# Chapter 4. SDF Domain

*Authors:* *Joseph T. Buck*
*Michael J. Chen*
*Paul Haskell*
*Alan Kamas*
*Alireza Khazeni*
*Edward A. Lee*
*David G. Messerschmitt*
*Soonhoi Ha*
*Shuvra Bhattacharyya*

*Other Contributors:* *Brian L. Evans*
*Asawaree Kalavade*
*Thomas M. Parks*
*Gregory S. Walter*

## 4.1 Introduction

Synchronous dataflow (SDF) is a data-driven, statically scheduled domain in Ptolemy. It is a direct implementation of the techniques given in [Lee87a] and [Lee87b]. "Data-driven" means that the availability of **`Particle`**s at the inputs of a star enables it. Stars without any inputs are always enabled. "Statically scheduled" means that the firing order of the stars is determined once, during the start-up phase. The firing order will be periodic. The SDF domain in Ptolemy is one of the most mature, with a large library of stars and demo programs. It is a simulation domain, but the model of computation is the same as that used in most of the code generation domains. A number of different schedulers, including parallelizing schedulers, have been developed for this model of computation.

### 4.1.1 Basic dataflow terminology

SDF is a special case of the dataflow model introduced by Dennis [Den75]. It is equivalent to the *computation graph* model of Karp and Miller [Kar66]. In the terminology of the dataflow literature, stars are called *actors*. An invocation of the **`go()`** method of a star is called a *firing*. Particles are called *tokens*.

When an actor fires, it consumes some number of tokens from its input arcs, and produces some number of output tokens. In synchronous dataflow, these numbers remain constant throughout the execution of the system. It is for this reason that this model of computation is suitable for synchronous signal processing systems, but not for asynchronous systems. The fact that the firing pattern is determined statically is both a strength and a weakness of this domain. It means that long runs can be very efficient, a fact that is heavily exploited in the code generation domains. But it also means that data-dependent flow of control is not allowed. This would require dynamically changing firing patterns. The Dynamic Dataflow (DDF) and Boolean Dataflow (BDF) domains were developed to support this.

### 4.1.2 Balancing production and consumption of tokens

Each porthole of each SDF star has an attribute that specifies the number of particles consumed (for input ports) or the number of particles produced (for output ports). When you connect two portholes with an arc, the number of particles produced on the arc by the source star may not be the same as the number of particles consumed from that arc by the destination star. To maintain a balanced system, the scheduler must fire the source and destination stars with different frequency.

Consider a simple connection between three blocks, as shown in figure 4-1. The symbols adjacent to the portholes, such as $N_{A1}$ represent the number of particles consumed or produced by that porthole when the star fires. For many signal processing stars, these numbers are simply one, indicating that only a single sample is consumed or produced when the star fires. But there are three basic circumstances in which these numbers differ from one:

- Vector processing in the SDF domain can be accomplished by consuming and producing multiple tokens on a single firing. For example, a star that computes a fast Fourier transform (FFT) will typically consume and produce $2^M$ samples when it fires, where $M$ is some integer. Examples of vector processing stars that work this way are **`FFTCx`**, **`Average`**, **`Burg`**, and **`LevDur`**. This behavior is quite different from the matrix stars, which operate on particles where each individual particle represents a matrix.
- In multirate signal processing systems, a star may consume $M$ samples and produce $N$, thus achieving a sampling rate conversion of $N/M$. For example, the **`FIR`** and **`FIRCx`** stars optionally perform such a sampling rate conversion, and with an appropriate choice of filter coefficients, can interpolate between samples. Other stars that perform sample rate conversion include **`UpSample`**, **`DownSample`**, and **`Chop`**.
- Multiple signals can be merged using stars such as **`Commutator`** or a single signal can be split into subsignals at a lower sample rate using the **`Distributor`** star.

To be able to handle these circumstances, the scheduler first associates a simple balance equation with each connection in the graph. For the graph in figure 4-1, the balance equations are

$$
\begin{aligned}
r_A N_{A1} &= r_C N_{C1} \\
r_A N_{A2} &= r_B N_{B1} \\
r_B N_{B2} &= r_C N_{C2}
\end{aligned}
$$

This is a set of three simultaneous equations in three unknowns. The unknowns, $r_A$, $r_B$, and $r_C$ are the *repetitions* of each actor that are required to maintain balance on each arc. The first task of the scheduler is to find the smallest non-zero integer solution for these repetitions. It is proven in [Lee87a] that such a solution exists and is unique for every SDF graph that is "con-

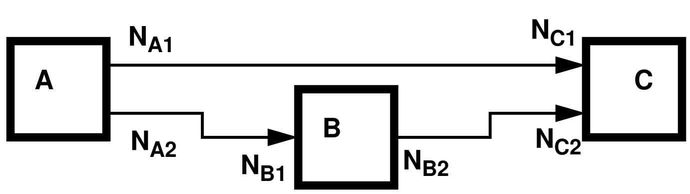



**FIGURE 4-1:** A simple connection of SDF stars, used to illustrate the use of balance equations in constructing a schedule.

sistent," as defined below.

### 4.1.3 Iterations in SDF

When running an SDF system under the graphical interface, you will have the opportunity to specify "when to stop." Since the SDF domain has no notion of time, this is not given in units of time. Instead, it is given in units of SDF iterations. At each SDF iteration, each star is fired the minimum number of times to satisfy the balance equations.

Suppose for example that star B in figure 4-1 is an **`FFTCx`** star with its parameters set so that it will consume 128 samples and produce 128 samples. Suppose further that star A produces exactly one sample on each output, and star C consumes one sample from each input. In summary,

$$N_{A1} = N_{A2} = N_{C1} = N_{C2} = 1$$
$$N_{B1} = N_{B2} = 128.$$

The balance equations become

$$r_A = r_C$$
$$r_A = 128 r_B$$
$$128 r_B = r_C.$$

The smallest integer solution is

$$r_A = r_C = 128$$
$$r_B = 1.$$

Hence, each iteration of the system includes one firing of the **`FFTCx`** star and 128 firings each of stars A and B.

### 4.1.4 Inconsistency

It is not always possible to solve the balance equations. Suppose that in figure 4-1 we have

$$N_{A1} = N_{A2} = N_{C1} = N_{C2} = N_{B1} = 1$$
$$N_{B2} = 2.$$

In this case, the balance equations have no non-zero solution. The problem with this system is that there is no sequence of firings that can be repeated indefinitely with bounded memory. If we fire A,B,C in sequence, a single token will be left over on the arc between B and C. If we repeat this sequence, two tokens will be left over. Such a system is said to be *inconsistent*, and is flagged as an error. The SDF scheduler will refuse to run it. If you must run such a system, change the domain of your graph to the DDF domain.

### 4.1.5 Delays

Delays are indicated in Pigi by small green diamonds that are placed on an arc. Most of the standard palettes of stars have the delay icon at the upper left. The delay has a single parameter, the number of samples of delay to be introduced. In the SDF domain, a delay with parameter equal to one is simply an initial particle on an arc. This initial particle may enable a star, assuming that the destination star for the delay arc requires one particle in order to fire. Hence, to avoid deadlock, all feedback loops much have delays. The SDF scheduler will flag the error if it finds a loop with no delays. There is currently no mechanism for setting the value of the initial particle. For most particle types, the initial value will be zero. For particles which hold matrices, the initial value is an empty Envelope, which must be checked for by stars which work on matrix inputs.

## 4.2 An overview of SDF stars

The "open-palette" command in pigi ("O") will open a checkbox window that you can use to open the standard palettes in all of the installed domains. For the SDF domain, the star library is large enough that it has been divided into sub-palettes. The top-level palette is shown in figure 4-2.

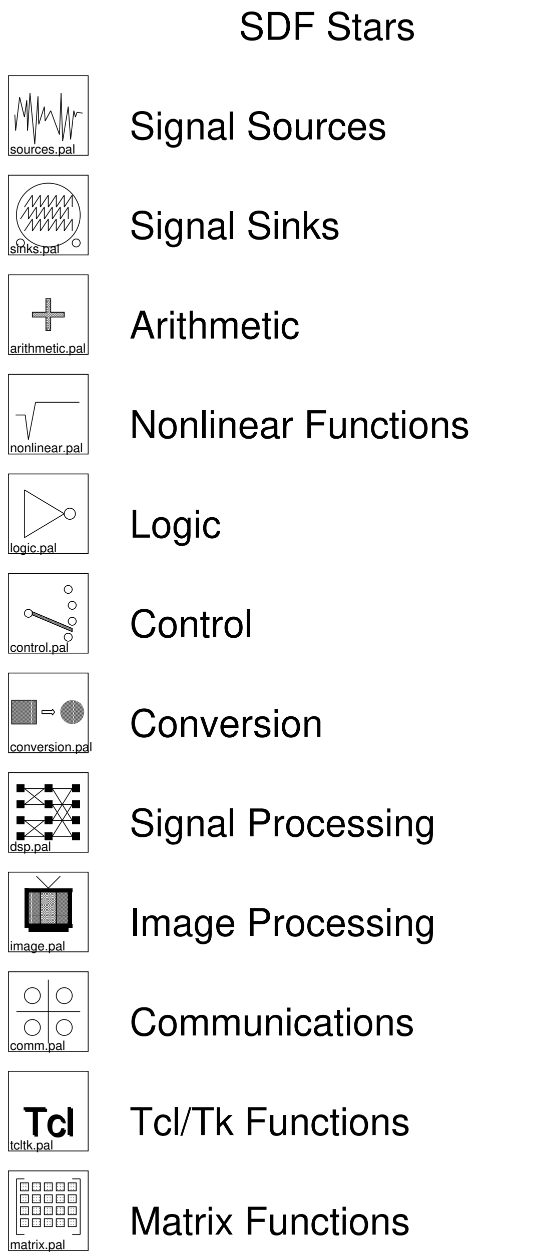



**FIGURE 4-2:** The top-level palette for accessing the library of SDF stars.

The "sources" palette contains signal generators of various types. The "sinks" palette contains various stars that display signals in different ways or write the value of signal samples out to files. The "arithmetic" palette contains basic adders, multipliers, subtracters, and integrators for all the standard scalar data types (floating point, complex, fixed-point, and fixed). The "nonlinear" palette contains stars that compute transcendental functions, such as logarithms, cosine, sine, as well a general table lookup stars. The "control" palette contains stars that manipulate the flow of tokens, such as forks, commutators, and distributors. The "conversion" palette contains stars that explicitly accomplish type conversion. The "dsp" palette contains various signal processing functions such as fixed and adaptive filters of various types and spectral estimation functions. The "image" palette contains stars for image and video signal processing. The "comm" palette contains stars that are specific to digital communications functions, such as pulse shapers, speech coders, and QAM encoders. The "Tcl" palette contains stars that interface to the Tcl interpreter or the associated X window toolkit Tk. The "matrix" palette contains matrix operators such as matrix addition and multiplication. More complex stars that use matrix operations internally can be found in other palettes, such as the singular value decomposition and Kalman filters in the "dsp' palette. Each palette is summarized in more detail below, with the stars listed as they appear in the palette from left to right then top to bottom. For complete documentation on the stars, refer to the *Star Atlas* volume of *The Almagest*.

In the listing below, whenever data types are not mentioned, double-precision floating point is used. Not all data types are represented in all stars. Type conversions, automatic or explicit, can be used to complete the collection.

The parameters of a star are shown in italics. More information about each star can be obtained using the on-line "profile" command (","), the on-line "man" command ("M"), or by looking in the *Star Atlas* volume of *The Almagest*.

At the top of each palette, for convenience, are instances of the delay icon, the bus icon, and the following star:

**`BlackHole`** Discard all inputs. This star is useful for discarding signals that are not useful.

### 4.2.1 Source stars

Source stars are stars with no inputs. They generate signals, and may represent external inputs to the system, constant data, or synthesized stimuli. In the dataflow model of computation, they are always enabled, and hence can be fired at any time. In the synchronous dataflow model, the frequency with which they are fired, relative to other stars in the system, is determined by the solution to the balance equations. The palette of source stars is shown in figure 4-3, and the stars are summarized below, in the order they appear in the palette.

#### Floating-point sources

**`Ramp`** Generate a ramp signal, starting at *value* (default 0.0) and incre-

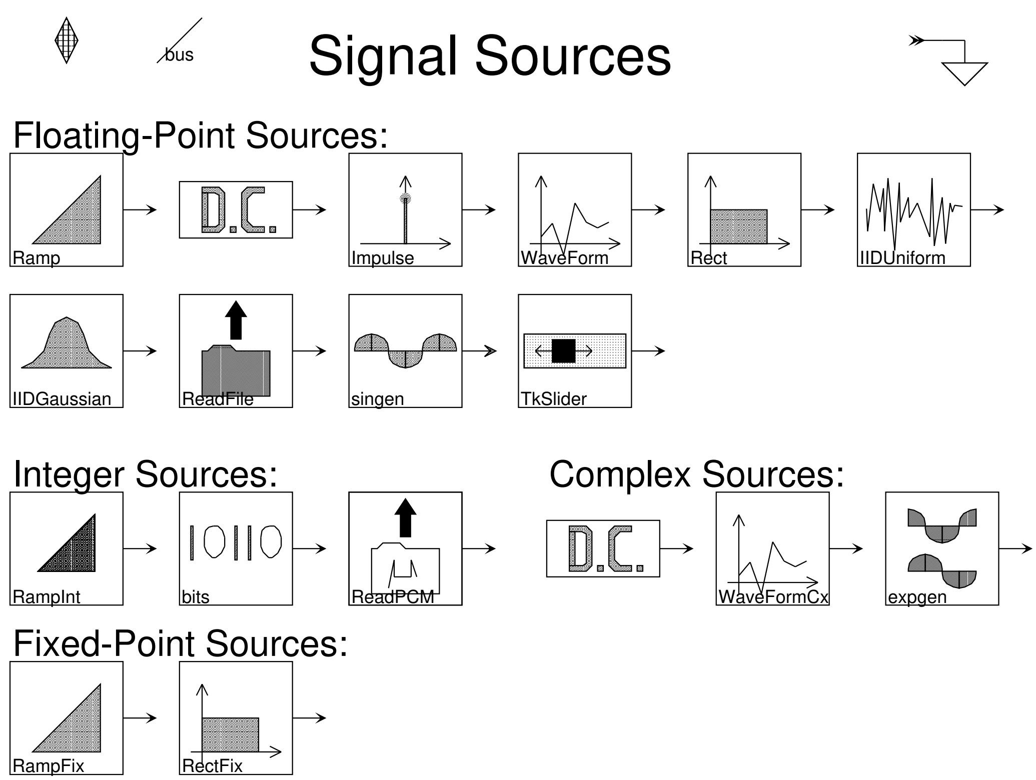



**FIGURE 4-3:** The palette of source stars for the SDF domain.

menting by step size *step* (default 1.0) on each firing.

**`Const`** Output a constant signal with value given by the *level* parameter (default 0.0).

**`Impulse`** Generate a stream of impulses of size *level* (default 1.0). The period is given by *period* (default 0). If *period* = 0 then only one impulse is generated.

**`WaveForm`** Output a waveform as specified by the array state *value* (default "1 -1"). You can get periodic signals with any period, and can halt a simulation at the end of the given waveform. The following table summarizes the capabilities:

| *haltAtEnd* | *periodic* | *period* | *operation* |
|---|---|---|---|
| NO | YES | 0 | The period is the length of the waveform |
| NO | YES | N>0 | The period is N |
| NO | NO | anything | Output the waveform once, then zeros |
| YES | anything | anything | Stop after outputting the waveform once |

The first line of the table gives the default settings. This star may be used to read a file by simply setting *value* to something of the form "< filename", preferably specifying a complete path.

**`Rect`** Generate a rectangular pulse of height *height* (default 1.0) and width *width* (default 8). If *period* is greater than zero, then the pulse is repeated with the given period.

**`IIDUniform`** Generate an i.i.d. uniformly distributed pseudo-random process. Output is uniformly distributed between *lower* (default 0) and *upper* (default 1).

**`IIDGaussian`** Generate a white Gaussian pseudo-random process with mean *mean* (default 0) and variance *variance* (default 1).

**`ReadFile`** Read ASCII data from a file. The simulation can be halted on end-of-file, or the file contents can be periodically repeated, or the file contents can be padded with zeros.

**`singen`** Generate a sine wave with frequency *frequency* (relative to the given *sample_rate*) and phase given by *phase_in_radians*. This is implemented as a galaxy.

**`TkSlider`** Output a value determined by an interactive on-screen scale slider.

### Integer sources

**`RampInt`** Ramp for integer values.

**`bits`** Produce "0" with probability *probOfZero*, else produce "1".

**ReadPCM** Read a binary μ-law encoded PCM file. Return one sample on each firing. The file format that is read is the same as the one written by the Play star. The simulation can be halted on end-of-file, or the file contents can be periodically repeated, or the file contents can be padded with zeros.

### Complex sources

**ConstCx** Constant source for complex values.

**WaveFormCx** Output a complex waveform as specified by the array state *value* (default “(1,0) (-1,0)”). Note that “(a,b)” means a + b j. The parameters work the same way as in the **WaveForm** star.

**expgen** Generate a complex exponential with the given frequency (relative to the *sample_rate* parameter).

### Fixed-point sources

**RampFix** Ramp for fixed-point values.

**RectFix** Generate a fixed-point rectangular pulse of height *height* (default 1.0). and width *width* (default 8). If *period* is greater than zero, then the pulse is repeated with the given period. The precision of *height* can be specified in bits.

## 4.2.2 Sink stars

The stars in the palette of figure 4-4 are those with no outputs. They display signals in various ways, or write them to files.

The first nine stars in this palette are all based on the **pxgraph** program. This program has many options, summarized in “pxgraph — The Plotting Program” on page 7-1. The differences between stars often amount to little more than the choice of default options. Some, however, preprocess the signal in useful ways before passing it to the **pxgraph** program. The first five icons actually correspond to only one star, with five different configurations. The first allows only one input signal, the second allows two, etc., until the fifth, which allows any number (notice the double arrow on the input port).

**XMgraph** (five icons) Generate a generic multi-signal plot.

**Waterfall** Plot a series of traces in the style of a waterfall plot. This is a type of three-dimensional plot used to show the evolution of signals or spectra. Optionally, each plot can be made opaque, so that lines that would appear behind the plot are eliminated.

**Xscope** Generate a multi-trace plot with the **pxgraph** program. Successive traces are overlaid on one another.

**XYgraph** Generate an X-Y plot with the **pxgraph** program. The X data is on “xInput” and the Y data is on “input”.

**Xhistogram** Generate a histogram with the **pxgraph** program. The parameter *binWidth* determines the bin width.

The remaining stars have no connection with the **pxgraph** program.

**Play** Play an input stream on the SPARC Station speaker. The *gain* state (default 1.0) multiplies the input stream before it is mu-law compressed and written. The inputs should be in the range of -32000.0 to 32000.0. The file is played at a fixed sampling rate of 8000 samples per second. When the wrapup method is called, a file of 8-bit μ-law samples is handed to a program named **ptplay** which plays the file. The **ptplay** program must be in your path.

**Printer** (two icons) Print out one sample from each input port per line. The *fileName* state specifies the file to be written; the special names <stdout> and <cout> which specify the standard output stream, as well as<stderr> and <cerr> which specify the standard error stream, are also supported.

The remaining stars in the palette are multiple configurations of only four stars. These stars all use the Tk toolkit associated with the Tcl language to create interactive, animated displays on

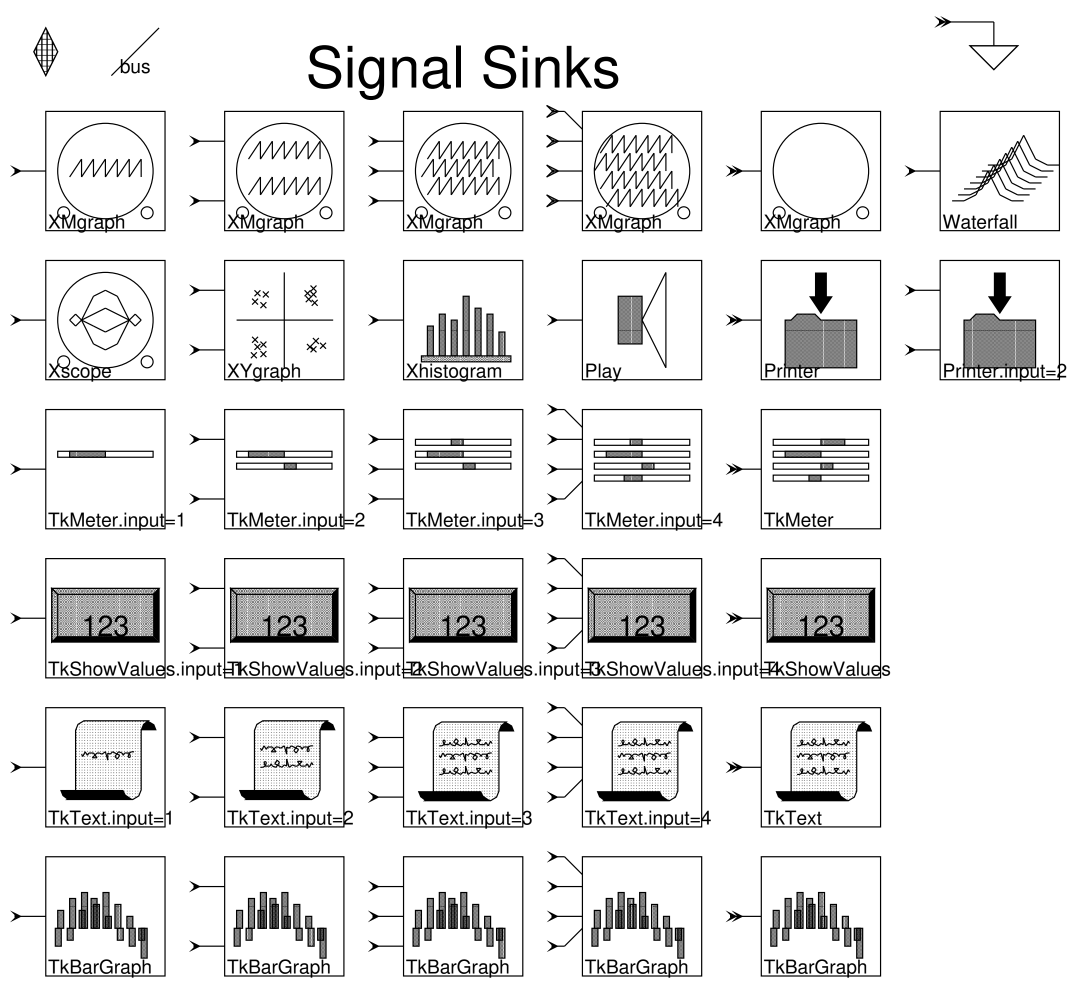



**FIGURE 4-4:** Sink stars in the SDF domain.

the screen.

**TkMeter** (five icons) Dynamically display the value of any number of input signals on a set of bar meters.

**TkShowValues** (five icons) Display the values of the inputs in textual form. The print method of the input particles is used, so any data type can be handled, although the space allocated on the screen may need to be adjusted.

**TkText** (five icons) Display the values of the inputs in a separate window, keeping a specified number of past values in view. The print method of the input particles is used, so any data type can be handled.

**TkBarGraph** (five icons) Dynamically display the value of any number of input signals in bar-chart form. The first 12 input signals will be assigned distinct colors. After that, the colors are repeated. The colors can be controlled using X resources.

### 4.2.3 Arithmetic stars

In principle, it would be possible to overload the basic arithmetic operators so that, for example, a single **Add** star could handle any data type. Our decision, however, was in favor of more explicit typing, in which there is an **Add** star for each particle type supported in the kernel. As before, when there is no data type suffix in the name of the star, the data type supported is double-precision floating point.

Many of the stars in this palette have more than one icon, as indicated in figure 4-5. Each such icon has a different configuration of ports. This is done for visual clarity in schematics. A port with a double arrowhead can accept any number of input signals. The first four rows of the palette each contain equivalent stars for floating-point, complex, fixed-point, and integer arithmetic, respectively. Listed by the roots of the names of the stars, they are:

**Add** (two icons) Output the sum of the inputs.

**Sub** Output the “pos” input minus all “neg” inputs.

**Mpy** (two icons) Output the product of the inputs.

**Gain** This is an amplifier; the output is the input multiplied by the *gain* (default 1.0).

**Average** Average some number of input samples or blocks of input samples. Blocks of successive input samples are treated as vectors.

The last row contains one additional arithmetic star:

**Integrator** This is an integrator with leakage, limits, and reset. With the default parameters, input samples are simply accumulated, and the running sum is the output. To prevent any resetting in the middle of a run, connect a **Const** source with value 0 to the “reset” input. Otherwise, whenever a non-zero is received on this input, the accumulated sum is reset to the current input (i.e.

no feedback).

Limits are controlled by the *top* and *bottom* parameters. If $top \leq bottom$, no limiting is performed (this is the default). Otherwise, the output is kept between *bottom* and *top*. If *saturate* = “YES”, saturation is performed. If *saturate* = “NO”, wrap-around is performed (this is the default). Limiting is performed before output.

Leakage is controlled by the *feedbackGain* parameter (default 1.0). The output is the data input plus *feedbackGain* $\times$ *state*, where *state* is the previous output.

### 4.2.4 Nonlinear stars

The nonlinear palette (figure 4-6) in the SDF domain includes transcendental functions, quantizers, table lookup stars, and miscellaneous nonlinear functions.

**cexp** Compute the complex exponential of the input.

**conj** Compute the complex conjugate of the input.

**Cos** Compute the cosine of its input, assumed to be an angle in radians.

**Dirichlet** Compute the normalized Dirichlet kernel (also called the aliased sinc function):

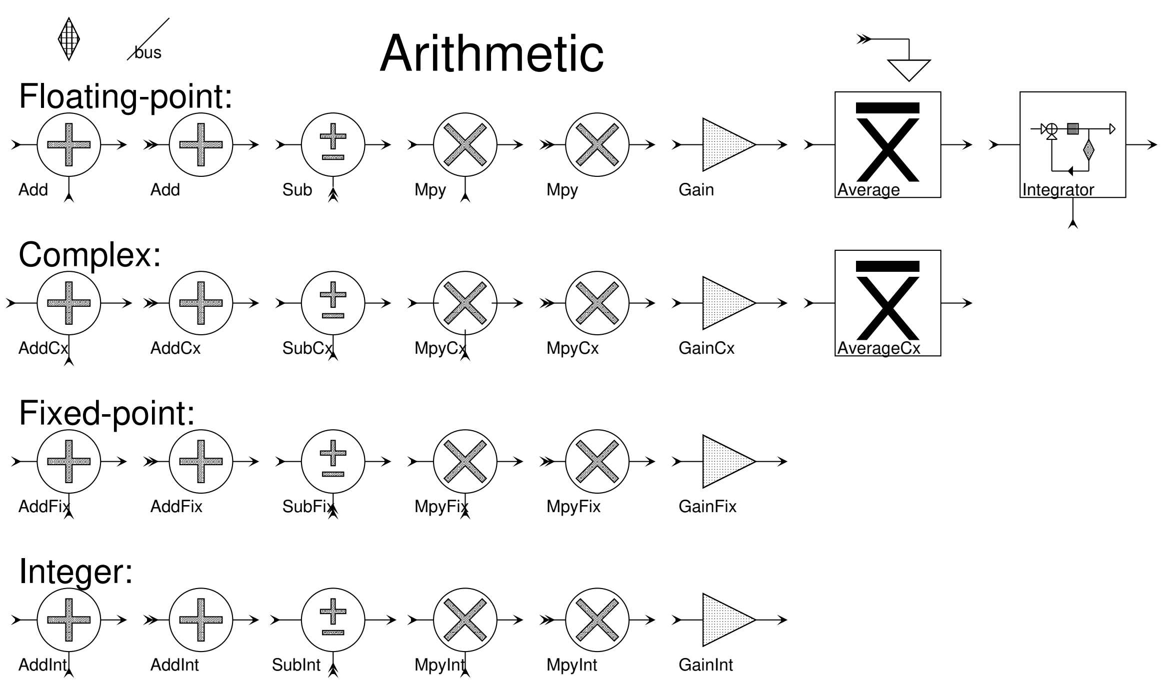



**FIGURE 4-5:** The arithmetic palette in the SDF domain. Note that several of the stars have more than one icon, each with a different configuration of ports.

$$d_N(x) = \frac{\sin(Nx/2)}{N \sin(x/2)}$$

The value of the normalized Dirichlet kernel at $x = 0$ is always 1, and the normalized Dirichlet kernel oscillates between −1 and +1. The normalized Dirichlet kernel is periodic in $x$ with a period of either $2\pi$ when $N$ is odd or $4\pi$ when $N$ is even.

**`Exp`** Compute the exponential function of the input.

**`expjx`** Compute the complex exponential function of the input.

**`Floor`** Output the greatest integer less than or equal to the input.

**`Limit`** The output of this star is the value of the input limited to the range between *bottom* and *top* inclusive.

**`LinQuantIdx`** Quantize the input to the number of levels given by the *levels* parameter. The quantization levels are uniformly spaced between *low* and *high* inclusive. Rounding down is performed, so that output level will equal *high* only if the input level equals or exceeds *high*. If the input is below *low*, then the quantized output will equal *low*. The quantized value is output to the "amplitude" port, while the index of the quantization level is output to the "stepNumber" port. This integer output is useful

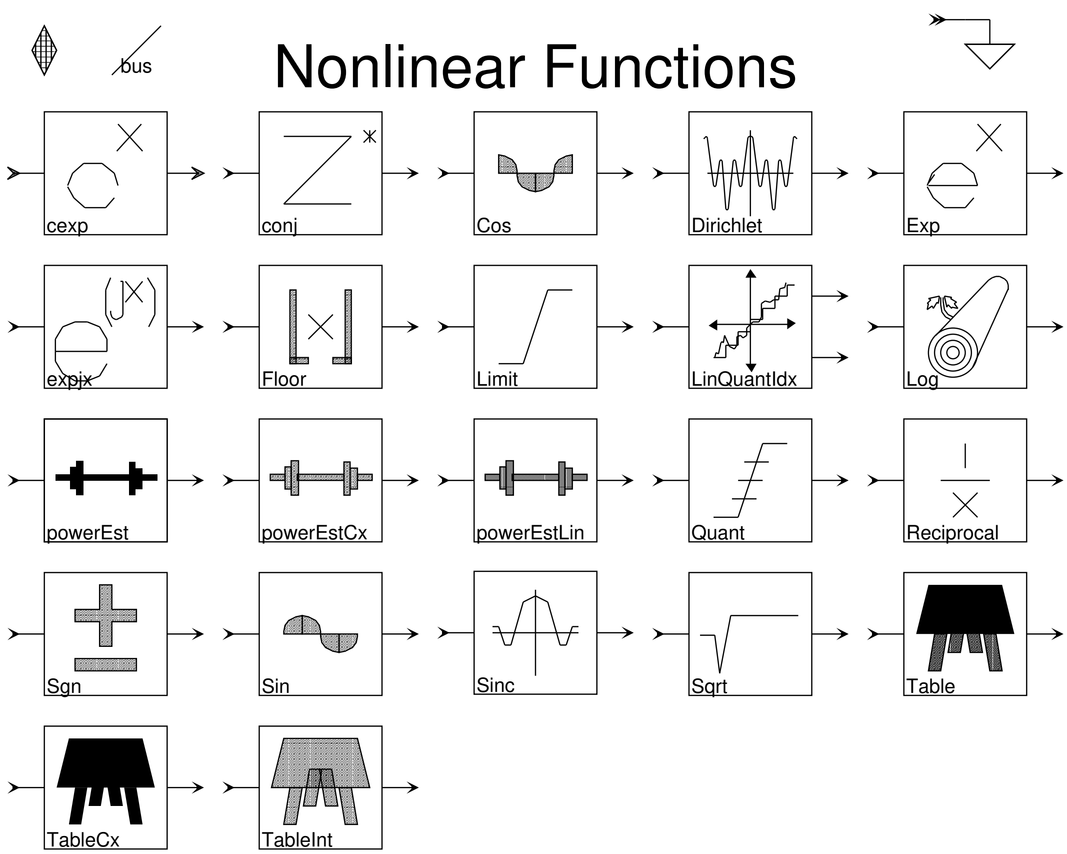



**FIGURE 4-6:** Palette of nonlinear stars for the SDF domain.

for stars that need an integer input, such as Thor stars.

**Log** Output the natural logarithm of the input value.

**powerEst** Estimate the power in dB by filtering the square of the input using a first-order filter with the time constant given as a number of sample periods.

**powerEstCx** Like **powerEst**, but for complex inputs.

**powerEstLin** Same as **powerEst**, but the output is on a linear scale instead of dB.

**Quant** Quantize the input value to one of N+1 possible output levels using N thresholds. For an input less than or equal to the n-th threshold, but larger than all previous thresholds, the output will be the n-th level. If the input is greater than all thresholds, the output is the N+1-th level. If level is specified, there must be one more level than thresholds; the default value for level is 0, 1, 2, ... N. This star is much slower than **LinQuantIdx**, so if possible, that one should be used instead.

**Reciprocal** Output the reciprocal of the input, with an optional magnitude limit. If the magnitude limit is greater than zero, and the input value is zero, then the output will equal the magnitude limit.

**Sgn** Compute the signum of the input. The output is ±1. Note that 0.0 maps into 1.

**Sin** Computes the sine of the input, assumed to be an angle in radians.

**Sinc** Computes the sinc of the input given in radians. The sinc function is defined as $\sin(x)/x$, with value 1.0 when $x = 0$.

**Sqrt** Computes the square root of the input.

**Table** This star implements a real-valued lookup table indexed by an integer-valued input. The input must lie between 0 and N-1, inclusive, where N is the size of the table. The *values* parameter specifies the table. Its first element is indexed by a zero-valued input. An error occurs if the input value is out-of-bounds.

**TableCx** Table lookup for complex values.

**TableInt** Table lookup for integer values.

### 4.2.5 Logic stars

The logic palette shown in figure 4-7 is made up of only three stars. Each star has multiple icons representing a variety of configurations.

**Test** (four icons) Compare two inputs. The test condition can be any of {EQ NE GT GE} or {== != > >=}, resulting in equals, not equals, greater than, or greater than or equals. The four icons represent these possibilities.

If *crossingsOnly* is **TRUE**, then the output is non-zero only when the outcome of the test changes from **TRUE** to **FALSE** or **FALSE** to **TRUE**. In this case, the first output is always **TRUE**.

**Multiple** (one icon) Output a 1 if top input is a multiple of bottom input.

**Logic** (19 icons) This star applies a logical operation to any number of inputs. The inputs are integers interpreted as Booleans, where zero is a **FALSE** and nonzero is a **TRUE**. The logical operations supported are {**NOT**, **AND**, **NAND**, **OR**, **NOR**, **XOR**, **XNOR**}, with any number of inputs.

### 4.2.6 Control stars

Control stars (figure 4-8) manipulate the flow of tokens. All of these stars are polymorphic; they operate on any data type. From left to right, top to bottom, they are:

**Fork** (three icons) Copy input particles to each output. Note that a fork is automatically inserted in a schematic when a single output is sent to more than one input. However, when a delay is needed on one of the connections, then an explicit fork star must be used.

**DownSample** Decimate by a given *factor* (default 2). The *phase* tells which sample of the last *factor* samples to output. If *phase* = 0, the most recent sample is the output, while if *phase* = *factor* −1 the

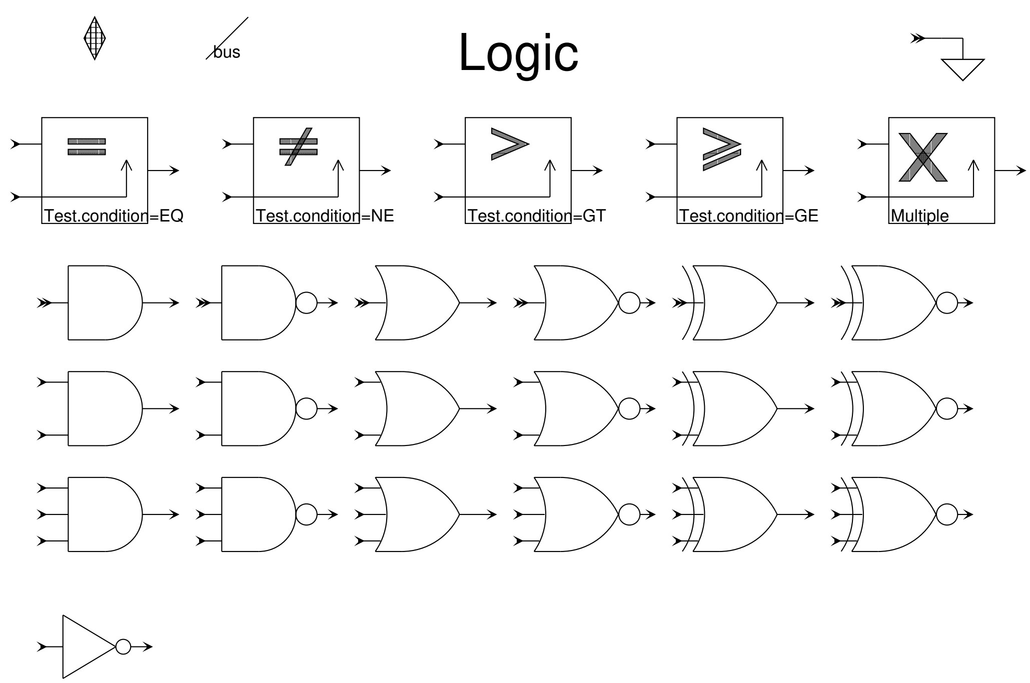



**FIGURE 4-7:** Logic stars in the SDF palette.

oldest sample is the output. *Phase* = 0 is the default. Note that *phase* has the opposite sense of the phase parameter in the **`UpSample`** star, but the same sense as the phase parameter in the **`FIR`** star.

**`UpSample`** Upsample by a given *factor* (default 2), filling with *fill* (default 0.0). The *phase* parameter tells where to put the sample in an output block. The default is to output it first (*phase* = 0). The maximum *phase* is *factor* − 1. Although the *fill* parameter is a floating-point number, if the input is of some other type, such as complex, the fill particle will be obtained by casting *fill* to the appropriate type.

**`Repeat`** Repeat each input sample a specified number of times.

**`Commutator`** (three icons) Synchronously combine $N$ input streams (where $N$ is the number of inputs) into one output stream. The star consumes $B$ input particles from each input (where $B$ is the *blockSize*), and produces $N \times B$ particles on the output. The first $B$ particles on the output come from the first input, the next $B$ particles from the next input, etc.

**`Distributor`** (three icons) Synchronously split one input stream into $N$ output streams, where $N$ is the number of outputs. The star consumes $N \times B$ input particles, where $B$ = *blockSize*, and sends the first $B$ particles to the first output, the next $B$ particles to the next output, etc.

**`Trainer`** Pass the value of the "train" input to the output for the first *trainLength* samples, then pass the "decision" input to the out-

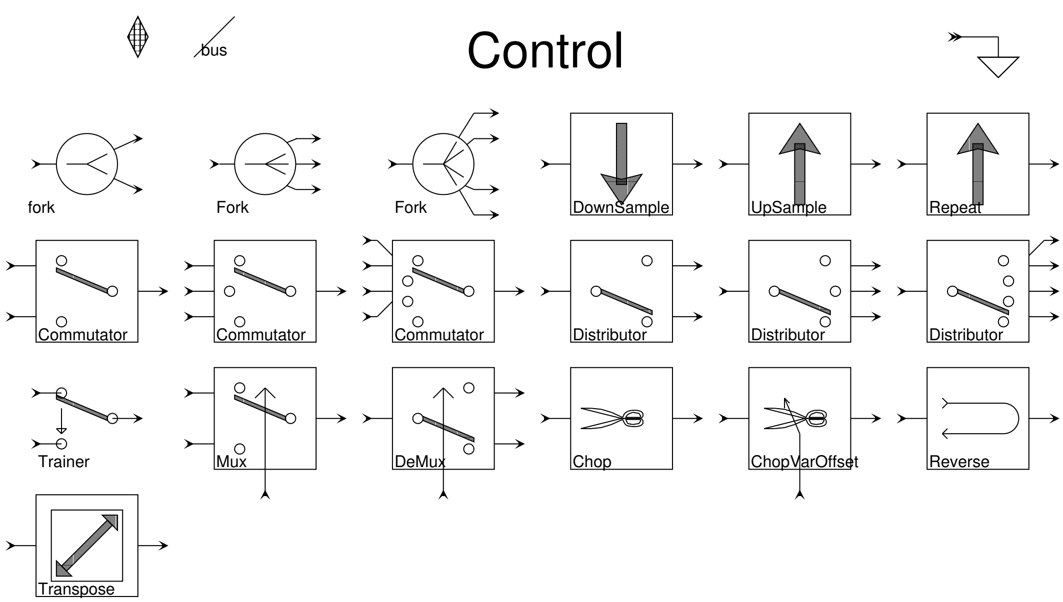



**FIGURE 4-8:** Control stars for the SDF domain.

put. This star is designed for use with adaptive equalizers that require a training sequence at start-up, but it can be used whenever one sequence is used during a start-up phase, and another sequence after that.

**Mux** Multiplex any number of inputs onto one output stream. $B$ particles are consumed on each input, where $B$ is the *blockSize*. But only one of these blocks of particles is copied to the output. The one copied is determined by the "control" input. Integers from 0 through $N-1$ are accepted at the "control" input, where $N$ is the number of inputs. If the control input is outside this range, an error is signaled.

**DeMux** Demultiplexes one input onto any number of output streams. The star consumes $B$ particles from the input, where $B$ is the *blockSize*. These $B$ particles are copied to exactly one output, determined by the "control" input. The other outputs get a zero of the appropriate type.

Integers from 0 through $N-1$ are accepted at the "control" input, where $N$ is the number of outputs. If the control input is outside this range, all outputs get zeros.

**Chop** On each execution, this star reads a block of *nread* particles and writes them to the output with the given offset. The number of particles written is given by *nwrite*. The output block contains all or part of the input block, depending on *offset* and *nwrite*. The *offset* specifies where in the output block the first (oldest) particle in the input block will lie. If *offset* is positive, then the first *offset* output particles will be either particles consumed on previous firings (if *use_past_inputs* parameter is YES), or zero (otherwise). If *offset* is negative, then the first *offset* input particles will be discarded.

**ChopVarOffset** This star has the same functionality as the **Chop** star except the offset parameter is determined at run time by a control input.

**Reverse** On each execution, read a block of $N$ samples (default 64) and write them out backwards.

**Transpose** Transpose a rasterized matrix (which is read as a sequence of particles, row by row, and written out the same way). The number of particles produced and consumed equals the product of *samplesInaRow* and *numberOfRows*. Contrast this with the transpose stars in the matrix palette, which operate on matrix particles.

### 4.2.7 Conversion stars

The palette in figure 4-9 shows a collection of stars for format conversions of various types. The first two rows contain stars with functions that are fundamentally different from the automatic type conversion performed by Ptolemy. From left to right, top to bottom, they are:

**`CxToRect`** Convert a complex input to real and imaginary parts.

**`RectToCx`** Convert real and imaginary inputs into a complex output.

**`RectToPolar`** Convert real and imaginary inputs into magnitude and phase form. The phase output is in the range $-\pi$ to $\pi$.

**`PolarToRect`** Convert magnitude and phase to rectangular form.

**`NumToBus`** (two icons) This star accepts an integer and outputs the low-order bits that make up the integer on a number of outputs, one bit per output. The number of outputs should not exceed the word size of an integer. This star is particularly useful for interfacing to digital logic simulation domains.

**`PCMBitCoder`** Encode voice samples for a 64 kbps bit stream using CCITT Recommendation G.711. The input is one 8 kHz sample of voice data and the output is the eight-bit codeword (the low-order 8 bits of an integer) representing the quantized samples.

**`MuLaw`** This star encodes its input into an 8 bit representation using the non-linear companding Mu law. It is similar to **`PCMBitCoder`**, but it does the conversion in a single star, rather than a galaxy.

**`PCMBitDecoder`** Decode 8-bit PCM codewords that were encoded using **`PCM-BitCoder`**.

**`MakeSeqATMCell`** Load a sequence of up to 384 sequential Boolean input bits into a **`SeqATMCell`** object. This is used for simulation of asynchro-

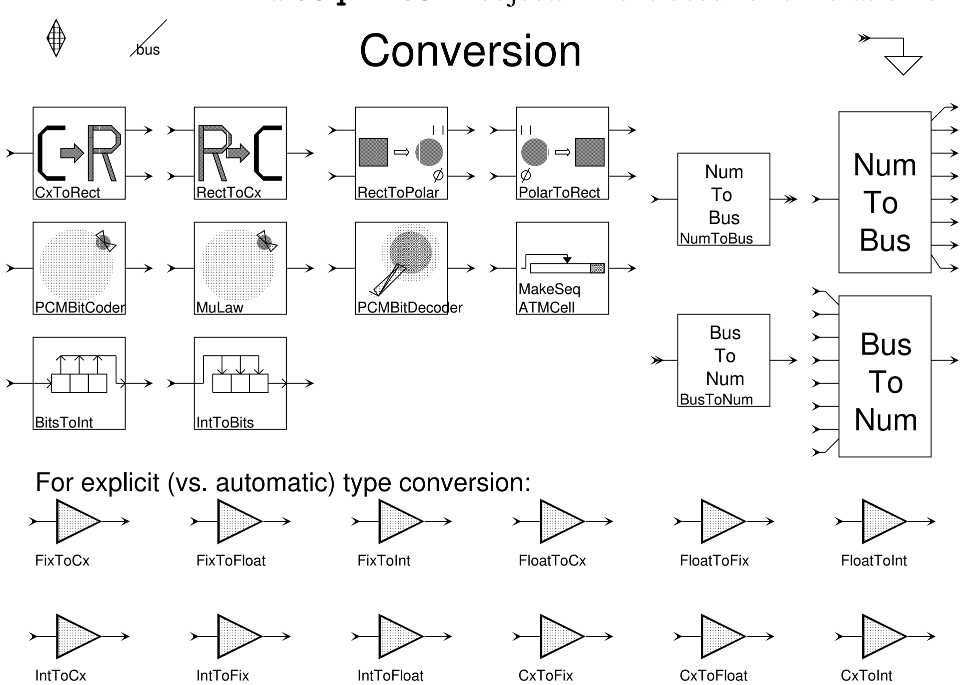



**FIGURE 4-9:** Type conversion stars for the SDF domain.

nous transfer mode (ATM) networks.

**BitsToInt** The integer input sequence is interpreted as a bit stream in which any non-zero value is a “one” bit. This star consumes *nBits* successive bits from the input, packs them into an integer, and outputs the resulting integer. The first received bit becomes the most significant bit of the output. If *nBits* is larger than the integer wordsize, then the first bits received will be lost. If *nBits* is smaller than the wordsize minus one, then the output integer will always be non-negative.

**IntToBits** Read the least significant *nBits* bits from an integer input, and output the bits as integers serially on the output, most significant bit first.

**BusToNum** (two icons) This star accepts a number of input bit streams, where this number should not exceed the wordsize of an integer. Each bit stream has integer particles with values 0, 3, or anything else. These are interpreted as binary 0, tri-state, or 1, respectively. When the star fires, it reads one input bit from each input. If any of the input bits is tri-stated, the output will be the previous output (or the initial value of the *previous* parameter if the firing is the first one). Otherwise, the bits are assembled into an integer word, assuming two's complement encoding, and sign extended. The resulting signed integer is sent to the output. This star is particularly useful for interfacing to digital logic simulation domains.

Automatic type conversion, as implemented in Ptolemy 0.5, has limitations. If a given output port has more than one destination, then all destinations must have the same type input. This is true even if an explicit **fork** star is used. Explicit type conversions are needed to get around this limitation. For this reason, the palette in figure 4-9 also contains a set of type conversions that behave exactly the same way the automatic type conversions behave.

**FixToCx** Convert a fixed-point input to a complex output.

**FixToFloat** Convert a fixed-point input to a floating-point output.

**FixToInt** Convert a fixed-point input to an integer output.

**FloatToCx** Convert a floating-point input to a complex output.

**FloatToFix** Convert a floating-point input to an fixed-point output.

**FloatToInt** Convert a floating-point input to an integer output.

**IntToCx** Convert an integer input to a complex output.

**IntToFix** Convert an integer input to a fixed-point output.

**IntToFloat** Convert an integer input to a floating-point output.

**CxToFix** Convert a complex input to a fixed-point output.

**CxToFloat** Convert a complex input to a floating-point output.

**CxToInt** Convert a complex input to an integer output.

### 4.2.8 Signal processing stars

The palette shown in figure 4-10 has icons for the library of signal processing functions. Simple time-domain filtering operations come first.

#### Filtering

**Biquad** A two-pole, two-zero IIR filter (a biquad). The default is a Butterworth filter with a cutoff at 0.1 times the sample frequency. The transfer function is

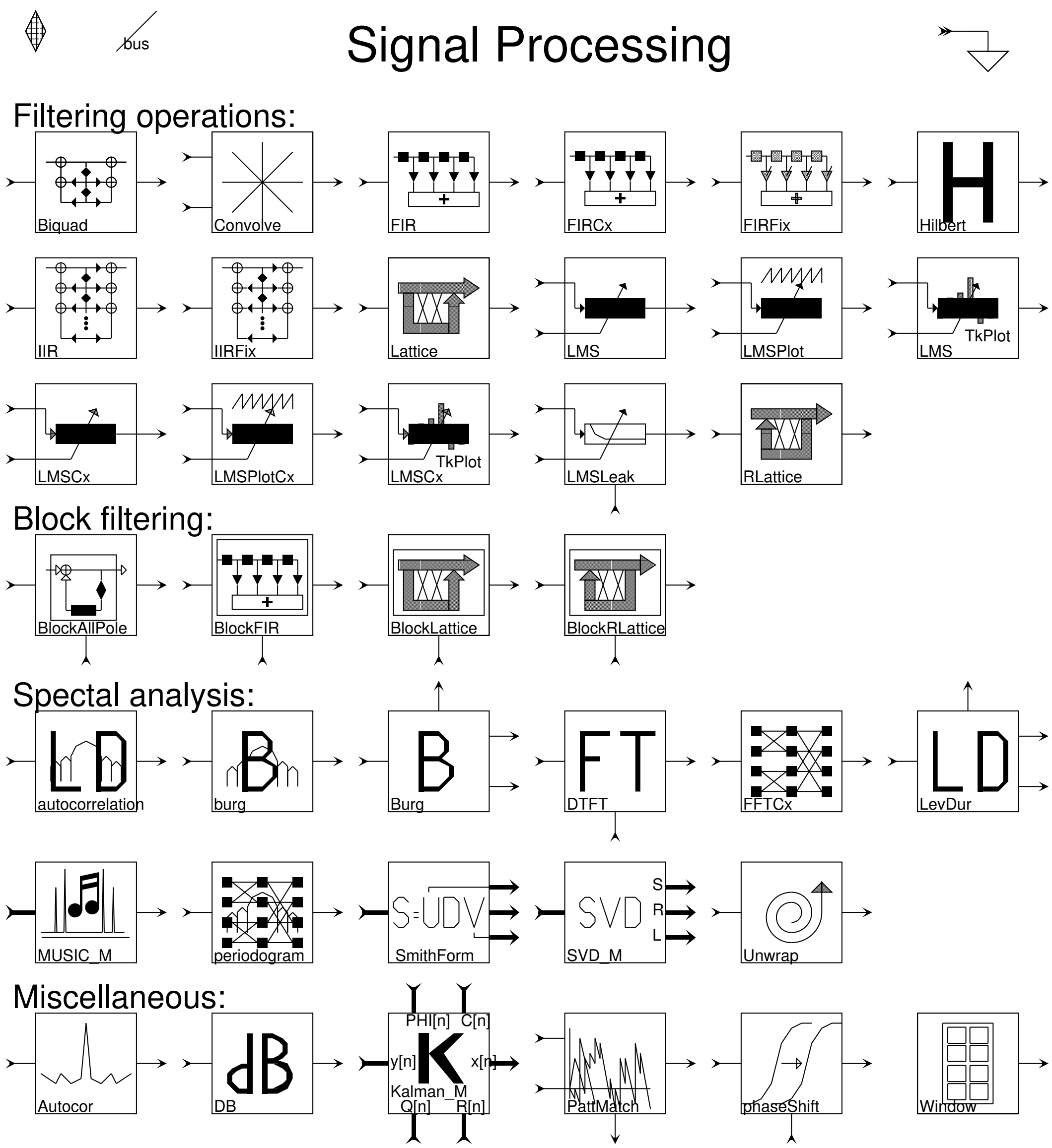



**FIGURE 4-10:** The signal processing (dsp) palette of the SDF domain.

$$H(z) = \frac{n_0 + n_1 z^{-1} + n_2 z^{-2}}{1 + d_1 z^{-1} + d_2 z^{-2}}.$$

**`Convolve`** Convolve two causal finite sequences. The *truncationDepth* parameter specifies the number of terms used in the convolution sum. Set *truncationDepth* larger than the number of output samples of interest.

**`FIR`** A finite impulse response (FIR) filter. Coefficients are specified by the *taps* parameter. The default coefficients give an 8th order, linear-phase, lowpass filter. To read coefficients from a file, replace the default coefficients with "< fileName", preferably specifying a complete path. Polyphase multirate filtering is also supported.

**`FIRCx`** A complex finite impulse response (FIR) filter. Coefficients are specified by the *taps* parameter. The default coefficients give an 8th order, linear phase, lowpass filter. To read coefficients from a file, use the syntax: "< fileName", preferably specifying a complete path. Real and imaginary parts should be paired with parentheses, e.g. (1.0, 0.0). Polyphase multirate filtering is also supported.

**`FIRFix`** A Finite Impulse Response (FIR) filter with fixed-point capabilities. The fixed-point coefficients are specified by the *taps* state variable. The default coefficients give an 8th order, linear phase lowpass filter. To read coefficients from a file, replace the default coefficients with "< fileName", preferably specifying a complete path. Polyphase multirate filtering is also supported.

**`Hilbert`** Output the (approximate) Hilbert transform of the input signal. This star approximates the Hilbert transform by using an FIR filter, and is derived from **`FIR`**.

**`IIR`** An infinite impulse response (IIR) filter implemented in direct form II. The transfer function is of the form

$$H(z) = G \frac{N(1/z)}{D(1/z)},$$

where $N()$ and $D()$ are polynomials. The parameter *gain* specifies $G$, and the floating-point arrays *numerator* and *denominator* specify $N()$ and $D()$, respectively. Both arrays start with the constant terms of the polynomial and decrease in powers of $z$ (increase in powers of $1/z$). Note that the constant term of $D$ is not omitted, as is common in other programs that assume it is always normalized to unity.

**`IIRFix`** This is a fixed-point version of the **`IIR`** star. The coefficient precision, input precision, accumulation precision, and output precision can all be separately specified.

**Lattice** An FIR lattice filter. The default reflection coefficients form the optimal predictor for a particular 4th-order AR random process. To read other reflection coefficients from a file, replace the default coefficients with "< fileName", preferably specifying a complete path.

**LMS** An adaptive filter using the LMS adaptation algorithm. The initial coefficients are given by the *taps* parameter. The default initial coefficients give an 8th order, linear phase lowpass filter. To read default coefficients from a file, replace the default coefficients with "< fileName", preferably specifying a complete path. This star, which is derived from **FIR**, supports decimation, but not interpolation.

**LMSPlot** This star is exactly like the **LMS** star, except that, in addition to the functions of **LMS**, it makes a plot of the tap coefficients. It can produce two types of plots: a plot of the final tap values or a plot that traces the time evolution of each tap value. The time evolution is obtained if the value of the parameter *trace* is "YES".

**LMSTkPlot** This star is just like the **LMS** star, but with an animated Tk display of the taps, plus associated controls.

**LMSCx** Complex version of **LMS**.

**LMSPlotCx** Complex version of **LMSPlot**. Separate plots are generated for the magnitude and phase of the filter coefficients.

**LMSCxTkPlot** This star is just like the **LMSCx** star, but with an animated Tk display of the taps, plus associated controls.

**LMSLeak** An LMS adaptive filter in which the step size is input (to the "step" input) every iteration. In addition, the *mu* parameter specifies a leakage factor in the updates of the filter coefficients.

**RLattice** A recursive (IIR) lattice filter. The default coefficients implement the synthesis filter for a particular 4th order AR random process.

### Block filtering

The next group of stars perform "block filtering," which means that on each firing, they read a set of input particles all at once, process them, and produce a set of output particles. The number of particles in a set is specified by the *blockSize* parameter.

**BlockAllPole** This star implements an all pole filter with the denominator coefficients of the transfer function externally supplied. For each set of coefficients, a block of input samples is processed, all in one firing. The transfer function is

$$H(z) = \frac{1}{1 - D(z)}$$

where the coefficients of $D(z)$ are externally supplied.

**BlockFIR** This star implements an FIR filter with coefficients that are periodically updated from the outside. For each set of coefficients, a block of input samples is processed, all in one firing.

**BlockLattice** A block forward lattice filter. It is identical to the **Lattice** star except that the reflection coefficients are updated each time the star fires by reading the "coefs" input. The *order* parameter indicates how many coefficient should be read. The *blockSize* parameter specifies how many data samples should be processed for each set of coefficients.

**BlockRLattice** A block recursive (IIR) lattice filter. It is identical to the **RLattice** star, except that the reflection coefficients are updated each time the star fires by reading the "coefs" input. The *order* and *blockSize* parameters have the same interpretation as in the **BlockLattice** star.

### Spectral analysis

The next group of stars are concerned with various signal analysis algorithms.

**autocorrelation** Estimate a power spectrum using the autocorrelation method, a method that uses the Levinson-Durbin algorithm to compute linear predictor coefficients, and then uses these coefficients to construct an approximate maximum entropy power spectrum estimate.

**burg** Estimate a power spectrum using Burg's method, a method that uses Burg's algorithm to compute linear predictor coefficients, and then uses these coefficients to construct an approximate maximum entropy power spectrum estimate.

**Burg** This star uses Burg's algorithm to estimate the linear predictor coefficients of an input random process. These coefficients are produced both in autoregressive form (on the "lp" output) and in lattice filter form (on the "refl" output). The "errPower" output is the power of the prediction error as a function of the predictor order. This star is used in the **burg** galaxy.

**DTFT** Compute the discrete-time Fourier transform (DTFT) at frequency points specified on the "omega" input.

**FFTCx** Compute the discrete-time Fourier transform of a complex input using the fast Fourier transform (FFT) algorithm. The parameter *order* (default 8) is the log base 2 of the transform size. The parameter *size* (default 256) is the number of samples read (<= 2^order). The parameter *direction* (default 1) is 1 for the forward, -1 for the inverse FFT.

**LevDur** This star uses the Levinson-Durbin algorithm to compute the linear predictor coefficients of a random process, given its autocorrelation function as an input. These coefficients are produced both in autoregressive form (on the "lp" output) and in lattice filter form (on the "refl" output). The "errPower" output is the power of the prediction error as a function of the predictor order.

**MUSIC_M** This star is used to estimate the frequencies of some specified number of sinusoids in a signal. The output is the eigenspectrum of a signal such that the locations of the peaks of the eigenspectrum correspond to the frequencies of the sinusoids in the signal. The input is the right singular vectors in the form generated by the **SVD_M** star. The MUSIC algorithm is used, where MUSIC stands for "multiple signal characterization."

**periodogram** Estimate a power spectrum using the periodogram method. This consists simply of computing the magnitude squared of the DFT of a set of observations of the signal. The FFT algorithm is used.

**SmithForm** Decompose an integer matrix *S* into one of its Smith forms *S* = *U D V*, where *U*, *D*, and *V* are simpler integer matrices. The Smith form decomposition for integer matrices is analogous to singular value decomposition for floating-point matrices.

**SVD_M** Compute the singular-value decomposition of a Toeplitz data matrix *A* by decomposing *A* into *A* = *UWV'*, where *U* and *V* are orthogonal matrices, and *V'* represents the transpose of *V*. *W* is a diagonal matrix composed of the singular values of *A*, and the columns of *U* and *V* are the left and right singular vectors of *A*.

**Unwrap** Unwraps a phase plot, removing discontinuities of magnitude $2\pi$. The star assumes that the phase never changes by more than PI in one sample period. It also assumes that the input is in the range $[-\pi,\pi]$.

## Miscellaneous signal processing blocks

**Autocor** Estimate an autocorrelation function by averaging input samples. Both biased and unbiased estimates are supported.

**DB** Convert input to a decibels (dB) scale. Zero and negative values are assigned the value *min* (default -100). The *inputIsPower* parameter should be set to "YES" if the input signal is a power measurement (vs. an amplitude measurement).

**Kalman_M** Output the state vector estimates of a Kalman filter using a one-step prediction algorithm.

**PattMatch** This star accepts a template and a search window. The template is slid over the window one sample at a time, and cross correlations are calculated at each step. The cross-correlations are output on the "values" output. The "index" output is the value of the time-shift which gives the largest cross correlation. This index refers to a position on the search window beginning with 0 corresponding to the earliest arrived sample of the search window that is part of the best match with the template.

**phaseShift** This galaxy applies a phase shift to a signal according to the "shift" input. If the "shift" input value is time varying, then its slope determines the instantaneous frequency shift.

**Window** Generate standard window functions or periodic repetitions of standard window functions. The possible functions are Rectangle, Bartlett, Hanning, Hamming, Blackman, and SteepBlackman. One period of samples is produced on each firing.

### 4.2.9 Image processing stars

The image processing stars contained in the palette in figure 4-11 were mostly written by Paul Haskell. Haskell's implementation defines a set of specialized message particles that each contain an entire image. These stars, therefore, operate on streams of images, typically consuming and producing one image on each firing.

The base class for all the image types is called **BaseImage**. It is based on Ptolemy's built-in **Message** type, which allows aggregate data types to be passed between stars. The **GrayImage** class, derived from **BaseImage**, is used to represent gray-scale images. The **DCTImage** class is used to represent images or image fragments that have been encoded using the discrete-cosine transform. The **MVImage** class is a bit more specialized; it stores a frame's worth of motion vectors. See the programmer's manual for further information about these classes.

### Reading and displaying images

**DisplayImage** Accept a black-and-white input **GrayImage** and generate output in PGM format. Send the output to a user-specified command (by default, "**$PTOLEMY/bin.$ARCH/xv**" is used).

The user can set the root filename of the displayed image (which will probably be printed in the image display window title bar) and can choose whether or not the image file is saved or deleted. The image frame number is appended to the root filename to form the complete filename of the displayed image.

**DisplayRGB** This is similar to **DisplayImage**, but accepts three color images (Red, Green, and Blue) from three input **GrayImage**s and generates a PPM format color image file. The image file is displayed using a user-specified command (by default, "**$PTOLEMY/bin.$ARCH/xv**" is used).

**`DisplayVideo`** Accept a stream of black-and-white images from input **`GrayImage`**s, save the images to files, and display the resulting files as a moving video sequence. This star requires that programs from the “Utah Raster Toolkit” be in your path. Although this toolkit is not included with Ptolemy, it is available for free. See this star’s long description (with the “look-inside” or “manual” commands in the Ptolemy menu) for information on how to get the toolkit.

The user can set the root filename of the displayed images (which probably will be printed in the display window title bar) with the *ImageName* parameter. If no filename is set, a default will be chosen.

The *Save* parameter can be set to “YES” or “NO” to choose whether the created image files should be saved or deleted. Each image’s frame number is appended to the root filename to form the image’s complete filename.

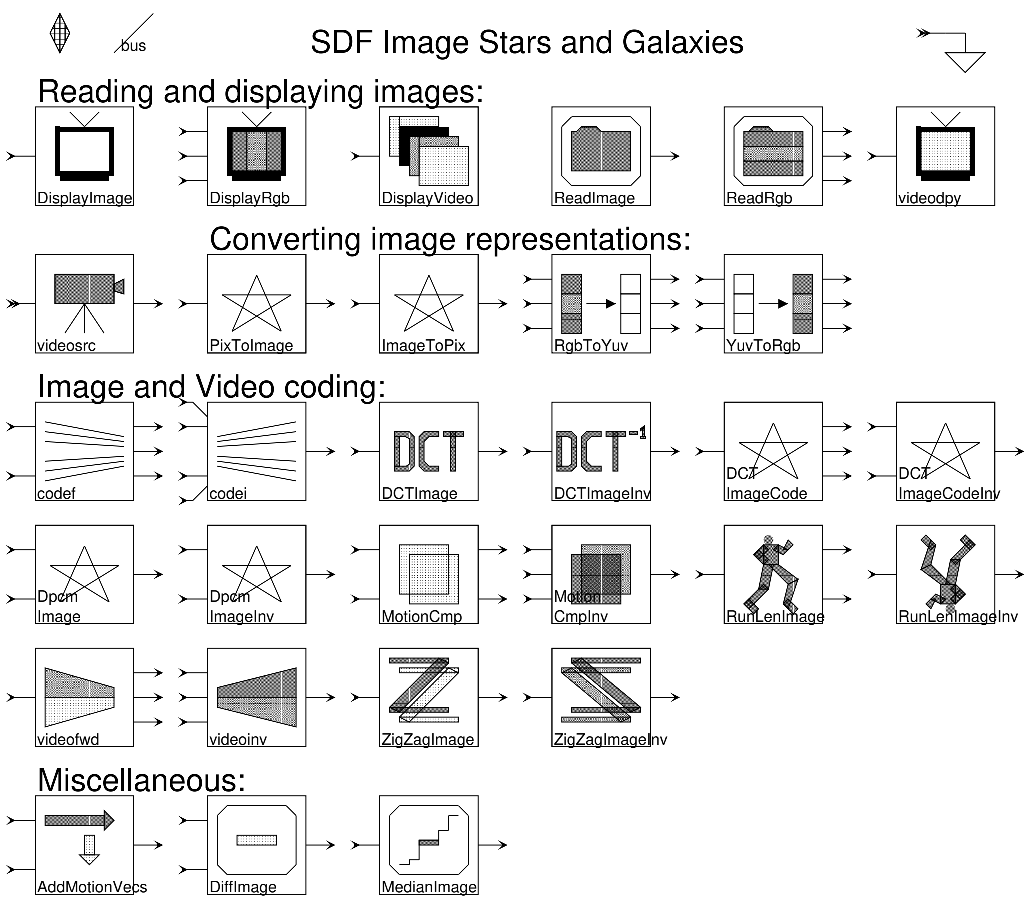



**FIGURE 4-11:** The Image processing palette in the SDF domain.

The *ByFields* parameter can be set to either “YES” or “NO” to choose whether the input images should be treated as interlaced fields that make up a frame or as entire frames. If the inputs are fields, then the first field should contain frame lines 1, 3, 5, etc. and the second field should contain lines 0, 2, 4, 6, etc.

**`ReadImage`** Read a sequence of PGM-format images from different files and send them out in **`Envelopes`** (containing data of type **`GrayImage`**).

If present, the character ‘#’ in the “fileName” state is replaced with the frame number to be read next. For example, if the “frameId” state is set to 2 and if the “fileName” state is “dir.#/pic#” then the files that are read and output are “dir.2/pic2”, “dir.3/pic3”, etc.

**`ReadRGB`** Read a PPM-format image from a file and send it out in three inputs — a Red, Green, and Blue image. Each image is of type **`GrayImage`**. The same mechanism for reading successive frames as in **`ReadImage`** is supported.

**`videodpy`** Display an image sequence in an X window. This is simply the **`SDFDisplayVideo`** star encapsulated in a galaxy so that it can be easily used in other domains.

**`videosrc`** Read in an image from a specified file. This is simply the **`SDFReadImage`** star encapsulated in a galaxy so that it can be easily used in other domains.

## Converting image representations

**`PixToImage`** Accept integer values, and assemble them into an image. **Warning:** this star produces large sample-rate changes.

**`ImageToPix`** Accept a black-and-white image from an input packet, and generate the individual pixels. A single firing will typically produce a very large number of output particles.

**`RGBToYUV`** Read three **`GrayImage`**s that describe a color image in RGB format and output three **`GrayImage`**s that describe an image in YUV format. No downsampling is done on the U and V signals.

**`YUVToRGB`** Read three **`GrayImage`**s that describe a color image in YUV format and output three **`GrayImage`**s that describe an image in RGB format.

## Image and video coding

**`codef`** This galaxy encodes a sequence of images using motion compensation, a discrete-cosine transform, quantization, and run-length encoding. The outputs are split into high priority and low priority, where corruption of the low priority data will impact the image less.

**codei** This galaxy inverts the encoding of the codef block, and outputs a reconstructed image sequence.

**DCTImage** This star takes a **GrayImage** input particle, computes the discrete cosine transform (DCT), and outputs a **DCTImage** particle.

**DCTImageInv** This star takes a **DCTImage** input and does inverse discrete cosine transform (DCT) coding and outputs a **GrayImage**.

**DCTImageCode** This star takes a **DCTImage** input particle, inserts "start of block" markers, run-length encodes it, and outputs the modified **DCTImage** particle. For the run-length encoding, all values with absolute value less than *Thresh* are set to 0.0, to help improve compression.

Run lengths are coded with a "start of run" symbol and then an (integer) run-length. *HiPri* DCT coefficients per block are sent to "hiport", the high-priority output. The remainder of the coefficients are sent to "loport", the low-priority output.

**Note:** This differs from the **SDFRunLen** star in that this block processes **DCTImages**, not **GrayImages**.

**DCTImageCodeInv** This star reads two coded **DCTImage**s (one high priority and one low-priority), inverts the run-length encoding, and outputs the resulting **DCTImage**. Protection is built in to avoid crashing even if some of the coded input data is affected by loss.

**Note:** This star is different from the **SDFRunLenInv** star. This one works on **DCTImage**s, not **GrayImage**s.

**DPCMImage** This star implements differential pulse code modulation of an image. If the "past" input is not a **GrayImage** or has size 0, pass the "input" directly to the "output". Otherwise, subtract the "past" from the "input" (with leakage factor *alpha*) and send the result to "output".

**DPCMImageInv** This star inverts differential pulse code modulation of an image. If the "past" input is not a **GrayImage** or has size 0, pass the "diff" directly to the "output". Otherwise, add the "past" to the "diff" (with leakage factor *alpha*) and send the result to "output".

**MotionCmp** If the "past" input is not a GrayImage (e.g. "dummyMessage"), copy the "input" image unchanged to the "diffOut" output and send a null field of motion vectors to the "mvOut" output. This should usually happen only on the first firing of the star.

For all other inputs, perform motion compensation and write the difference frames and motion vector frames to the corresponding outputs.

This star can be used as a base class to implement slightly different motion compensation algorithms. For example, synchronization techniques can be added or reduced-search motion compensation can be performed.

**`MotionCmpInv`** For NULL inputs on "mvIn" (motion-vector-in), copy the "diffIn" input unchanged to the output and discard the "pastIn" input. (A NULL input usually indicates the first frame of a sequence.)

For non-NULL "mvIn" inputs, perform inverse motion compensation and write the result to "output".

**`RunLenImage`** Accept a **`GrayImage`** and run-length encode it. All values closer than *Thresh* to *meanVal* are set to *meanVal* to help improve compression.

Run lengths are coded with a start symbol of *meanVal* and then a run-length between 1 and 255. Runs longer than 255 must be coded in separate pieces.

**`RunLenImageInv`** Accept a **`GrayImage`** and inverse run-length encode it.

**`ZigZagImage`** This star zig-zag scans a **`DCTImage`** and outputs the result. This is useful before quantization.

**`ZigZagImageInv`** This star inverse zig-zag scans a **`DCTImage`**.

### Miscellaneous image blocks

**`AddMotionVecs`** Over each block in the input image, superimpose an arrow indicating the size and direction of the corresponding motion vector.

**`DiffImage`** Accept two black-and-white images from input **`GrayImage`**s, take their absolute difference, scale the absolute difference, and pass the result to the output.

**`MedianImage`** Accept an input **`GrayImage`**, median-filter the image, and send the result to the output. Filter widths of 1, 3, 5 work well. Any length longer than five will take a long time to run.

Median filtering is useful for removing impulse-type noise from images. It also smooths out textures, so it is a useful pre-processing step before edge detection. It removes inter-field flicker quite well when displaying single frames from a moving sequence.

## 4.2.10 Communication stars

The limited set of communication stars that have been developed are shown in figure 4-12, and summarized below. Many of these are galaxies, and should be viewed as examples of systems that a user can create.

### Sources and encoders

**bits** Produce "0" with probability *probOfZero*, else produce "1".

**QAM4** Encode an input bit stream into a 4-QAM (or 4-PSK) complex symbol sequence.

**QAM16** Encode an input bit stream into a 16-QAM complex symbol sequence.

**ask** Generate an amplitude-shift keyed (ASK) signal encoding a random sequence of bits using a raised-cosine pulse.

**ADPCMCoder** Implement adaptive differential pulse code modulation using an **LMS** star. Both the quantized and unquantized prediction-error signals are available as outputs.

**ADPCMDecoder** Decode the quantized prediction error signal produced by **ADPCMCoder.**

**ADPCMToBits** Convert the quantized prediction error of the **ADPCMCoder** galaxy into a bit stream. The quantized prediction error has 16 possible levels, so this galaxy produces 4 bits in each output sample.

**ADPCMFromBits** Convert a bit stream encoded with **ADPCMToBits** back to float-

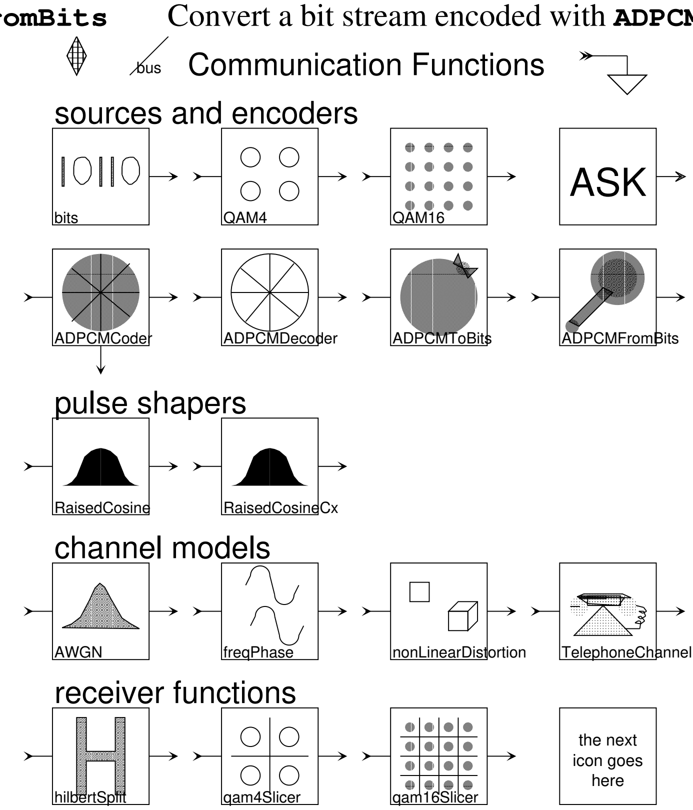



**FIGURE 4-12:** Communication stars in the SDF domain.

ing-point values. The 4 low-order bits of the input integer are changed to 1 of 16 floating-point values scaled by *range*.

## Pulse shapers

**`RaisedCosine`** An **`FIR`** filter with a magnitude frequency response shaped like the standard raised cosine or square-root raised cosine used in digital communication. By default, the star upsamples by a factor of 16, so 16 outputs will be produced for each input unless the *interpolation* parameter is changed.

**`RaisedCosineCx`** This galaxy uses the **`RaisedCosine`** star to implement an FIR filter for complex inputs with a raised cosine or square-root raised cosine transfer function.

## Channel models

**`AWGNchannel`** Model an additive Gaussian white noise channel with optional linear distortion.

**`freqPhase`** Impose frequency offset and/or phase jitter on a signal in order to model channels, such as telephone channels, that suffer these impairments.

**`nonLinearDistortion`**
Generate second and third harmonic distortion by squaring and cubing the signal, and adding the results in controlled proportion to the original signal.

**`telephoneChannel`**
Simulate impairments commonly found on a telephone channel, including additive Gaussian noise, linear and nonlinear distortion, frequency offset, and phase jitter.

## Receiver functions

**`hilbertSplit`** This galaxy implements a phase splitter, in which the real-valued input signal is converted to an (approximate) analytic signal. The signal is filtered by the Hilbert block to generate the imaginary part of the output, while the real part is gotten by creating a matching delay.

**`qam4Slicer`** This galaxy implements a slicer (decision device) for a 4-QAM (or equivalently, 4-PSK) signal. The output decision is a complex number with +1 or -1 for each of the real or imaginary parts.

**`qam16Slicer`** This galaxy implements a slicer (decision device) for a 16-QAM complex signal. The output decision is a complex number with +1, -1, +3, or -3 for each of the real or imaginary parts.

### 4.2.11 Tcl stars

Most of the stars that interface to Tcl appear in palettes that reflect their function. For instance, all the stars beginning with **`Tk`** in the "sinks" palette are actually Tcl stars derived from **`TclScript`**. This is the most generic Tcl star, with no useful function on its own. It must have a Tcl script associated with it to make it useful. There is a chapter of the Programmer's Manual of the The Almagest devoted to how to write such scripts. The complete palette of Tcl stars, which includes many stars that also appear in other palettes, is shown in figure 4-13.

**`TclScript`** Invoke a Tcl script that can optionally define a procedure that is invoked every time the star fires. That procedure can read the star's inputs and update the value of the outputs.

### 4.2.12 Matrix stars

The stars in the matrix palette (figure 4-14) operate on particles that represent matrices with floating-point, fixed-point, complex, or integer entries. Most of the work is done in the

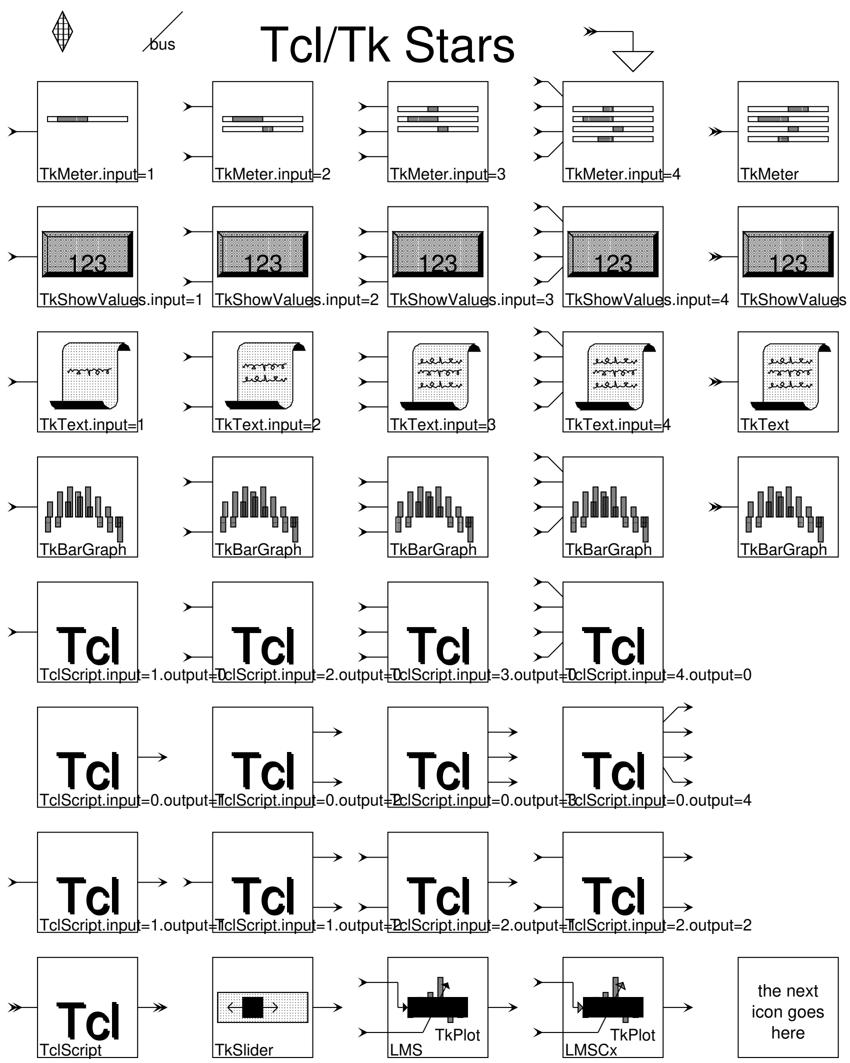



**FIGURE 4-13:** The Tcl/Tk palette includes many stars that also appear in other palettes.

underlying matrix classes, **`FloatMatrix`**, **`ComplexMatrix`**, **`FixMatrix`**, and **`IntMatrix`**. These classes are treated as ordinary particles. In Pigi, matrix types are indicated with thick terminal stems, where the color of the terminal stem corresponds to the data type of the matrix elements.

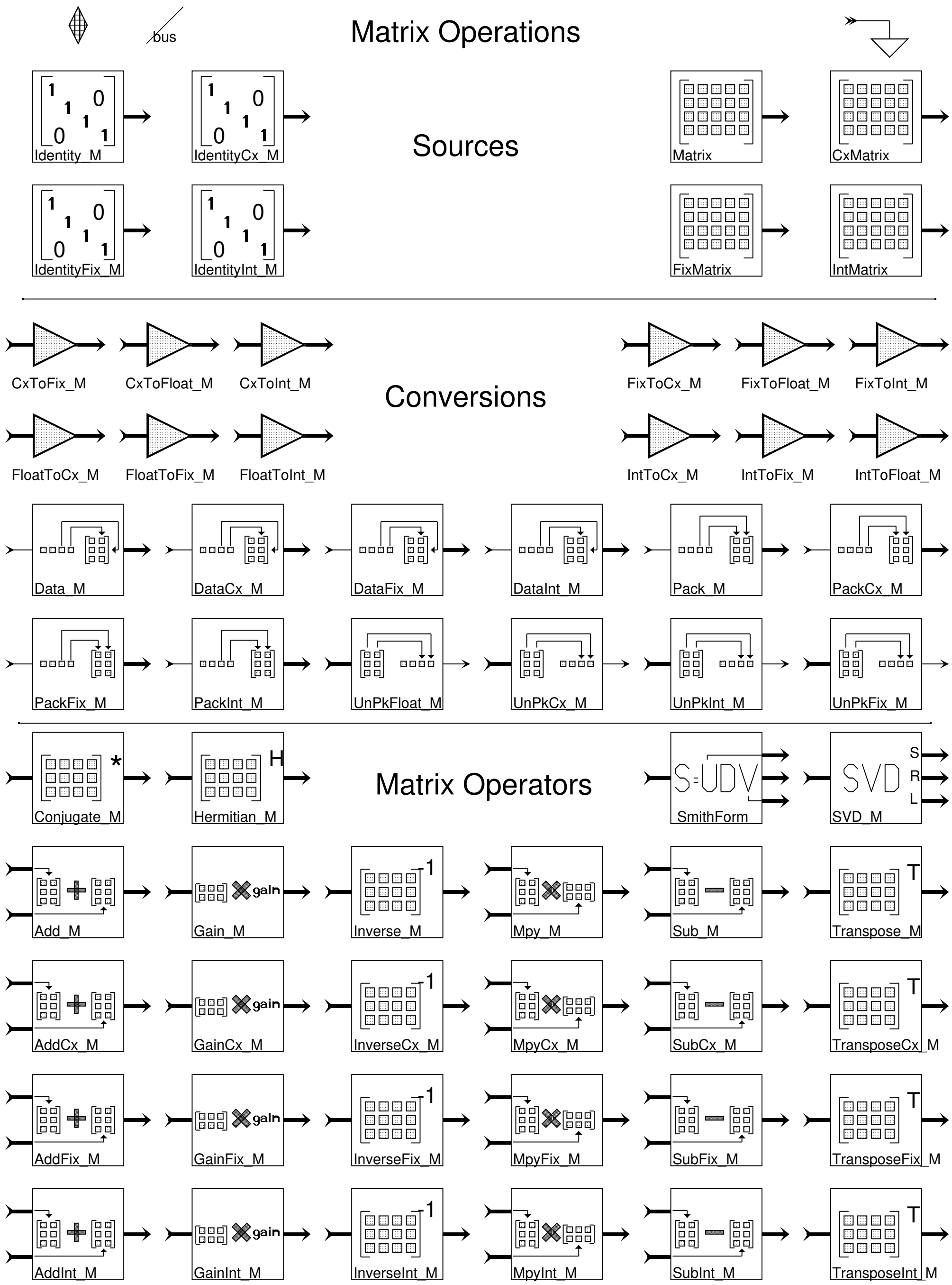



**FIGURE 4-14:** The matrix palette in the SDF domain. These stars operate on matrices encapsulated in a particles.

### Sources

Each of the following stars appears in four versions, one for each data type. Only the floating-point version is listed here.

**`Identity_M`** Output a floating-point identity matrix.

**`Matrix`** Produce a matrix with floating-point entries. The entries are read from the array parameter *FloatMatrixContents* in rasterized order: i.e. for a MxN matrix, the first row is filled from left to right using the first N values from the array.

### Type conversion

The following type conversions construct a new matrix of the destination type by converting each element of the old matrix as it is copied to the new one. For **`FixMatrix`** types, the precision is specified as a parameter of the conversion star. The actual conversions are implemented using the cast conversion in the underlying class, except for the conversions to the **`FixMatrix`** type which are more complex because they involve possible changes in precision and require a rounding option. The stars provided are:

**`CxToFix_M`**

**`CxToFloat_M`**

**`CxToInt_M`**

**`FixToCx_M`**

**`FixToFloat_M`**

**`FixToInt_M`**

**`FloatToCx_M`**

**`FloatToFix_M`**

**`FloatToInt_M`**

**`IntToCx_M`**

**`IntToFix_M`**

**`IntToFloat_M`**

The following conversions perform more interesting functions. They also come in four versions, one for each data type, and again we only list the floating-point version.

**`Data_M`** Generate a floating-point data matrix X, with dimensions (numRows,numCols), from a stream of numRows+numCols-1 input particles organized as shown below:

$$X = \begin{bmatrix} x(M-1) & x(M-2) & \dots & x(0) \\ x(M) & x(M-1) & \dots & x(1) \\ \dots & \dots & \dots & \dots \\ x(N-1) & x(N-2) & \dots & x(N-M) \end{bmatrix}$$

Here numRows = N-M+1 and numCols = M. This Toeplitz matrix is the form of the matrix that is required by the **`SVD_M`** star, among others.

**`Pack_M`** Produce a matrix with floating-point entries constructed from floating-point input particles. The inputs are put in the matrix in rasterized order, e.g. for a MxN matrix, the first row is filled from left to right using the first N input particles.

**`UnPk_M`** Read a floating-point matrix and output its elements, row by row, as a stream of floating-point particles.

### Matrix operations

The following blocks are functions defined only for the **`ComplexMatrix`** data type.

**`Conjugate_M`** Conjugate a matrix.

**`Hermitian_M`** Perform a Hermitian transpose (conjugate transpose) on the input matrix.

The following blocks also appear in the signal processing palette.

**`SmithForm`** Decompose an integer matrix *S* into one of its Smith forms *S* = *U D V*, where *U*, *D*, and *V* are simpler integer matrices. The Smith form decomposition for integer matrices is analogous to singular value decomposition for floating-point matrices.

**`SVD_M`** Compute the singular-value decomposition of a Toeplitz data matrix *A* by decomposing *A* into *A* = *UWV'*, where *U* and *V* are orthogonal matrices, and *V'* represents the transpose of *V*. *W* is a diagonal matrix composed of the singular values of *A*, and the columns of *U* and *V* are the left and right singular vectors of *A*.

The following operations are arranged at the bottom of the palette with one row for each data type. We list below only the floating point data type, from left to right.

**`Add_M`** Add two floating-point matrices.

**`Gain_M`** Multiply a floating-point matrix by a scalar gain value.

**`Inverse_M`** Invert a square floating-point matrix.

**`Mpy_M`** Multiply two floating-point matrices *A* and *B* to produce matrix *C*. Matrix *A* has dimensions (*numRows*,*X*). Matrix *B* has dimensions (*X*,*numCols*). Matrix *C* has dimensions (*numRows*,*numCols*). The user need only specify *numRows* and *numCols*. An error will be generated if the number of columns in *A* does not match the number of rows in *B*.

**`Sub_M`** Subtract floating-point matrix B from A.

**`Transpose_M`** Transpose a floating-point matrix read as a single particle.

## 4.3 An overview of SDF demos

A rather large number of SDF demo programs have been developed. These can serve as valuable illustrations of the possibilities. Almost every star is illustrated in the demos. Because of the large number, the demos are organized into a set of palettes. A top-level palette, shown in figure 4-15, contains an icon for each demo palette.

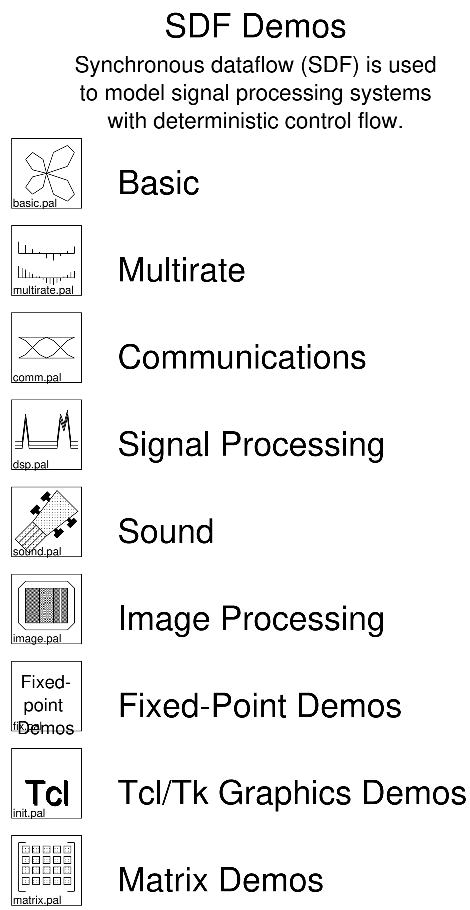



**FIGURE 4-15:** The top-level demo palette for the SDF domain.

### 4.3.1 Basic demos

These demos illustrate the use of certain stars without necessarily performing functions that are inherently interesting. The palette is shown in figure 4-16. The demos are described below from left to right, top to bottom.

**`butterfly`** This system uses sines and cosines to compute a curve known as the butterfly curve, invented by T. Fay. The curve is plotted in polar form.

**`comparison`** Compare two sinusoidal signals using the **`Test`** star.

**`complexExponential`** Generate and plot a complex exponential.

**`freqPhaseOffset`** Impose frequency jitter and phase offset on a sinusoid using the **`freqPhase`** SDF block.

**`gaussian`** Generate a Gaussian white noise signal, and plot its histogram and estimated autocorrelation.

**`integrator`** Demonstrate the features of the integrator star, such as limiting, leakage, and resetting.

**`muxDeMux`** Demonstrate the **`Mux`** and **`DeMux`** stars, which perform multiplexing and demultiplexing. Contrast with the **`scramble`** demo below.

**`pseudoRandom`** Generate a pseudo random sequence of zeros and ones using a maximal-length shift register and test its randomness by estimating it autocorrelation.

**`quantize`** Demonstrate the use of the **`Quantizer`** star.

**scramble** This system rearranges the order of samples of signal using the **Commutator** and **Distributor** stars. Note that because these are multirate stars, one iteration involves more than one sample. Contrast with the **muxDeMux** demo above.

**sinMod** Modulate a sinusoid my multiplying by another sinusoid.

**tbus** This is a simple illustration of the bus facility in Ptolemy, in which multiple signals are combined onto a single graphical connection.

### 4.3.2 Multirate demos

The demos with icons shown in figure 4-17 illustrate synchronous dataflow principles as applied to multirate signal processing problems. These are arranged roughly in order of sophistication.

**broken** This is an example of an inconsistent SDF system. It fails to run, generating an error message instead.

**interp** This system uses an **FIR** filter to upsample by a factor of 8 and linearly interpolate between samples.

**multirate** Upsample a sinusoidal signal by a ratio of 5/2 using a polyphase lowpass interpolating **FIR** filter.

**analytic** This system uses a **FIRCx** filter to reduce the sample rate of a sinusoid by a factor of 8/5, and at the same time produce a complex approximately analytic signal (one that has no negative frequency components).

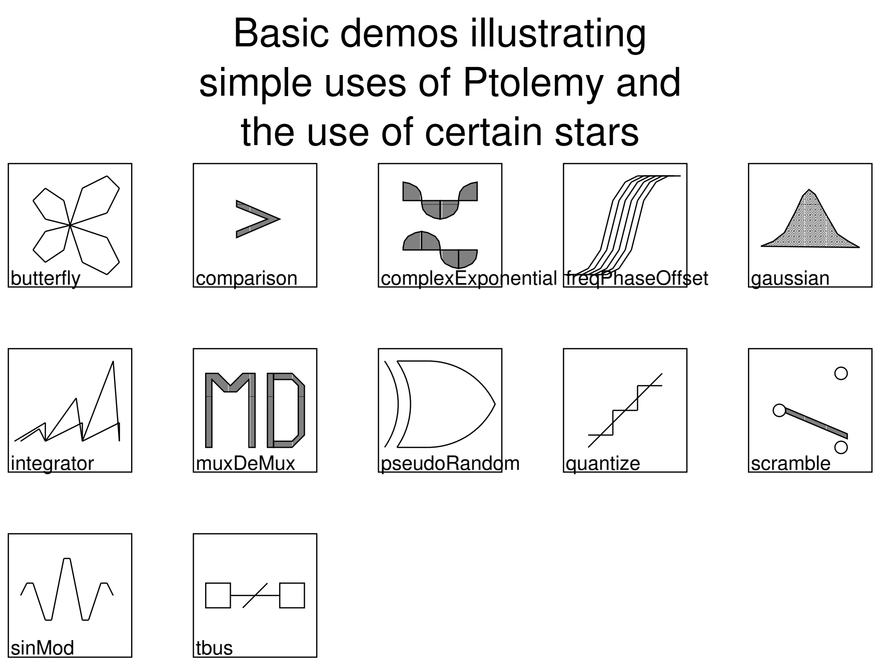



**FIGURE 4-16:** Palette for a set of basic demos for the SDF domain.

**filterBank** This universe implements an eight-level perfect reconstruction one-dimensional filter bank based on the biorthogonal wavelet decomposition.

**downSample** Convert from the digital audio tape sampling rate (48 kHz) to the compact disc sampling rate (44.1 kHz). The conversion is performed in multiple stages for better performance.

**upSample** Convert from the compact disc sampling rate (44.1 kHz) to the digital audio tape sampling rate (48 kHz). The conversion is performed in multiple stages for better performance.

**filterBank-NonUniform**
This universe implements a simple split of the frequency domain into two non-uniform frequency bands.

### 4.3.3 Communications demos

The palette shown in figure 4-18 points to some examples of digital communication systems and channel simulators. The palette is fairly sparse, since we have not yet undertaken

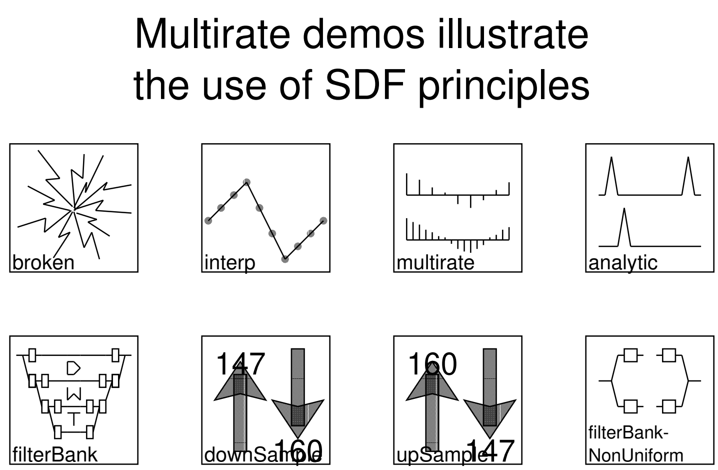



**FIGURE 4-17:** Multirate signal processing demos in the SDF domain.

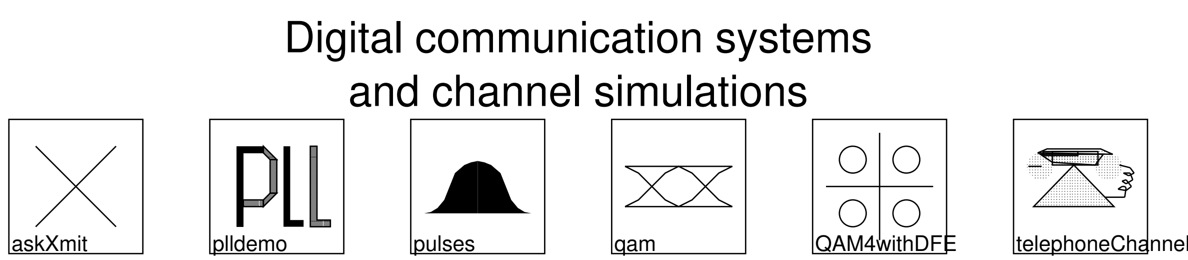



**FIGURE 4-18:** Communication system demos in the SDF domain.

a systematic effort to build up applications of this type.

**askXmit** Synthesize an amplitude-shift keyed (ASK) signal with a 100% excess bandwidth raised cosine pulse.

**plldemo** Simulate a fourth-power optical phase-locked loop with laser phase noise and additive Gaussian white noise operating on a complex baseband envelope model of the signal.

**pulses** Generate raised cosine and square-root raised cosine pulses and demonstrates matched filtering with the square-root raised cosine pulse.

**qam** This universe produces a 16-point quadrature amplitude modulated (QAM) signal and displays the eye diagram for the in-phase part, the constellation, and the modulated transmit signal.

**QAM4withDFE** This is a model of a digital communication system that uses quadrature amplitude modulation (QAM) and a fractionally spaced decision feedback equalizer.

**telephoneChannelTest**
Assuming a sampling rate of 8 kHz, a sinusoid at 500 Hz is transmitted through a simulation of a telephone channel with additive Gaussian noise, nonlinear distortion, and phase jitter.

### 4.3.4 Digital signal processing demos

A fairly large number of signal processing applications are represented in the palette shown in figure 4-19. Several of these serve as good examples to help in solving the exercises included at the end of the chapter.

**adaptFilter** An **LMS** adaptive filter converges so that its transfer function matches that of a fixed **FIR** filter.

**allPole** Two realizations of an all-pole filter are shown to be equivalent. One uses an **FIR** filter in a feedback path, the other uses the **BlockAllPole** star.

**animatedLMS** An **LMS** adaptive filter is configured as in the **adaptFilter** demo, but this time the filter taps are displayed as they adapt.

**animatedLMSCx** A complex LMS adaptive filter is configured as in the **adaptFilter** demo, but in addition, user-controlled noise is added to the feedback loop using an on-screen slider to control the amount of noise. The filter taps are displayed as they adapt.

**cep** Given the coefficients of any polynomial, this demo uses the cepstrum to find a minimum-phase polynomial. Thus, given the coefficients of the denominator polynomial of an unstable filter, this demo will compute the coefficients of a stable denominator polynomial that has the same magnitude frequency response.

**chaos** This is simple demonstration of chaos, in which the phase-space

| | |
|---|---|
| | plot of the famous Henon map is given. |
| **convolve** | Convolve two rectangular pulses in order to demonstrate the **Convolve** block. |
| **dft** | Compute a discrete Fourier transform of a finite signal using the **FFT** star. The magnitude and phase (unwrapped) are plotted. |
| **doppler** | A sine wave is subjected to four successive amounts of doppler shift. The doppler shift is accomplished by the **phaseShift** galaxy, which forms an analytic signal (using a Hilbert transform) that modulates a complex exponential. |
| **dtft** | Demonstrate the **DTFT** star, showing how it is different from the **FFTCx** star. Specifically, the range, number, and spacing of frequency samples is arbitrary. |
| **freqsample** | This system designs FIR filters using the frequency sampling method. Samples of the frequency response are converted into FIR filter coefficients. |
| **lattice** | Demonstrate the use of lattice filters to synthesize an auto- |

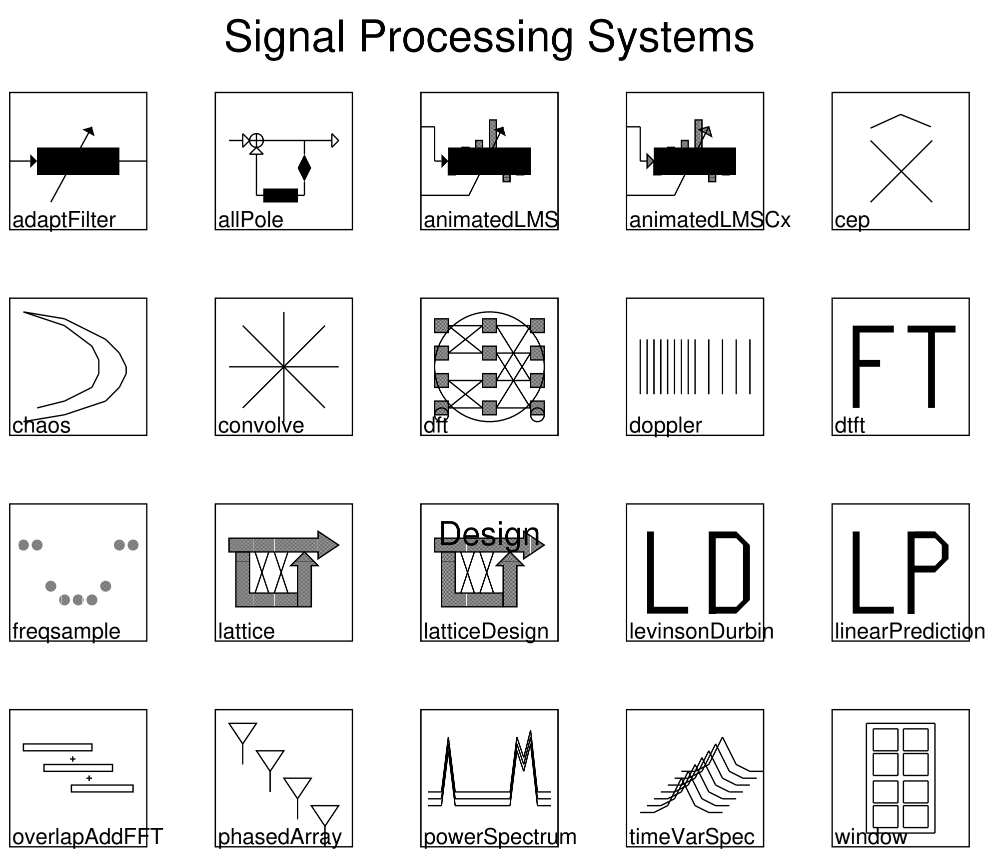



**FIGURE 4-19:** Signal processing applications in the SDF domain.

regressive (AR) random process.

**latticeDesign** Use of Levinson-Durbin algorithm to design a lattice filter with a specified transfer function.

**levinsonDurbin** Use the Levinson-Durbin algorithm to estimate the parameters of an AR process.

**linearPrediction** This demo performs linear prediction on a test signal consisting of three sinusoids in colored, Gaussian noise. Two mechanisms (Burg's algorithm and an **LMS** adaptive filter) for linear prediction are compared.

**overlapAdd/FFT** Convolution is implemented in the frequency domain using overlap and add.

**phasedArray** This demo simulates a plane wave approaching a phased array with four sensors. The plane wave approaches from angles starting from head on and slowly rotating 360 degrees. The response of the antenna is plotted as a function of direction of arrival in polar form.

**powerSpectrum** Compare three methods for estimating a power spectrum of a signal with three sinusoids plus colored noise. The three methods are the periodogram method, the autocorrelation method, and Burg's method.

**timeVarSpec** A time-varying spectrum is computed using the autocorrelation method and displayed using a waterfall plot.

**window** Generate and display four window functions and the magnitude of their Fourier transforms. The windows displayed are the Hanning, Hamming, Blackman, and steep Blackman.

### 4.3.5 Sound-making demos

The demos in the palette in figure 4-20 assume that a program called "ptplay" is in your path, and that it accepts data of an appropriate format and will play it over a workstation speaker at an 8kHz sample rate. If you are using a Sun SPARC Station, these conditions will most likely be satisfied, if your path is correct. The samples are written into a file before they are played. Since a large number of samples must be generated, these demos can take some

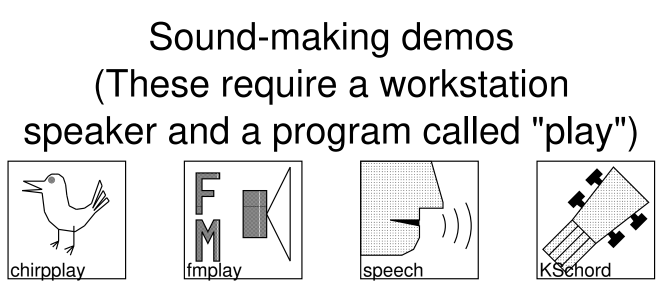



**FIGURE 4-20:** Sound-making demos in the SDF domain.

time to run. By contrast, the CGC domain has some audio demos that generate sounds in real time at 8kHz, assuming a reasonably fast workstation.

**chirpplay** Chirp generator that plays on the workstation speaker.

**KSchord** Simulation of plucked string sounds using the Karplus-Strong algorithm

**fmplay** Sound generator using FM modulation that plays on the workstation speaker.

**speech** This system reads a speech signal from a file, and encodes it at two bits per sample using adaptive differential pulse code modulation with a feedback-around-quantizer structure. It then reconstructs the signal from the quantized data. The original and reconstructed speech are played over the workstation speaker.

### 4.3.6 Image and video processing demos

The demos in figure all read images from files on the workstation disk, process them, and then display them. Some of the demos process short sequences of images, thus illustrating video processing in Ptolemy. They all use the image classes described in "Image processing stars" on page 4-23. The set of demos in this palette does not reflect the richness of possibilities. See the DE domain for more image and video signal processing applications in the context of packet-switched network simulations. The video display requires that the Utah Raster Toolkit be installed and available in the user's path.

**ColorImage** A universe that converts an RGB color image to YUV and back, and then displays it on the workstation screen.

**CompareMedian** A universe that median filters an image to reduce artifacts due to interleaved scanning in video sequences.

**DctImage** A universe that performs discrete cosine transform (DCT) coding of an image sequence.

**DpcmImage** A universe that performs differential pulse code modulation (DPCM) on an image sequence.

**MC_DCT** A universe that performs motion compensation and DCT encoding of video.

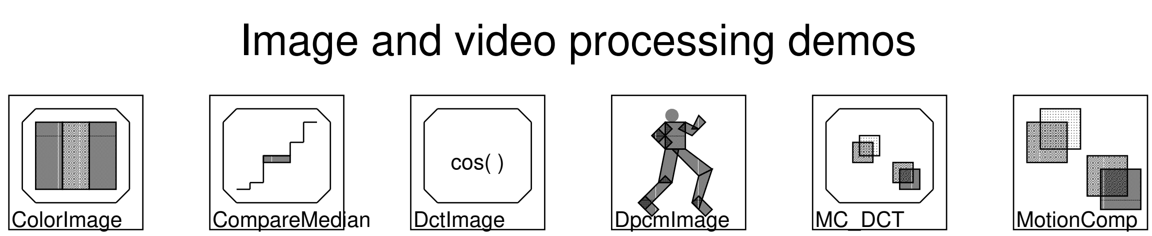



**FIGURE 4-21:** Image processing demos in the SDF domain.

**MotionComp** A universe that does motion compensation video coding.

### 4.3.7 Fixed-point demos

The demos shown in figure 4-22 illustrate the use of fixed-point stars in the SDF domain. These stars are used to model hardware implementation with finite precision.

**fixConversion** A simple demo illustrating the different masking options available.

**fixFIR** This demo shows the effect of filter tap precision on the filter response.

**fixIIR** A simple demo illustrating the sensitivity of different IIR filter structures to finite precision.

**fixIIRdf** A simple demo comparing a fourth-order direct-form IIR filter implemented with floating-point arithmetic to a similar filter implemented with fixed-point arithmetic.

**fixLimitCycle** This demo demonstrates the limit-cycle phenomenon due to finite precision effects.

**fixMpyTest** A simple demo showing that tests fixed-point multiplication over a range of numbers by comparing it against floating-point multiplication. The results should be the same.

**fixRamp** A simple demo illustrating the different overflow handling mechanisms.

### 4.3.8 Tcl/Tk demos

**animatedLMS** (See the signal processing section).

**animatedLMSCx** (See the signal processing section).

**sinWaves** Demonstrate **TkBarGraph** by generating and displaying a

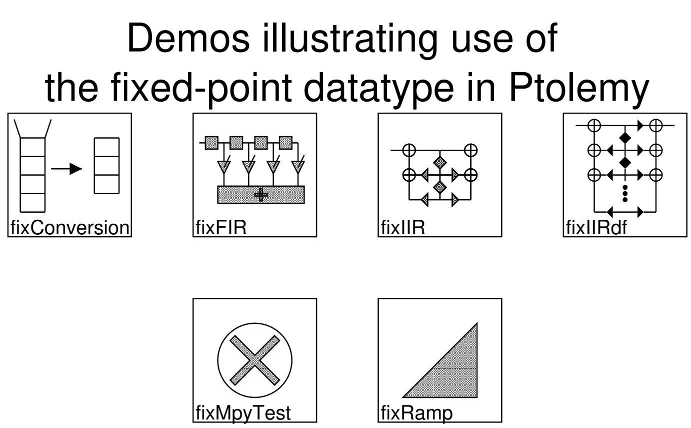



**FIGURE 4-22:** These demos illustrate fixed-point effects in signal processing systems.

complex exponential.

**tclScript** Demonstrate **TclScript** by generating two interactive X window follies that consist of circles that move in a playing field.

**tkMeter** Demonstrate **TkMeter** by creating three bar meters. The first oscillates sinusoidally. The second displays a random number between zero and one. The third displays a random walk.

**tkShowValues** Demonstrate **TkShowValues** and **TkText** by displaying the ASCII form of two ramp sequences.

**wildColors** Demonstrate **TkBarGraph** by creating a bar graph that simultaneously displays 12 random sequences.

### 4.3.9 Matrix demos

The systems in figure 4-23 demonstrate the use of matrix particles in Ptolemy. These particles are quite new, so the library of demonstrations is still fairly small.

**MatrixTest1** This demo shows the use of the Matrix stars that have one input. These include the operations inverse, transpose, and multiply by a scalar gain for all matrix types. Also conjugate and hermitian transpose are available for the complex matrix type.

**MatrixTest2** This demo shows the use of some simple Matrix stars with two inputs. These include multiply, add, and subtract.

**MatrixTest3** This demo shows the use of the Matrix conversion stars. These convert between the scalar particles and the matrix particles as well as between the various matrix types.

**Kalman_M** This demo compares the convergence properties of a Kalman filter to those of an LMS filter when addressing the problem of adaptive equalization of a process in noise.

**SVD_MUSIC_1** This demo shows the use of singular-value decomposition

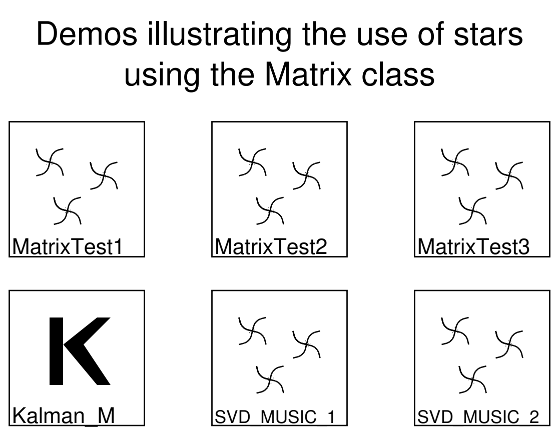



**FIGURE 4-23:** Demonstrations of matrix operations in Ptolemy.

(SVD) and the Multiple-Signal Characterization (MUSIC) algorithm to identify the frequency of a single sinusoid in a signal that has two different signal to noise ratios.

**SVD_MUSIC_2** This demo shows the use of the Multiple-Signal Characterization (MUSIC) algorithm to identify three sinusoids in noise that have frequencies very close to each other.

## 4.4 Targets

As is typical of simulation domains, the SDF domain does not have many targets. The three included in the 0.5 distribution are described below. To choose one of these targets, with your mouse cursor in a schematic window, execute the Edit:edit-target command (or just type "T"). You will get a list of the available **Target**s in the SDF domain. The "default-SDF" target is normally selected by default. When you click "OK", dialog box appears with the parameters of the target. You can edit these, or accept the defaults. The next time you run the schematic, the selected target will be used.

### 4.4.1 Default SDF target

The default SDF target has a simple set of options:

**logFile** The name of a file into which the scheduler will write the final schedule.

**loopScheduler** A Boolean specifying whether to attempt to compact the schedule for forming looping structure (see below).

**schedulePeriod** A floating-point number defining the time taken by one iteration through the schedule. This is not needed for pure SDF systems, but if SDF systems are mixed with timed domains, such as DE, then this will determine the amount of simulated time taken by one iteration.

The SDF scheduler determines the order of execution of stars in a system at start time. It performs most of its computation during its **setup()** phase. The default scheduler (gotten if **loopScheduler** is NO) exactly implements the method described in [Lee87a] for sequential schedules. If there are sample rate changes in a program graph, some parts of the graph are executed multiple times. The default scheduler does not attempt to generate loops; it simply generates a linear list of blocks to be executed. For example, if star A is executed 100 times, the generated schedule includes 100 instances of A. A loop scheduler will include in its "looped" schedule (where possible) only one instance of A and indicate the repetition count of A, as in (100 A). For simulation, a long unstructured list might be tolerable, but not in code generation (The SDF schedulers are also used in the code generation for a single processor target).

Neglecting the overhead due to each loop, an optimally compact looped schedule is one that contains only one instance of each actor, and we refer to such schedules as *single appearance schedules*. For example, the looped schedule (3 A)(2 B), corresponding to the firing sequence AAABB, is a single appearance schedule, whereas the schedule AB(2 A)B is not.

By setting the **loopScheduler** target parameter to YES, we can select a scheduler developed by J. Buck. Before applying the default scheduling algorithm, this algorithm collects actors into a hierarchy of clusters. This clustering algorithm consists of alternating a "merging" step and a "looping" step until no further changes can be made. In the merging step, blocks connected together are merged into a cluster if there is no sample rate change between them and the merge will not introduce deadlock. In the looping step, a cluster is looped until it is possible to merge it with the neighbor blocks or clusters. Since this looping algorithm is conservative, some complicated looping possibilities are not always discovered.

### 4.4.2 The SDF LoopTarget

A more sophisticated looping algorithm, available in an alternative target called the **LoopTarget**, was developed by adding postprocessing steps to the above algorithm to handle cases that it cannot handle. For lack of a better name, we call this technique "SJS scheduling", for the first initials of the designers (Shuvra Bhattacharyya, Joe Buck, and Soonhoi Ha). We apply the previously described scheduling algorithm as the first pass. In the second pass, we attempt to decompose the graph into a hierarchy of acyclic graphs [Bha93b], for which a compact looped schedule can easily be constructed. Cyclic subgraphs that cannot be decomposed by this method, called *tightly interdependent subgraphs*, are expanded to acyclic precedence graphs in which looping structures are extracted by the techniques developed in [Bha94a] and extensions to these techniques developed by Ha. This scheduling option is selected when the **loopTarget** is chosen instead of the default SDF target. The target options are:

**logFile**

**schedulePeriod**

They have the same interpretation as for the default target.

When there are sample rate changes in the program graph, the default SDF scheduler may be much slower than the loop schedulers, and in code generation, the resulting schedules may lead to unacceptably large code size. Buck's scheduler provides a fast way to get compact looped schedules for many program graphs, although there are no guarantees of optimality. The somewhat slower SJS scheduler is guaranteed to find a single appearance schedule whenever one exists [Bha93c]. Furthermore, a schedule generated by the SJS scheduler contains only one instance of each actor that is not contained in a tightly interdependent subgraph.

The looped result can be seen by setting the **logFile** target parameter. That file will contain all the intermediate procedures of looping and the final scheduling result. The loop scheduling algorithms are usually used in code generation domains, not in the simulation SDF domain. Refer to the CG domain documentation for a detailed discussion.

### 4.4.3 Compile-SDF target

A third target in the SDF domain, called **compile-SDF**, is experimental; it has a number of known problems, so it should be viewed at this time as a concept demonstration only. We have chosen to release it in this preliminary state so that it could serve as a concept demonstration. Instead of executing a simulation by invoking the **go()** methods of stars from within the Ptolemy process, it generates a C++ program that implements the universe, links it with appropriate parts of the Ptolemy kernel, and then invokes that system. The schedule is constructed statically, so the generated program has no scheduler linked in. Instead, the generated code directly invokes the **go()** methods of the stars. The target parameters are:

**directory** The directory into which to place the generated code. The default value is **$HOME/PTOLEMY_SYSTEMS**.

**LoopingLevel** If the value is "0", no attempt will be made to construct a looped schedule. This can result in very large programs for multirate systems, since the invocation of the go() methods will be in line. Setting the level to "1" invokes a quick a simple loop scheduler. Setting it to "2" invokes the more sophisticated SJS loop scheduler, which can take considerably more time to execute.

If you wish to try the SDF target, open any of the basic SDF demos in figure 4-16, change to the **compile-SDF** target, and run the system. You can then examine the source code and makefile the are placed in the specified directory. An executable with the same name as the name of the demo will also be placed in that directory. This is a stand-alone executable that does not require any part of the Ptolemy system to run. For example, if you choose the **butterfly** demo in figure 4-16, your destination directory will contain the following files:

```
butterfly
butterfly.cc
code.cc
make.template
makefile
```

The first of these is executable. Try executing it. You can modify the number of sample points generated using a command-line argument. For example, to generate 1,000 points instead of 10,000, type

```
butterfly 1000
```

The **compile-SDF** target is an example of a code-generation **Target** within the SDF domain. So, in a very fundamental way, a **Target** defines the way a system is executed. The default target is essentially an interpreter. The **compile-SDF** target synthesizes a standalone program and then executes it on the native workstation. It should be viewed merely as an example of the kinds of extensions users can build. More elaborate targets parallelize the code and execute the resulting programs on remote hardware. Targets can be defined by users and can make use of existing Ptolemy schedulers. Knowledgeable users can also define their own schedulers.

The compile-SDF target has a number of known problems:

- The resulting C++ program is unnecessarily large (a minimum of about half a megabyte) because many unnecessary Ptolemy objects get linked in. You can create a much leaner program using the CGC domain.
- Error messages during the compile or run are sent to the standard output rather than displayed in a window on the screen.
- Currently, the target does not invoke the **begin()** methods of the stars.

- If you specify a relative directory for the destination directory, instead of the absolute directory as done in the default, then the location of the directory will be relative to the current working directory of the Ptolemy system. It is easy to lose track of what that is.
- Generating code can take quite a bit of time, particularly if a multirate system is used with the **LoopingLevel** parameter set to “0”. Unfortunately, there is no convenient way to interrupt the code generation process.
- Implicit forks are not currently correctly handled. Consequently, whereas the target works for simple systems, more elaborate systems inevitably cause problems.
- The bus icon is not supported.

We would welcome any assistance in fixing these problems.

## 4.5 Exercises

The exercises in this section were developed by Alan Kamas, Edward Lee, and Kennard White for use in the undergraduate and graduate digital signal processing classes at U. C. Berkeley. If you are assigned these exercises for a class, you should turn in printouts of well-labeled schematics, showing all non-default parameter values, and printouts of relevant plots. Combining multiple plots into one can make comparisons more meaningful, and can save paper. Use the **XMgraph** star with multiple inputs.

### 4.5.1 Modulation

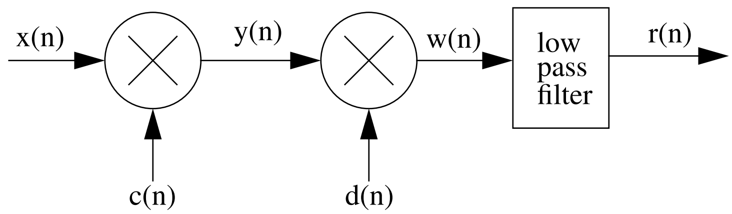
This problem explores amplitude modulation (AM) of discrete-time signals. It makes extensive use of FFTs. These will be used to approximate the discrete-time Fourier transform (DTFT). In subsequent exercises, we will study artifacts that can arise from this approximation. For our purposes here, the output of the **FFTCx** block will be interpreted as samples of the DTFT in the interval from 0 (d.c.) to $2\pi$.

Frequencies in many texts are normalized. To make this exercise more physically meaningful, you should assume a sampling frequency of 128kHz ($T = 7.8\mu\text{sec}$ sampling period). Thus the 0 to $2\pi$ range of frequencies (in radians per sample) translates to a range of 0 to 128kHz. On your output graphs, you should clearly label the units of the x-axis. The *xUnits* parameter of the **XMGraph** star can be used to do this. If the FFT produces $N = 256$ samples, representing the range from 0 to $f_s = 128$ kHz., then *xUnits* should be $f_s/N = 500$ Hz. Thus each sample out from the FFT will represent 500Hz. Keep in mind that a DTFT is actually periodic, and that only one cycle will be shown.

With default parameters, the **FFTCx** star will read 256 input samples and produce 256 complex output samples. This gives adequate resolution, so just use the defaults for this exercise. The section “Iterations in SDF” on page 4-3 will tell you, for instance, that you should run your systems for one iteration only. The section “Particle types” on page 3-17 explains how to properly manage complex signals. For this exercise, you should only plot the magnitude of the **FFTCx** output, ignoring the phase.

The overall goal is to build a modulation system that transmits a speech or music signal $x(n)$ using AM modulation. The transmitted signal is $y(n)$. The receiver demodulates y(n) to get the recovered signal $r(n)$. The system is working if $r(n) = x(n)$. Commercial AM radio uses carrier frequencies from 500kHz to 2MHz; however, we will use carriers around 32kHz.

This makes the results of the modulation easier to see. The system you will develop (after several intermediate phases) is shown below:




1. The first task is to figure out how to use the **FFTCx** star to plot the magnitude of a DTFT. Begin by generating a signal where you know the DTFT. Use the **Rect** star to generate a rectangular pulse

$$x(n) = \sum_{k=0}^{M} \delta(n-k)$$

   for $M = 4$ and $M = 10$. Plot the magnitude of the DTFT. It would be a good idea at this point to make a galaxy that will output the magnitude of the DTFT of the input signal. Be sure the axis of your graph is labeled with the frequencies in Hz, assuming a sampling frequency of 128kHz.

2. The signal generated above does not have narrow bandwidth. The next task will be to generate a signal $x(n)$ with narrower bandwidth so that the effects of modulating it can be seen more clearly and so there are fewer artifacts. A distinctive and convenient lowpass signal can be generated by feeding an impulse into the **RaisedCosine** star (found in the "communications" palette). Set the parameters of the **RaisedCosine** star as follows:
   *length*: 256
   *symbol_interval*: 8
   *excessBW*: 0.5
   Leave the *interpolation* parameter on its default value. The detailed functionality of this star is not important: we are just using it to get a signal we can work with conveniently. Plot the time domain signal and its magnitude DTFT. What is the bandwidth (single-sided), in Hz, of the signal? Use the -6dB point (amplitude at 1/2 of the peak) as the band edge. The signal was chosen to have roughly the bandwidth of a typical AM broadcast signal.

3. The next task is to modulate the signal $x(n)$ generated in part (2) with a sine wave. Construct a 32 kHz sine wave using the **singen** galaxy and let it be the carrier $c(n)$ ; then produce $y(n) = x(n)c(n)$ . Graph the DTFT of $y(n)$ . What is the bandwidth of $y(n)$ ? Change the carrier to 5kHz, and graph the FFT of y(n). Explain in words what has happened. Keep the carrier at 5kHz, and determine what the largest possible bandwidth is for $x(n)$ so that $y(n)$ will not have any significant distortion.

4. The next step it to build the demodulator. First multiply again by the same carrier, $d(n) = c(n)$ , and plot the magnitude DTFT of the result. Explain in words what about this spectrum is directly attributable to the discrete-time nature of the problem. In other words, what would be different if this problem were solved in continuous time?

5. To complete the demodulation, you need to filter out the double frequency terms. Use the **FIR** filter star with its default coefficients. This is not a very good lowpass filter, but it is a lowpass filter. Explain in words exactly how the resulting signal is different from the original baseband signal. How would you make it more like the original? Do you think it is enough like the original to be acceptable for AM broadcasting?

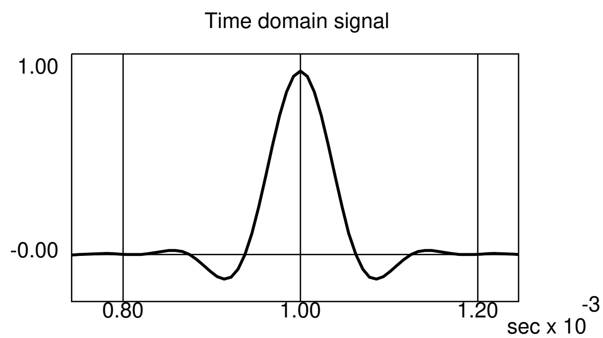
### 4.5.2 Sampling and multirate

This exercise explores sampling and multirate systems. As with the previous exercise, this one makes extensive use of FFTs to approximate the DTFT in the interval from 0 (d.c.) to $2\pi$ (normalized) or the sampling frequency $f_s$ (unnormalized).

1. The first task is to generate an interesting signal that we can operate on. We will begin with the same signal used in the previous exercise, generated by feeding an impulse into the **RaisedCosine** star. Set the parameters of the **RaisedCosine** star as follows:

   *length*: 256
   *symbol_interval*: 8
   *excessBW*: 0.5
   *interpolation*: 1

   Unlike the previous exercise, you should not leave the *interpolation* parameter on its default value. The time domain should look like the following (after zooming in on the central portion):

   Assume as in the exercise “Modulation” on page 4-46 a sampling frequency of 128kHz. Use the **FFTCx** to compute and plot the magnitude DTFT, properly labeled in absolute frequency. In other words, instead of the normalized sampling frequency $2\pi$, use the actual sampling frequency, 128kHz. Carefully and completely explain in words what would be different about this plot if the signal were a continuous-time signal and the plot of the spectrum were its Fourier transform instead of a DTFT.

2. Subsample the above signal at 64kHz, 32kHz, and 16kHz. To do this, use the **DownSample** star (in the “control” palette) with downsampling factors of 2, 4, and 8. Compare the magnitude spectra. It would be best to plot them on the same plot. To do this, you will need to keep the number of samples consistent in all signal paths. Since the **DownSample** star produces only one sample for every $N$ it consumes, the **FFTCx** star that gets its data should have its *size* parameter proportional to $1/N$ for each path.

   **Warning:** If you fail to make the numbers consistent, you will either get an error message, or your system will run for a very long time. Please be sure you understand synchronous dataflow. Read “Iterations in SDF” on page 4-3.

Answer the following questions:

a. Which of the downsampled signals have significant aliasing distortion?

b. What is the smallest sample rate you can achieve with the downsampler without getting aliasing distortion?

3. The next task is to show that sometimes subsampling can be used to demodulate a modulated signal.

   a. First, modulate our "interesting signal" with a *complex exponential* at frequency 32kHz. The complex exponential can be generated using the **`expgen`** galaxy in the sources palette. Plot the magnitude spectrum, and explain in words how this spectrum is different from the one obtained in the exercise "Modulation" on page 4-46, which modulates with a cosine at 32kHz.

   b. Next, demodulate the signal by downsampling it. What is the appropriate downsampling ratio?

4. The next task is to explore upsampling.

   a. First, generate the signal we will work with by downsampling the original "interesting signal" at a 32kHz sample rate (a factor of 4 downsampling). Then upsample by a factor of 4 using the **`UpSample`** star. This star will just insert three zero-valued samples for each input sample. Compare the magnitude spectrum of the original "interesting signal" with the one that has been downsampled and then upsampled. Explain in words what you observe.

   b. Instead of upsampling with the **`UpSample`** star, try using the **`Repeat`** star. Instead of filling with zeros, this one holds the most recent value. This more closely emulates the behavior of a practical D/A converter. Set the *numTimes* parameter to 4. Compare the magnitude spectrum to that of the original signal. Explain in words the difference between the two. Is this a better reconstruction than the zero-fill signal of part (a)?

   c. Use the **`Biquad`** star with default parameters to filter the output of the **`Repeat`** star from part (b). Does this improve the signal? Describe in words how the signal still differs from the original.

### 4.5.3 Exponential sequences, transfer functions, and convolution

This exercise explores rational Z transform transfer functions.

1. Generate an exponential sequence $a^n u(n)$, with $a = 0.9$, and convolve it with a square pulse of width 10. For this problem, use the following brute-force method for generating the exponential sequence. Observe that

$$a^n = e^{ln(a^n)} = e^{n(ln(a))}.$$

   You can use the **`Const`** star to generate a constant $a$, feed that constant into the **`Log`** star, multiply it by the sequence $n \times u(n)$ generated using the **`Ramp`** star, and feed the result into the **`Exp`** star. For your display, try the following options to the **`XMgraph`** star: "-P -nl -bar".

2. A much more elegant way to generate an exponential sequence is to implement a filter with an exponential sequence as its impulse response. Generate the sequence

$$a^n u(n)$$

by feeding an impulse (**Impulse** star) into a first order filter (**IIR** star). Try various values for $a$, including negative numbers and values that make the filter unstable.

a. Let $h(n) = a^n u(n)$ and $x(n) = b^n u(n)$, where $a = 0.9$ and $b = 0.5$. Generate these two sequences using the method above, and convolve them using the convolver block. Now find $h(n) * x(n)$ *without* using a convolver block. Print your block diagram, and don't forget to mark the parameter values on it.

b. Given the Z transform

$$H(z) = \frac{1 - 0.995z^{-1}}{1 - 1.99z^{-1} + z^{-2}},$$

use Ptolemy to find and print the inverse Z transform $h(n)$. Find the poles and zeros of the transfer function and use them to explain the impulse response you observe.

3. Generate the following sequences:

$$x(n) = (0.95)^n \sin(0.1n)\, u(n)$$

$$y(n) = (0.95)^n \sin(0.2n)\, u(n),$$

where $u(n)$ is the unit step function. Estimate the peak value of each signal. Note that you can zoom in xgraph by drawing a box around the region of interest.

4. Given the following difference equation:

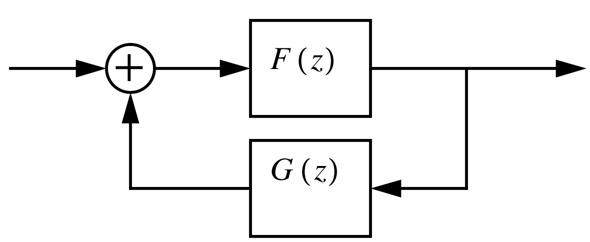
$$y(n) = 2x(n) + 0.75y(n-1) + 0.125y(n-2)$$

find $H(z)$ so that $Y(z) = X(z)H(z)$. Write it down. Use Ptolemy to generate a plot of $h(n)$. Plot $y(n)$ when $x(n)$ is a rectangular pulse of width 5. Assume $y(n) = 0$ for $n < 0$.

5. This problem explores feedback systems. An example of an "all-pole" filter is

$$H(z) = \frac{1}{1 - 2z^{-1} + 1.91z^{-2} - 0.91z^{-3} + 0.205z^{-4}}.$$

Although there are plenty of zeros (at $z = 0$), they don't effect the magnitude frequency response. Hence the name. Although this can be implemented in Ptolemy using the **IIR** star, you are to implement it using only one or more **FIR** star(s) in the standard feedback configuration:




Find $F(z)$ an $G(z)$ to get an overall transfer function of $H(z)$. Then implement it as a feedback system in Ptolemy and plot the impulse response. Is the impulse response infinite in extent?

**Note:** For a feedback system to be implementable in discrete-time, it must have at least one unit delay ($z^{-1}$) in the loop. Ptolemy needs for this delay to be explicit, not hidden in the tap values of a filter star. For this reason, you should factor a $z^{-1}$ term out of $G(z)$ and implement it using the delay icon (a small green diamond). Note that the delay is not a star, and is not connected as a star. It just gets placed on top of an arc, as explained in "Using delays" on page 3-41. Also note that Ptolemy requires you to use an explicit **`Fork`** (in the control palette) if you are going to put a delay on a net with more than one destination.

### 4.5.4 Linear phase filtering

You can compute the frequency response of a filter in Ptolemy by feeding it an impulse, and connecting the output to an **`FFTCx`** star. Recall that you will only need to run your system for **one** iteration when you are using an FFT, or you will get several successive FFT computations. The output of the FFT is complex, but may be converted to magnitude and phase using a complex to real (**`CxToRect`**) followed by a rectangular to polar (**`RectToPolar`**) converter stars. You can also examine the magnitude in dB by feeding it through the **`DB`** star before plotting it.

1. Build an FIR filter with real, symmetric tap values. Use any coefficients you like, as long as they are symmetric about a center tap. Look at the phase response. Is it linear, modulo $\pi$? Experiment with several sets of tap values, maintaining linear phase. Try long filters and short filters. Experiment with the phase unwrapper star (**`Unwrap`**), which attempts to remove the $2\pi$ ambiguity, keeping continuous phase. Choose your favorite linear-phase filter, and turn in the plots of its frequency response, together a plot of its tap values.
2. For the filter you used in (1), what is the group delay? How is the group delay related to the slope of the phase response?
3. Build an FIR filter with odd-symmetric taps (anti-symmetric). Find the phase response of this filter, and compare it to that in (1). Generate a sine wave (using the **`singen`** galaxy) and feed it into your filter. What is the phase difference (in radians) between the input cosine and the output? Try different frequencies.
4. Although linear phase is easy to achieve with FIR filters, it can be achieved with other filters using signal reversal. If you run the same signal forwards and backwards through the same filter, you can get linear phase. Given an input $x(n)$ and a filter $h(n)$, compute the output $y(n)$ as follows:

$$g(n) = h(n) * x(n)$$

$$r(n) = h(n) * g(-n)$$

$$y(n) = r(-n) .$$

   Obviously, this operation is not causal. Let $f(n)$ be such that

$$y(n) = f(n) * x(n) .$$

   Find $f(n)$ in terms of $h(n)$. If $h(n)$ is causal, will $f(n)$ also be causal? Find the frequency response $F(\omega)$, and express it in terms of $|H(\omega)|$ and $\angle H(\omega)$. It will help if you assume all signals are real.

5. All signals in Ptolemy start at time zero, so it is impossible to generate the signal $g(-n)$ used above. However, you can collect a block of samples and reverse them, getting $g(N-n)$, using the **Reverse** star. This introduces an extra delay of $N$ samples. Use a first-order IIR filter (with an exponentially decaying impulse response) to implement $h(n)$. First verify that the above methodology yields an impulse response that is symmetric in time. Then measure the phase response. You can use Ptolemy to adjust the computed phase output to remove the effect of the large delay offset (the center of your symmetric pulse is nowhere near zero). Compare your result against the theoretical prediction in (4).

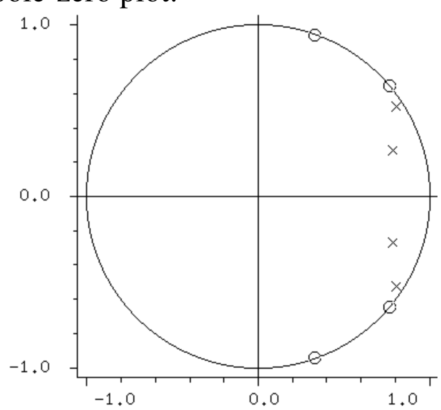
   **Hint:** You will want the block size of the **Reverse** star to match that used for the **FFTCx** star. Then just run the system through one iteration. Also, you should delay your impulse into the first filter by half the block size. This will ensure a symmetric impulse response, which is what you want for linear phase. The center of symmetry should be half the block size.

### 4.5.5 Coefficient quantization

1. You will experiment with the following transfer function:

$$H(z) = \frac{1 - 2.1872z^{-1} + 3.0055z^{-2} - 2.1872z^{-3} + z^{-4}}{1 - 3.1912z^{-1} + 4.1697z^{-2} - 2.5854z^{-3} + 0.6443z^{-4}},$$

   which has the following pole-zero plot:




   This is a fourth order elliptic filter.

   a. Implement this filter in the canonical direct form, or direct form II (using the **IIR** star). Plot the magnitude frequency response in dB, and verify that it is what you expect from the pole-zero plot.

   b. The transfer function can be factored as follows, where the poles nearest the unit circle and the zeros close to those poles appear in the second term:

$$H(z) = \left(\frac{1 - 0.6511z^{-1} + z^{-2}}{1 - 1.5684z^{-1} + 0.6879z^{-2}}\right)\left(\frac{1 - 1.5321z^{-1} + z^{-2}}{1 - 1.6233z^{-1} + 0.9366z^{-2}}\right).$$

   Implement this as a cascade of two second order sections (using two **IIR** stars). Verify that the frequency response is the same as in part (a). Does the order of the two second order sections affect the magnitude frequency response?

2. You will now quantize the coefficients for implementation in two's complement digital hardware. Assume in all cases that you will use enough bits to the left of the binary point to

represent the integer part of the coefficients perfectly. The left-most bit is the most significant bit. You will only vary the number of bits to the right of the binary point, which represent the fractional part. With zero bits to the right of the binary point, you can only represent integers. With one bit, you can represent fractional parts that are either .0 or .5. Other possibilities are given in the table below:

| number of bits right of the binary point | possible values for the fractional part |
|---|---|
| 2 | .0, .25, .5, .75 |
| 3 | .0, .125, .25, .375, .5, .625, .75, .875 |
| 4 | .0, .0625, .125, .1875 .25, .3125, .375, ... |

You can use the **`IIRFix`** star to implement this. First, we will study the effects of coefficient quantization only. To minimize the impact of fixed-point internal computations in the **`IIRFix`** star, set the *InputPrecision, AccumulationPrecision,* and *OutputPrecision* to 16.16 (meaning 16 bits to the right and 16 bits to the left of the binary point) getting more than adequate precision.

a. For the cascaded second-order sections of problem 1a, quantize the coefficients with two bits to the right of the binary point. Compare the resulting frequency response to the original. What has happened to the pole closest to the unit circle? Do you still have a fourth-order system? Does the order of the second order sections matter now?

b. Repeat part (a), but using four bits to the right of the binary point. Does this look like it adequately implements the intended filter?

3. Direct form implementations of filters with order higher than two are especially subject to coefficient quantization errors. In particular, poles may move so much when coefficients are quantized that they move outside the unit circle, rendering the implementation unstable. Determine whether the direct form implementation of problem (1a) is stable when the coefficients are quantized. Try 2 bits to the right of the binary point and 4 bits to the right of the binary point. You should plot the impulse response, not the frequency response, to look for instability. How many bits to the right of the binary point do you need to make the system stable?

4. Experiment with the other precision parameters of the **`IIRFix`** star. Is this filter more sensitive to accumulation precision than to coefficient precision?

5. Many applications require a very narrowband lowpass filter, used to extract the d.c. component of a signal. Unfortunately, the pole locations for second-order direct form 2 structures are especially sensitive to coefficient quantization in the region near $z = 1$. Consequently, they are not very well suited to implementing very narrowband lowpass filters.

a. The following transfer function is that of a second-order Butterworth lowpass filter:

$$H(z) = \frac{1 + 2z^{-1} + z^{-2}}{1 - 1.9293z^{-1} + 0.9317z^{-2}}.$$

Find and sketch the pole and zero locations of this filter. Compute and plot the magnitude frequency response. Where is the cutoff frequency (defined to be 3dB below the peak)?

b. Quantize the coefficients to use four bits to the right of the binary point. How many bits to the left of the binary point are required so that all the coefficients can be represented in the same format? Compute and plot the magnitude frequency response of this new filter. Explain why it is so different. What is wrong with it?

c. The following transfer function is a bit better behaved when quantized to four bits to the right of the binary point:

$$H(z) = \frac{1 + 2z^{-1} + z^{-2}}{1 - 1.7187z^{-1} + 0.7536z^{-2}}.$$

It is also a second order Butterworth filter. Determine where its 3dB cutoff frequency is. Quantize the coefficients to four bits right of the binary point, and determine how closely the resulting filter approximates the original.

d. Use the filter from part (c) (possibly used more than once), together with **`Upsample`** and **`Downsample`** stars to implement a lowpass filter with a cutoff of 0.05 radians. Implement both the full precision and quantized versions. Describe qualitatively the effectiveness of this design. Your input and output sample rate should be the same, and the objective is to pass only that part of the input below 0.05 radians to the output unattenuated.

### 4.5.6 FIR filter design

This lab explores FIR filter design by windowing and by the Parks-McClellan algorithm.

1. Use the **`Rect`** star to generate rectangular windows of length 8, 16, and 32. Set the amplitude of the windows so that they have the same d.c. content (so that the Fourier transform at zero will be the same).
   a. Find the drop in dB at the peak of the first side-lobe in the frequency domain. Also find the position (in Hz, assuming the sampling interval $T = 1$) of the peak of the first side-lobe. Is the dB drop a function of the length of the window? What about the position?
   b. Find the drop in dB at the side-lobe nearest $\pi$ radians (the Nyquist frequency) for each of the three window lengths. What relationship would you infer between window length and this drop?
2. Repeat problem 1 with a Hanning window instead of a rectangular window. Be sure to set the *period* parameter of the **`Window`** star to a negative number in order to get only one instance of the window.
3. An ideal low-pass filter with cutoff at $\omega_c$ has impulse response

$$h(n) = \frac{\sin(\omega_c n)}{\pi n}.$$

   This impulse response can be generated for any range $-M \le n \le M$ using the **`Raised-Cosine`** star from the communications subpalette, or the **`Sinc`** star from the nonlinear subpalette. This star is actually an FIR filter, so feed it a unit impulse. Its output will be shaped like $h(n)$ if you set the “excess bandwidth” to zero. Set its parameters as follows:

*length*: 64 (the length of the filter you want)
*symbol_interval*: 8 (the number of samples to the first zero crossing)
*excessBW*: 0.0 (this makes the output ideally lowpass).
*interpolation*: 1

a. What is the theoretical cutoff frequency $\omega_c$ given that $h(8) = 0$ is the first zero crossing in the impulse response? Give your answer in Hz, assuming that the sampling interval $T = 1$.

b. Multiply the 64-tap impulse response gotten from the **`RaisedCosine`** star by Hanning and steep Blackman windows, and plot the original 64-tap impulse response together with the two windowed impulse responses. Which impulse responses end more abruptly on each end?

c. Compute and plot the magnitude frequency response (in dB) of filters with the three impulse responses plotted in part (b). You will want to change the parameter of the **`FFTCx`** star to get more resolution. You can use an *order* of 9 (which corresponds to a 512 point FFT). You can also set the *size* to 64 since the input has only 64 non-zero samples. Describe qualitatively the difference between the three filters. What is the loss at $\omega_c$ compared to d.c.?

4. In this problem, you will use the rather primitive FIR filter design software provided with Ptolemy. The program you will use is called "**`optfir`**"; it uses the Parks-McClellan algorithm to design equirripple FIR filters. See "optfir — equirriple FIR filter design" on page C-1 for an explanation of how to use it. The main objective in this problem will be to compare equirripple designs to the windowed designs of the previous problem.

   a. Design a 64 tap filter with the passband edge at (1/16)Hz and stopband edge at (0.1)Hz. This corresponds very roughly to the designs in problem 3. Compare the magnitude frequency response to those in problem 3. Describe in words the qualitative differences between them. Which filters are "better"? In what sense?

   b. The filter you designed in part (a) should end up having a slightly wider passband than the designs in problem 3. So to make the comparison fair, we should use a passband edge smaller than (1/16)Hz. Choose a reasonable number to use and repeat your design.

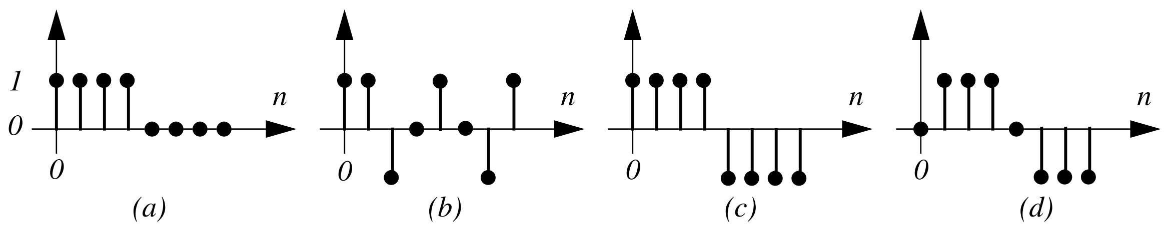
   c. Experiment with different transition band widths. Draw some conclusions about equirripple designs versus windowed designs.

### 4.5.7 The DFT (discrete Fourier transform)

This exercise explores the DFT, FFT, and circular convolution. Ptolemy has both a **`FFTCx`** (complex FFT) and a **`DTFT`** star in the "dsp" palette. The **`FFTCx`** star has an *order* parameter and a *size* parameter. It consumes *size* input samples and computes the DFT of a periodic signal formed by repeating these samples with period $2^{order}$. Only integer powers of two are supported. If $size \le 2^{order}$, then the unspecified samples are given value zero. This can also be viewed as computing samples of the DTFT of a finite input signal of length *size,* padded with $2^{order} - size$ zeros. These samples are evenly spaced from d.c. to $2\pi$, with spacing $2\pi/N$, where $N = 2^{order}$.

The **`DTFT`** star, by contrast, computes samples of the DTFT of a finite input signal at *arbitrary* frequencies (the frequencies are supplied at a second input port). If you are interested in computing even spaced samples of the DTFT in the whole range from d.c. to the sampling frequency, the **`DTFT`** star would be far less efficient than the **`FFTCx`** star. However, if you are interested in only a few samples of the DTFT, then the **`DTFT`** star is more efficient. For this exercise, you should use the **`FFTCx`** star.

1. Find the 8 point DFT (*order* = 3, *size* = 8) of each of the following signals:




   Plot the magnitude, real, and imaginary parts on the same plot. Ignoring any slight round-off error in the computer, which of the DFTs is purely real? Purely imaginary? Why? Give a careful and complete explanation. **Hint:** Do not rely on implicit type conversions, which are tricky to use. Instead, explicitly use the **`CxToReal`** and **`RectToPolar`** stars to get the desired plots.

2. Let

$$x(n) = \begin{cases} 1, & \text{if } n = 0, 1, 2, \text{ or } 3 \\ 0, & \text{otherwise} \end{cases}$$

   as in (a) above. Compute the 4, 8, 16, 32, and 64 point DFT using the **`FFTCx`** star. Plot the 64 point DFT. Explain why the 4 point DFT is as it is, and explain why the progression does what it does as the order of the DFT increases.

3. Assuming a sample rate of 1 Hz, compare the 128 point FFT (*order* = 7, *size* = 128) of a 0.125 Hz cosine wave to the 128 point FFT of a 0.123 Hz cosine wave. It is easy to observe the differences in the magnitude, so you should plot only the magnitude of the output of the **`FFTCx`** star. Explain why the DFTs are so different.

4. For the same 0.125 Hz signal of problem 3, compute a DFT of order 512 using only 128 samples, padded by zeros (*order* = 9, *size* = 128; the zero padding will occur automatically). Explain the difference in the magnitude frequency response from that observed in problem 3. Do the same for the 0.123 Hz signal. Is its magnitude DFT much different from that of the 0.125 Hz cosine? Why or why not?

5. Form a rectangular pulse of width 128 and plot its magnitude DFT using a 512 point FFT (*order* = 9, *size* =512). How is this plot related to those in problem 4? Multiply this pulse by 512 samples of a 0.125 Hz cosine wave and plot the 512 point DFT. How is this related to the plot in problem 4? Explain. **Reminder:** If you get an error message "unresolvable type conflict" then you are probably connecting a float signal to both a float input and a complex input. You can use explicit type conversion stars to correct the problem.

6. To study circular convolution, let

$$y(n) = \begin{cases} 1, & \text{if } n = 1, 2, 3, 4, 5, \text{ or } 6 \\ 0, & \text{otherwise} \end{cases}$$

and let $x(n)$ be as given in problem 2. Use the **FFTCx** star to compute the 8 point circular convolution of these two signals. Which points are affected by the overlap caused by circular convolution? Compute the 16 point circular convolution and compare.

### 4.5.8 Whitening filters

This exercise, and all the remaining ones in this chapter, involve random signals.

1. Implement a filter with two zeros, located at $z = a \pm ja$, where $a = 0.8$, one pole at $z = 0$, and one pole at $z = 0.9$. You may use the **Biquad** or **IIR** star in the "dsp" palette. Filter white noise with it to generate an ARMA process. Then design a whitening filter that converts the ARMA process back into white noise. Demonstrate that your system does what is desired by whatever means seems most appropriate.
2. Implement a causal FIR filter with two zeros at $z = a$ and $z = a^*$, where

$$a = 0.9e^{j\pi/4}.$$

   Plot its magnitude frequency response and phase response, using the **Unwrap** star to remove discontinuities in the phase response. Then implement a second filter with two zeros at $1/a$ and $1/a^*$. Adjust the gain of this filter so that it is the same at d.c. as the first filter. Verify that the magnitude frequency responses are the same. Compare the phases. Which is minimum phase? Then implement an allpass filter which when cascaded with the first filter yields the second. Plot its magnitude and phase frequency response.

### 4.5.9 Wiener filtering

1. Generate an AR (auto-regressive) process $x(n)$ by filtering white Gaussian noise with the following filter:

$$G(z) = \frac{1}{1 - 2z^{-1} + 1.91z^{-2} - 0.91z^{-3} + 0.205z^{-4}}.$$

   You can implement this with the **IIR** filter star. The parameters of the star are:

   *gain*: A float: $g$
   *numerator*: A list of floats separated by spaces: $a_0\ a_1\ a_2\ ...$
   *denominator*: A list of floats separated by spaces: $b_0\ b_1\ b_2\ ...$

   where the transfer function is:

$$H(z) = g\left(\frac{a_0 + a_1 z^{-1} + a_2 z^{-2} + ...}{b_0 + b_1 z^{-1} + b_2 z^{-2} + ...}\right).$$

   More interestingly, you can implement the filter with an **FIR** filter in the feedback loop. Try it both ways, but turn in the latter implementation.
2. Define the "desired" signal to be

$$d(n) = g(n) * x(n) + w(n),$$

where $w(n)$ is a white Gaussian noise process with variance 0.5, uncorrelated with $x(n)$, and $g(n)$ is the impulse response of a filter with the following transfer function:

$$G(z) = 1 + 2z^{-1} + 3z^{-2} + 4z^{-3}.$$

Generate $d(n)$.

3. Design a Wiener filter for estimating $d(n)$ from $x(n)$. Verify that the power of the error signal $e(n) = d(n) - y(n)$ is equal to the power of the additive white noise $w(n)$.
4. Use an adaptive **`LMS`** filter to perform the same function as the fixed Wiener filter in part 3. Use the default initial tap values for the **`LMS`** filter star. Compare the error signals for the adaptive system to the error signal for fixed system by comparing their power. How closely does the **`LMS`** filter performance approximate that of the fixed Wiener filter? How does its performance depend on the adaptation step size? How quickly does it converge? How much do its final tap value look like the optimal Wiener filter solution?

**Ptolemy Hint**: The **`powerEst`** galaxy (in the nonlinear palette) is convenient for estimating power. For the **`LMS`** star, to examine the final tap values, set the *saveTapsFile* parameter to some filename. This file will appear in your home directory (even if you started pigi in some other directory). To examine this file, just type "**`pxgraph -P filename`**" in any shell window. The -P option causes each point to shown with a dot. You may also wish to experiment with the LMSTkPlot star to get animated displays of the filter taps as they adapt.

### 4.5.10 Adaptive equalization

1. Generate random sequence of $\pm 1$ using the **`IIDUniform`** and **`Sgn`** stars. This represents a random sequence of bits to be transmitted over a channel. Filter this sequence with the following filter (the same filter used in "Wiener filtering" on page 4-57):

$$A(z) = \frac{1}{1 - 2z^{-1} + 1.91z^{-2} - 0.91z^{-3} + 0.205z^{-4}}.$$

Assume this filter represents a channel. Observe that it is very difficult to tell from the channel output directly what bits were transmitted. Filter the channel output with an **`LMS`** adaptive filter. Try two mechanisms for generating the error used to update the **`LMS`** filter taps:

a. Subtract the **`LMS`** filter output from the transmitted bits directly. These bits may be available at a receiver during a start-up, or "training" phase, when a known sequence is transmitted.

b. Use the **`Sgn`** star to make decisions from the **`LMS`** filter output, and subtract the filter output from these decisions. This is a decision-directed structure, which does not assume that the transmitted bits are known at the receiver.

To get convergence in reasonable time, it may be necessary to initialize the taps of the LMS filter with something reasonably close to the inverse of the channel response. Try initializing each tap to the integer nearest the optimal tap value. Experiment with other initial tap values. Does the decision-directed structure have more difficulty adapting than the "training" structure that uses the actual transmitted bits? You may wish to experiment with the **`LMSTkPlot`** block to get animated displays of the filter taps.

2. For the this problem, you should generate an AR process by filtering Gaussian white noise with the following filter:

$$A(z) = \frac{1}{1 - 1.94z^{-1} + 0.98z^{-2}}.$$

Construct an optimal one-step forward linear predictor for this process using the **FIR** star, and a similar adaptive linear predictor using the **LMS** star. Display the two predictions and the original process on the same plot. Estimate the power of the prediction errors and the power of the original process. Estimate the prediction gain (in dB) for each predictor. For each predictor, how many fewer bits would be required to encode the prediction error power vs. the original signal with the same quantization error? Assume the number of bits required for each signal to have the same quantization error is determined by the $4\sigma$ rule, which means that full scale is equal to four standard deviations.

3. Modify the AR process so that is generated with the following filter:

$$A(z) = \frac{1}{1 - 1.2z^{-1} + 0.6z^{-2}}.$$

Again estimate the prediction gain in both dB and bits. Explain clearly why the prediction gain is so much lower.

4. In the file **$PTOLEMY/src/domains/sdf/demo/speech.lin** there are samples from two seconds of speech sampled at 8kHz. You need not use all 16,000 samples. The samples are integer-valued with a peak of around 20,000. You may want to scale the signal down. Use your one-step forward linear predictor with the LMS algorithm to compute the prediction error signal. Measure the prediction gain in dB, and note that it varies widely for different speech segments. Identify the segments where the prediction gain is greatest, and explain why. Identify the segments where the prediction gain is small and explain why it is so. Make an engineering decision about the number of bits that can be saved by this coder without appreciable degradation in signal quality. You can read the file using the **WaveForm** star.

### 4.5.11 ADPCM speech coding

For the same speech file you used in the last assignment, **$PTOLEMY/src/domains/sdf/demo/speech.lin**, you are to construct an adaptive differential pulse code modulation (ADPCM) coder using the "feedback around quantizer" structure and an LMS filter to form the approximate linear prediction. Be sure to connect your LMS filter so that at the receiver, if there are no transmission errors, an LMS filter can also be used in a feedback path, and the LMS filter will exactly track the one in the transmitter. You will use various amounts of quantization.

To assess the ADPCM system, reconstruct the speech signal from the quantized residual, subtract this from the original signal, and measure the noise power. If you have a workstation with a speaker available, listen to the sound, and compare against the original.

1. In your first experiment, do not quantize the signal. Find a good step size, verify that the feedback around quantizer structure works, measure the reconstruction error power and prediction gain. Does your reconstruction error make sense? Compare your prediction gain

result against that obtained in the previous lab. It should be identical, since all you have changed is to use the feedback-around-quantizer structure, but you are not yet using a quantizer.

Assume you have a communication channel where you can transmit $N$ bits per sample. You will now measure the signal quality you can achieve with ADPCM compared to simple PCM (pulse code modulation) over the same channel. In PCM, you directly quantize the speech signal to $2^N$ levels, whereas in ADPCM, you quantize the prediction error to $2^N$ levels. For a given $N$, you should choose the quantization levels carefully. In particular, the quantization levels for the ADPCM case should not be the same as those for the PCM case. Given a particular prediction gain $G$, what should the relationship be? You should use the **`Quant`** star to accomplish the quantization in both cases. A useful way to set the parameters of the **`Quant`** star is as follows (shown for $N = 2$ bits, meaning 4 quantization levels):

*thresholds*: (-1*s) (0) (1*s)

*levels*: (-1.5*s) (-0.5*s) (0.5*s) (1.5*s)

where "s" is a universe parameter. This way, you can easily experiment with various quantization spacings without having to continually retype long sequences of numbers.

For each $N$, you should compare (a) the ADPCM encoded speech signal and (b) the PCM encoded speech signal to the original speech signal. You should make this comparison by measuring the power in the differences between the reconstructed signals and the original. How does this difference compare to the prediction gain?

2. Use $N = 3$ bits.
3. Use $N = 2$ bits.

### 4.5.12 Spectral estimation

In the Ptolemy "dsp" palette there are three galaxies that perform three different spectral estimation techniques. These are the (1) periodogram, (2) autocorrelation method using the Levinson-Durbin algorithm, and (3) Burg's method. The latter two compute linear predictor coefficients, and then use these to determine the frequency response of a whitening filter for the random process. The magnitude squared of this frequency response is inverted to get an estimate of the power spectrum of the random process. Study these and make sure you understand how they work. You are going to use all three to construct power spectral estimates of various signals and compare them. In particular, note how many input samples are consumed and produced. If you display all three spectral estimates on the same plot, then you must generate the same number of samples for each estimate. You will begin using only the Burg galaxy.

1. In this problem, we study the performance of Burg's algorithm for a simple signal: a sinusoid in noise. First, generate a sinusoid with period equal to 25 samples. Add Gaussian white noise to get an SNR of 10 dB.
   a. Using 100 observations, estimate the power spectrum using order 3, 4, 6, and 12th order AR models. You need not turn in all plots, but please comment on the differences.
   b. Fix the order at 6, and construct plots of the power spectrum for SNR of 0, 10, 20, and 30 dB. Again comment on the differences.

c. When the AR model order is large relative to the number of data samples observed, an AR spectral estimate tends to exhibit spurious peaks. Use only 25 input samples, and experiment with various model orders in the vicinity of 16. Experiment with various signal to noise ratios. Does noise enhance or suppress the spurious peaks?

d. Spectral line splitting is a well-known artifact of Burg's method spectral estimates. Specifically, a single sinusoid may appear as two closely spaced sinusoids. For the same sinusoid, with an SNR of 30dB, use only 20 observations of the signal and a model order of 15. For this problem, you will find that the spectral estimate depends heavily on the starting phase of the sinusoid. Plot the estimate for starting phases of 0, 45, 90, and 135 degrees of a cosine wave.

2. In this problem, we study a synthetic signal that roughly models both voiced and unvoiced speech.

a. First construct a signal consisting of white noise filtered by the transfer function

$$H(z) = \frac{1}{1 - 1.2z^{-1} + 0.6z^{-2}}.$$

Then estimate its power spectrum using three methods, a periodogram, the autocorrelation method, and Burg's method. Use 256 samples of the signal in all three cases, and order-8 estimates for the autocorrelation and Burg's methods. Increase and decrease the number of inputs that you read. Does the periodogram estimate improve? Do the other estimates improve? How should you measure the quality of the estimates? What order would work better than 8 for this estimate?

b. Instead of exciting the filter $H(z)$ with white noise, excite it with an impulse stream with period 20 samples. Repeat the spectral estimate experiments. Which estimate is best? Does increasing the number of input samples observed help any of the estimates? With the number of input samples observed fixed at 256, try increasing the order of the autocorrelation and Burg's estimates. What is the best order for this particular signal? Note that deciding on an order for such estimates is a difficult problem.

c. Voiced speech is often modeled by an impulse stream into an all-pole filter. Unvoiced speech is often modeled by white noise into an all-pole filter. A reasonable model includes some of both, with more noise if the speech is unvoiced, and less if it is voiced. Mix noise and the periodic impulse stream at the input to the filter $H(z)$ in various ratios and repeat the experiment. Does the noise improve the autocorrelation and Burg estimates, compared to estimates based on pure impulsive excitation? You should be able to get excellent estimates using both the autocorrelation and Burg's methods. You may wish to run some of these experiments with 1024 input samples.

### 4.5.13 Lattice filters

In the Ptolemy "dsp" palette there are four lattice filter stars called: **`Lattice`**, **`RLattice`**, **`BlockLattice`**, and **`BlockRLattice`**. The "R" refers to "Recursive", so the "**`RLattice`**" stars are inverse filters (IIR), while the "**`Lattice`**" stars are prediction-error filters (FIR). The "Block" modifier allows you to connect the **`LevDur`** or **`Burg`** stars to the Lattice filters to provide the coefficients. A block of samples is processed with a given set of coefficients, and then new coefficients can be loaded.

1. Consider an FIR lattice filter with the following values for the reflection coefficients: 0.986959, -0.945207, 0.741774, -0.236531.
   a. Is the inverse of this filter stable?
   b. Let the transfer function of the FIR lattice filter be written

$$H(z) = 1 + h_1 z^{-1} + h_2 z^{-2} + \ldots + h_M z^{-M}$$

   Use the Levinson-Durbin algorithm to find $h_1, \ldots, h_M$. Experiment with various methods to estimate the autocorrelation. Turn in your estimates of $h_1, \ldots, h_M$.
   c. Use Ptolemy to verify that an FIR filter with your computed tap values 1, $h_1, \ldots, h_M$ has the same transfer function as the lattice filter.
2. In this problem, we compare the biased and unbiased autocorrelation estimates for troublesome sequences.
   a. Construct a sine wave with a period of 40 samples. Use 64 samples into the **`Autocor`** star to estimate its autocorrelation using both the biased and unbiased estimate. Which estimate looks more reasonable?
   b. Feed the two autocorrelation estimates into the **`LevDur`** star to estimate predictor coefficients for various prediction orders. Increase the order until you get predictor coefficients that would lead to an unstable synthesis filter. Do you get unstable filters for both biased and unbiased autocorrelation estimates?
   c. Add white noise to the sine wave. Does this help stabilize the synthesis filter?
   d. Load your reflection coefficients into the **`BlockLattice`** star and compute the prediction error both the biased and unbiased autocorrelation estimate. Which is a better predictor?