Source Coding

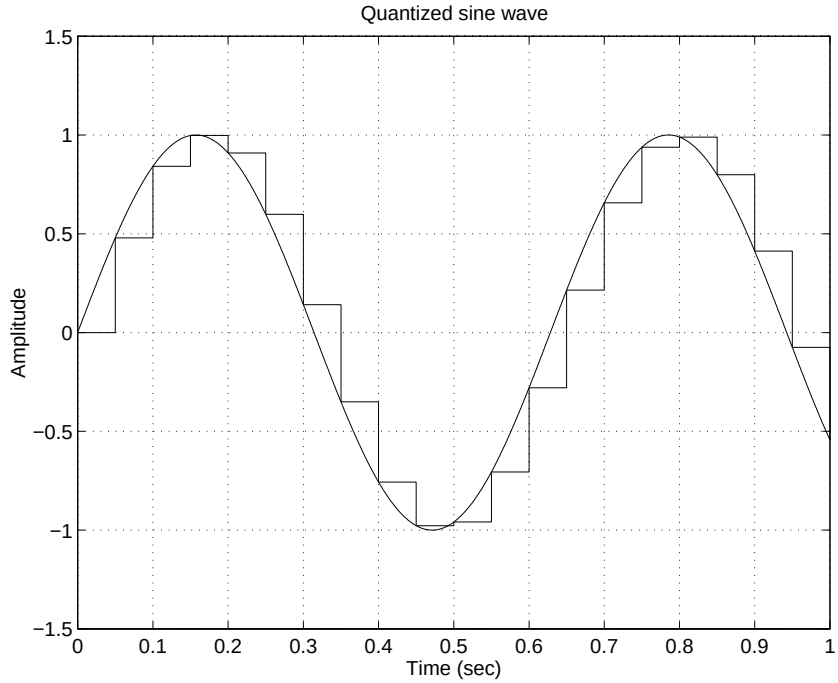
The Communications Toolbox includes some basic functions for source coding. Source coding, also known as quantization or signal formatting, includes the concepts of analog-to-digital conversion and data compression.

Source coding divides into two basic procedures: source encoding and source decoding. Source encoding converts a source signal into a digital code using a quantization method. The source coded signal is represented by a set of integers {0, 1, 2, ..., N-1}, where N is finite. Source decoding recovers the original information signal sequence using the source coded signal. This toolbox includes two source coding quantization methods: scalar quantization and predictive quantization. A third source coding method, vector quantization, is not included in this toolbox.

Scalar Quantization

Scalar quantization is a process that assigns a single value to inputs that are within a specified range. Inputs that fall in a different range of values are assigned a different single value. An analog signal is in effect digitized by

scalar quantization. For example, a sine wave, when quantized, will look like a rising and falling stair step:



Scalar quantization requires the use of a mapping of N contiguous regions of the signal range into N discrete values. The N regions are defined by a partitioning that consists of N-1 distinction partition values within the signal range. The partition values are arranged in ascending order and assigned indices ranging from 1 to N-1. Each region has an index that is determined by this formula:

$$indx(x) = \begin{cases} 0 & x \le \text{partition}(1) \\ i & \text{partition}(i) < x \le \text{partition}(i+1) \\ N-1 & \text{partition}(N-1) < x \end{cases}$$

For a signal value x, the index of the corresponding region is indx(x).

To implement scalar quantization you must specify a length N-1 vector *partition* and a length N vector *codebook*. The vector partition, as its name

implies, divides the signal input range into N regions using N-1 partition values. The partition values must be in strictly ascending order. The codebook is a vector that assigns a value, typically either an endpoint of the region or some average value of the interval, to each region defined in the partition. Since each region must have an assigned output value, the length of the codebook must equal the length of the partition. Another way to view this is that the codebook functions as a table lookup with each element assigned to a partition.

The index value *indx(x)* is the output of the quantization encode function. The codebook contains the values that correspond to sample points. There is no function for quantization decoding, which is simply constructing the quantized signal using the stream of index values output by the quantizer. Construct the quantized signal by using the MATLAB command:

```
y = codebook(indx+1);
```

In general, a codebook has the following relation with the vector partition:

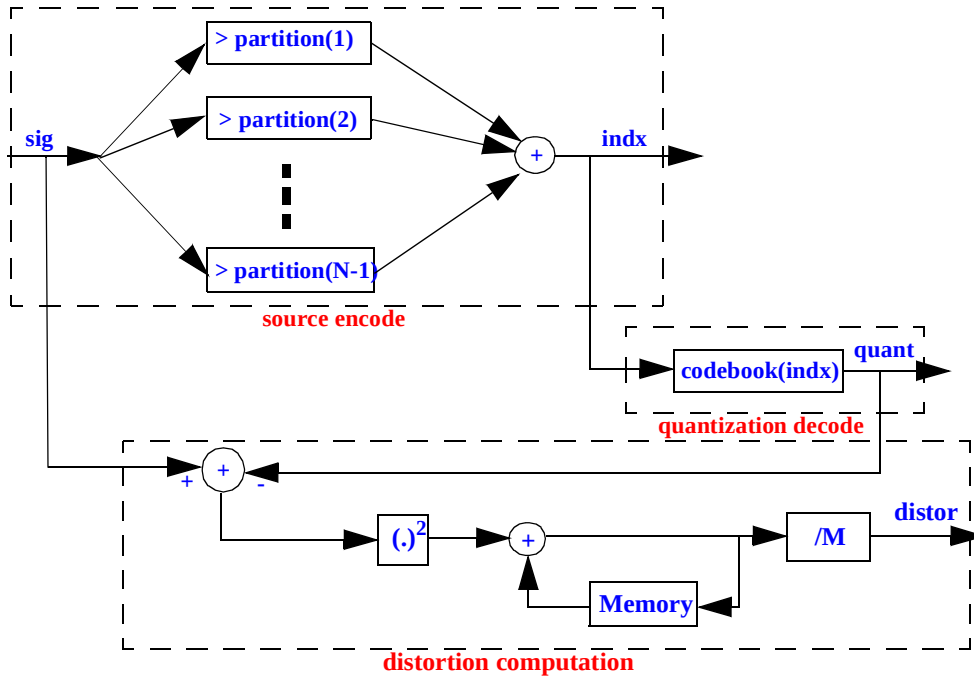
$$\begin{aligned}\text{codebook}(1) \le \text{partition}(1) \le \text{codebook}(2) \le \text{partition}(2) \le \ldots \\ \ldots \le \text{codebook}(N-1) \le \text{partition}(N-1) \le \text{codebook}(N)\end{aligned}$$

The quality of the quantization, called the distortion, is the mean-square error between the original signal data `sig` and the quantized signal quan:

$$\text{distortion} = \frac{1}{\text{M}} \sum_{i=1}^{M} (\text{sig(i)} - \text{quan(i)})^2$$

where M is the number of samples of the source signal `sig`.

The computation procedure for the quantization source coding and decoding is shown in the figure below:



The dashed square at the top of this figure is the source encode algorithm, which assigns an index after deciding in which region the input signal value falls. The dashed square in the middle of the figure is the quantization decode algorithm, which maps the input index to whatever value the codebook assigns to that particular index. The dashed square at the bottom of the figure is the distortion computation, which calculates a cumulative average in which M is the total number of points used in the computation.

The MATLAB function `quantiz` computes all three outputs shown in the above figure. The Simulink Scalar Quantizer block is available in the source code sublibrary.

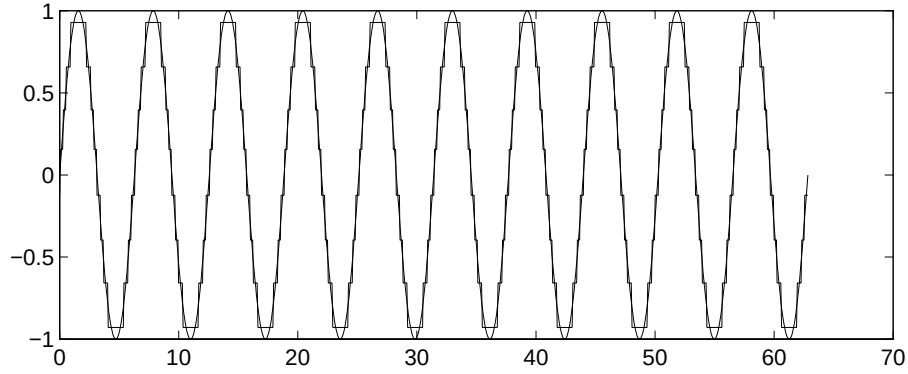
Training of Partition and Codebook Parameters

The key functions in quantization are the assignment of the partition and codebook parameters. In large signal sets with a fine quantization scheme, the

selection of all the correct parameters can be tedious. In the Communications Toolbox you can train these two parameters by using the MATLAB function `lloyds`. To train the parameters, you must prepare a training set, which typically represents function input data. The function `lloyds` finds the partition and codebook parameter vectors by minimizing the distortion using the provided training data. Here is an example of the data training for a sinusoidal signal:

```
N = 2^3; % three bits transfer channel
t = [0:1000]*pi/50;
sig = sin(t); % one complete period of sinusoidal signal
[partition,codebook] = lloyds(sig,N);
[indx,quant,distor] = quantiz(sig,partition,codebook);
plot(t,sig,t,quant,'--');
```

In the above commands, `sig` is a sinusoidal signal to be quantized. The peak amplitude of the input signal must be one. The trained codebook and the partition can be used for sinusoidal signals of any frequency. The above code generates a figure that compares the original signal (the smooth curve) to the quantized signal (the digital curve):



The decoding procedure is simple using the basic MATLAB computation format. Use the following command to obtain the decoded result:

```
quant = codebook(indx+1)
```

Companders

The quantization discussed above is linear. In certain applications, you may need to quantize a signal based on the power level of the input signal. In this case, it is common to use a logarithm computation before the quantization operation. Since a simple logarithm computation can only handle a positive signal, some modification of the input signal is needed. The logarithm computation is known as a *compressor*. The reverse computation of a compressor is called an *expander*. The combination of a compressor and expander is called a compander (compress and expand). This toolbox supports two companders: the μ-law and A-law companders. The selection of either method is a matter of user preference.

The MATLAB function `compand` is designed for compander computation. The Simulink block library includes four blocks for the μ-law and A-law compander computations: μ-Law Compressor, μ-Law Expander, A-Law Compressor, and A-Law Expander.

μ-law Compander

For a given signal x, the output y of the μ-law compressor is

$$y = \frac{V\log(1+\mu|x|/V)}{\log(1+\mu)}\operatorname{sgn}(x)$$

where V is the peak value of signal x, which is also the peak value of y. μ is the μ-law parameter of the compander. The function `log` is the natural logarithm and `sgn` is the sign function.

The μ-law expander is the inverse of the compressor:

$$x = \frac{V}{\mu}(e^{|y|\log(1+\mu)/V} + 1)\operatorname{sgn}(y)$$

The MATLAB function for μ-law companding is `compand`. The corresponding Simulink μ-Law Compressor and μ-Law Expander blocks also support μ-law companding.

A-law Compander

For a given signal x, the output y of the A-law compressor is

$$y = \begin{cases} \dfrac{A|x|}{1+\log A}\operatorname{sgn}(x) & \text{for } 0 \le |x| \le A/V \\ \dfrac{V(1+\log A|x|/V)}{1+\log A}\operatorname{sgn}(x) & \text{for } A/V < |x| \le V \end{cases}$$

where V is the peak value of signal x, which is also the peak value of y. A is the A-law parameter of the compander. The function log is the natural logarithm and sgn is the sign function.

The A-law expander is the inverse of the compressor:

$$x = \begin{cases} |y|\dfrac{1+\log A}{A}\operatorname{sgn}(y) & \text{for } 0 \le |y| \le \dfrac{V}{1+\log A} \\ e^{|y|(1+\log A)/V - 1}\dfrac{V}{A}\operatorname{sgn}(y) & \text{for } \dfrac{V}{1+\log A} < |y| \le V \end{cases}$$

The MATLAB function for A-law companding is compand. The corresponding Simulink A-Law Compressor and A-Law Expander blocks also support A-law companding.

Predictive Quantization

The quantization introduced in the “Scalar Quantization” section is usually implemented when there is no *a priori* knowledge about the transmitted signal. In practice, a communications engineer often has some *a priori* information about the message signals. An engineer can use this information to predict the next signal to be transmitted based on past signal transmissions; i.e., he or she can use the past data set $x=\{x(k-m),\ldots,x(k-2), x(k-1)\}$ to predict $x(k)$ by using some function f(.). The most common way to implement predictive quantization is to use the differential pulse code modulation (DPCM) method. The Communications Toolbox provides the tools necessary to implement a DPCM predictive quantizer.

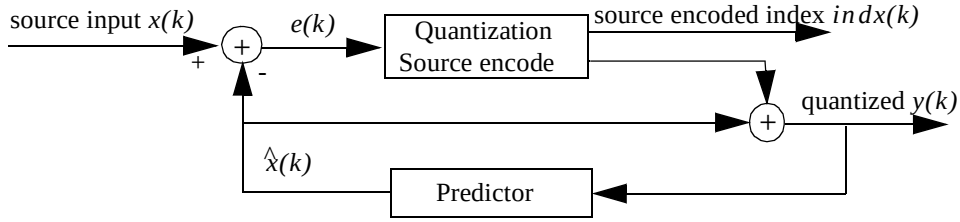
Differential Pulse Code Modulation

Using the past data set and predictor as described above, the predicted value is assumed to be

$$\hat{x}(k) = f(x(k-m), ..., x(k-2), x(k-1))$$

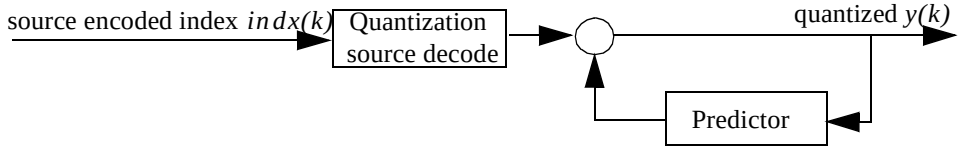
where k is the computation step index. The function f(.) is called the *predictor*; the integer m is the predictive order. The predictive error $e(k) = x(k) - \hat{x}(k)$ is quantized by using the method discussed in the "Scalar Quantization" section.

The structure of predictive quantization is:



This method is known as the differential pulse code modulation method (DPCM). In the figure, indx(k) is the source encoded index, and y(k) is the quantized output. The DPCM method transfers the bit length reduced indx(k) instead of the real data x(k). At the receiving side, a quantization decoder recovers the quantized y(k) from indx(k).

The figure below shows the quantization source decoding method:



The predictor must be the same one used in the encoding figure.

This toolbox uses a linear predictor:

$$\hat{x}(k) = p(1)x(k-1) + ... + p(m-1)x(k-m+1) + p(m)x(k-m)$$

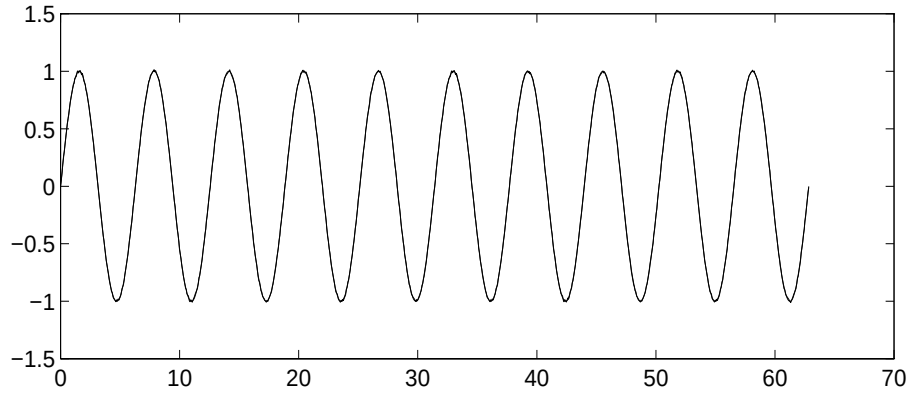
The transfer function of this predictor is represented by a polynomial. The vector p_trans = $[0, p(1) \ldots p(k-m+1), p(k-m)]$ represents the finite impulse response (FIR) transfer function:

A special case of the DPCM source code method is the widely used delta-modulation method, in which the linear predictor is a first order predictor with

The Communications Toolbox provides MATLAB functions `dpcmenco` and `dpcmdeco` for the source encoding and source decoding using the DPCM method. This toolbox also provides the function `dpcmopt`, which uses a set of training data to generate an optimal transfer function of the predictor `p_trans`, the `partition`, and the `codebook`. The training data represents the input signal used in the DPCM quantization. For example, you can use `dpcmopt` to find the parameters needed to encode/decode a sinusoidal signal using the delta-modulation method. This example is a continuation of the example provided in the “Scalar Quantization” section:

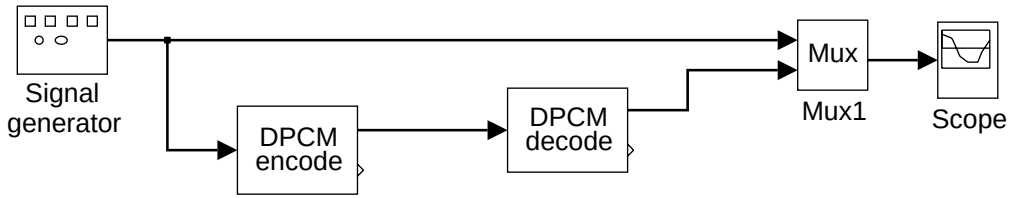
```
% Generate the optimal predictive transfer function,
% the partition, and the codebook.
[p_trans,partition,codebook] = dpcmopt(sig,1,N);
% Encode the signal using DPCM.
indx = dpcmenco(sig,codebook,partition,p_trans);
% Decode using DPCM.
quant = dpcmdeco(indx,codebook,p_trans);
% Compare the original and the quantized signal.
plot(t,sig,t,quant,'--')
```

Note that the sample time is important in the DPCM quantization. The figure below shows the plot generated from the code:



The predictor must be the same one used in the encoding figure. Comparing the result generated in this example to the one generated by scalar quantization, notice that the DPCM quantization is of much better quality. The distortion here is 6.2158e-5, which is much lower than the distortion value of 5.002e-3 achieved by the scalar quantization. Both methods used three bit symbols in the quantization.

Simulink blocks for the DPCM encode and decode are available in the source code sublibrary. A simple block diagram example of using DPCM encode and decode blocks for source coding is:



This block diagram encodes a generated signal using the DPCM method and then recovers the signal by DPCM decoding. The scope in the block diagram compares the original signal with the quantized signal. The curve displayed on the scope is the same as the curve shown in the plot generated from the MATLAB code.

Purpose

Produce a quantization index and a quantized output value

Syntax

```
index = quantiz(sig,partition);
[index,quants] = quantiz(sig,partition,codebook);
[index,quants,distor] = quantiz(sig,partition,codebook);
```

Description

`index = quantiz(sig,partition)` returns the quantization levels in the real vector signal `sig` using the parameter `partition`. `partition` is a real vector whose entries are in strictly ascending order. If `partition` has length n, then `index` is a column vector whose kth entry is

- 0 if $sig(k) \le partition(1)$
- m if $partition(m) < sig(k) \le partition(m+1)$
- n if $partition(n) < sig(k)$

`[index,quants] = quantiz(sig,partition,codebook)` is the same as the syntax above, except that `codebook` prescribes a value for each partition in the quantization and `quants` contains the quantization of `sig` based on the quantization levels and prescribed values. `codebook` is a vector whose length exceeds the length of `partition` by one. `quants` is a row vector whose length is the same as the length of `sig`. `quants` is related to `codebook` and `index` by

```
quants(ii) = codebook(index(ii)+1);
```

where `ii` is an integer between 1 and `length(sig)`.

`[index,quants,distor] = quantiz(sig,partition,codebook)` is the same as the syntax above, except that `distor` estimates the mean square distortion of this quantization data set.

Examples

The command below rounds several numbers between 1 and 100 up to the nearest multiple of ten. `quants` contains the rounded numbers, and `index` tells which quantization level each number is in.

```
[index,quants] = quantiz([3 34 84 40 23],10:10:90,10:10:100)

index =

     0
     3
```

```
     8
     3
     2


quants =

    10    40    90    40    30
```

See Also

`lloyds`, `dpcmenco`, `dpcmdeco`

Purpose

Source code mu-law or A-law compressor or expander

Syntax

```
out = compand(in,Mu,v);
out = compand(in,Mu,v,'mu/compressor');
out = compand(in,Mu,v,'mu/expander');
out = compand(in,A,v,'A/compressor');
out = compand(in,A,v,'A/expander');
```

Description

`out = compand(in,param,v)` implements a μ-law compressor for the input vector `in`. `Mu` specifies μ and `v` is the input signal's maximum magnitude. `out` has the same dimensions and maximum magnitude as `in`.

`out = compand(in,Mu,v,'mu/compressor')` is the same as the syntax above.

`out = compand(in,Mu,v,'mu/expander')` implements a μ-law expander for the input vector `in`. `Mu` specifies μ and `v` is the input signal's maximum magnitude. `out` has the same dimensions and maximum magnitude as `in`.

`out = compand(in,A,v,'A/compressor')` implements an A-law compressor for the input vector `in`. The scalar `A` is the A-law parameter, and `v` is the input signal's maximum magnitude. `out` is a vector of the same length and maximum magnitude as `in`.

`out = compand(in,A,v,'A/expander')` implements an A-law expander for the input vector `in`. The scalar `A` is the A-law parameter, and `v` is the input signal's maximum magnitude. `out` is a vector of the same length and maximum magnitude as `in`.

Note The prevailing parameters used in practice are μ = 255 and A = 87.6.

Examples

The examples below illustrate the fact that compressors and expanders perform inverse operations.

```
compressed = compand(1:5,87.6,5,'a/compressor')
```

```
compressed =

    3.5296    4.1629    4.5333    4.7961    5.0000

expanded = compand(compressed,87.6,5,'a/expander')

expanded =

    1.0000    2.0000    3.0000    4.0000    5.0000
```

Algorithm

For a given signal x, the output of the μ-law compressor is

$$y = \frac{V \log(1 + \mu|x|/V)}{\log(1 + \mu)} \operatorname{sgn}(x)$$

where V is the maximum value of the signal x, μ is the μ-law parameter of the compander, log is the natural logarithm, and sgn is the signum function (`sign` in MATLAB).

The output of the A-law compressor is

$$y = \begin{cases} \dfrac{A|x|}{1 + \log A} \operatorname{sgn}(x) & \text{for } 0 \le |x| \le \dfrac{V}{A} \\ \dfrac{V(1 + \log(A|x|/V))}{1 + \log A} \operatorname{sgn}(x) & \text{for } \dfrac{V}{A} < |x| \le V \end{cases}$$

where A is the A-law parameter of the compander and the other elements are as in the μ-law case.

See Also

`quantiz`, `dpcmenco`, `dpcmdeco`

References

Sklar, Bernard, *Digital Communications: Fundamentals and Applications*, Englewood Cliffs, N.J., Prentice-Hall, 1988.

Source Coding

Source coding in communication systems converts arbitrary real-world information to an acceptable representation in communication systems. This section provides some basic techniques as examples of solving the source coding problems using Simulink and MATLAB. This toolbox includes the source coding techniques of signal quantization and differential pulse code modulation (DPCM).

This section also includes compander techniques. Compander is the name for the combination of compressor and expander. Data compression is important for transforming a signals with different power level transformation.

This figure shows the Source Coding Sublibrary:

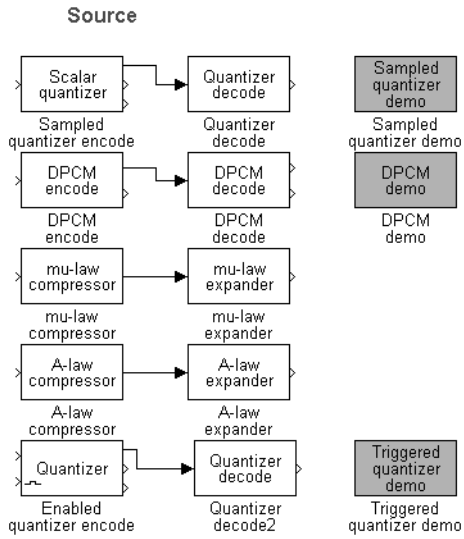


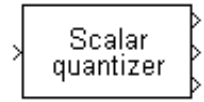
Figure 6-6: Source Coding Sublibrary

Source Coding Reference Table

This table lists the Simulink blocks in the Source Coding Sublibrary. (They are listed alphabetically in this table for your convenience.):

Block Name	Description
A-Law Compressor	Compresses data using an A-law compander
A-Law Expander	Recovers compressed data using an A-law compander
DPCM Decode	Recovers DPCM quantized signals
DPCM Encode	Quantized input data signals
μ-Law Compressor	Compresses data using a μ-law compander
μ-Law Expander	Recovers compressed data using a μ-law compander
Quantization Decode	Recovers signals quantized by the Signal Quantizer or the Triggered Signal Quantizer block
Signal Quantizer	Quantizes an input signal
Triggered Signal Quantizer	Quantizes an input signal when triggered

Signal Quantizer



Catagory Signal Quantization

Location Source Coding Sublibrary

Description The Signal Quantizer block encodes a message signal using scalar quantization. The block uses the finite length of a digit to represent an analog signal. Please refer to chapter 3, the *Tutorial*, for the general principles of quantization computation. Note that you may lose computation accuracy in the quantization processing.

In quantization, the major parameters are **Quantization partition** and **Quantization codebook**. **Quantization partition** is a strict ascending ordered vector, which contains the partition points used in dividing up the input data. **Quantization codebook** is a quantization value vector with length equal to the (length + 1) of the **Quantization partition**. If the input value is less than the `i`th element of **Quantization partition** (and greater than (`i`-1)th element, if any), the quantization value equals to the `i`th element in the **Quantization codebook**.

The figure below shows the quantization process:

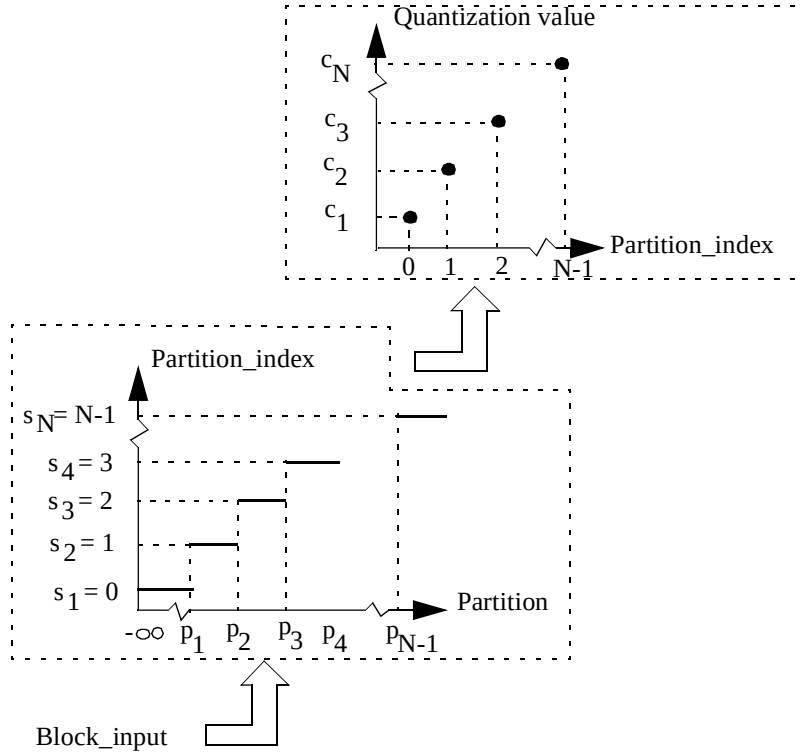
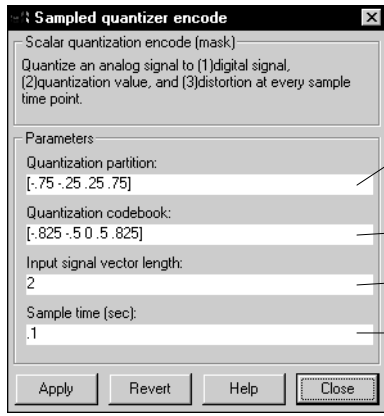


Figure 6-7: Quantization

This block has one input port and three output ports. The input port takes the analog signal. The three output ports output, from top to bottom, the quantization indexFigure 6-7:, the distortion value, and the quantization valueFigure 6-7:. The distortion is a measurement of the quantization error. The vector lengths of all three outputs are equal to the vector length of the input. The quantization block can accept a vector input. When the input is a vector, each output port outputs a vector with the vector length equal to the input vector length. The block processes each element of the vector independently; it performs the quantization at the sample time.

You can use the function `lloyds` to train the available data to obtain the expected partition and codebook vectors.

Dialog Box



A length N vector, where N is the number of symbols in the symbol set. This must be a strictly ascending ordered vector.

A length N-1 strictly ascending ordered vector.

Specify the length of the input signal.

Specify the sample time. When this parameter is a two-element vector, the second element is the offset value.

Characteristics

No. of Inputs/Outputs	1/3
Vectorized Inputs/ Outputs	Yes/Yes
Input Vector Width	Auto
Output Vector Width	Same as the input vector width
Scalar Expansion	N/A
Time Base	Discrete time
States	N/A
Direct feedthrough	Yes

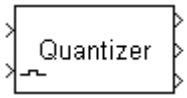
Pair Block

Quantization Decode

Equivalent M-function

`quantiz` for quantization computation

`lloyds` for partition and codebook training using the available data



Triggered Signal Quantizer

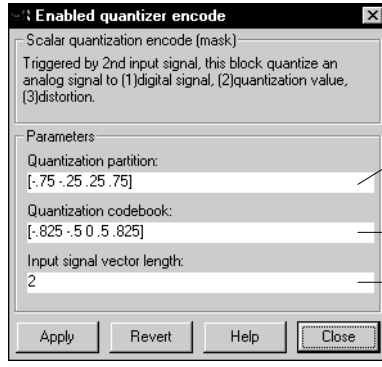
Catagory Signal Quantization

Location Source Coding Sublibrary

Description The Trigger Signal Quantizer block performs quantization when a trigger signal occurs. This block is similar to the Signal Quantizer block except that the quantization processing is controlled by the second input port of this block, the trigger signal. This block renews its output when the scalar signal from the second input port is a nonzero signal. Please refer to the Signal Quantizer block for a discussion of scalar quantization.

This block has two input ports and three output ports. The quantizer block takes message input from the first input port. It takes the trigger signal from the second input port. The three output ports output quantization index, quantization value, and quantization distortion. When the message input is a vector, the three outputs are also vectors with their vector length equal to the input vector length. Each element in the vector is independently processed.

Dialog Box



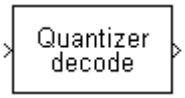
A length N vector, where N is the number of partition values. This must be a strictly ascending ordered vector.

A length N-1 strictly ascending ordered vector.

Specify the length of the input signal.

Triggered Signal Quantizer

Characteristics	No. of Input/Outputs	2/3
	Vectorized No. 1 Input	Yes
	Vectorized No. 2 Input	No
	Vectorized Outputs	Yes
	No. 1 Input Vector Width	Auto
	Output Vector Width	Same as the input vector width
	Scalar Expansion	N/A
	Time Base	Triggered
	States	N/A
	Direct feedthrough	Yes
Pair Block	Quantization Decode	
Equivalent M-function	`quantiz`	



Quantization Decode

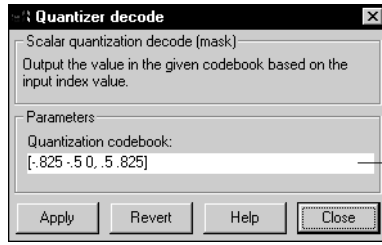
Catagory Decoding

Location Source Coding Sublibrary

Description The Quantization Decode block recovers a message from a quantized signal by finding the quantization value from quantization index. The input of this block is the quantization index, which contains the elements in $S = [s_1, s_2, \ldots s_N] = [0, 1, \ldots N-1]$. The output of this block is quantized value, which contains the elements in $C = [c_1, c_2, \ldots, c_N]$. The vector S and C are introduced in the Signal Quantizer block.

This implementation of this block uses a look-up table.

Dialog Box



A length N vector, where N+1 is the number of partitions. The ith element is the quantization output for the (i-1)th quantization index. The default value for the codebook is [-0.825 -0.5 0 0.5 0.825].

Characteristics

No. of Inputs/ Outputs	1/1
Vectorized Inputs/ Outputs	Yes/Yes
Input Vector Width	Auto
Scalar Expansion	N/A
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Pair Blocks Signal Quantizer, Triggered Signal Quantizer

Equivalent M-function There is no quantization decode function in this toolbox. For decode computation, use the command:

```
y = codebook(quantiz_index + 1);
```

DPCM Encode

Catagory Encoding

Location Source Coding Sublibrary

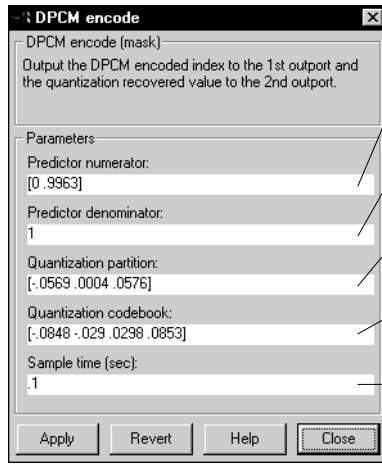
Description The DPCM (Differential Pulse Code Modulation) Encode block quantizes an input signal. This method uses a predictor to estimate the possible value of the signal at the next step based on the past information into the system. The predictive error is quantized.

This method is specially useful to quantize a signal with a predictable value. This block uses the Signal Quantizer block. Refer to the Signal Quantizer block for a discussion of the codebook and partition concepts.

The predictor in this toolbox is assumed to be a linear predictor. You can use the function `dpcmopt` to train the parameters used in this block: **Predictor numerator, Predictor denominator, Quantization partition**, and **Quantization codebook**. You must input the numerator and denominator of the predictor's transfer function, but the output of `dpcmopt` provides only the numerator. In most DPCM applications, the denominator of predictor transfer function is 1, which means that the predictor is a FIR filter.

When the numerator of the predictor transfer function is a first-order polynomial with the first element (zero-order element) equal to one, the DPCM is a delta modulation.

Dialog Box



A vector containing the coefficients in ascending order of the numerator of the predictor transfer function.

A vector containing the coefficients in ascending order of the denominator of the predictor transfer function. Usually this parameter is set to 1.

A length N vector, where N+1 is the number of partition values. This must be a strictly ascending ordered vector.

A length N+1 strictly ascending ordered vector that specifies the output values assigned to each partition.

The calculation sample time. When this parameter is a two-element vector, the second element is the offset value.

Characteristics

No. of Inputs	1/2
Vectorized Inputs/ Outputs	No/No
Time Base	Discrete time
States	N/A
Direct feedthrough	Yes

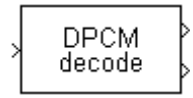
Pair Block

DPCM Decode

Equivalent M-function

`dpcmenco`

DPCM Decode

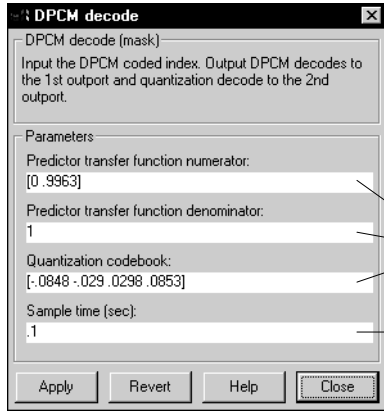


Catagory Decoding

Location Source Coding Sublibrary

Description The DPCM (Differential pulse code modulation) Decode block recovers a quantized signal. This block inputs the DPCM encoded index signal and outputs the recovered signal to the first output port and the predictive error to the second output port.

Dialog Box



Match these parameters to the ones used in the corresponding DPCM Encode block.

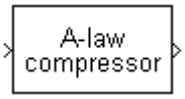
The calculation sample time. When this parameter is a two-element vector, the second element is the offset value.

Characteristics

No. of Inputs	1/2
Vectorized Inputs/ Outputs	No/No
Scalar Expansion	N/A
Time Base	Discrete time
States	N/A
Direct feedthrough	Yes

Pair Block DPCM Encode

Equivalent M-function `dpcmdec`



A-Law Compressor

Catagory Data Compression

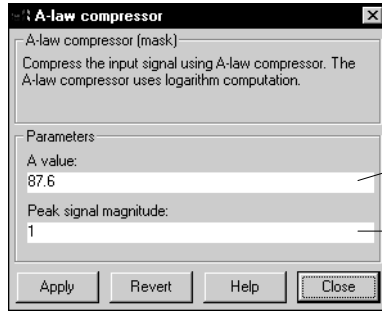
Location Source Coding Sublibrary

Description The A-Law Compression block performs data compression. The formula for the A-law compressor is

$$y = \begin{cases} \dfrac{A|x|}{1+\log A}\,\mathrm{sgn}(x) & \text{for } 0 \le |x| \le A/V \\ \dfrac{V(1+\log A|x|/V)}{1+\log A}\,\mathrm{sgn}(x) & \text{for } A/V < |x| \le V \end{cases}$$

The parameters to be specified in the A-law compressor are the *A* value and the peak magnitude *V*. The most commonly used *A* value in practice is 87.6.

Dialog Box



The parameter A in the A-law compressor equation.

Specify the peak value for the input signal. This is the parameter *V* in the above equation, and the output peak magnitude as well.

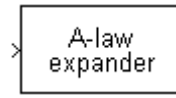
Characteristics

No. of Inputs/Output	1/1
Vectorized Inputs/ Outputs	Yes/Yes
Scalar Expansion	N/A
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Pair Block A-Law Expander

Equivalent M-function `compand`

A-Law Expander



Catagory Data Decompression

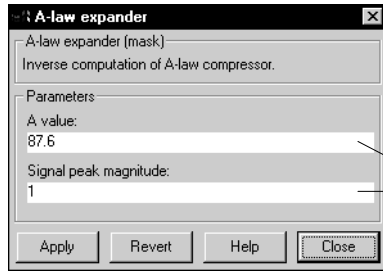
Location Source Coding Sublibrary

Description The A-Law Expander block recovers compressed data. The formula for the A-law expander is the inverse of the compressor function:

$$x = \begin{cases} |y|\dfrac{1+\log A}{A}\operatorname{sgn}(y) & \text{for } 0 \le |y| \le \dfrac{V}{1+\log A} \\ e^{|y|(1+\log A)/V - 1}\dfrac{V}{A}\operatorname{sgn}(y) & \text{for } \dfrac{V}{1+\log A} < |y| \le V \end{cases}$$

You must specify the A value and the peak magnitude V.

Dialog Box



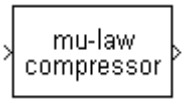
Match these parameters to the ones used in the corresponding A-Law Compressor block.

Characteristics

No. of Inputs/Output	1/1
Vectorized Inputs/ Outputs	Yes/Yes
Scalar Expansion	N/A
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Pair Block A-Law Compressor

Equivalent M-function `compand`



μ-Law Compressor

Catagory Data Compression

Location Source Coding Sublibrary

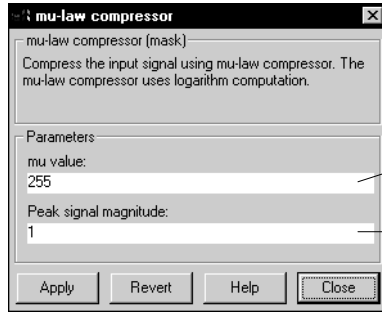
Description The μ-Law Compressor block performs data compression. The formula for the μ-law compressor is:

$$y = \frac{V\log(1+\mu|x|/V)}{\log(1+\mu)}\operatorname{sgn}(x)$$

The parameters to be specified in the μ-law compressor are μ value and the peak magnitude V. The most commonly used μ value in practice is 255.

This block has one input and one output. It takes the x value and outputs the y value described in the above equation.

Dialog Box



The parameter μ in the μ-law compressor equation. The default value is 255.

Specify the peak magnitude of the input signal. This parameter is V in the above equation and the output peak magnitude as well.

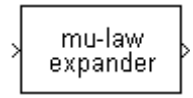
Characteristics

No. of Inputs/Output	1/1
Vectorized Inputs/ Outputs	Yes/Yes
Scalar Expansion	N/A
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Pair Block μ-Law Expander

Equivalent M-function `compand`

μ-Law Expander



Catagory Data Compression

Location Source Coding Sublibrary

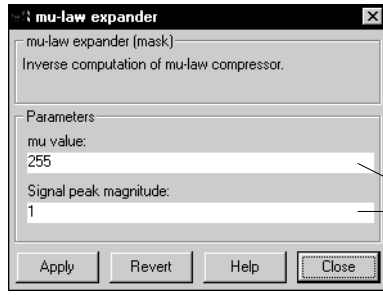
Description The μ-Law Expander block recovers a signal from compressed data. The formula for the μ-law expander is the inverse of the compressor function:

$$x = \frac{V}{\mu}(e^{|y|\log(1+\mu)/V} + 1)\,\mathrm{sgn}(y)$$

Same as the μ-law compressor, the parameters to be specified in the μ-law compressor are μ value and the peak magnitude *V*.

This block takes *y* as the input and outputs *x*.

Dialog Box



Match these parameters to the ones used in the corresponding μ-Law Compressor block.

Characteristics

No. of Inputs/Output	1/1
Vectorized Inputs/ Outputs	Yes/Yes
Scalar Expansion	N/A
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Pair Block μ-Law Compressor

Equivalent M-function `compand`