UNIT V: TOOL BOXES 10 hrs

Simulink – Simulink model for a dead zone system, nonlinear system – Applications in DSP – Computation of DFT & FFT – Filter structure – IIR & FIR filter design – Applications in Communication PCM, DPCM, DM, DTMF- Interfacing of Matlab with event driven simulators.

Simulink is built on top of MATLAB, so you must have MATLAB to use Simulink. It is included in the Student Edition of MATLAB and is also available separately from The MathWorks, Inc. Simulink is widely used in industry to model complex systems and processes that are dif cult to model with a simple set of differential equations. Simulink provides a graphical user interface that uses various types of elements called *blocks* to create a simulation of a dynamic system, that is, a system that can be modeled with differential or difference equations whose independent variable is time. For example, one block type is a multiplier, another performs a sum, and still another is an integrator. The Simulink graphical interface enables you to position the blocks, resize them, label them, specify block parameters, and interconnect the blocks to describe complicated systems for simulation.

Type simulink in the MATLAB Command window to start Simulink. The Simulink Library Browser window opens. The Simulink blocks are located in libraries. Depending on what other MathWorks products are installed, you might see additional items in this window, such as the Control System Toolbox and State flow. These provide additional Simulink blocks, which can be displayed by clicking on the plus sign to the left of the item. As Simulink evolves through new versions, some libraries are renamed and some blocks are moved to different libraries, so the library we specify here might change in later releases. The best way to locate a block, given its name, is to type its name in the search pane at the top of the Simulink Library Browser. When you press Enter, Simulink will take you to the block location. To create a new model, click on the icon that resembles a clean sheet of paper, or select **New** from the **File** menu in the browser. Anew **Untitled** window opens for you to create the model. To select a block from the Library Browser, double-click on the appropriate library, and a list of blocks.

10.5 Transfer-Function Models

The equation of motion of a mass-spring-damper system is

$$m\ddot{y} + c\dot{y} + ky = f(t) \tag{10.5–1}$$

As with the Control System toolbox, Simulink can accept a system description in transfer-function form and in state-variable form. (See Section 9.5 for a discussion of these forms.) If the mass-spring system is subjected to a sinusoidal forcing function $f(t)$, it is easy to use the MATLAB commands presented thus far to solve and plot the response $y(t)$. However, suppose that the force $f(t)$ is created by applying a sinusoidal input voltage to a hydraulic piston that has a *dead-zone* nonlinearity. This means that the piston does not generate a force until the input voltage exceeds a certain magnitude, and thus the system model is piecewise linear.

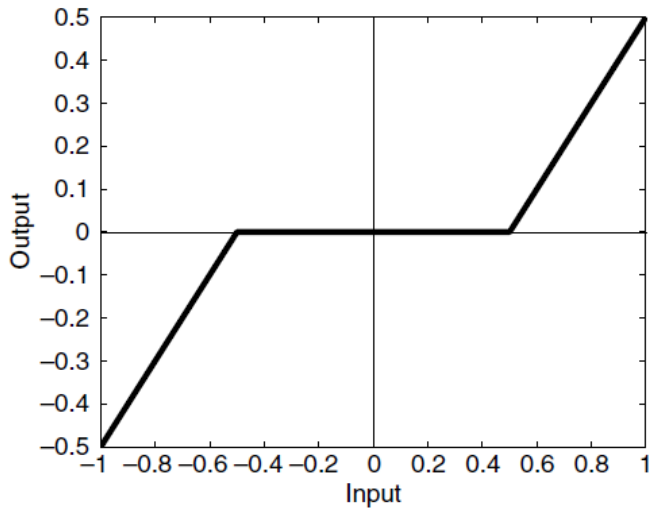


Figure 10.5–1 A dead-zone nonlinearity

A graph of a particular dead-zone nonlinearity is shown in Figure 10.5–1. When the input (the independent variable on the graph) is between −0.5 and 0.5, the output is zero. When the input is greater than or equal to the upper limit of 0.5, the output is the input minus the upper limit. When the input is less than or equal to the lower limit of −0.5, the output is the input minus the lower limit. In this example, the dead zone is symmetric about 0, but it need not be in general.

Simulations with dead-zone nonlinearities are somewhat tedious to program in MATLAB, but are easily done in Simulink. The following example illustrates how it is done.

Create and run a Simulink simulation of a mass-spring-damper model (Equation 10.5–1) using the parameter values $m = 1$, $c = 2$, and $k = 4$. The forcing function is the function $f(t) = \sin 1.4t$. The system has the dead-zone nonlinearity shown in Figure 10.5–1.

■ Solution

To construct the simulation, do the following steps.

1. Start Simulink and open a new Model window as described previously.
2. Select and place in the new window the Sine Wave block from the Sources library. Double-click on it, and set the Amplitude to 1, the Frequency to 1.4, the Phase to 0, and the Sample time to 0. Click **OK.**
3. Select and place the Dead Zone block from the Discontinuities library, double-click on it, and set the Start of dead zone to −0.5 and the End of dead zone to 0.5. Click **OK.**
4. Select and place the Transfer Fcn block from the Continuous library, double-click on it, and set the Numerator to `[1]` and the Denominator to `[1, 2, 4]`. Click **OK.**
5. Select and place the Scope block from the Sinks library.

6. Once the blocks have been placed, connect the input port on each block to the outport port on the preceding block. Your model should now look like Figure 10.5–2.
7. Set the Stop time to 10.
8. Run the simulation. You should see an oscillating curve in the Scope display.

It is informative to plot both the input and the output of the Transfer Fcn block versus time on the same graph. To do this:

1. Delete the arrow connecting the Scope block to the Transfer Fcn block. Do this by clicking on the arrow line and then pressing the **Delete** key.
2. Select and place the Mux block from the Signal Routing library, double-click on it, and set the Number of inputs to 2. Click **OK.**
3. Connect the top input port of the Mux block to the output port of the Transfer Fcn block. Then use the same technique to connect the bottom input port of the Mux block to the arrow from the outport port of the Dead Zone block. Just remember to start with the input port. Simulink will sense the arrow automatically and make the connection. Your model should now look like Figure 10.5–3.
4. Set the Stop time to 10, run the simulation as before, and bring up the Scope display. You should see what is shown in Figure 10.5–4. This plot shows the effect of the dead zone on the sine wave.

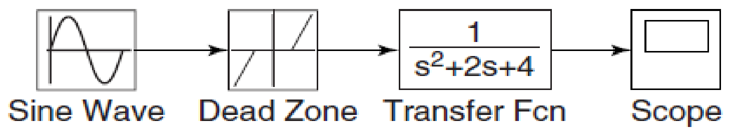


Figure 10.5–2 The Simulink model of dead-zone response.

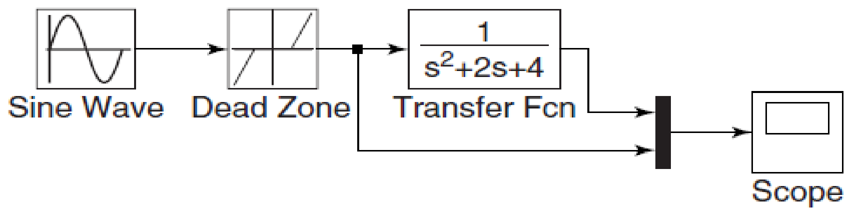


Figure 10.5–3 Modi cation of the dead-zone model to include a Mux block.

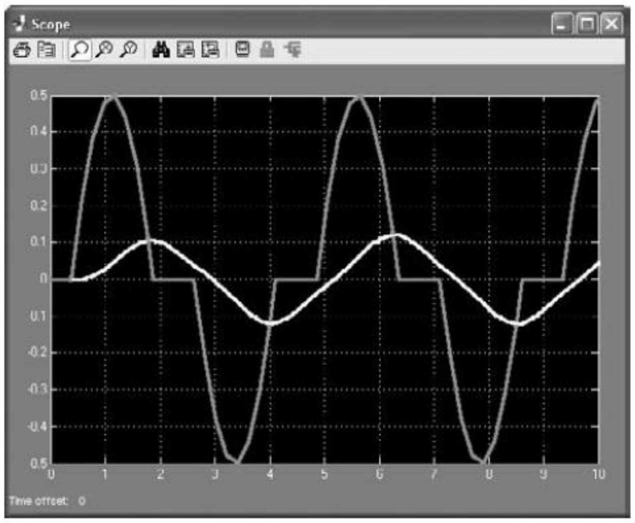


Figure 10.5–4 The response of the dead-zone model.

10.6 Nonlinear State-Variable Models

Nonlinear models cannot be put into transfer-function form or the state-variable form $\dot{\mathbf{x}} = \mathbf{Ax} + \mathbf{Bu}$. However, they can be simulated in Simulink. The following example shows how this can be done.

The pendulum shown in Figure 10.6–1 has the following nonlinear equation of motion, if there is viscous friction in the pivot and if there is an applied moment $M(t)$ about the pivot

$$I\ddot{\theta} + c\dot{\theta} + mgL \sin \theta = M(t)$$

where I is the mass moment of inertia about the pivot. Create a Simulink model for this system for the case where $I = 4$, $mgL = 10$, $c = 0.8$, and $M(t)$ is a square wave with an amplitude of 3 and a frequency of 0.5 Hz. Assume that the initial conditions are $\theta(0) = \pi/4$ rad and $\dot{\theta}(0) = 0$.

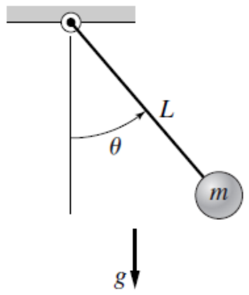


Figure 10.6–1
A pendulum.

■ Solution

To simulate this model in Simulink, de ne a set of variables that lets you rewrite the equation as two rst-order equations. Thus let $\omega = \dot{\theta}$. Then the model can be written as

$$\dot{\theta} = \omega$$

$$\dot{\omega} = \frac{1}{I}[-c\omega - mgL \sin \theta + M(t)] = 0.25[-0.8\omega - 10 \sin \theta + M(t)]$$

Integrate both sides of each equation over time to obtain

$$\theta = \int \omega \, dt$$

$$\omega = 0.25 \int [-0.8\omega - 10 \sin \theta + M(t)] \, dt$$

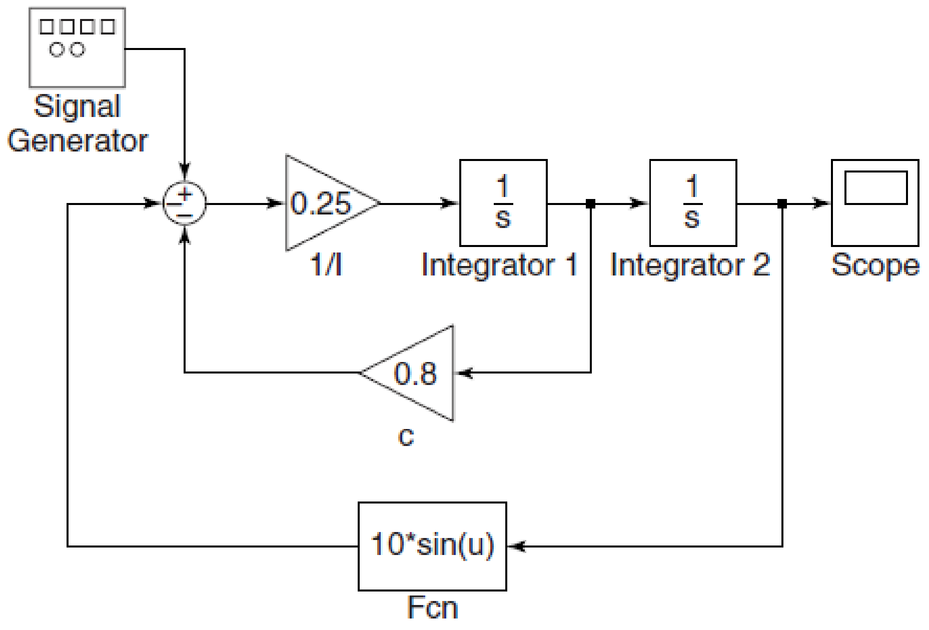


Figure 10.6–2 Simulink model of nonlinear pendulum dynamics.

We will introduce four new blocks to create this simulation. Obtain a new Model window and do the following.

1. Select and place in the new window the Integrator block from the Continuous library, and change its label to Integrator 1 as shown in Figure 10.6–2. You can edit text associated with a block by clicking on the text and making the changes. Double-click on the block to obtain the Block Parameters window, and set the Initial condition to 0 [this is the initial condition $\dot{\theta}(0) = 0$]. Click **OK.**

2. Copy the Integrator block to the location shown and change its label to Integrator 2. Set its initial condition to $\pi/4$ by typing `pi/4` in the Block Parameters window. This is the initial condition $\theta(0) = \pi/4$.
3. Select and place a Gain block from the Math Operations library, double-click on it, and set the Gain value to 0.25. Click **OK.** Change its label to `1/I`. Then click on the block, and drag one of the corners to expand the box so that all the text is visible.
4. Copy the Gain box, change its label to c, and place it as shown in Figure 10.6–2. Double-click on it, and set the Gain value to 0.8. Click **OK.** To ip the box left to right, right-click on it, select **Format,** and select **Flip.**
5. Select and place the Scope block from the Sinks library.
6. For the term 10 sin θ, we cannot use the Trig function block in the Math library because we need to multiply the sin θ by 10. So we use the Fcn block under the User-De ned Functions library (Fcn stands for function). Select and place this block as shown. Double-click on it, and type `10*sin(u)` in the expression window. This block uses the variable u to represent the input to the block. Click **OK.** Then ip the block.
7. Select and place the Sum block from the Math Operations library. Double-click on it, and select round for the Icon shape. In the List of Signs window, type `+--`. Click **OK.**

8. Select and place the Signal Generator block from the Sources library. Double-click on it, select square wave for the Wave form, 3 for the Amplitude, 0.5 for the Frequency, and Hertz for the Units. Click **OK.**
9. Once the blocks have been placed, connect arrows as shown in the gure.
10. Set the Stop time to 10, run the simulation, and examine the plot of $\theta(t)$ in the Scope. This completes the simulation.

Applications in DSP

dsp_links : Identify whether blocks in model are current, deprecated, or obsolete

Syntax

1. dsp_links
2. dsp_links('modelname')

Description

dsp_links returns a structure with three elements that identify whether the Signal Processing Blockset blocks in the current model are current, deprecated, or obsolete. Each element represents one of the three block categories and contains a cell array of strings. Each string is the name of a library block in the current model.

dsp_links('modelname') returns the three-element structure for the specified model.

Examples

Display block support information for the specified model, and then find the name of the first current block:

```
sys = 'dspcochlear';
load_system(sys)          % Load the dspcochlear model
links = dsp_links(sys)    % Run dsp_links on the model
links.current{1}          % Find the name of the first current block
```

dsplib: Open top-level Signal Processing Blockset library

Description

dsplib opens the top-level Signal Processing Blockset block library model.

Dspstartup:Configure Simulink environment for signal processing systems

Description

dspstartup configures Simulink environment parameters with settings appropriate for a typical signal processing project. You can use the dspstartup function in the following ways: At the MATLAB command line. Doing so configures the Simulink environment in your current session for signal processing projects.
By adding a call to the dspstartup function from your startup.m file. When you do so, MATLAB configures your Simulink environment for typical signal processing projects each time you launch MATLAB. When the function successfully configures your Simulink environment, MATLAB displays the following message in the command window. Changed default Simulink settings for signal processing systems (dspstartup.m).

The dspstartup.m file executes the following commands. See Model and Block Parameters in the Simulink documentation.

```
set_param(0, ...
      'SingleTaskRateTransMsg',        'error', ...
      'multiTaskRateTransMsg',         'error', ...
      'Solver',                        'fixedstepdiscrete', ...
      'SolverMode',                    'SingleTasking', ...
      'StartTime',                     '0.0', ...
      'StopTime',                      'inf', ...
      'FixedStep',             'auto', ...
      'SaveTime',               'off', ...
      'SaveOutput',            'off', ...
      'AlgebraicLoopMsg',      'error', ...
      'SignalLogging',         'off');
```

Examples
Add a call to the dspstartup function from your startup.m file:

To find out if there is a startup.m file on your MATLAB path, run the following code at the MATLAB command line:
which startup.m

If MATLAB returns 'startup.m' not found., see Startup Options in the MATLAB documentation to learn more about the startup.m file. If MATLAB returns a path to your startup.m file, open that file for editing. edit startup.m
Add a call to the dspstartup function. Your startup.m file now resembles the following code sample:

```
%STARTUP   Startup file
%   This file is executed when MATLAB starts up, if it exists
%   anywhere on the path.  In this case, the startup.m file
%   runs the dspstartup.m file to configure the Simulink
%   environment with settings appropriate for typical
%   signal processing projects.
dspstartup;
```

liblinks: Check model for blocks from specific Signal Processing Blockset libraries
Syntax
liblinks(lib)
liblinks(lib,sys)
liblinks(lib,sys,c)

Description
liblinks(lib) returns a cell array of strings that lists the blocks in the current model that are linked to the specified libraries. The input lib provides a cell array of strings with the library names. Use the library name visible in the title bar when you open a library model.

liblinks(lib,sys) acts on the named model sys.

liblinks(lib,sys,c) changes the foreground color of the returned blocks to the color c. Possible values of c are 'blue', 'green', 'red', 'cyan', 'magenta', 'yellow', or 'black'.

rebuffer_delay: Number of samples of delay introduced by buffering and unbuffering operations
Syntax
d = rebuffer_delay(f,n,v)
d = rebuffer_delay(f,n,v,'mode')

Description
d = rebuffer_delay(f,n,v) returns the delay, in samples, introduced by the Buffer or Unbuffer block in multitasking operations.

d = rebuffer_delay(f,n,v,'mode') returns the delay, in samples, introduced by the Buffer or Unbuffer block in the specified tasking mode.

Input Arguments
f : Frame size of the input to the Buffer or Unbuffer block.
n : Size of the output buffer. Specify one of the following:

The value of the Output buffer size parameter, if you are computing the delay introduced by a Buffer block.

1, if you are computing the delay introduced by an Unbuffer block. Amount of buffer overlap. Specify one of the following:

The value of the Buffer overlap parameter, if you are computing the delay introduced by a Buffer block. 0, if you are computing the delay introduced by an Unbuffer block.

'mode'

The tasking mode of the model. Specify one of the following options:
'singletasking'
'multitasking'
Default: 'multitasking'
Definitions
Multitasking :When you run a model in MultiTasking mode, Simulink processes groups of blocks with the same execution priority through each stage of simulation based on task priority. Multitasking mode helps to create valid models of real-world multitasking systems, where sections of your model represent concurrent tasks. The Tasking mode for periodic sample times parameter on the Solver pane of the Configuration Parameters dialog box controls this setting.

Singletasking: When you run a model in SingleTasking mode, Simulink processes all blocks through each stage of simulation together. The Tasking mode for periodic sample times parameter on the Solver pane of the Configuration Parameters dialog box controls this setting.
Examples
Compute the delay introduced by a Buffer block in a multitasking model: Open a model containing a Buffer block. For this example, open the doc_buffer_tut4 model by typing doc_buffer_tut4 at the MATLAB command line. Double-click the Buffer block to open the block mask. Verify that you have the following settings:

Output buffer size = 3
Buffer overlap = 1
Initial conditions = 0
Based on these settings, two of the required inputs to the rebuffer_delay function are as follows:

n = 3
v = 1

To determine the frame size of the input signal to the Buffer block, open the Signal From Workspace block mask. Verify that you have the following settings:

Signal = sp_examples_src

Sample time = 1
Samples per frame = 4

Because Samples per frame = 4, you know the f input to the rebuffer_delay function is 4.

After you verify the values of all the inputs to the rebuffer_delay function, determine the delay that the Buffer block introduces in this multitasking model. To do so, type the following at the MATLAB command line:

```
d = rebuffer_delay(4,3,1)
d =     8
```

Compute the delay introduced by an Unbuffer block in a multitasking model: Open a model containing an Unbuffer block. For this example, open the doc_unbuffer_ref1 model by typing doc_unbuffer_ref1 at the MATLAB command line. To determine the frame size of the input to the Buffer block, open the Signal From Workspace block mask by double-clicking the block in your model. Verify that you have the following settings:
Signal = sp_examples_src
Sample time = 1
Samples per frame = 3

Because Samples per frame = 3, you know the f input to the rebuffer_delay function is 3. Use the rebuffer_delay function to determine the amount of delay that the Unbuffer block introduces in this multitasking model. To compute the delay introduced by the Unbuffer block, use f = 3, n = 1 and v = 0.

```
d = rebuffer_delay(3,1,0)
d =    3
```

DFT

Spectral analysis is the process of identifying component frequencies in data. For discrete data, the computational basis of spectral analysis is the *discrete Fourier transform (DFT)*. The DFT transforms time- or space-based data into frequency-based data.

The DFT of a vector x of length n is another vector y of length n:

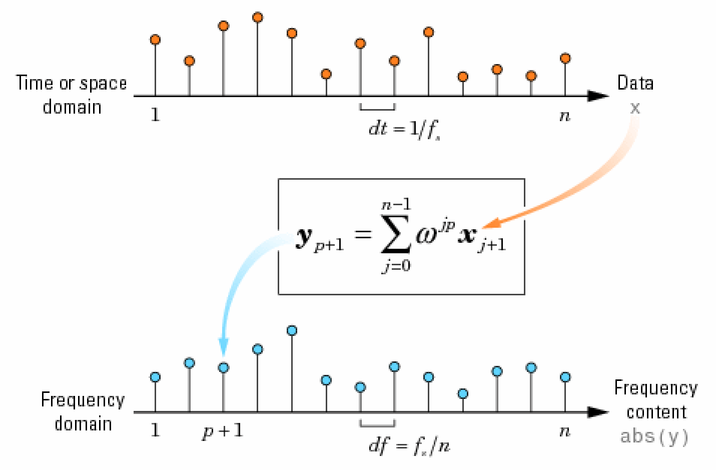
$$y_{p+1} = \sum_{j=0}^{n-1} \omega^{jp} x_{j+1}$$

where ω is a complex nth root of unity:

$$\omega = e^{-2\pi i/n}$$

This notation uses i for the imaginary unit, and p and j for indices that run from 0 to n–1. The indices p+1 and j+1 run from 1 to n, corresponding to ranges associated with MATLAB vectors.

Data in the vector x are assumed to be separated by a constant interval in time or space, $dt = 1/f_s$ or $ds = 1/f_s$, where f_s is the *sampling frequency*. The DFT y is complex-valued. The absolute value of y at index p+1 measures the amount of the frequency $f = p(f_s / n)$ present in the data.



The first element of y, corresponding to zero frequency, is the sum of the data in x. This DC component is often removed from y so that it does not obscure the positive frequency content of the data.

FFT

DFTs with a million points are common in many applications. Modern signal and image processing applications would be impossible without an efficient method for computing the DFT.

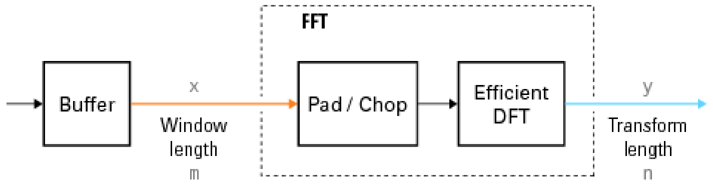
Direct application of the definition of the DFT (see Discrete Fourier Transform (DFT)) to a data vector of length n requires n multiplications and n additions—a total of $2n^2$ floating-point operations. This does not include the generation of the powers of the complex nth root of unity ω. To compute a million-point DFT, a computer capable of doing one multiplication and addition every microsecond requires a million seconds, or about 11.5 days.

Fast Fourier Transform (FFT) algorithms have computational complexity $O(n \log n)$ instead of $O(n^2)$. If n is a power of 2, a one-dimensional FFT of length n requires less than $3n \log_2 n$ floating-point operations (times a proportionality constant). For $n = 220$, that is a factor of almost 35,000 faster than $2n^2$.

The MATLAB functions `fft`, `fft2`, and `fftn` (and their inverses `ifft`, `ifft2`, and `ifftn`, respectively) all use fast Fourier transform algorithms to compute the DFT.

> **Note** MATLAB FFT algorithms are based on FFTW, "The Fastest Fourier Transform in the West" (`http://www.fftw.org`). See `fft` and `fftw` for details.

When using FFT algorithms, a distinction is made between the *window length* and the *transform length*. The window length is the length of the input data vector. It is determined by, for example, the size of an external buffer. The transform length is the length of the output, the computed DFT. An FFT algorithm pads or chops the input to achieve the desired transform length. The following figure illustrates the two lengths.



The execution time of an FFT algorithm depends on the transform length. It is fastest when the transform length is a power of two, and almost as fast when the transform length has only small prime factors. It is typically slower for transform lengths that are prime or have large prime factors. Time differences, however, are reduced to insignificance by modern FFT algorithms such as those used in MATLAB. Adjusting the transform length for efficiency is usually unnecessary in practice.

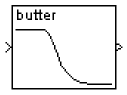
Digital IIR Filter Design (Obsolete)

Design and implement IIR filter

Library

`dspobslib`

Description



> **Note** The Digital IIR Filter Design block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Digital Filter block.

The Digital IIR Filter Design block designs a discrete-time (digital) IIR filter in a lowpass, highpass, bandpass, or bandstop configuration, and applies it to the input using the Direct-Form II Transpose Filter (Obsolete) block.

An M-by-N sample-based matrix input is treated as $M*N$ independent channels, and an M-by-N frame-based matrix input is treated as N independent channels. In both cases, the block filters each channel independently over time, and the output has the same size and frame status as the input.

The **Design method** parameter allows you to specify Butterworth, Chebyshev type I, Chebyshev type II, and elliptic filter designs. Note that for the bandpass and bandstop configurations, the actual filter length is twice the **Filter order** parameter value.

Filter Design	Description
Butterworth	The magnitude response of a Butterworth filter is maximally flat in the passband and monotonic overall.
Chebyshev type I	The magnitude response of a Chebyshev type I filter is equiripple in the passband and monotonic in the stopband.
Chebyshev type II	The magnitude response of a Chebyshev type II filter is monotonic in the passband and equiripple in the stopband.
Elliptic	The magnitude response of an elliptic filter is equiripple in both the passband and the stopband.

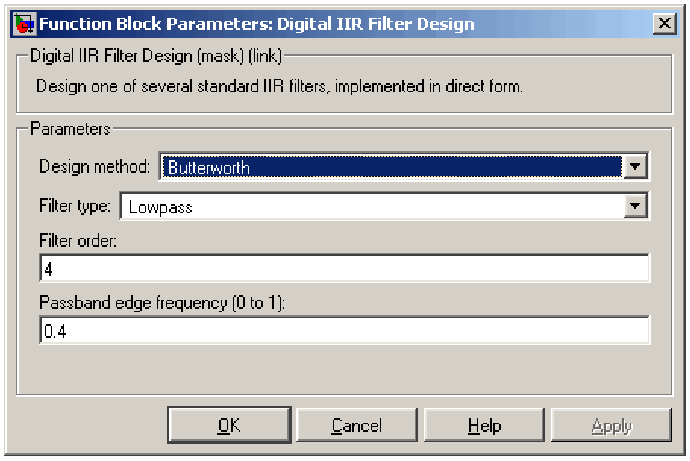
The design and band configuration of the filter are selected from the **Design method** and **Filter type** pop-up menus in the dialog box. For each combination of design method and band configuration, an appropriate set of secondary parameters is displayed.

The table below lists the available parameters for each design/band combination. For lowpass and highpass band configurations, these parameters include the passband edge frequency f_{np}, the stopband edge frequency f_{ns}, the passband ripple R_p, and the stopband attenuation R_s. For bandpass and bandstop configurations, the parameters include the lower and upper passband edge frequencies, f_{np1} and f_{np2}, the lower and upper stopband edge frequencies, f_{ns1} and f_{ns2}, the passband ripple R_p, and the stopband attenuation R_s. Frequency values are normalized to half the sample frequency, and ripple and attenuation values are in dB.

	Lowpass	Highpass	Bandpass	Bandstop
Butterworth	Order, f_{np}	Order, f_{np}	Order, f_{np1}, f_{np2}	Order, f_{np1}, f_{np2}
Chebyshev Type I	Order, f_{np}, R_p	Order, f_{np}, R_p	Order, f_{np1}, f_{np2}, R_p	Order, f_{np1}, f_{np2}, R_p
Chebyshev Type II	Order, f_{ns}, R_s	Order, f_{ns}, R_s	Order, f_{ns1}, f_{ns2}, R_s	Order, f_{ns1}, f_{ns2}, R_s
Elliptic	Order, f_{np}, R_p, R_s	Order, f_{np}, R_p, R_s	Order, f_{np1}, f_{np2}, R_p, R_s	Order, f_{np1}, f_{np2}, R_p, R_s

The digital filters are designed using Signal Processing Toolbox software's filter design commands `butter`, `cheby1`, `cheby2`, and `ellip`.

Dialog Box


Function Block Parameters: Digital IIR Filter Design

Digital IIR Filter Design (mask) (link)

Design one of several standard IIR filters, implemented in direct form.

Parameters

Design method: Butterworth

Filter type: Lowpass

Filter order: 4

Passband edge frequency (0 to 1): 0.4

The parameters displayed in the dialog box vary for different design/band combinations. Only some of the parameters listed below are visible in the dialog box at any one time.

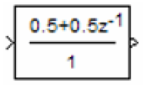
Discrete FIR Filter

Model FIR filters

Library

Discrete

Description



The Discrete FIR Filter block independently filters each channel of the input signal with the specified digital FIR filter. The block can implement static filters with fixed coefficients, as well as time-varying filters with coefficients that change over time. You can tune the coefficients of a static filter during simulation.

This block filters each channel of the input signal independently over time, treating each element of the input as an individual channel. The output dimensions equal those of the input, except in single-input/multi-output mode.

The outputs of this block numerically match the outputs of the Signal Processing Blockset Digital Filter Design block and of the Signal Processing Toolbox `dfilt` object.

This block supports the Simulink state logging feature. See States in the *Simulink User's Guide* for more information.

Filter Structure Support

You can change the filter structure implemented with the Discrete FIR Filter block by selecting one of the following from the **Filter structure** parameter:

- `Direct form`
- `Direct form symmetric`
- `Direct form antisymmetric`
- `Direct form transposed`
- `Lattice MA`

You must have an available Signal Processing Blockset software license to run a model with any of these filter structures other than direct form.

Specifying Initial States

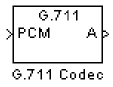
The Discrete FIR Filter block initializes the internal filter states to zero by default, which has the same effect as assuming that past inputs and outputs are zero. You can optionally use the **Initial states** parameter to specify nonzero initial conditions for the filter delays.

To determine the number of initial states you must specify and how to specify them, see the table on valid initial states. The **Initial states** parameter can take one of the forms described in the next table.

Valid Initial States

Initial Condition	Description
Scalar	The block initializes all delay elements in the filter to the scalar value.
Vector or matrix (for applying different delay elements to each channel)	Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel: • The vector length equal the product of the number of input channels and the number of delay elements in the filter, `#_of_filter_coeffs-1` (or `#_of_reflection_coeffs` for `Lattice MA`). • The matrix must have the same number of rows as the number of delay elements in the filter, `#_of_filter_coeffs-1` (`#_of_reflection_coeffs` for `Lattice MA`), and must have one column for each channel of the input signal.

PCM



The G711 Codec block is a logarithmic scalar quantizer designed for narrowband speech. Narrowband speech is defined as a voice signal with an analog bandwidth of 4 kHz and a Nyquist sampling frequency of 8 kHz. The block quantizes a narrowband speech input signal so that it can be transmitted using only 8-bits. The G711 Codec block has three modes of operation: encoding, decoding, and conversion. You can choose the block's mode of operation by setting the **Mode** parameter.

If, for the **Mode** parameter, you choose `Encode PCM to A-law`, the block assumes that the linear PCM input signal has a dynamic range of 13 bits. Because the block always operates in saturation mode, it assigns any input value above $2^{12}-1$ to $2^{12}-1$ and any input value below -2^{12} to -2^{12}. The block implements an A-law quantizer on the input signal and outputs A-law index values. When you choose `Encode PCM to mu-law`, the block assumes that the linear PCM input signal has a dynamic range of 14 bits. Because the block always operates in saturation mode, it assigns any input value above $2^{13}-1$ to $2^{13}-1$ and any input value below -2^{13} to -2^{13}. The block implements a mu-law quantizer on the input signal and outputs mu-law index values.

If, for the **Mode** parameter, you choose `Decode A-law to PCM`, the block decodes the input A-law index values into quantized output values using an A-law lookup table. When you choose `Decode mu-law to PCM`, the block decodes the input mu-law index values into quantized output values using a mu-law lookup table.

If, for the **Mode** parameter, you choose `Convert A-law to mu-law`, the block converts the input A-law index values to mu-law index values. When you choose `Convert mu-law to A-law`, the block converts the input mu-law index values to A-law index values.

> **Note** Set the **Mode** parameter to `Convert A-law to mu-law` or `Convert mu-law to A-law` only when the input to the block is A-law or mu-law index values.

DPCM

The quantization in the section Quantizing a Signal requires no *a priori* knowledge about the transmitted signal. In practice, you can often make educated guesses about the present signal based on past signal transmissions. Using such educated guesses to help quantize a signal is known as *predictive quantization*. The most common predictive quantization method is differential pulse code modulation (DPCM).

The functions `dpcmenco`, `dpcmdeco`, and `dpcmopt` can help you implement a DPCM predictive quantizer with a linear predictor.

Back to Top

DPCM Terminology

To determine an encoder for such a quantizer, you must supply not only a partition and codebook as described in Representing Partitions and Representing Codebooks, but also a *predictor*. The predictor is a function that the DPCM encoder uses to produce the educated guess at each step. A linear predictor has the form

```
y(k) = p(1)x(k-1) + p(2)x(k-2) + ... + p(m-1)x(k-m+1) + p(m)x(k-m)
```

where `x` is the original signal, `y(k)` attempts to predict the value of `x(k)`, and `p` is an `m`-tuple of real numbers. Instead of quantizing `x` itself, the DPCM encoder quantizes the *predictive error*, `x-y`. The integer `m` above is called the *predictive order*. The special case when `m = 1` is called *delta modulation*.

Back to Top

Representing Predictors

If the guess for the `k`th value of the signal `x`, based on earlier values of `x`, is

```
y(k) = p(1)x(k-1) + p(2)x(k-2) +...+ p(m-1)x(k-m+1) + p(m)x(k-m)
```

then the corresponding predictor vector for toolbox functions is

```
predictor = [0, p(1), p(2), p(3),..., p(m-1), p(m)]
```

> **Note** The initial zero in the predictor vector makes sense if you view the vector as the polynomial transfer function of a finite impulse response (FIR) filter.

drivermanager

Construct database drivermanager object

Syntax

```
dm = drivermanager
```

Description

`dm = drivermanager` constructs a database drivermanager object which comprises the properties for all loaded database drivers. Use `get` and `set` to obtain and change the properties of dm.

Examples

- `dm = drivermanager` creates a database drivermanager object dm.
- `get(dm)` returns properties of the drivermanager object dm.

DTMF Generator and Receiver

This demo illustrates a dual-tone multifrequency (DTMF) generator and receiver. The model includes a bandpass filter bank receiver, a spectrogram plot of the generated tones, and a shift register to store the decoded digits. An alternate form of the demo, dspdtmf_audio.mdl, includes real-time soundcard audio on all platforms except Solaris™.

Contents

DTMF Generator

DTMF signaling uses two tones to represent each key on the touch pad. There are 12 distinct tones. When any key is pressed the tone of the column and the tone of the row are generated. As an example, pressing the '5' button generates the tones 770 Hz and 1336 Hz. In this demo, use the number 10 to represent the '*' key and 11 to represent the '#' key.

The frequencies were chosen to avoid harmonics: no frequency is a multiple of another, the difference between any two frequencies does not equal any of the frequencies, and the sum of any two frequencies does not equal any of the frequencies.

The frequencies of the tones are as follows:

```
        1209 Hz     1336 Hz     1477 Hz

697 Hz     1           2           3
770 Hz     4           5           6
852 Hz     7           8           9
941 Hz     *           0           #
```

DTMF Receiver

Running the Demo

When you run the simulation, the spectrogram of the received tone will be constructed. If you use the version of the model designed for audio hardware, the received tone is played through the system soundcard. The detected dialed numbers will be shown on the numeric display scope. The following parameters can be adjusted:

- Frequency bias for each tone (from the DTMF Generator mask dialog)
- Channel noise power and signal gain (from the Channel mask dialog)
- Receiver bandpass filter frequency bandwidth (from the DTMF Receiver mask dialog)