2 Representing Signals and Systems

Signal processing is rich in representations for signals. Choosing an appropriate form for operating on a signal is essential to many signal processing tasks. Signals and Systems handles a variety of signal forms, including both continuous and discrete formulas. Systems that act on signals are represented in operator form.

■ 2.1 Analog Signals

A variety of interesting types of analysis can be performed on analog signals. These signals are assumed to take on a value from a continuous range, over all time (or space). These signals are represented in traditional *Mathematica* notation, for instance, `Sin[2 Pi t]` for a sinusoidal signal.

Signals and Systems enhances *Mathematica* by allowing the user to employ a variety of functions common in signal processing that are not found in *Mathematica* proper.

`ContinuousPulse[`*width*`, t]`	a pulse with unit height, with t running from 0 to *width*
`DiracDelta[t]`	the Dirac δ function, which represents an impulse in a continous domain
`DirichletSinc[k,` *omega*`]`	the normalized Dirichlet aliased sinc function, defined as `Sin[k omega/2]/(k Sin[omega/2])`
`Sinc[t]`	the sinc function, which is defined as `Sin[t]/t`
`UnitFunction[n][t]`	the n^{th} -order unit singularity function
`UnitStep[t]`	the unit step function, which is defined as 0 for t less than zero, and 1 for t greater than or equal to zero

Some common analog signal processing signals.

Functions such as `Sinc` and `DirichletSinc` correctly evaluate to the limit at zero, unlike a simple definition. Various rules to improve their behavior under numeric evaluation, differentiation, and integration are also implemented.

There is a general class of functions called unit singularity functions that involve discontinuities. These are actually functionals, but useful representations as functions have been devised, at least for real-valued input. `UnitFunction[n][t]` can be used to enter the n^{th}-order unit singularity function. It will automatically be rewritten in terms of the `DiracDelta` function (0^{th}-order singularity function) and the `UnitStep` function (1^{st}-order singularity function). `DiracDelta` and `UnitStep` are actually drawn from standard *Mathematica* functions. Signals and Systems also allows you to call these functions by `ContinuousStep` and `ContinousImpulse`, respectively.

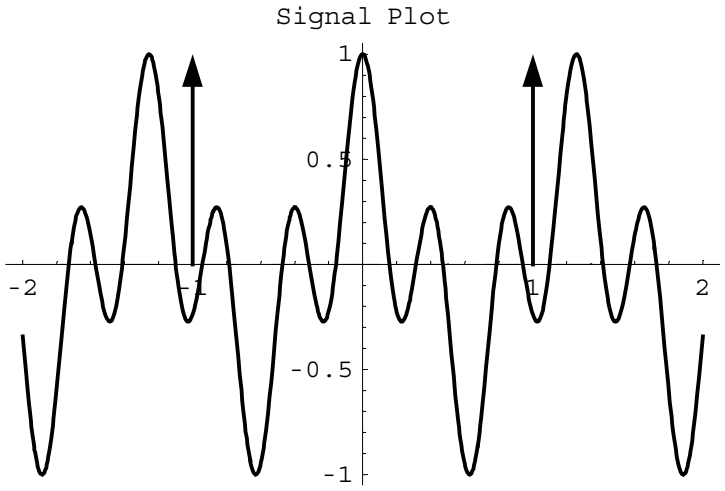
The default definition at the discontinuities of the pulse and step functions used by Signals and Systems is that the value is `1` at the leading discontinuity and, for the `ContinuousPulse` function, `0` at the trailing step.

- First, verify that the signal processing functions are available.

```
In[1]:= Needs["SignalProcessing`"]
```

- Here we plot a signal that includes the `DirichletSinc` function, with an impulse at $\omega = 1$ and at $\omega = -1$.

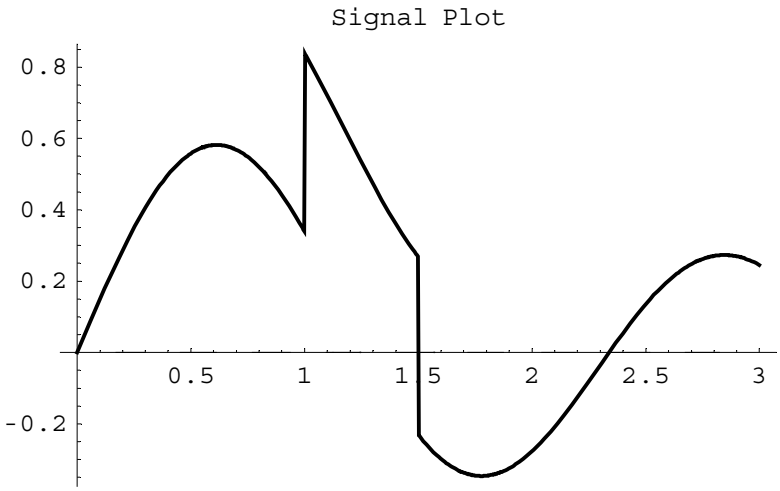
```
In[2]:= SignalPlot[
            DirichletSinc[4, 10 omega] +
                DiracDelta[omega - 1] +
                DiracDelta[omega + 1],
            {omega, -2, 2}
        ]
```




```
Out[2]= - Graphics -
```

- Signals can be described by standard *Mathematica* functions, as well.

```
In[3]:= SignalPlot[
            BesselJ[1, 3 t] +
                ContinuousPulse[0.5, t - 1]/2,
            {t, 0, 3}
        ]
```

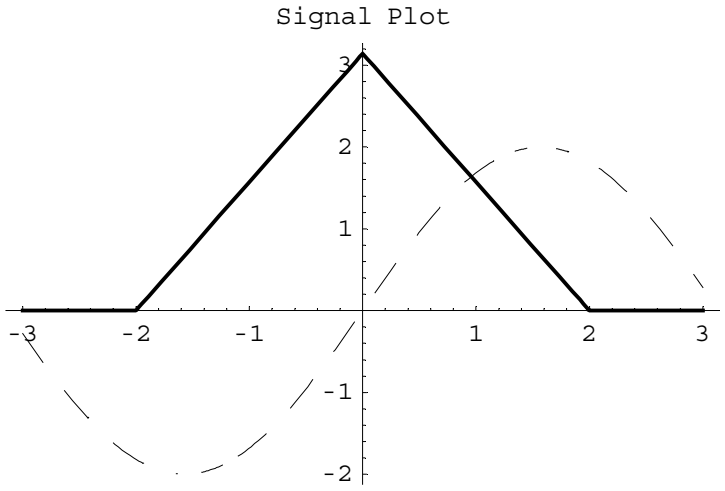



```
Out[3]= - Graphics -
```

While a signal's "time" variable is often restricted to the real domain for the purposes of Signals and Systems (for instance, when creating plots), the entire expression representing the signal may take on complex values. An example of this can be found in the result of transforms such as the Fourier transform.

- A complex-valued function is displayed. The real part is represented by the solid line, while the dashed line is the complex part.

```
In[4]:= SignalPlot[
            FourierTransform[Sinc[t]^2 +
                DiracDelta[t + 1] - DiracDelta[t - 1],
                t, w],
            {w, -3, 3}
        ]
```

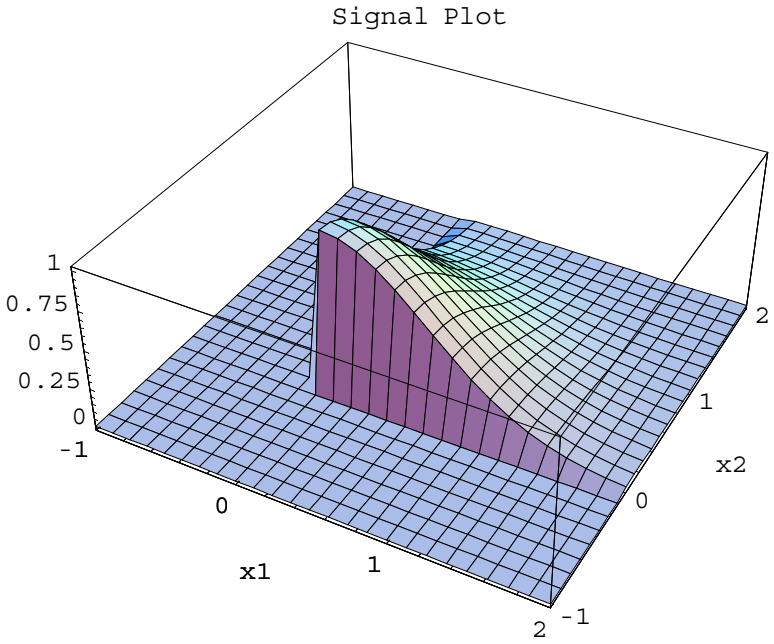



```
Out[4]= - Graphics -
```

Many functions in Signals and Systems can also handle multidimensional signals. Functions with multidimensional equivalents are called by placing the variables in a list in the case of transform functions, or by giving the additional dimensions as extra arguments, for instance, in `UnitStep[`x_1`, `x_2`, ...]`.

- Here is a two-dimensional signal with Gaussian characteristics in the upper-right quadrant.

```
In[5]:= SignalPlot3D[
            Exp[-(x1^2 + x2^2)] UnitStep[x1, x2],
            {x1, -1, 2},
            {x2, -1, 2}
        ]
```




```
Out[5]= - SurfaceGraphics -
```

The unit singularity functions are represented in terms of Dirac delta functions (for singularity functions of nonnegative order) and the unit step function (for singularity functions of negative order). These are particularly useful in representing continuous transforms of polynomials. Powers of these functions will automatically simplify to the function itself when appropriate.

- Here is a list of the representations for some unit singularity functions.

```
In[6]:= Table[UnitFunction[n][t], {n, -2, 2}]

Out[6]= {t UnitStep[t], UnitStep[t],
         DiracDelta[t], DiracDelta'[t], DiracDelta''[t]}
```

Piecewise Representations

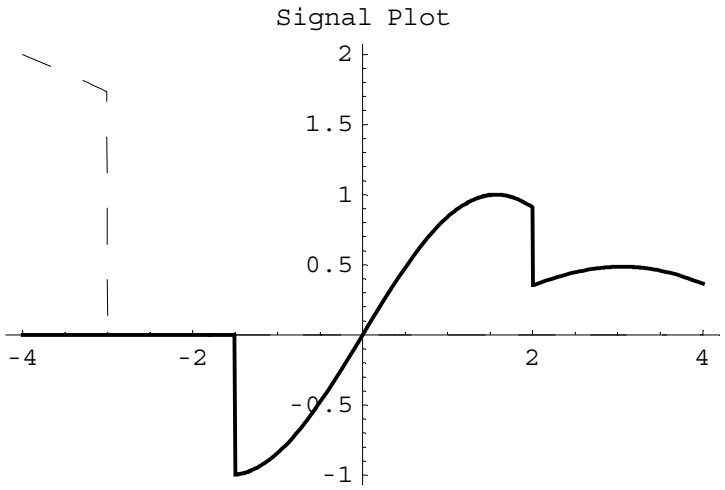
`ContinuousPiecewiseData[`	piecewise representation
`{{`fun_1`,` l_1`,` u_1`},`	of a function, taking on values of
`{`fun_2`,` l_2`,` u_2`},...},` *var*`]`	fun_i in *var* in the interval from l_i to u_i

Representation for piecewise-continuous function.

Piecewise-continuous signals can also be specified, and many Signals and Systems functions can operate on signals given in this form. A piecewise function is given as a list of elements, each of the form `{`*fun*`,` *min*`,` *max*`}` where *min* and *max* give the range over which the signal takes the value given by *fun*. This list is wrapped in a data type called `ContinuousPiecewiseData`, allowing the independent variable to be tracked. The full syntax is thus `ContinuousPiecewiseData[`*list*`,` *var*`]`. Multidimensional signals are not supported with this notation.

- Here is a plot of a function defined in piecewise form. Note that the function may be complex, that it takes the value zero where a segment is not defined, and that a segment may range to `Infinity` (or from `-Infinity`).

```
In[7]:= SignalPlot[
          ContinuousPiecewiseData[
              {{Sqrt[x], -5, -3},
               {Sin[x], -1.5, 2},
               {BesselJ[2, x], 2, Infinity}},
              x
          ],
         {x, -4, 4}
        ]
```




```
Out[7]= - Graphics -
```

`ToContinuousPiecewiseData` [*function*, *var*]	convert *function* into piecewise form in terms of the variable *var*
`Normal`[*piecewise*]	convert the piecewise data type into a sum of continuous step and pulse functions

Converting to and from the continuous piecewise data type.

`ContinuousPiecewiseData` effectively represents a sum of functions multiplied by `ContinuousPulse` functions. As a result, at the boundary of two pieces, the piecewise data type takes the value of the second piece (recall that the leading discontinuity of the `ContinuousPulse` is defined as 1, while the trailing edge is defined as 0). You can

convert between these representations with the functions `ToContinuousPiecewiseData[expr, var]` and `Normal[expr]` (the sum-of-pulses form is the canonical representation).

- Here we convert a function from step form to piecewise form.

```
In[8]:= ToContinuousPiecewiseData[
            a t^2 ContinuousPulse[4, t] + UnitStep[t - 2],
            t
        ]

Out[8]= ContinuousPiecewiseData[{{a t^2, 0, 2}, {1 + a t^2, 2, 4}, {1, 4, ∞}}, t]
```

- The function is converted back into segments determined by the piecewise data structure, not the original step form.

```
In[9]:= Normal[%]

Out[9]= (1 + a t^2) ContinuousPulse_2[-2 + t] +
         a t^2 ContinuousPulse_2[t] + UnitStep[-4 + t]
```

`DeltaArea[a]`	represents a Dirac delta function of area *a* inside a piecewise representation

The `DeltaArea` representation.

A Dirac delta function is an odd beast with infinite value at a point, but finite area under the curve. To retain the information about the area under a delta function, the `DeltaArea` object is introduced. It is used inside the piecewise representation for any function involving Dirac deltas.

- Here is the representation for a function involving Dirac delta functions.

```
In[10]:= ToContinuousPiecewiseData[
             7 DiracDelta[t] + 3 DiracDelta[t + 3] + Sin[t],
             t
         ]

Out[10]= ContinuousPiecewiseData[
          {{DeltaArea[3], -3, -3}, {DeltaArea[7], 0, 0}, {Sin[t], -∞, ∞}}, t]
```

■ 2.2 Discrete Signals

A discrete signal is a signal defined only for integer values of the independent variable(s). Several representations, including explicit sequences of values, formulas, and piecewise formulas are available for use in Signals and Systems. Note that a discrete signal is differentiated from a digital signal in that the dependent variable can take on a continuous range of values, while for a digital signal, the dependent variable must also be discrete. Digital signals are thus a subset of discrete signals, and both are handled identically by Signals and Systems.

Signals as Discrete Data

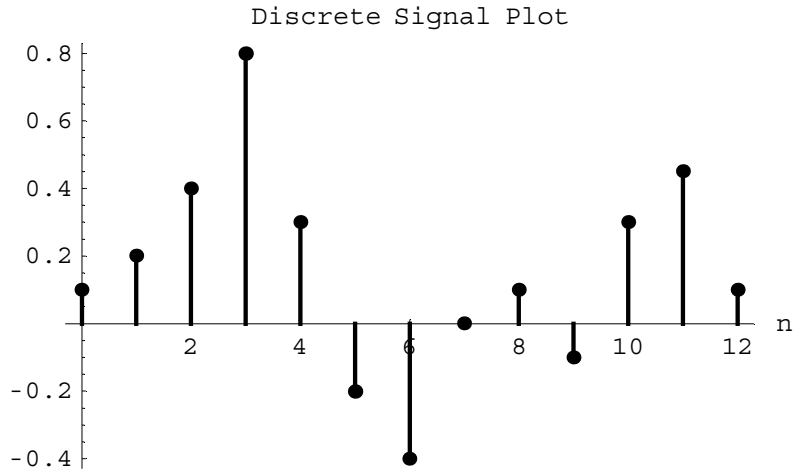
`SampledData[`*list*`,` *variable*`]`	vector of data representing a discrete signal with the first point at zero
`SampledData[`*list*`,` *start*`,` *variable*`]`	data whose initial value is at the index *start*
`SampledData[`*matrix*`, {`v_1`,` v_2`, ...}]`, `SampledData[`*matrix*`,` *startvector*`,` *variables*`]`	multidimensional data sampled on a regular grid, possibly offset from the origin by a starting vector

Data object for discrete data in vector form.

In *Mathematica* we can represent an explicit sequence of data values as a list of numbers, using the form $\{y_1, y_2, y_3, \ldots\}$. For signals, information about the independent variable and the support is also required. We handle this by placing the list in a special data type called `SampledData`. We can display signals of this form very conveniently.

- This is an arbitrary sequence of numbers representing a signal.

```
In[11]:= DiscreteSignalPlot[SampledData[
           {0.1, 0.2, 0.4, 0.8, 0.3, -0.2,
           -0.4, 0, 0.1, -0.1, 0.3, 0.45, 0.1},
          n], {n, 0, 12}]
```




```
Out[11]= - Graphics -
```

As is common in signal processing, we assume that the first item in the list corresponds to a value of 0 for the independent variable when plotting or computing with sequences in Signals and Systems. Note that this is contrary to standard *Mathematica* usage where the first element in a list is given the index 1. You should take care to distinguish between structural manipulations on lists of numbers and computations with a signal expressed as sampled data.

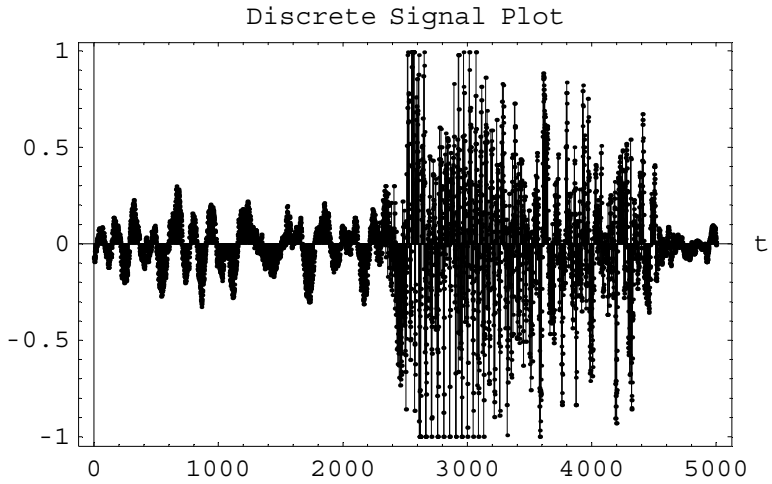
Sequences of data most often correspond to sampled signals, possibly read from a file.

- First we read in a data file of a digitized audio signal.

```
In[12]:= snd = ReadList[ToFileName[{"SignalProcessing", "Documentation",
          "English"},"snd1.dat"], Number];
```

- We now display the first 5000 samples of the signal.

```
In[13]:= DiscreteSignalPlot[
             SampledData[snd, t], {t, 0, 4999},
             Frame -> True
         ]
```

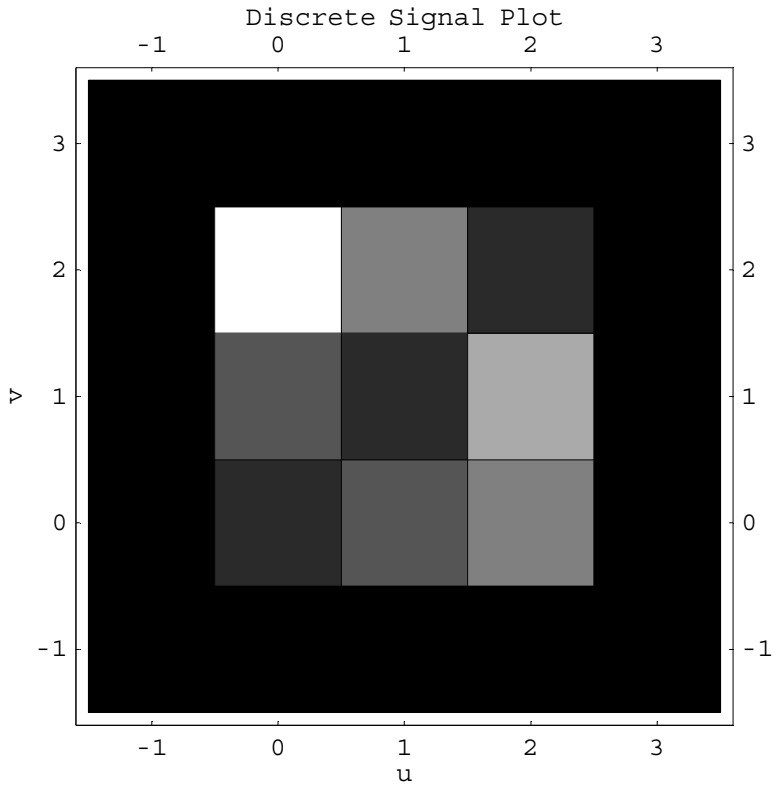



```
Out[13]= - Graphics -
```

Multidimensional data are represented on an array, with the values of the independent variables represented by the locations in the array. Several graphical representations can be made of multidimensional discrete data, including density plots and surfaces where the intersections of the polygons making up the surface correspond to the data points.

- A multidimensional discrete data set can be displayed in a density plot by `DiscreteSignalDensityPlot`. The coloring is that used by `DensityGraphics`, which by default displays the smallest value in black and the largest in white.

```
In[14]:= DiscreteSignalDensityPlot[
           SampledData[
               {{1,2,3},
                {2,1,4},
                {6,3,1}},
               {u, v}
           ],
         {u, -1, 3}, {v, -1, 3}
       ]
```




```
Out[14]= - DensityGraphics -
```

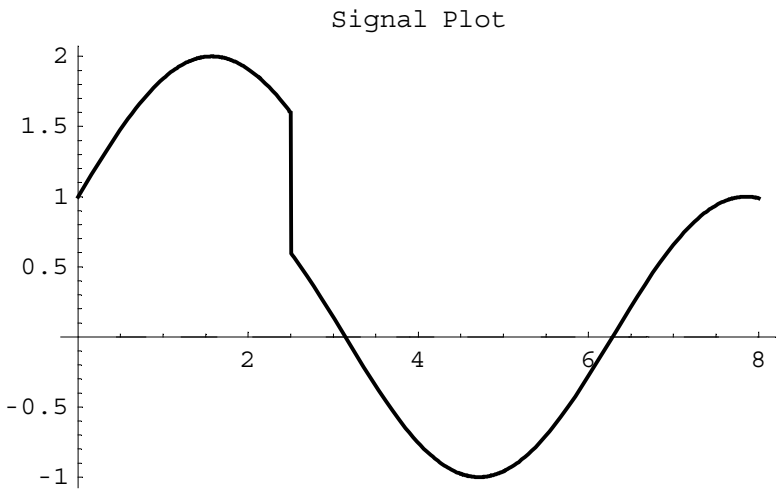
`ToSampledData[` *fun*`,` `{`*var*`,` *start*`,` *stop*`}]`	convert *fun* into a `SampledData` object, with samples stepping by 1 between the integers *start* and *stop* in the variable *var*
`ToSampledData[` *fun*`,` $range_1$ `,` $range_2$ `, … ,` $range_n$ `]`	sample *fun* in *n* dimensions

Creating a `SampledData` object from a function.

It is often useful to work with discrete data generated from a function, rather than working with the function itself (as described in the next section). The function `ToSampledData` can be used to create the data object. It will make the object out of integer values of a variable between a specified starting value and endpoint.

- Here is a continuous signal.

```
In[15]:= SignalPlot[
             Sin[x] + ContinuousPulse[0.4 x],
            {x, 0, 8}
         ]
```




```
Out[15]= - Graphics -
```

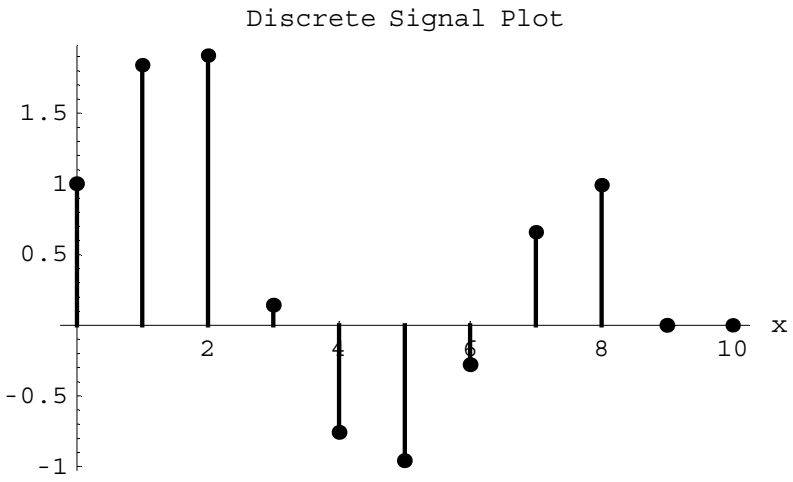
- We have created a SampledData object from the formula.

```
In[16]:= dat = N[ToSampledData[
            Sin[x] + ContinuousPulse[0.4 x],
            {x, 0, 8}
          ]]

Out[16]= SampledData[{1., 1.84147, 1.9093, 0.14112,
           -0.756802, -0.958924, -0.279415, 0.656987, 0.989358}, 0, x]
```

- This is a plot of the constructed data object.

```
In[17]:= DiscreteSignalPlot[
            dat,
            {x, 0, 10}
          ]
```




```
Out[17]= - Graphics -
```

Signals in Formulas

Representation of a sequence in the form of a formula greatly expands the scope of signals that can be expressed. For instance, signals of infinite extent are easily manipulated as formulas.

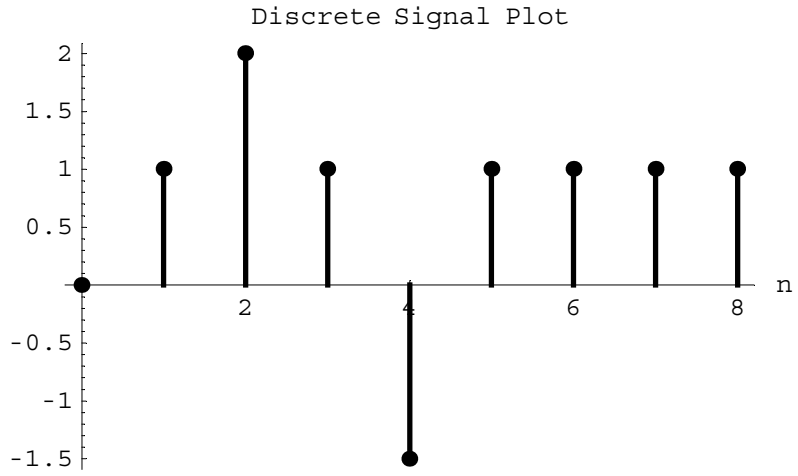
DiscreteDelta[*n*]	a discrete impulse, with value 1 at the origin, and value 0 everywhere else
DiscretePulse[*len*, *n*]	a train of *len* discrete unit impulses, with the first impulse at *n*=0
DiscreteStep[*n*]	a discrete step, which is 0 at *n* less than zero, and 1 at *n* greater than or equal to zero

Some useful discrete signals.

As with analog signals, several functions are provided for representation of discrete signals in the form of formulas. The DiscreteDelta can also be called DiscreteImpulse, to parallel the analog ContinuousImpulse.

- Here is a discrete signal superposing a step and two impulses.

```
In[18]:= DiscreteSignalPlot[DiscreteStep[n - 1] +
           DiscreteDelta[n - 2] -
           2.5 DiscreteDelta[n - 4],
         {n, 0, 8}]
```




```
Out[18]= - Graphics -
```

`UpsampledLattice[`*m*`,` *value*`,` *fun*`,` *variable*`]`	an upsampled signal, with every m^{th} sample taking on the current value of *fun*, while other samples take on *value*

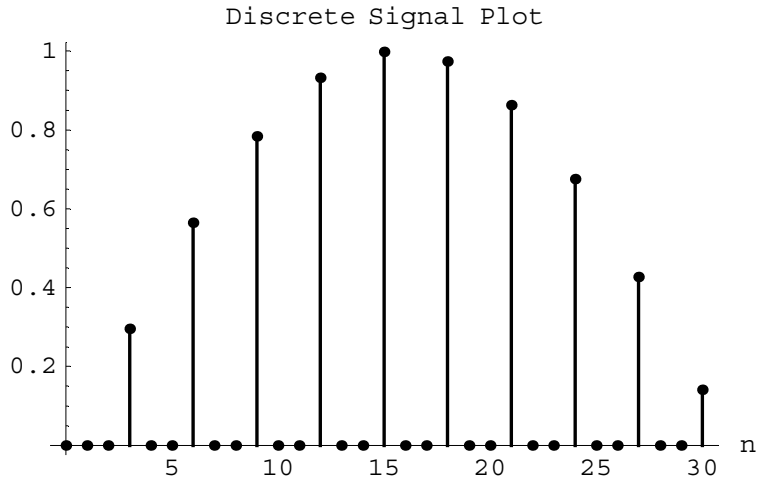
Signal generated as a result of signal processing operations.

Some signal processing operations result in signals that do not have a convenient closed-form representation in terms of traditional functions. It is useful to have an object representing this sort of signal. The `UpsampledLattice` function is one such signal, representing a discrete signal that has been upsampled by the insertion of $n-1$ instances of some value in between each sample. Note that the function *fun* in the specification is one that has already been transformed by an upsampling operation, not the function before upsampling.

Multidimensional upsampling is also represented by `UpsampledLattice`, where *m* is a resampling matrix, and *n* is a vector of variables. Then where $m^{-1}\, n$ falls on the integer lattice, it takes on the value of the function at *n*. `UpsampledLattice` takes on zero for all other points on the integer lattice.

- This is a sinusoidal signal that has been upsampled by a factor of three, with zeros inserted.

```
In[19]:= DiscreteSignalPlot[
             UpsampledLattice[3, 0, Sin[n/10], n],
             {n, 0, 30}
         ]
```




```
Out[19]= - Graphics -
```

Piecewise Signals

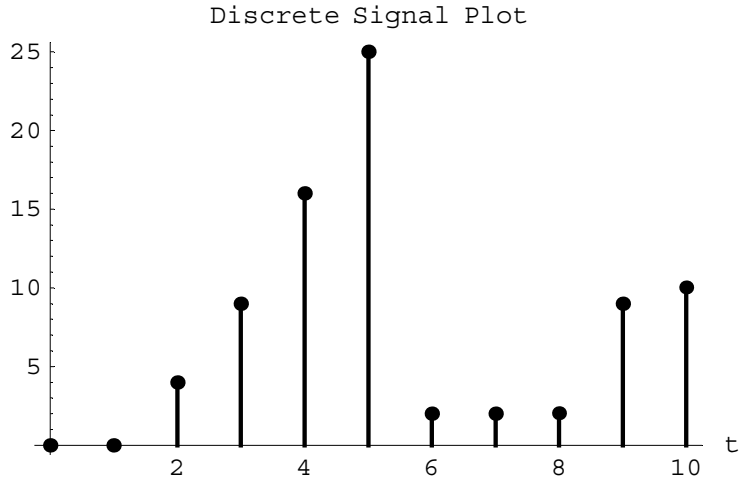
`DiscretePiecewiseData[{{{`fun_1`,` l_1`,` u_1`},` `{`fun_2`,` l_2`,` u_2`},` `…},` *var*`]`	piecewise representation of a function, taking on values of fun_i in *var* in the interval from l_i to u_i

Representation for a piecewise-defined discrete function.

As with continuous signals, discrete signals can also be described in a piecewise form. The basic syntax is identical (with the exchange of `DiscretePiecewiseData` for `ContinuousPiecewiseData`), but simplified by nonoverlapping integer endpoints for the intervals.

- Here is a plot of a discrete signal with a parabolic, a constant, and a linear segment. Note that the signal takes the value zero outside of the defined intervals.

```
In[20]:= DiscreteSignalPlot[
             DiscretePiecewiseData[
                 {{t^2, 2, 5},
                  {2, 6, 8},
                  {t, 9, 10}},
                 t
             ],
            {t, 0, 10}
         ]
```




```
Out[20]= - Graphics -
```

`ToDiscretePiecewiseData[`*signal*`, `*var*`]`	convert a discrete signal to its piecewise form in terms of the variable *var*
`Normal[`*piecewise*`]`	convert a piecewise signal to a formula in terms of step and pulse functions

Conversion to and from the piecewise data type.

- Here is a discrete signal written as a formula converted to piecewise form.

```
In[21]:= ToDiscretePiecewiseData[
           2 t DiscretePulse[4, t - 2] +
             Exp[t] UnitStep[t - 8],
           t
         ]

Out[21]= DiscretePiecewiseData[{{2 t, 2, 5}, {ℯ^t, 8, ∞}}, t]
```

- We can convert it back into a formula.

```
In[22]:= Normal[%]

Out[22]= 2 t DiscretePulse4[-2 + t] + ℯ^t DiscreteStep[-8 + t]
```

■ 2.3 Windows

Windows are special signals of finite extent designed for modifying other signals to enhance various forms of analysis, such as spectral analysis. Signals and Systems provides a variety of continuous and discrete windows that can be incorporated into your computations. Of course, you can easily define a window as any weighting function multiplied by a pulse, but the window objects specified here have the additional advantage of allowing the functions to retain their identities as windows, which can assist you in understanding the computations later.

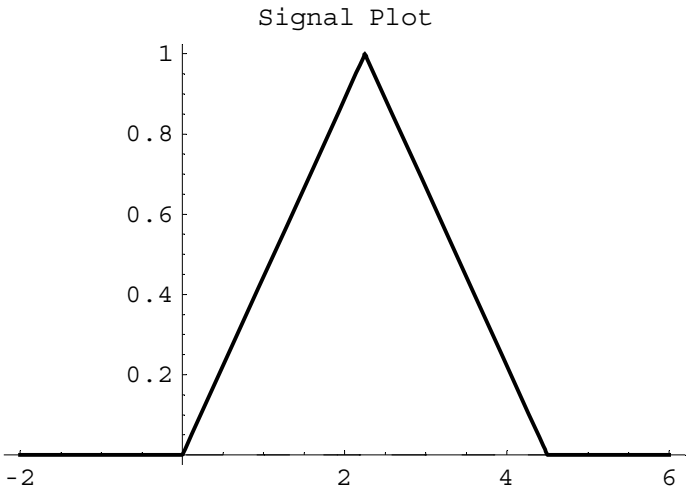
`ContinuousWindow[`*type*`,` *length*`,` *var*`]`	a continuous window in the variable *var*
`DiscreteWindow[`*type*`,` *length*`,` *var*`]`	a discrete window in the variable *var*

Specifications of windows.

These window objects can be manipulated much as you would any other signal.

- A window can be plotted. Here we have a triangular window.

```
In[23]:= SignalPlot[
            ContinuousWindow[Triangular, 4.5, t],
            {t, -2, 6}
         ]
```

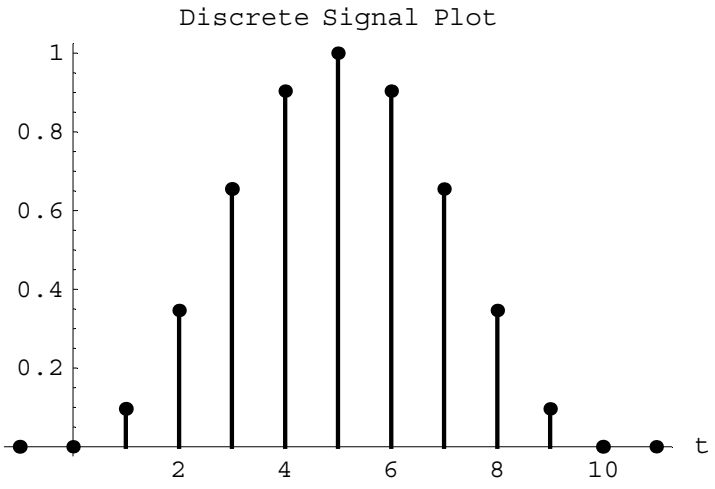



```
Out[23]= - Graphics -
```

- Here is a discrete window. Note that some window types have parameters controlling their shape. This parameterized Hanning window corresponds to the usual nonparametric definition when the parameter's value

  is $\frac{1}{2}$.

```
In[24]:= DiscreteSignalPlot[
             DiscreteWindow[Hanning[1/2], 10, t],
             {t, -1, 11}
         ]
```




```
Out[24]= - Graphics -
```

`Rectangular`, `Dirichlet`	standard rectangular window
`Triangular`	triangular window
`Bartlett`, `Fejer`	alternate names for the triangular window
`Sinusoidal[`*power*`]`	a sinusoidal window of the form `Sin[`*var* π`/`*length*`]^`*power*
`Hanning[`*alpha*`]`	the Hann window defined as *alpha* `-(1 -` *alpha*`) Cos[2` *var* π`/`*length*`]`
`Hamming`	equivalent to `Hanning[25/46]`
`BlackmanHarris[` a_0 , a_1 , a_2 `]`	the three-parameter Blackman-Harris window, which is a sum of two harmonic functions (weighted by the parameters) and the constant 1
`BlackmanHarris[` a_0 , a_1 , a_2 , a_3 `]`	the four-parameter Blackman-Harris window
`Blackman`	the exact Blackman window, equivalent to `BlackmanHarris[7938/18608, 9240/18608, 1430/18608]`

Some window types implemented in Signals and Systems.

- Some windows are implemented in terms of other windows, or are special cases of other windows.

```
In[25]:= ContinuousWindow[Blackman, 30, t]
```

$$\text{Out[25]= ContinuousWindow}\left[\text{BlackmanHarris}\left[\frac{3969}{9304}, \frac{1155}{2326}, \frac{715}{9304}, 0\right], 30, t\right]$$

- The formula for each window can be found by application of `Normal`.

```
In[26]:= Normal[%]
```

$$\text{Out[26]= ContinuousPulse}_{30}[t]\left(\frac{3969}{9304} - \frac{1155\,\text{Cos}\left[\frac{\pi t}{15}\right]}{2326} + \frac{715\,\text{Cos}\left[\frac{2\pi t}{15}\right]}{9304}\right)$$

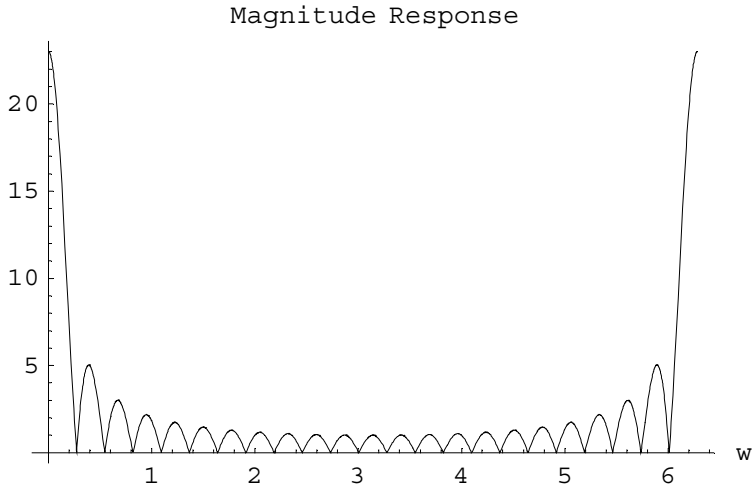
DiscreteTimeFourierWindow[*type*, *length*, *var*]	the discrete-time Fourier transform of a discrete window
DiscreteFourierWindow[*type*, *length*, *var*]	the discrete Fourier transform of a discrete window

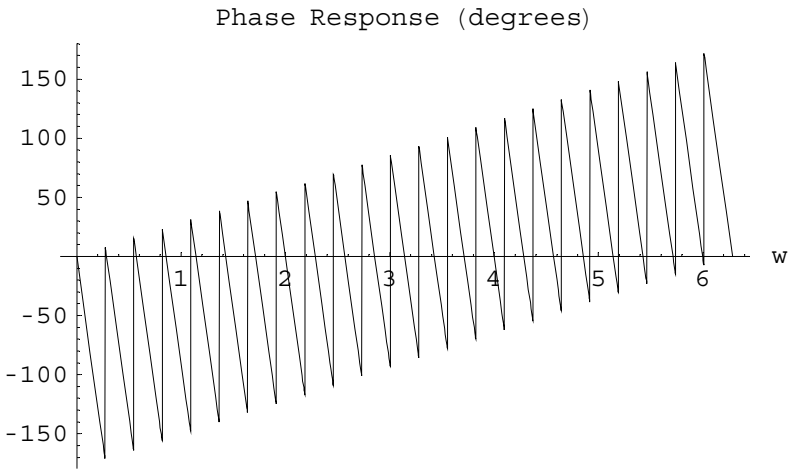
Representations of transforms of discrete windows for frequency analysis.

Closed forms for the transforms of many types of windows are well known. The routines allow abstract specification of the discrete and discrete-time Fourier transforms of discrete windows for manipulation of convolution and transform operations, for instance, as well as easy analysis of the windows themselves.

- The rectangular window has particularly high sidelobes. These more than offset the ease of computation with this window in many applications.

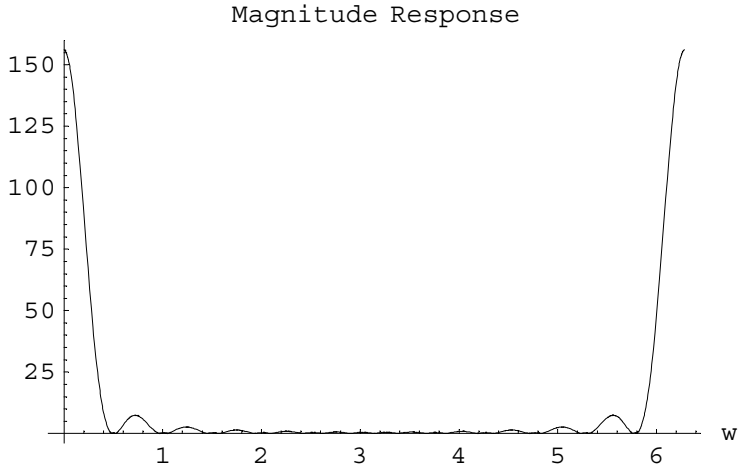
```
In[27]:= MagnitudePhasePlot[
             DiscreteTimeFourierWindow[
                 Rectangular, 22, w
             ],
             {w, 0, 2 Pi},
             PlotPoints -> 100
         ];
```

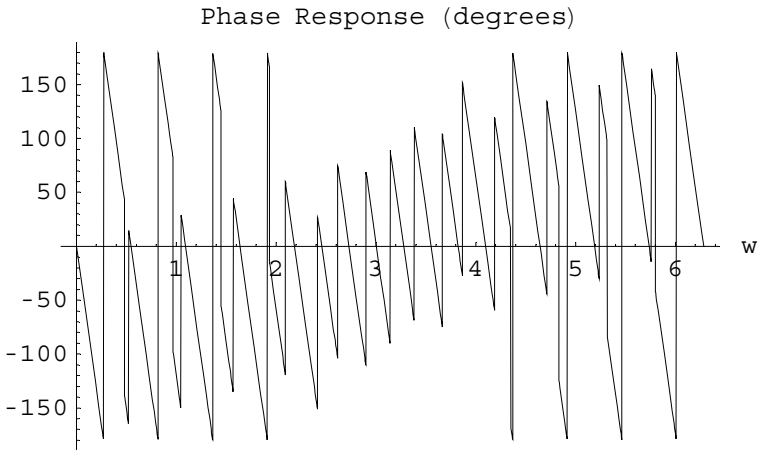




- Here is the frequency response of a triangular window, which has noticeably better sidelobe characteristics.

```
In[28]:= MagnitudePhasePlot[
             DiscreteTimeFourierWindow[
                 Triangular, 22, w
             ],
             {w, 0, 2 Pi},
             PlotPoints -> 100
         ];
```





■ 2.4 Systems as Operators

A system essentially transforms one signal into another. Strictly speaking, standard mathematical functions such as those described in previous sections are systems, but it is convenient to think of functions as describing an initial signal. Systems can be manipulated as parameterized transformations, and can in principle be rewritten in useful ways that represent different hardware implementations.

We can take this behavior and implement it in *Mathematica* using an operator notation, in which the parameterized system description is written as a head that takes the signal (or in some cases, signals) as the argument, as in `system[`*param*$_1$`, `*param*$_2$`, … ][`*signal*`]`.

- This represents a system that downsamples by 2, acting on a periodic signal.

```
In[29]:= Downsample[2, n][ Cos[Pi n/10] ]
```

$$\text{Out[29]= Downsample}_{2,n}\left[\text{Cos}\left[\frac{n\,\pi}{10}\right]\right]$$

`EvaluateOperators[` *expr*`]`	evaluate the response of signal processing operators in *expr* to their input signals

The `EvaluateOperators` function.

- To find the output generated by passing the signal through the system, we apply the `EvaluateOperators` function.

```
In[30]:= EvaluateOperators[%]

Out[30]= Cos[n π / 5]
```

Note that the standard operator syntax in Signals and Systems is *nonevaluating*. This means that a system description is established in a form that allows easy cascading of operators. It also makes system rewriting to perform design tasks convenient. To evaluate a system applied to a signal, use `EvaluateOperators` to force the response of the systems to be generated. If you wish to immediately evaluate the system, you can usually use a functional form, where the signal becomes the first argument of the system description.

- A system applied to a signal in functional form immediately returns the output signal.

```
In[31]:= Downsample[Cos[Pi n/10], 2, n]

Out[31]= Cos[n π / 5]
```

Signals and Systems provides many common signal processing operators. Some operators, because of equivalent definitions, can be used equally well on continuous or discrete signals. Other operators process only one of these types of signals.

`Shift[`*quantity*`, `*var*`][`*signal*`]`	shift *signal* by *quantity* in the variable *var*
`ReverseSignal[`*var*`][`*signal*`]`	reverse the sense of the independent variable *var* in the signal
`Periodic[`*period*`, `*var*`][`*signal*`]`	make *signal* periodic by creating an infinite sum of the signal shifted left and right by integer multiples of the period

Signal processing operators that act on both continuous and discrete signals.

The `Shift` operator is equivalent to adding an offset to the independent variable in the signal. `ReverseSignal` is equivalent to inverting the sense of the independent variable (multiplying it by -1). The `Periodic` operator is generally applied to signals of limited duration, as it is formed of an infinite summation. These summations will often not converge

for infinite signals. You can use these operators in multidimensional form by giving the parameters as lists (except for the period, which can be either a list or a matrix), where the list has a number of elements equivalent to the dimensionality of the operator.

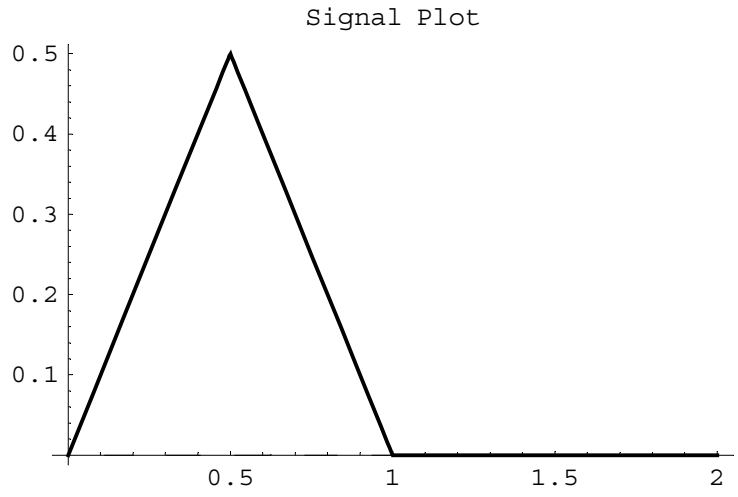
- This represents a triangular pulse.

```
In[32]:= tri = t ContinuousPulse[0.5, t] +
             (1 - t) ContinuousPulse[0.5, t - 0.5]
```

```
Out[32]= (1 - t) ContinuousPulse0.5[-0.5 + t] + t ContinuousPulse0.5[t]
```

- Here is what the pulse looks like.

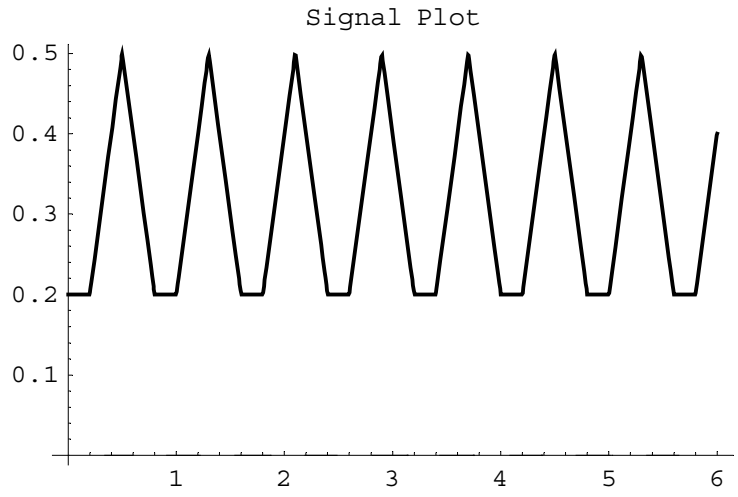
```
In[33]:= SignalPlot[tri, {t, 0, 2}]
```




```
Out[33]= - Graphics -
```

- We can make the pulse periodic, with a period of 0.8. Note the effects of aliasing.

```
In[34]:= SignalPlot[Periodic[0.8, t][tri],
           {t, 0, 6}]
```




```
Out[34]= - Graphics -
```

- Applying `EvaluateOperators` creates the signal resulting from the application of the operators, which consists of shifting and reversing the signal in this example.

```
In[35]:= EvaluateOperators[
             ReverseSignal[t][Shift[4, t][t^2]]
         ]
```

```
Out[35]= (-4 - t)^2
```

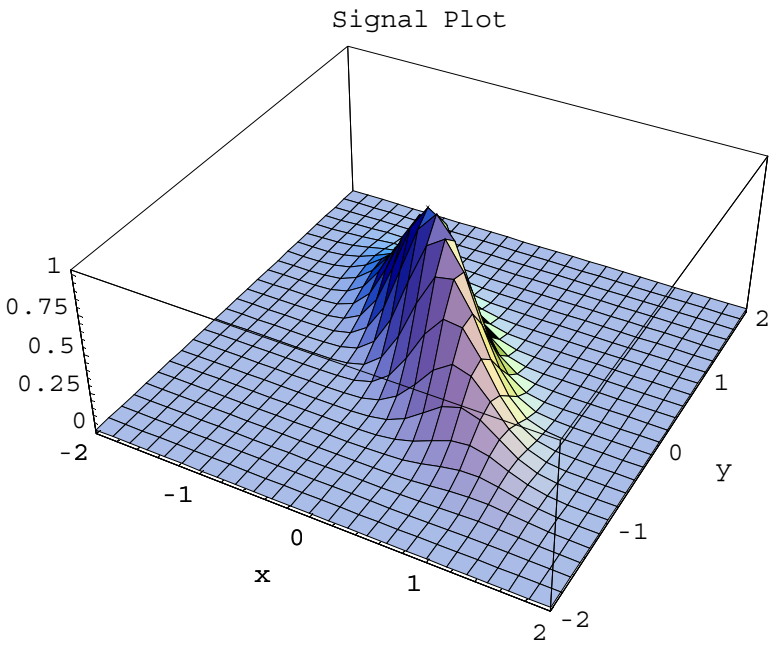
`RotateSignal[` *angle*`, {`v_1`, `v_2`}][`*signal*`]`	rotate *signal* in the plane determined by variables v_1 and v_2 by *angle* in radians
`ScaleSignal[`*factor*`, `*var*`][`*signal*`]`	compress *signal* by *factor* in the variable *var*
`AliasSignal[`*m*`, `*omega*`][`*signal*`]`	alias a continuous frequency spectrum given by *signal* by replicating it every `2 Pi/`*m* in *omega*

Signal processing operators for continuous signals.

`RotateSignal` is used with multidimensional signals. It rotates the signal in the plane corresponding to the two variables. The rotation occurs around the origin. The angle of the rotation is in the direction given by rotating from the axis specified by the second variable toward the first. The `ScaleSignal` operator effectively multiplies the specified variable by a constant factor. Both `ScaleSignal` and `AliasSignal` can also be used in multidimensional form.

- This is a Gaussian curve that has been scaled and rotated.

```
In[36]:= SignalPlot3D[
             RotateSignal[Pi/4, {x, y}][
                 ScaleSignal[3, y][
                     Exp[-(x^2 + y^2)]
                 ]
             ],
             {x, -2, 2}, {y, -2, 2},
             PlotPoints -> 25
         ]
```




```
Out[36]= - SurfaceGraphics -
```

`AliasSignal` is always rewritten in terms of the `Periodic` operator. It is shorthand for the aliasing of a signal representing a continuous frequency spectrum.

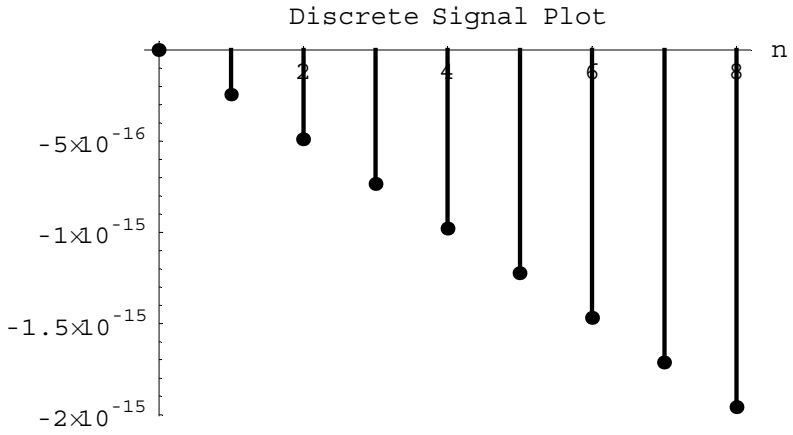
`Upsample[`*k*`, `*var*`][`*signal*`]`	upsample *signal* in *var*, inserting *k* zeros between every sample
`Downsample[`*m*`, `*var*`][`*signal*`]`	downsample *signal* in *var* by taking every m^{th} element, starting with the 0^{th} element
`CircularShift[`*index*`, `*length*`, `*var*`][`*signal*`]`	rotate *signal* by *index* elements to the right in the variable *var*, while treating the signal as a repeating sequence of given length
`Difference[`*order*`, `*var*`][`*signal*`]`	compute the backward difference of the signal to the specified order in the variable *var*
`Interleave[`*var*`][`$signal_1$`, `$signal_2$`, ...]`	interleave the samples of the signals in terms of the variable *var*

Signal processing operators that act on discrete signals.

The upsampling and downsampling functions operate on discrete signals. When the signal is multidimensional, the sampling factor should be an integer sampling matrix, and the variable should be replaced by a list of variables.

- This is an example of how downsampling can cause aliasing problems. Information that the signal is sinusoidal has been lost.

```
In[37]:= DiscreteSignalPlot[
             Downsample[4, n][Sin[Pi/2 n]],
             {n, 0, 8}
         ]
```

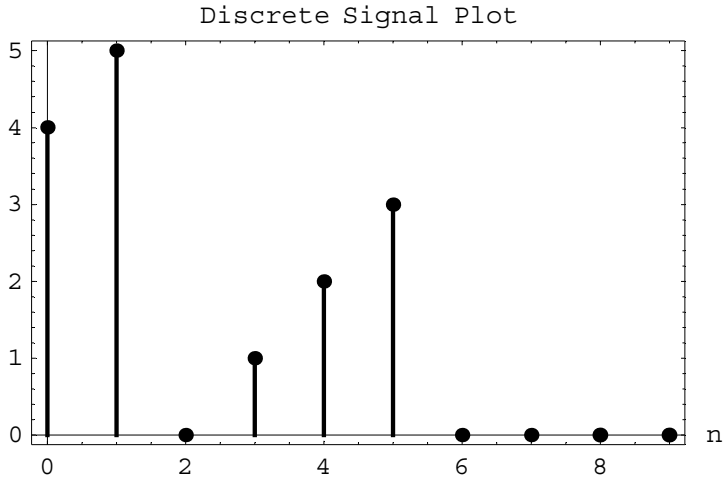



```
Out[37]= - Graphics -
```

`CircularShift` acts as a ring buffer of given length with a specified rotation value, so each sample at position *n* is shifted by the formula `Mod[`*n* `-` *value*`,` *length*`]`.

■ Here is a ramp that has had a circular shift applied to it.

```
In[38]:= DiscreteSignalPlot[
             CircularShift[2, 6, n][n],
             {n, 0, 9},
             Frame -> True
         ]
```




```
Out[38]= - Graphics -
```

■ Here is the third-order backward difference of the signal `Sin[n]`.

```
In[39]:= EvaluateOperators[Difference[3, n][Sin[n]]]

Out[39]= 3 Sin[1 - n] - 3 Sin[2 - n] + Sin[3 - n] + Sin[n]
```

The `Interleave` operator is unique in that it acts on multiple signals, multiplexing them by effectively taking one sample from each input signal in sequence. (This is performed algebraically, so the signals can be arbitrary discrete formulas.) The output may contain `UpsampledLattice` objects, since the general procedure is to upsample each signal by the total number of signals, shift each successive signal by its position in the sequence, and add together the resulting signals.

- Interleave the two periodic signals. Note that the result is in terms of `UpsampledLattice` objects.

```
In[40]:= EvaluateOperators[
           Interleave[n][Sin[n], Cos[n]]
         ]
```

$$\text{Out[40]= UpsampledLattice}\left[2, 0, \text{Cos}\left[\frac{1}{2}\ (-1+n)\right], -1+n\right] + \text{UpsampledLattice}\left[2, 0, \text{Sin}\left[\frac{n}{2}\right], n\right]$$

The various filter objects and transforms (Laplace, Fourier, Z, etc.) defined in Signals and Systems also have operator forms. These are documented further in Chapters 4 and 5.

`Summation[`*index*`, `*min*`, `*max*`, `*step*`][`*signal*`]`	represent a summation over a signal parameter

Operator-like objects in Signals and Systems.

Signals and Systems contains some other common operator-like objects, though they do not have the same behavior as the preceding operators. It is useful for various signal representation and signal processing tasks to keep a summation in an unevaluated form. The operator `Summation[`*index*`, `*min*`, `*max*`]` represents the summation of a signal incrementing *index* from *min* to *max*. It does not transform a signal variable the way other operator functions do, but is instead a compact notation for a complicated signal expression.

- Here is a summation, perhaps representing the frequency response of a filter. Note that if this were written with `Sum`, it would immediately evaluate because of the exact numeric limits.

```
In[41]:= expr = Summation[m, 1, 3][a[m] Exp[I d[m] w]]
```

$$\text{Out[41]= Summation}_{m=1}^{3}\left[e^{i\,w\,d[m]}\,a[m]\right]$$

- This is the evaluated summation. The result is not a signal in `m`, as it would be with other operators in Signals and Systems.

```
In[42]:= EvaluateOperators[expr]
```

$$\text{Out[42]= } e^{i\,w\,d[1]}\,a[1] + e^{i\,w\,d[2]}\,a[2] + e^{i\,w\,d[3]}\,a[3]$$

`PolyphaseUpsample[` *up*`,` *filter*`,` *var*`][`*signal*`]`	efficient structure for filtering by the specified FIR filter, then upsampling by factor *up*
`PolyphaseDownsample[`*down*`,` *filter*`,` *var*`][`*signal*`]`	efficient structure for downsampling by the factor *down*, then filtering by the specified FIR filter
`PolyphaseResample[` *up*`,` *filter*`,` *down*`,` *var*`][`*signal*`]`	efficient structure for upsampling by factor *up*, passing through the given FIR filter, then downsampling by the factor *down*

Representations of polyphase operations.

Signals and Systems also has the capability to represent polyphase operations, although at this time they are nonevaluating. Currently, these representations are solely for use when exporting to `Ptolemy` (see Section 7.2 for more information). These polyphase systems use an operator syntax in the expectation that they will be evaluating in a future version of the routines.

■ 2.5 Modified Standard Functions

The behavior of several standard functions has been modified somewhat for Signals and Systems. For the most part, these are extensions to existing functions so that they may operate properly on objects introduced in the application. However, a few other functions were adjusted to use assumptions common to signal processing and related fields.

■ Both `Det` and `Dot` have been extended to work with scalars. This is because of the parallels between one-dimensional and multidimensional resampling operations.

■ For typical signals and systems manipulations in the continuous domain, we can make an assumption of analaticity. To improve this behavior, the `Analytic` option of `Limit` has been set to `True` by default, as opposed to the usual `False` value.

■ Calculus functions such as `Integrate`, `Derivative`, and `Limit` have been extended to cover many of the new functions introduced in Signals and Systems.

■ The standard `LaplaceTransform`, `FourierTransform`, and `ZTransform` functions and their inverses have been overridden by the Signals and Systems versions. You can access the standard system versions via `System`LaplaceTransform`, etc.

■ `UnitStep` and `DiracDelta` are system functions. Signals and Systems removes the `Listable` attribute from these functions for backward compatibility with earlier versions of Signals and Systems.

■ Improved TeX formatting has been added to several standard operators, such as `Re` and `Im`.