter
Set Quantization Parameters

Current Filter Information
Filter structure:
Direct form II transposed
Convert structure...
Filter:
Source: Designed
Order: 46
Stable: Yes
Sections: 1

Quantization
[✓] Turn quantization on

Filter Specifications

Filter Type
● Lowpass
○ Highpass
○ Bandpass
○ Bandstop
○ Differentiator

Design Method
○ IIR Butterworth
● FIR Equiripple

Filter Order
○ Specify order: 10
● Minimum order

Window Specifications
Window: Kaiser
Parameter: 1

Frequency Specifications
Units: Hz
Fs: 48000
Fpass: 9600
Fstop: 12000

Magnitude Specifications
Units: dB
Apass: 1
Astop: 80

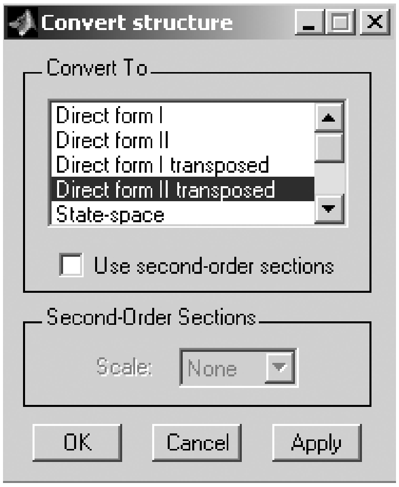
Design Filter

Computing filter response... Done

Figure A.9 **Filter Design & Analysis Tool** window

Designer window in SPTool (Fig. A.5). The design steps and available features that can be used to view filter characteristics are also similar. However, FDATool is a more advanced filter-design tool that includes additional filter types such as differentiator, Hilbert transform, multiband, arbitrary magnitude and group delay, Nyquist, and raised-cosine. FDATool also has an additional option that allows the default filter structure (direct-form II transposed) to be converted to different structures, such as direct-form I, direct-form II, direct-form I transposed, state-space, and its lattice equivalents, as shown in Fig. A.10.

In addition, FDATool is a very powerful tool for investigating quantized filters. Once a filter has been designed and verified, we can turn on the quantization mode by clicking on the **Set Quantization Parameters** icon located at the bottom-left side of window and then check the box **Turn quantization on**. The bottom panel of the **Filter Design & Analysis Tool** window changes, as shown in Fig. A.11. We can select (1) **Mode**, (2) **Round mode**, (3) **Overflow mode**, and (4) **Format** for representing coefficients, input, output, multiplicands, products, and sums of the filter. The options and explanations for setting these quantizer properties are listed in Table A.8. This new panel also provides a useful option for limiting filter coefficients to less than 1 by scaling the input of the filter, which can be accomplished by clicking on the



Convert structure

Convert To

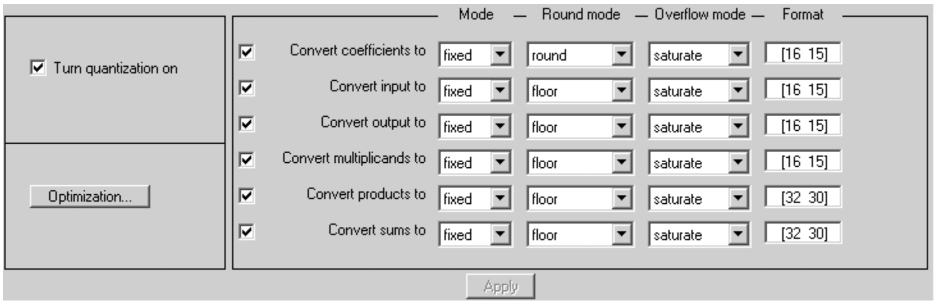
- Direct form I
- Direct form II
- Direct form I transposed
- [✓ Direct form II transposed]
- State-space

☐ Use second-order sections

Second-Order Sections

Scale: None

Figure A.10 **Convert Structure** window



☑ Turn quantization on

	Mode	Round mode	Overflow mode	Format
☑ Convert coefficients to	fixed	round	saturate	[16 15]
☑ Convert input to	fixed	floor	saturate	[16 15]
☑ Convert output to	fixed	floor	saturate	[16 15]
☑ Convert multiplicands to	fixed	floor	saturate	[16 15]
☑ Convert products to	fixed	floor	saturate	[32 30]
☑ Convert sums to	fixed	floor	saturate	[32 30]

Figure A.11 Setting quantization parameters in FDATool

Optimization button (the meanings of these options are explained in Chapter 7). Furthermore, the direct-form IIR filter can also be converted to a cascade of second-order sections for a more stable implementation. This option can be activated by clicking on **Edit** → **Convert to Second-Order** sections. This option is only applicable to IIR filters and is explained in more detail in Chapter 7.

In addition to these features, the new version of FDATool also provides an option for frequency transformation, which can be accessed by clicking on the **Transform Filter** icon. This option transforms the original designed filter to another filter with different characteristics. It also contains a **Filter Realization Wizard** (Realize Model) that allows the user to create a fixed-point or floating-point filter blockset, which can be used in the Simulink environment. Furthermore, filters can also be imported from variables or discrete-time filter objects in the workspace by clicking on the **Import Filter** icon.

Besides using FDATool in specifying the quantized filter, we can also construct a quantized filter object using the function `qfilt` in MATLAB. For example,

```
Hq=qfilt('quantizer',{'float',[32 8],'round'})
```

TABLE A.8 Quantizer Properties

Quantizer property	Description
Mode:	
`fixed`	Specifies fixed-point arithmetic (default)
`ufixed`	Unsigned fixed-point calculations
`float`	Specifies floating-point arithmetic
`double`	Specifies double-precision, floating-point arithmetic
`single`	Specifies single-precision, floating-point arithmetic
Round mode:	
`ceil`	Rounds a value to the nearest integer towards $+\infty$
`convergent`	As in `ceil` (If tie, round down if the next-to-last bit is even; up if odd)
`fix`	Rounds a value to the nearest integer toward 0
`floor`	Rounds a value to the nearest integer toward $-\infty$
`round`	Rounds a value to the nearest integer (default): rounds a negative number toward $-\infty$, a positive number toward $+\infty$, and ties toward $+\infty$
Overflow mode:	
`saturate`	Sets the overflowed values to the maximum or minimum values (default)
`wrap`	Maps overflow values to the number range using modular arithmetic
Format:	
`[wl,fl] (fixed)`	Default value is [16 15] for Q.15 Maximum wordlength `wl` = 53 bits, `fl`: fractional length
`[wl,exp] (float)`	`exp` up to 11 bits; 64-bit `>wl>exp,` `exp`: exponent length
`[32,8] (single)`	IEEE-754 single precision (`exp` = 8, `fl` = 23, `sign` = 1)
`[64,11] (double)`	IEEE-754 double precision (`exp` = 11, `fl` = 52, `sign` = 1)

creates a quantized-filter object that is quantized to single-precision, floating-point format and that uses round mode for rounding coefficients and arithmetic results. The object is defined as a structure data type, and direct referencing can modify its properties after the object is constructed. For example, to change the value of the scaling factor at the input of the filter to 0.5, we can use the following command:

```
Hq.scalevalues = 0.5
```

FDATool also allows the user to construct the other quantized objects listed in Table A.9. A detailed explanation can be found in the *Filter Design Toolbox User's Guide* [4].

A useful report showing the minimum, maximum, number of overflows, number of underflows, and number of operations of the most recent application of `F` (quantized filter object, quantized FFT object, or quantizer object) can be produced by issuing the following command:

```
qreport(F)
```

TABLE A.9 Quantizer, Quantized Filter, and Quantized FFT

Quantized function	Description
`qfft`	Constructor for quantized FFT objects
`qfilt`	Constructor for quantized filter objects
`quantizer`	Constructor for quantizer objects
`unitquantizer`	Constructor for unit-quantizer objects

Finally, the latest FDATool also provides a set of adaptive-filter functions using different adaptive algorithms for updating the filter coefficients listed in Table 9.1. Examples of using adaptive-filter functions are given in Chapter 9.

A.3 USING SIMULINK

Simulink provides a useful interactive interface for use in designing dynamic systems. This section introduces some basic functional blocks and their operations. Detailed information can be found in the *Simulink*: *User's Guide* [8]. To start Simulink from the MATLAB environment, type the following command:

```
simulink
```

or click on the Simulink icon at the top of the MATLAB window. A **Simulink Library Browser** appears, which contains the main Simulink blocks and all of the available Simulink blocksets, as shown in Fig. A.12. In this section, we examine the main Simulink Block Library. The DSP Blockset [9] and the Fixed-Point Blockset [10] are explained in Section A.4.

Right-click on **Simulink**, shown in Fig. A.12, and click on the menu **Open the 'Simulink' Library** to open the Simulink **Library** window, as shown in Fig. A.13. The functional blocks (libraries) include the following:

1. **Continuous** contains a set of blocks that perform numeric derivative, integration, delay, etc.
2. **Discrete** contains a set of blocks that perform discrete-dynamic systems, zero-order hold, delay, etc.
3. **Look-up Tables** contains a set of blocks that perform table lookups.
4. **User-Defined Functions** contains a set of MATLAB functions, S-functions, etc.
5. **Math operations** performs math functions such as scaling, dot product, product, trigonometric operations, and logical operations, etc.
6. **Discontinuities** contains a set of blocks for quantizer, saturation, relay, switch, nonlinear functions, etc.
7. **Signals Routing** manipulates the signal flow, storage and reading of data, etc.
8. **Signal Attributes** determines the data-type conversion line probing for data width and sample time, specify attribute of signal line, etc.

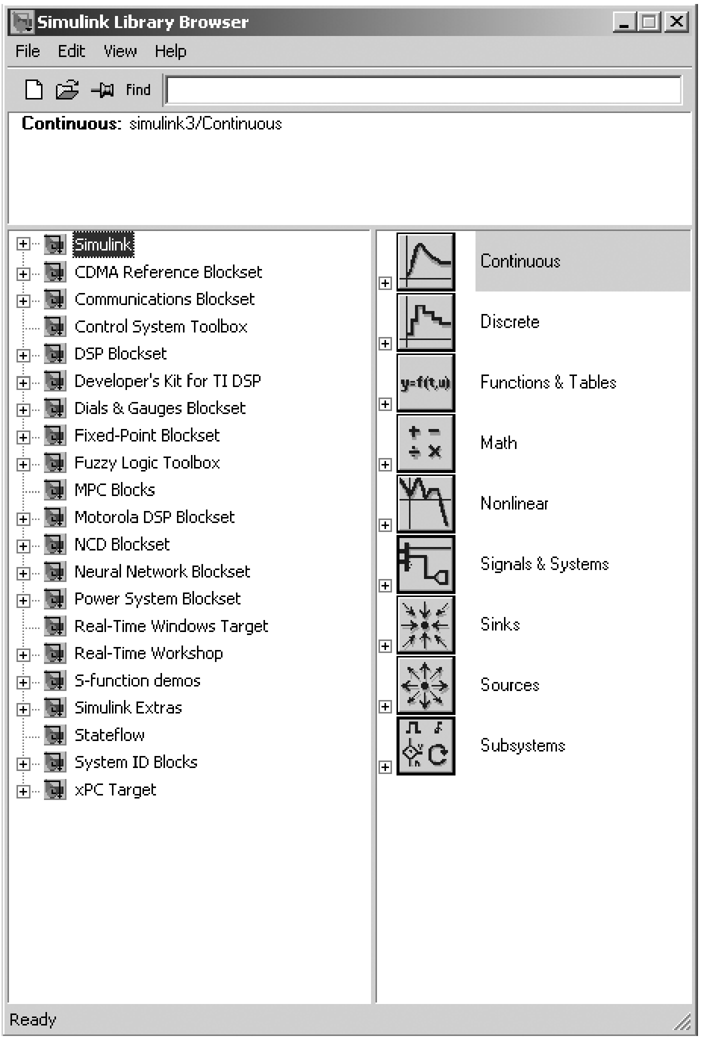


Figure A.12 **Simulink Library Browser** window

9. **Sinks** contains a set of destination blocks such as scope, files, workspace, graphs, etc.
10. **Sources** contains all source elements such as signal generator, random number, noise, clock, etc.
11. **Port & Subsystems** contains useful blocks that allow external trigger, conditional switching, iterative system, subsystem clock, I/O port, etc.
12. **Model Verification** checks the dynamic range of signals, input resolution, and static upper and lower bounds, etc.
13. **Model-Wide Utilities** provides documentation of model and linearization of running model.

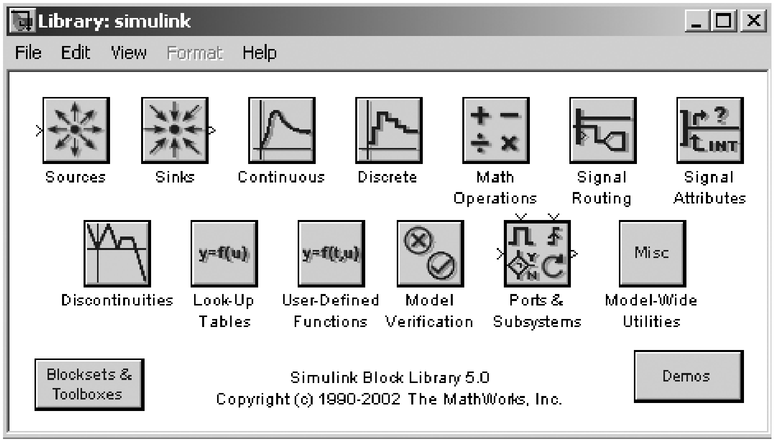


Figure A.13 Simulink **Library** window

To begin the block-building process, simply click on the **Create a new model** icon 🗋 to open a new worksheet. In order to use these functional blocks, click on the selected block icon and drag it into the opened worksheet. Most blocks (except the sink and source blocks) have at least one input and one output node. We can connect two blocks by clicking on the output node of the first block and dragging the mouse to the input node of the second block. A quicker way of connecting two blocks is to click on the first (source) block and hold down the Ctrl key while also clicking on the second (destination) block. These two blocks are connected automatically.

In this section, we use a simple example to show the processes of designing and building a simple dynamic digital system in Simulink from scratch. The complete system is shown in Fig. A.14. The step-by-step procedure of building this complete design involves the following steps:

Step 1: Click on **File** → **New** → **Model** in the **Simulink Library Browser**, or simply click on the **Create a new model** icon 🗋 to open a new worksheet.

Step 2: Select a sinusoidal source by double-clicking on **DSP Blockset**, and then click on **DSP Sources**. Select **Sine Wave** and drag it into the worksheet. Double-click on this block and set the following parameters: Amplitude = 1, Frequency = 1,000, and Sample time = 1/10,000.

Step 3: Select a white-noise source by clicking on **Random Source** (in **DSP Blockset** → **DSP Sources**) and drag it into the worksheet. Double-click on this block, and set the following parameters: Source Type = Gaussian, Mean = 0, Variance = 1, and Sample time = 1/10,000.

Step 4: Attach a gain block to the output of each source. The gain block is located in **Simulink** → **Math Operations** → **Gain**. Drag two gain blocks into the worksheet, set the gain for the sinusoidal source to 1, and set the gain for the noise source to 0.1. Link the output of the source to the input of the gain block by clicking on the output node of the source block. Then, complete the connection by dragging the mouse to the input node of the gain block, as shown in Fig. A.14.

Step 5: Combine the two signal sources using a summing block. Click on **Sum** (in **Simulink → Math Operations**) to select the summing block, and drag it into the worksheet. Connect it to the outputs of the Gain blocks using the method described in Step 4.

Step 6: Apply a digital bandpass filter to the output of the summing junction to enhance the sinewave and attenuate the noise. The digital filter is located at **DSP Blockset**. Click on **Filtering**, and double-click on **Filter Designs**. Drag **Digital Filter Design** (FDATool block) to the worksheet and double-click on the block to open a window similar to that shown in Fig. A.9. We can specify the following parameters for designing the bandpass filter: Filter Type = Bandpass, Design Method = FIR → Equiripple, Filter order = minimum order, Fs = 10,000, Fstop1 = 900, Fpass1 = 950, Fpass2 = 1,050, Fstop2 = 1,100, Astop1 = 40 dB, Apass = 1 dB, and Astop2 = 40 dB. After entering all of the parameters, click on the **Design Filter** button to start the filter design and implementation.

Step 7: Examine the signals before and after the bandpass filter using a **Spectrum Scope**. This scope can be found in **DSP Blockset → DSP Sinks**. Drag it to the worksheet. We can also view the output from the summing junction (noisy sinewave) by using another **Spectrum Scope**. Drag the second **Spectrum Scope** into the worksheet, click on the input node, and drag the mouse to the line between the summing junction and the bandpass filter. Enable the **buffer input** option of the **Spectrum Scope**, and use the default FFT length. This configuration allows the user to compare the difference between the original signal and the filtered signal.

Step 8: Set the Simulink parameters before running the simulation once the Simulink model has been completed, as shown in Fig. A.14. Click on **Simulation** and select **Simulation parameters** from the menu to open the window shown in Fig. A.15. Set the parameters as shown in Fig. A.15 for a 10-second simulation at a sampling frequency of 10,000 Hz. Close the window and start the simulation by clicking on the button ▶. Two spectrum plots are displayed that show that the processed signal has a 40 dB reduction in noise power. To stop the simulation, simply click on the button ■.

We can save the simulation model shown in Fig. A.14 to a file `bpf.mdl` and use the file later in the MATLAB environment by typing

```
bpf
```

In the second example, we open a dynamic system (from Simulink demo [8]) by typing

```
combfilter
```

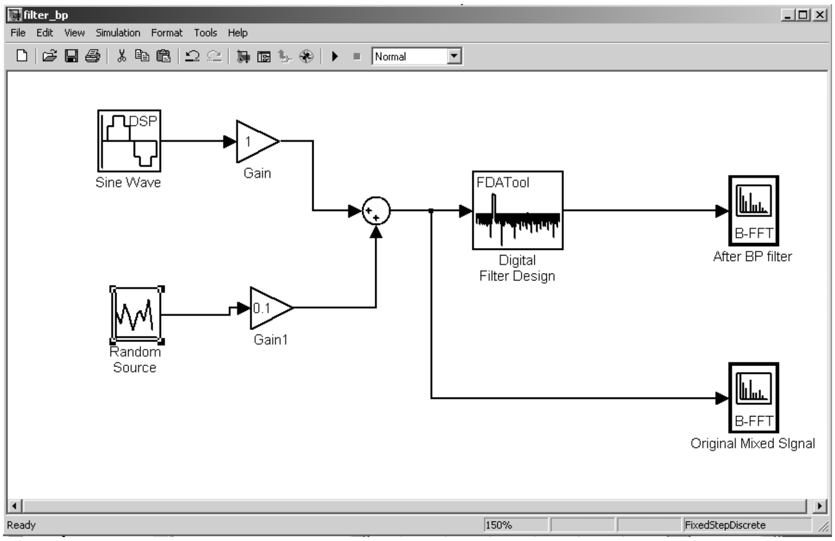
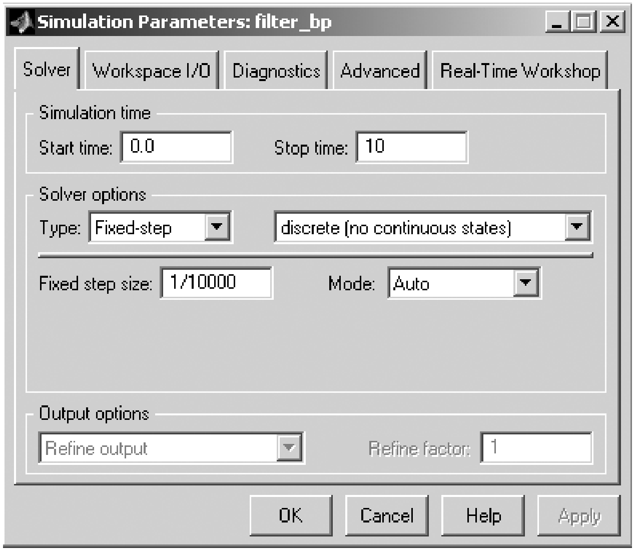


Figure A.14 A simple Simulink model for digital filtering



Simulation time
Start time: 0.0 Stop time: 10
Solver options
Type: Fixed-step discrete (no continuous states)
Fixed step size: 1/10000 Mode: Auto
Output options
Refine output Refine factor: 1

Figure A.15 Simulation parameters to run the filter simulation

This Simulink model is shown in Fig. A.16. This example illustrates the differences between digital filtering using single (upper path) and double (lower path) precisions. We can click on the block and drag it around. Double-click on any block to show either the internal block's detail or open up the block's parameters window. For example, double-click on the **Signal Waveform** block to open the **Block Parameters** menu, which allows the user to adjust the waveform type, amplitude, and frequency of the signal. The user can change the default 1 Hz sinewave into a 2 Hz squarewave with an amplitude of 50.

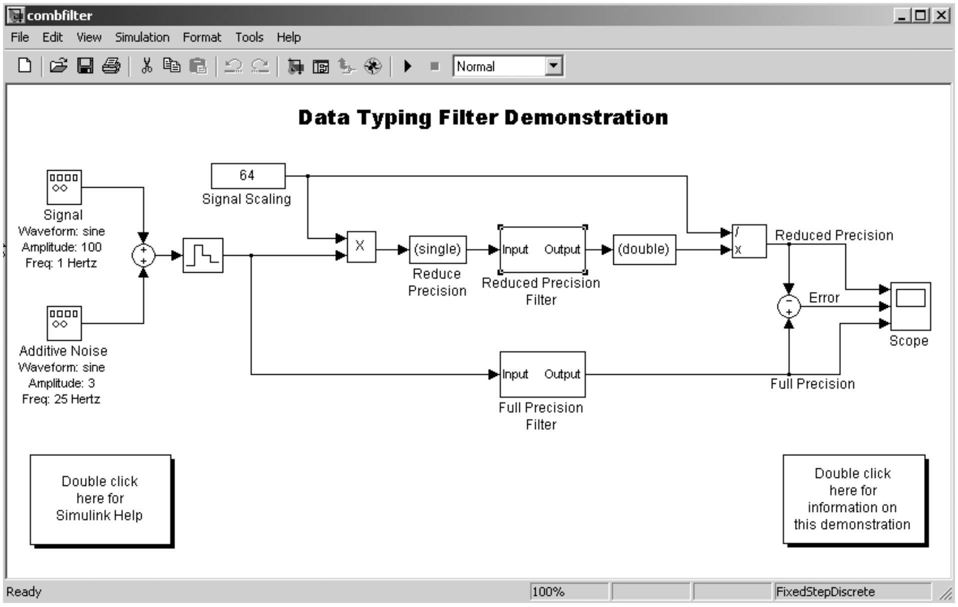
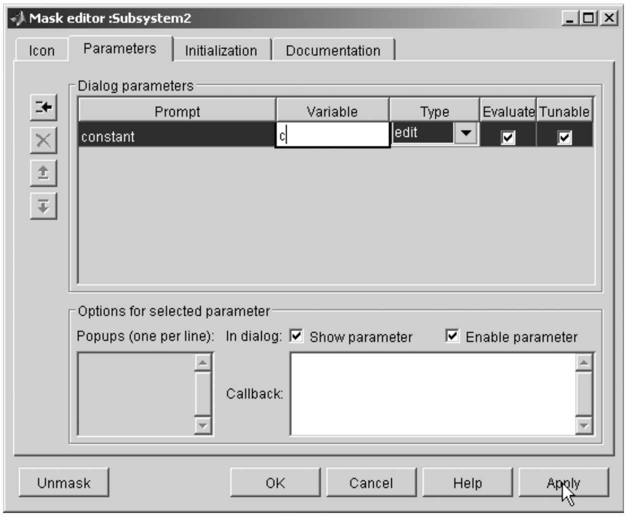


Figure A.16 A Simulink demo showing single- and double-precision filtering

Several blocks can be grouped together to form a subsystem. The subsystem can be created by selecting multiple blocks using Shift-click or by using the mouse cursor to form a bounding box that includes these blocks. We then select **Create subsystem** under the **Edit** menu to form a subsystem. The details of the subsystem can be viewed by double-clicking on it. For example, in the preceding Simulink demo, we can select the **Signal Scaling** block and the **Multiply** block to form a subsystem. Double-click on this new subsystem to view the internal details.

The subsystem can be masked to allow the user to customize the dialog box and icon for this subsystem. Click on the subsystem, and select **Mask Subsystem** under the **Edit** menu. A **Mask Editor** window is opened, as shown in Fig. A.17. Click on the **Parameters** pane, and then click on the **Add** icon to specify the attributes of mask parameters including prompt, variables, and control type. In this example, the parameters **constant** and **c** are specified in the **Prompt** and **Variable** fields, respectively. Make sure that the **Type** entry is selected as **edit** and that both **Evaluate** and **Tunable** are turned on. The block description is defined in the **Documentation** pane, as shown in Fig. A.18. The **Documentation** menu specifies the dialog box once the subsystem block is double-clicked.

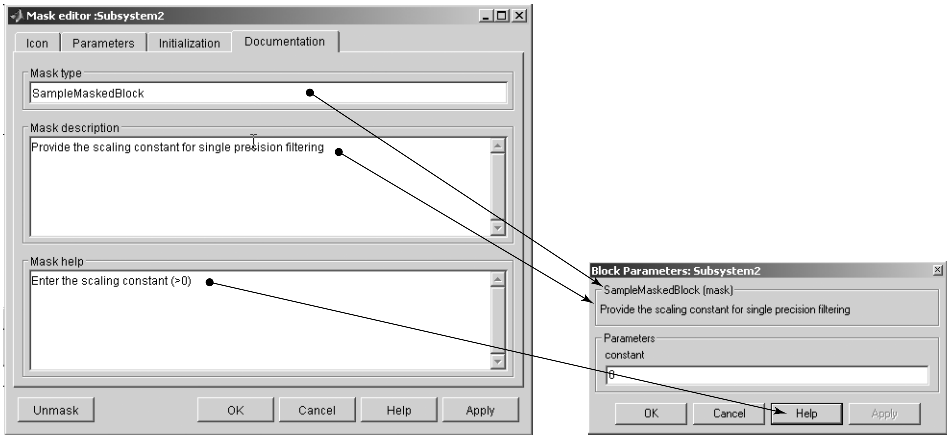
Before starting Simulink, the simulation parameters must be specified by selecting **Simulation Parameters** under the **Simulation** menu. The **Simulation Parameters** window appears, as shown in Fig. A.19. The start and stop time for the simulation can be specified in the **Start time** (0 sec) and **Stop time** (4 sec) fields. Simulink provides a number of solvers for the simulations. In DSP, the solver option is set to **Fixed-step** (take the same step size during simulation) and **discrete** (no continuous states). The **Fixed step size** field can be set to **auto** or to the sampling period of the



Prompt	Variable	Type	Evaluate	Tunable
constant	c	edit	✓	✓

In dialog: ✓ Show parameter ✓ Enable parameter

Figure A.17 **Mask editor** window



Mask type: SampleMaskedBlock

Mask description: Provide the scaling constant for single precision filtering

Mask help: Enter the scaling constant (>0)

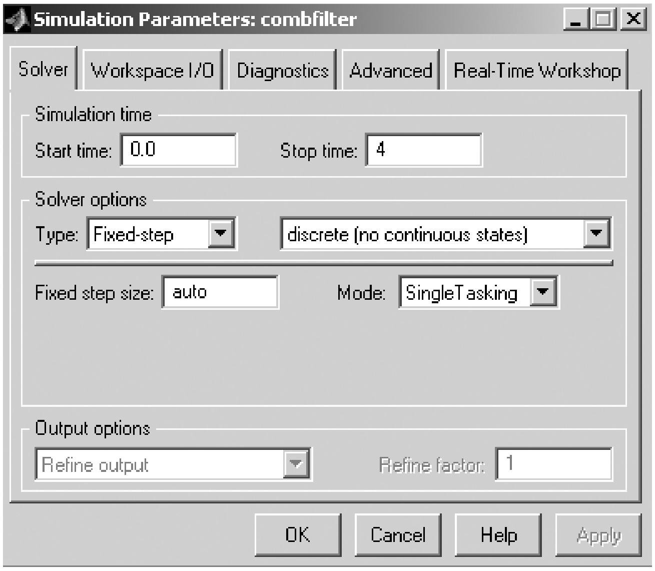
SampleMaskedBlock (mask)
Provide the scaling constant for single precision filtering

Parameters
constant: 8

Figure A.18 Block documentation

DSP system. Finally, the mode is set to **SingleTasking** (for a single-tasking system with a single-sampling rate). Other available modes are **MultiTasking** (a multisampling rate is used in the system) and **Auto** (automatically adjust between rates). Click on **OK** and start the simulation by clicking on the play icon ▶.

The **Scope** block displays the outputs from the reduced-precision and full-precision filters, and the error (middle plot) between these precisions is shown in Fig. A.20. The user can also double-click on the **Reduce Precision** block to change the **data type** to **int8**, **int16**, or **int32**, and observe the differences in comparison with the double-precision comb filter. More information on the usage of various blocks and settings can be found in the *Simulink: User's Guide* [8].


Simulation Parameters: combfilter

Solver | Workspace I/O | Diagnostics | Advanced | Real-Time Workshop

Simulation time

Start time: 0.0 Stop time: 4

Solver options

Type: Fixed-step discrete (no continuous states)

Fixed step size: auto Mode: SingleTasking

Output options

Refine output Refine factor: 1

OK Cancel Help Apply

Figure A.19 **Simulation Parameters** window

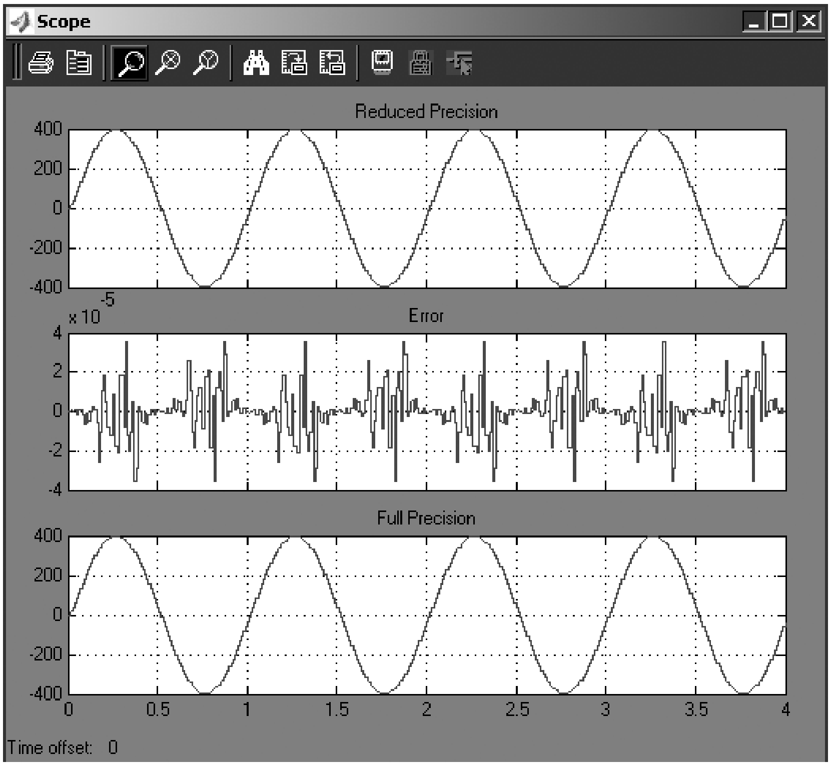


Figure A.20 **Scope** display

Additional Simulink tools can be used for simulating a dynamic system. These tools include:

1. Simulink Accelerator
2. Model Differencing tool
3. Profiler
4. Model Coverage tool

The **Simulink Accelerator** speeds up the execution of Simulink simulations. This tool uses part of the Real-Time Workshop to create and compile C code to replace the code that Simulink uses in normal mode. This mode can be selected by clicking on the **Simulink** menu and selecting **Accelerator**. Alternatively, the user can select **Accelerator** from the menu located in the middle of the toolbar .

The **Model Differencing** tool is available under the **Tools** menu of Simulink. It finds and displays the differences between two Simulink models, and the differences are reported in the Model Differences Report.

To activate the profiler, simply click on **Profiler** under Simulink's **Tools** menu. The Simulink Profiler collects and profiles performance data while the model is being simulated. When the simulation finishes, Simulink generates a report that describes the time taken to execute each functional block. This feature allows the user to identify the blocks or subsystems that require more time for execution and thus are targets for optimization.

The **Model Coverage** tool validates the user's models. It analyzes the execution of blocks that serve as decision points in the model. These blocks include switch, triggered subsystem, enabled subsystem, absolute value, saturation, stateflow charts, etc. The user can run the coverage tool by selecting **Coverage Settings** from Simulink's **Tools** menu. When the **Coverage Settings** dialog box is displayed, simply check **Enable Coverage Reporting**. Note that both model-coverage reporting and acceleration mode cannot be enabled at the same time because Simulink disables coverage reporting if the accelerator option is enabled.

Several Simulink blocksets, such as DSP Blockset, Fixed-Point Blockset, Communication Blockset, Code Division Multiple Access (CDMA) Reference Blockset, and many others, are available to run in the Simulink environment. The following section describes two important blocksets that are related to DSP applications.

A.4 USING BLOCKSETS

A blockset is a set of special blocks that are included in an application library for handling the application tasks in Simulink. The most commonly used blocksets for DSP applications are DSP and Fixed-Point blocksets [9, 10].

A.4.1 DSP Blockset

The DSP Blockset is a collection of signal-processing blocks for use with Simulink. To access the DSP Blockset, type the following in the MATLAB command window:

```
dsplib
```

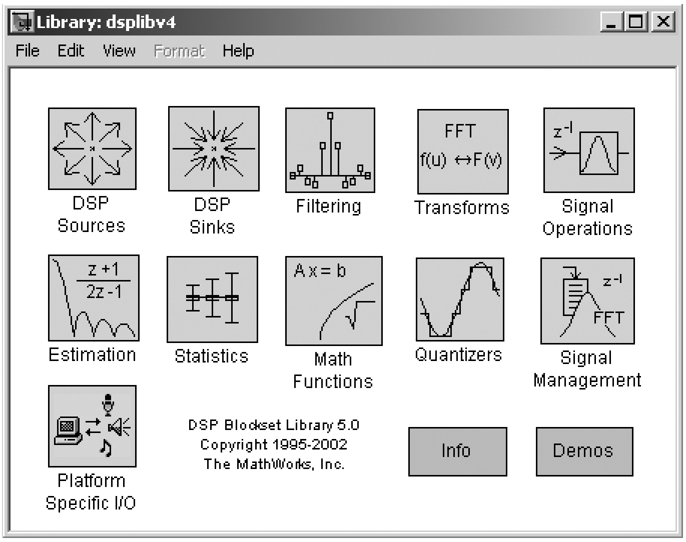


Figure A.21 Functions available in the DSP Blockset

Many important DSP algorithm blocks are included in the DSP Blockset Library. As shown in Fig. A.21, this library includes the following blocks:

1. **DSP Sources** contains special sources for generating signals such as chirp, discrete impulse, sinewave, clock, etc. It also contains blocks that capture signals from workspaces, wave devices, and wave files.
2. **DSP Sinks** contains functional blocks for counter, matrix, time, vector, and spectrum scope. It also contains a block for writing signals into workspaces, wave devices, and wave files.
3. **Filtering** contains three main sections: (1) filter design (contains many FIR- and IIR-filter designs and structures), (2) multirate filters (contains decimation, interpolation, analysis and synthesis, and wavelet analysis and synthesis filters), and (3) adaptive filters (contains LMS, Kalman, and RLS adaptive filters).
4. **Transforms** contains a set of transforms that include FFT, IFFT, discrete cosine transform (DCT), inverse discrete cosine transform (IDCT), and real and complex cepstrum.
5. **Signal Operations** provides a set of commonly used DSP operations such as convolution, upsample, downsample, integer and fractional delay, zero padding, sample-and-hold, etc.
6. **Estimation** contains three sections: (1) linear prediction (contains autocorrelation), (2) power-spectrum estimation (contains magnitude-square FFT, Burg, covariance, Yule-Walker, and STFT), and (3) parametric estimation (Burg, covariance, and Yule-Walker AR estimators).
7. **Statistics** contains a set of blocks for computing autocorrelation, correlation, histogram, variance, median, mean, etc.

8. **Math Functions** contains three functional groups: (1) math operations (dB gain, conversion, normalization, complex exponential, etc.), (2) matrix and linear algebra (matrix factorization, matrix inversion, linear-system solvers, pseudo inversion, matrix multiplication, matrix square, Toeplitz, etc.), and (3) polynomial-function evaluation.
9. **Quantizers** contains several blocks for quantizer and uniform encoder and decoder.
10. **Signal Management** contains four groups: (1) switches and counters, (2) buffer for buffering and unbuffering signals, (3) matrix and signal indexing, and (4) signal attributes that convert the dimension of the signal, check signal and frame status, etc.
11. **Platform Specific I/O** contains a set of blocks that take in sound signals from a standard audio device and output to a standard audio device in real time. It also contains a set of blocks that read from and write to a file with a standard `.wav` format.

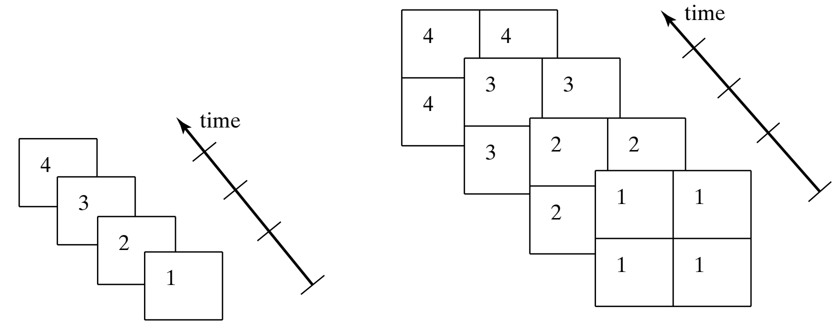
The DSP Blockset has the flexibility to handle single and multichannel signals, as will as sample and frame-based signals. Therefore, the following four possible types of input signal can be implemented in Simulink, as shown in Fig. A.22: (1) sample-based single-channel signals, (2) sample-based multichannel signals, (3) frame-based single channel signals and, (4) frame-based multichannel signals.

Frame-based (block) processing can accelerate real-time systems, as explained in Chapter 3. In real-time DSP systems, data acquisition can be carried out at a higher rate when a block of samples is transferred to the processor and when these block samples are processed at once. In contrast, sample processing interrupts the processor at every sample. Therefore, frame-based processing minimizes the overhead incurred in interrupting the processor. However, we have to consider the latency introduced by frame-based processing in some applications, as explained in Chapter 3.

In Simulink models, frame-based and sample-based signals are indicated by a double line ($\Rightarrow$)and single line ($\rightarrow$), respectively. A sample-based signal can be converted to a frame-based signal by using the **Buffer** block, and an **Unbuffer** block converts a frame-based signal back to a sample-based signal. The DSP Sources Library provides a set of blocks for creating sample-based and frame-based signals.

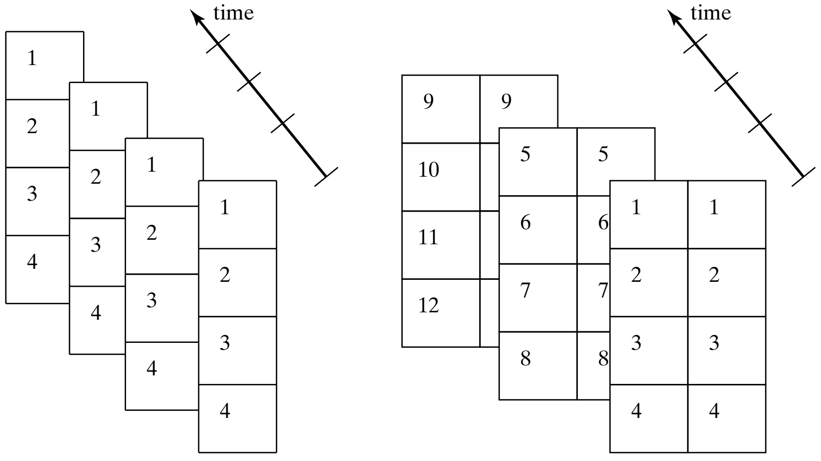
Two different delays affect Simulink models: (1) computational delay and (2) algorithmic delay. Computational delay depends on how fast the computer hardware and software can execute a block of data. There are several methods to reduce computational delay, such as using frame-based processing or using the Real-Time Workshop to generate generic real-time code for specific hardware. Algorithmic delay is intrinsic to the algorithm and is independent of the processor. Algorithmic delay is commonly implemented in Simulink using an integer-delay block. It can also occur in certain conditions known as tasking latency, which arises from the synchronization requirements of Simulink's tasking mode. For example, the multirate blocks operating in multitasking mode are subject to tasking latency.

More information on and examples of using the DSP Blockset can be found in the *DSP Blockset User's Guide* [9].



(a) Sample-based, single-channel

(b) Sample-based, multichannel (4 channels)



(c) Frame-based single-channel (4 samples per frame)

(d) Frame-based multichannel (2 channels, 1 frame per channel, 4 samples per frame)

Figure A.22 Different types of signal processing available in Simulink

A.4.2 Fixed-Point Blockset

The Fixed-Point Blockset [10] simulates effects commonly encountered in fixed-point systems such as digital filtering. The Fixed-Point Blockset extends the functional blocksets of the standard Simulink. This blockset allows the user to develop, simulate, analyze, and implement DSP systems using fixed-point arithmetic. A fixed-point simulation is vital for resolving finite-wordlength problems associated with different algorithms before entering the coding and implementation stages of the development cycle. Such a simulation greatly enhances the accuracy of the code and speeds up system-development time. Some important features of this blockset include the following:

1. Integer, fractional, and generalized fixed-point data types with a wordlength from 1 to 128 bits

2. IEEE-754 format with single and double precisions, as well as non-IEEE standards
3. Methods to control the overflow, scaling, and rounding of fixed-point data
4. An interface tool to profile the statistics of the simulation
5. Generation of C code (integer type only) for execution on a given embedded processor

Figure A.23 shows the window that contains the blocks in the Fixed-Point Blockset Library. This window can be opened by typing

```
fixpt
```

in the MATLAB command window. The window shows that the library contains the following different blocks: (1) **Math**, (2) **Data Type Conversion & Propagation**, (3) **Look-Up tables**, (4) **Logic & Comparison**, (5) **Filters** (for performing filtering), (6) **Delays & Holds**, (7) **Select** (for selecting one from many inputs), (8) **Nonlinear** (for performing different nonlinear threshold functions), (9) **Calculus**, (10) **Bits**, (11) **Sources**, (12) **FixPt GUI**, (13) **Demos**, and (14) **Edge Detect**.

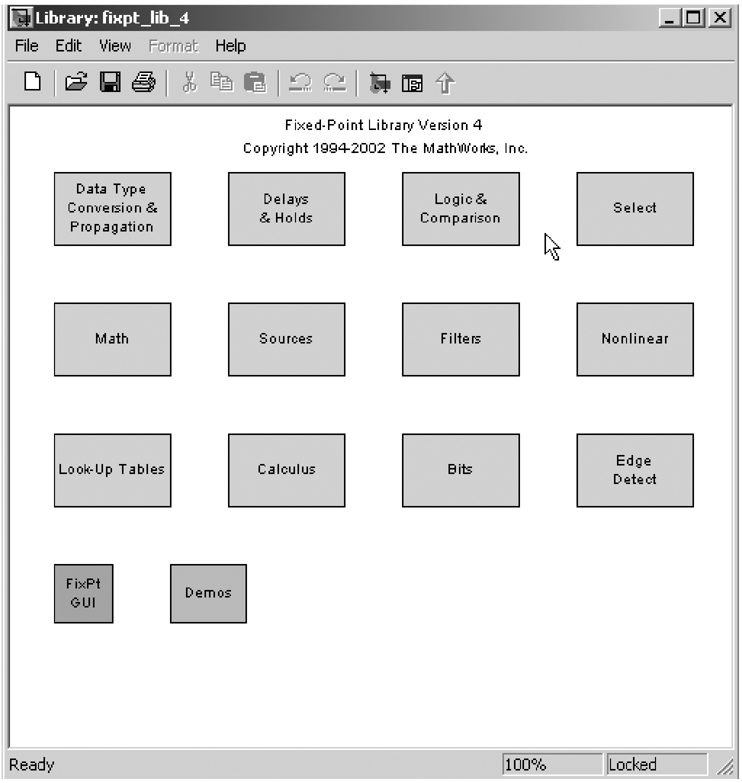


Figure A.23 Main window for the Fixed-Point Blockset Library

Configure Fixed-Point Blocks

The Fixed-Point Blockset provides an option for specifying the **output data type** via the block dialog box. Possible options are listed in Table A.10. In the signed and unsigned integer representation (`sint`, `uint`), the default binary point is to the right of the LSB. In the unsigned fractional-data type (`ufrac`), the default binary point is to the left of the MSB. In the signed fractional-data type (`sfrac`), the binary point is just to the right of the MSB (sign bit). In order to provide flexibility in defining different Q formats (or generalized fixed-point numbers) with a user-specified binary point, the Fixed-Point Blockset provides generalized signed and unsigned fixed-point numbers (`sfix`, `ufix`), and the binary point is determined from the value in the output scaling. For example, if the scaling factor is 2^E, the binary point is located E bits left of the LSB. The final data type is the floating-point data format, which can be represented by IEEE-754 single precision `float('single')`, IEEE-754 double precision `float('double')`, and a non-IEEE format expressed as `float(w,e)` for a total of `w` bits with `e` exponent bits, (`w-e-1`) fractional bits, and 1 sign bit.

The option of selecting output scaling is applicable to fixed-point data types. There are two general scaling modes: binary point-only scaling and slope/bias scaling. In binary point-only scaling mode, power-of-two scaling (e.g., 2^{-E}, where E is unrestricted) is used. This scaling has the effect of moving the binary point to the left by E bits. In slope/bias scaling mode, a slope of $S = F2^E$ is realized, where a bias of B bits is used to scale the integer number Q as $SQ + B$. Note that 2^E specifies the binary point, and F is the fractional slope, which is normalized to $1 \leq F < 2$. Slope/bias scaling is a superset and contains power-of-two scaling, which can be considered under slope/bias scaling with $F = 1$ and $B = 0$. In general, slope/bias scaling mode allows more flexible scaling and maximizes usage of the finite number of bits.

Fixed-point numbers can be rounded using the following rounding modes:

1. Zero, which rounds toward zero and is similar to the `fix` function in MATLAB.
2. Nearest, which rounds toward the nearest representable number and is similar to the `round` function.

TABLE A.10 Definition of Output Data Types

Option	Description
`uint(n)`	`n`-bit unsigned integer
`sint(n)`	`n`-bit signed integer
`ufrac(n,g)`	`n`-bit unsigned fractional number
`sfrac(n,g)`	`n`-bit signed fractional number
	Note: the number of guard bits, `g` (optional), lies at the left of the default binary point
`ufix(n)`	`n`-bit unsigned fixed-point number
`sfix(n)`	`n`-bit signed fixed-point number
`float('single')`	IEEE-754 single-precision, floating-point number
`float('double')`	IEEE-754 double-precision, floating-point number
`float(w,e)`	Total bits `w` and exponent bits `e`

3. **Ceiling**, which rounds toward positive infinity and is similar to the `ceil` function.
4. **Floor**, which rounds toward negative infinity and is similar to the `floor` function.

Overflow handling in the Fixed-Point Blockset can be set by checking the **Saturate to max or min when overflow occurs** check box to specify the use of saturated mode. If unchecked, overflow wraps to the other end of the number range.

An example of an analog-to-digital model (shown in Fig. A.24) is used to illustrate the effects of using different wordlengths in representing digitized samples with the Fixed-Point Blockset. To start the simulation, type

```
fxpadc
```

in the MATLAB command line. The user can double-click on any block and adjust the default parameters. The **Signal Generator** block is configured to generate a sinewave with double-precision amplitude on the interval [−2, +2]. The **Zero-Order Hold** block simulates the sampling of the continuous sinewave, and the **Dbl To FixPt1** block converts the double-precision, floating-point number to a fixed-point representation. Once the desired parameters have been specified, the user can click on the play button ▶ to start the simulation.

In the example, different fixed-point implementations using (1) Q.15 `sfrac(16)`, (2) Q1.14 `sfix(16)` with scaling of 2^{-14}, and (3) 8-bit wordlength `sfix(8)` with scaling of 2^{-4} can be implemented individually inside the **Dbl to FixPt1** block. The result for a Q.15 representation is shown in Fig. A.25(a), which is unable to cover the entire range of the input signal due to the fact that the dynamic range of a Q.15 number is [−1, +0.99]. By introducing an additional integer bit as in the Q1.14 format, the dynamic range increases to [−2, +1.99], but with reduced precision. The result is illustrated in Fig. A.25(b), which shows very little difference between the double-precision and Q1.14 representation. A coarser quantization

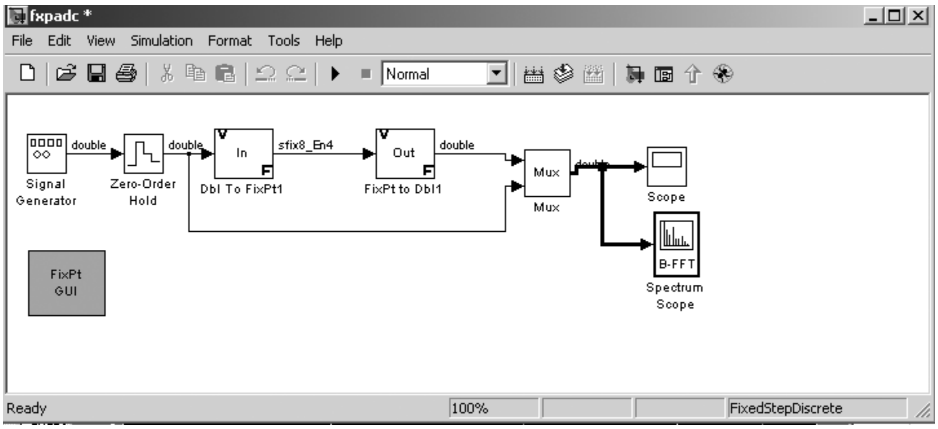
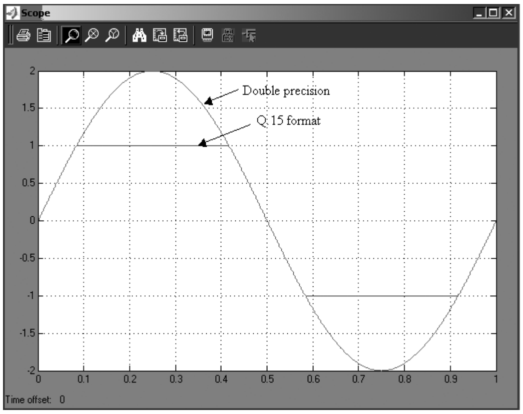
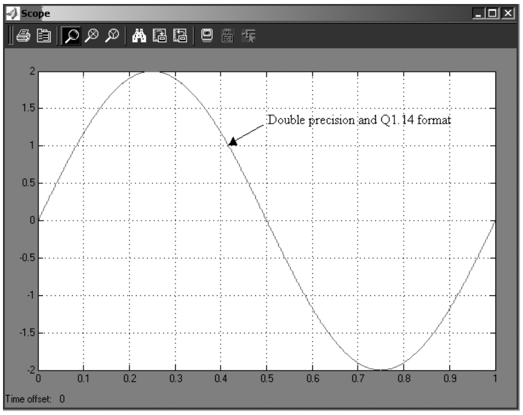


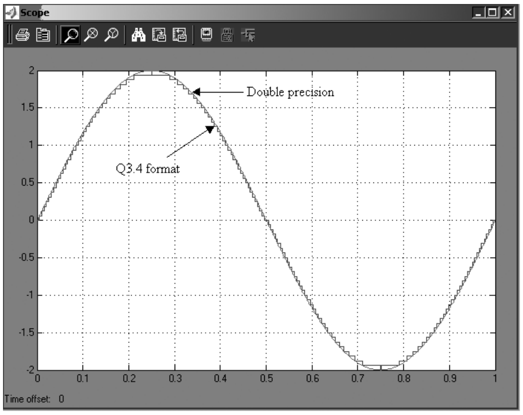
Figure A.24 Simulink block diagram for a fixed-point implementation of ADC



(a)



(b)



(c)

Figure A.25 Different data representations: (a) Q.15 and double precision, (b) Q1.14 and double precision, (c) Q3.4 and double precision

step is observed in Fig. A.25(c) for the 8-bit wordlength, where the binary point is the fourth bit to the left of the rightmost bit.

C-Code Generation in Fixed-Point Blockset

After the system has been designed and analyzed using the Fixed-Point Blockset, C code can be generated directly using the Real-Time Workshop [11]. The C code generated from the fixed-point block uses only integer types and can be used directly on embedded fixed-point processors, which can be either a fixed-point or floating-point architecture. The code can also be generated for testing on a rapid prototyping system such as xPC [12] and a real-time window target [13]. It can also generate code for non-real-time testing on a computer running on any supported operating system. Please refer to reference [10] for detailed information on the step-by-step generation of fixed-point C code.

A.5 MATLAB LINK FOR CODE COMPOSER STUDIO

MATLAB has introduced a new toolbox called MATLAB Link for CCS, which establishes bidirectional links between MATLAB, CCS, and TMS320 processors [14]. As shown in Fig. A.26, the MATLAB Link for CCS connects MATLAB to Texas Instruments software and hardware. It allows the MATLAB user to create an object that links to CCS and the real-time data exchange (RTDX) so that the user can transfer data to and from the processor without halting it. In other words, the user can open the CCS environment, download the program, communicate with the DSP processor, access the processor's registers and memories, and perform data logging while the DSP processor is running. This capability allows the user to change a parameter or variable in MATLAB and transfer the value into the running DSP processor in order to tune and alter the algorithm in real time.

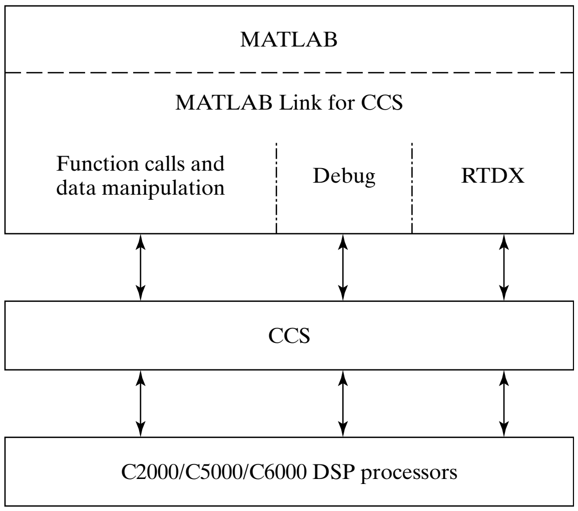

MATLAB
MATLAB Link for CCS
Function calls and data manipulation
Debug
RTDX
CCS
C2000/C5000/C6000 DSP processors

Figure A.26 Block diagram of MATLAB Link for CCS

The current version of MATLAB Link for CCS can link the TMS320C6701 Evaluation Module (EVM) TMS320C6711 DSK, C6000 and C5000 simulators, and other DSP boards that are supported in the setup utility. In this section, we give a brief overview of some important features of MATLAB Link for CCS and show how to link it to the C5000 simulator.

First, we can check whether the link for CCS is installed in MATLAB by typing

```
help ccslink
```

in the MATLAB command line. If the software is installed, it returns a list of commands for analysis and debugging with CCS. Next, we can check the boards and processors that are installed on the computer by typing

```
ccsboardinfo
```

which returns a list of boards and processors that are installed and recognized by CCS.

In order to select the C5000 simulator, type

```
[boardnum,procnum]=boardprocsel
```

and select the C5400 (or C5500) simulator. Note that if only a single board (or simulator) is installed in the system, this step can be skipped. After successful selection of the board or simulator, we can create the link between MATLAB and CCS by typing the following MATLAB command:

```
cc = ccsdsp('boardnum',boardnum,'procnum',procnum);
```

We notice that CCS is placed in the background, and we can view the status by typing

```
disp(cc)
```

In addition, we can set the visibility for the CCS window by typing

```
visible(cc,1)
```

The next step is to load a project file (e.g., `ccstut_54xx.pjt`) into CCS using the following MATLAB commands:

```
projfile = fullfile(matlabroot,'toolbox','ccslink','ccsdemos',
'ccstutorial', 'ccstut_54xx.pjt');
projpath = fileparts(projfile)
open(cc,projfile)      % load a file into CCS
cd(cc,projpath)        % change the working directory that CCS uses
```

However, we cannot build the CCS project in the MATLAB window. The user must build the project in CCS by clicking on the **Build-All** icon in the CCS window. After the project is built, we can load the `ccstut_54xx.out` file using the command

```
load (cc,'ccstut_54xx.out')    % transfer program file to target
                               % processor
```

After loading, we can set the breakpoint, reset the program counter, and run the program using the following commands:

```
halt(cc)                    % terminate the execution running on
                            % the target
restart(cc)                 % reset the PC to start of program
run(cc,'runtohalt',30)      % run to breakpoint with timeout=30 sec
```

Data located in memory on the target processor can be transferred to the MATLAB environment. For example, to read the data `ddtav` and `idtav` from CCS to the MATLAB workspace, type the following commands:

```
ddtav = read(cc,address(cc,'ddat'),'single',4)
idtav = read(cc,address(cc,'idat'),'int16',4)
```

MATLAB supports several data types. In the preceding example, the data type `single` is used instead of `double`, which is not supported by the C5000 simulator.

Besides reading from the target processor, the user can also write data to memory on the target processor. For example, we can modify `ddtav` and `idtav` as follows:

```
write(cc,address(cc,'ddat'),single([3.14 -10.3 exp(-2)
     sin(pi/2)]));
write(cc,address(cc,'idat'),int16([1:4]));
```

In addition, DSP registers can be viewed and modified. A simple example that changes the value of ACC A is shown as follows:

```
regal=cc.regread('AL','binary')
dec2hex(regal)
cc.regwrite('AL',hex2dec('683'),'binary')
dec2hex(cc.regread('AL','binary'))
```

More detailed information on MATLAB Link for CCS and how to use links for RTDX, which requires actual hardware (EVM or DSK) to be hooked up to the host computer, are documented in [14].

SUGGESTED READINGS

1 The MathWorks. *Using MATLAB*. Version 6, 2002.
2 The MathWorks. *Using MATLAB Graphics*. Version 6, 2002.
3 The MathWorks. *Signal Processing Toolbox User's Guide*. Version 6, 2002.

4 The MathWorks. *Filter Design Toolbox User's Guide*. Version 2, 2002.
5 The MathWorks. *Communication Toolbox User's Guide*. Version 2, 2002.
6 The MathWorks. *Image Processing Toolbox User's Guide*. Version 3, 2002.
7 The MathWorks. *Wavelet Toolbox User's Guide*. Version 2, 2002.
8 The MathWorks. *Simulink: User's Guide*. Version 5, 2002.
9 The MathWorks. *DSP Blockset User's Guide: For Use with Simulink*. Version 5, 2002.
10 The MathWorks. *Fixed-Point Blockset User's Guide: For Use with Simulink*. Version 4, 2002.
11 The MathWorks. *Real-Time Workshop User's Guide: For Use with Simulink*. Version 5, 2002.
12 The MathWorks. *xPC Target User's Guide: For Use with Simulink*. Version 2, 2002.
13 The MathWorks. *Real-Time Window Target User's Guide: For Use with Simulink*. Version 2.1, 2002.
14 The MathWorks. *MATLAB Link for Code Composer Studio*. Version 1, 2002