APPENDIX B

THE DELTA-SIGMA TOOLBOX

Getting Started

Go to `http://www.mathworks.com/matlabcentral/fileexchange/` and search for `delsig`. Download and unzip the `delsig.zip` file. Add the `delsig` directory to the MATLAB path. To improve simulation speed, compile the `simulateDSM.c` file by typing `mex simulateDSM.c` at the MATLAB prompt. Do the same for `simulateMS.c`.

The Delta-Sigma toolbox requires the Signal Processing toolbox and the Control Systems toolbox; the `clans` and `designPBF` functions also require the Optimization toolbox.

The following conventions are used throughout the Delta-Sigma toolbox:

- Frequencies are normalized; $f = 1$ corresponds to the sampling frequency, f_s.
- Default values for function arguments are shown following an equals sign in the parameter list. To use the default value for an argument, omit the argument if it is at the end of the list, otherwise use `NaN` (not-a-number) or `[]` (the empty matrix) as a place-holder.
- The loop-filter of a general delta-sigma modulator is described with an $ABCD$ matrix. See "Modulator Model Details" on page 533 for a description of this matrix.

Understanding Delta-Sigma Data Converters, *Second Edition.* By Shanthi Pavan, Richard Schreier, and Gabor C. Temes.

Demonstrations and Examples

dsdemo1	Demonstration of the `synthesizeNTF` function. Noise transfer function synthesis for a 5th-order lowpass modulator, both with and without optimized zeros, plus an 8th-order bandpass modulator with optimized zeros.
dsdemo2	Demonstration of the `simulateDSM`, `predictSNR` and `simulateSNR` functions: time-domain simulation, SNR prediction using the describing function method of Ardalan and Paulos, spectral analysis and signal-to-noise ratio calculation. Lowpass, bandpass, multi-bit lowpass examples are given.
dsdemo3	Demonstration of the `realizeNTF`, `stuffABCD`, `scaleABCD` and `mapABCD` functions: coefficient calculation and dynamic range scaling.
dsdemo4	Audio demonstration of MOD1 and MOD2 with sinc^n decimation.
dsdemo5	Demonstration of the `simulateMS` function: simulation of the element selection logic of a mismatch-shaping DAC.
dsdemo6	Demonstration of the `designHBF` function. Hardware-efficient halfband filter design and simulation.
dsdemo7	Demonstration of the `findPIS` function: positively-invariant set computation.
dsexample1	Discrete-time modulator design example.
dsexample2	Continuous-time lowpass modulator design example.

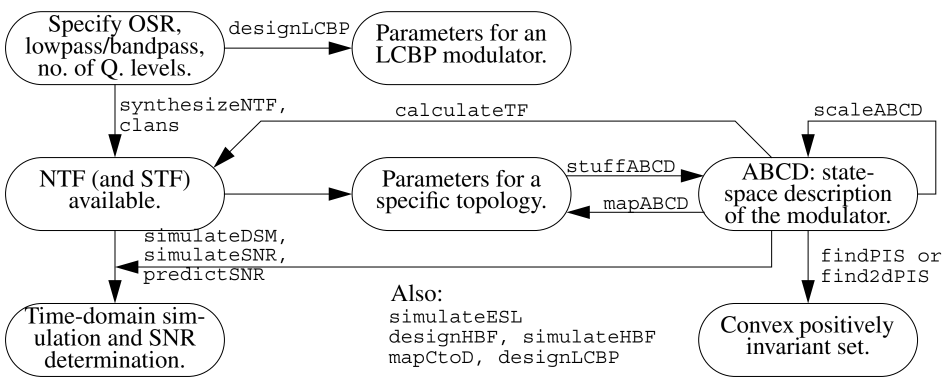


Figure B.1 Flowchart of key $\Delta\Sigma$ toolbox functions.

Key Functions

```
ntf = synthesizeNTF(order=3,R=64,opt=0,H_inf=1.5,f0=0)
```
page 505
```
ntf = clans(order=4,R=64,Q=5,rmax=0.95,opt=0)
```
page 506
```
ntf = synthesizeChebyshevNTF(order=3,R=64,opt=0,H_inf=1.5,f0=0)
```
page 507

Synthesize a noise transfer function.

```
[v,xn,xmax,y] = simulateDSM(u,ABCD,nlev=2,x0=0)
[v,xn,xmax,y] = simulateDSM(u,ntf,nlev=2,x0=0)
```
page 508

Simulate a delta-sigma modulator with a given input.

```
[snr,amp] = simulateSNR(ntf,OSR,amp=...,
                        f0=0,nlev=2,f=1/(4*R),k=13)
```
page 509

Determine the SNR versus input amplitude curve by simulation.

```
[a,g,b,c] = realizeNTF(ntf,form='CRFB',stf=1)
```
page 510

Convert a noise transfer function into coefficients for the specified topology.

```
ABCD = stuffABCD(a,g,b,c,form='CRFB')
```
page 511

Calculate the ABCD matrix given the parameters of the specified topology.

```
[a,g,b,c] = mapABCD(ABCD,form='CRFB')
```
page 511

Convert the ABCD matrix into the parameters of the specified topology.

```
[ABCDs, umax] = scaleABCD(ABCD,nlev=2,f=0,xlim=1,ymax=nlev+2)
```
page 512

Perform dynamic range scaling on a delta-sigma modulator described by ABCD.

```
[ntf,stf] = calculateTF(ABCD,k=1)
```
page 513

Calculate the NTF and STF of a delta-sigma modulator described by the given ABCD matrix, assuming a quantizer gain of k.

```
[sv,sx,sigma_se,max_sx,max_sy] =
    simulateMS(v,mtf,M=16,d=0,dw=[1-],sx0=[0-])
```
page 514

Simulate the element-selection logic of a mismatch-shaping DAC.

Functions for Continuous-Time Systems

`[ABCDc,tdac2]= realizeNTF_ct(ntf,form='FB',tdac,ordering=[1:n], bp=zeros(-),ABCDc)` page 516
Realize an NTF with a continuous-time loop-filter.

`[sys, Gp] = mapCtoD(sys_c,t=[0 1],f0=0)` page 517
Map a continuous-time system to a discrete-time system whose impulse response matches the sampled pulse response of the original continuous-time system. See `dsexample2`.

`H = evalTFP(Hs,Hz,f)` page 518
Compute the value of the product of the continuous-time transfer function H_s and the discrete-time transfer function H_z at frequencies f. Use this function to evaluate the signal transfer function of a CT $\Delta\Sigma$ ADC system.

Functions for Quadrature Systems

`ntf = synthesizeQNTF(order=3,OSR=64,f0=0,NG=-60,ING=-20)` page 519
Synthesize a noise transfer function for a quadrature delta-sigma modulator.

`[v,xn,xmax,y] = simulateQDSM(u,ABCD|ntf,nlev=2,x0=0)` page 520
Simulate a quadrature delta-sigma modulator with the given input.

`ABCD = realizeQNTF(ntf,form='FB',rot=0,bn)` page 521
Convert a quadrature noise transfer function into a complex ABCD matrix for the specified structure.

`ABCDr = mapQtoR(ABCD)` and `[ABCDq ABCDp] = mapR2Q(ABCDr)` page 522
Convert a complex matrix into its real equivalent and vice versa.

`[ntf stf intf istf] = calculateQTF(ABCDr)` page 523
Calculate the noise and signal transfer functions of a quadrature modulator.

`[sv,sx,sigma_se,max_sx,max_sy]= simulateQESL(v,mtf,M=16,sx0=[0-])` page 524
Simulate the Element Selection Logic of a quadrature differential DAC.

Note: `simulateSNR` works for a quadrature modulator if given a complex NTF or ABCD matrix; `simulateDSM` can also be used for a quadrature modulator if given an `ABCDr` matrix and a 2-element `nlev` vector.

Specialty Functions

`[f1,f2,info] = designHBF(fp=0.2,delta=1e-5,debug=0)` page 525
Design a Saramäki half-band filter for use in a decimation or interpolation filter.

`y = simulateHBF(x,f1,f2,mode=0)` page 527
Simulate a Saramäki half-band filter in the time domain.

`[C, e, x0] = designPBF(N,M,pb,pbr,sbr,ncd,np,ns,fmax)` page 528
Design a symmetric polynomial-based filter (PBF) according to Hunter's method.

`[snr,amp,k0,k1,sigma_e2 = predictSNR(ntf,OSR=64,amp=...,f0=0)` page 529
Predict the SNR versus input amplitude curve using the describing function method.

`[s,e,n,o,Sc] = findPIS(u,ABCD,nlev=2,options)` page 530
Find a convex positively-invariant set for a delta-sigma modulator.

```
[data, snr] = findPattern(N=1024,OSR=64,ntf,ftest,Atest,
                          f0=0,nlev=2,quadrature=0,dbg=0)
```
page 532
Create a length-N data record which has good spectral properties when repeated.

Utility Funtions

Delta-Sigma Utility

`mod1, mod2`
Set the ABCD matrix, NTF and STF of the standard 1^{st}- and 2^{nd}-order modulators.

`snr = calculateSNR(hwfft,f,nsig=1)`
Estimate the SNR given the in-band bins of a windowed FFT and the location of the input.

`[A B C D] = partitionABCD(ABCD, m)`
Partition ABCD into A, B, C, D for an m-input state-space system.

`H_inf = infnorm(H)`
Compute the infinity norm (maximum absolute value) of a z-domain transfer function.

`y = impL1(ntf,n=10)`
Compute n points of the impulse response from the comparator output back to the comparator input for the given NTF.

`y = pulse(S,tp=[0 1],dt=1,tfinal=10,nosum=0)`
Compute the sampled pulse response of a continuous-time system.

`sigma_H = rmsGain(H,f1,f2)`

Compute the root mean-square gain of the discrete-time transfer function H in the frequency band $[f_1, f_2]$.

General Utility

`dbv(), dbp(), undbv(), undbp(), dbm(), undbm()`
The dB equivalent of voltage/power quantities, and their inverse functions.

`window = ds_hann(N)`
A Hann window of length N. Unlike MATLAB's original `hanning` function, `ds_hann` does not smear tones which are located exactly in an FFT bin (i.e. tones having an integral number of cycles in the given block of data). MATLAB 6's `hanning(N,'periodic')` function and MATLAB 7's `hann(N,'periodic')` function are the same as `ds_hann(N)`.

`mag = zinc(f,n=64,m=1)`
Calculate the magnitude response of a cascade of m sinc_n filters at frequencies f.

Graphing Utility

`plotPZ(H,color='b',markersize=5,list=0)`
Plot the poles and zeros of a transfer function.

`plotSpectrum(X,fin,fmt)`
Plot a smoothed spectrum.

`figureMagic(xRange,dx,xLab, yRange,dy,yLab, size)`
Performs a number of formatting operations for the current figure, including axis limits, ticks and labelling.

`printmif(file,size,font,fig)`
Print a figure to an Adobe Illustrator file and then use `ai2mif` to convert it to FrameMaker MIF format. `ai2mif` is an improved version of the function of the same name originally written by Deron Jackson <djackson@mit.edu>.

`[f,p] = logsmooth(X,inBin,nbin)`
Smooth the FFT X, and convert it to dB. See also `bplogsmooth` and `bilogplot`.

synthesizeNTF

Synopsis: `ntf = synthesizeNTF(order=3,OSR=64,opt=0,H_inf=1.5,f0=0)`
Synthesize a noise transfer function (NTF) for a delta-sigma modulator.

Input

`order`	The order of the NTF. `order` must be even for bandpass modulators.
`OSR`	The oversampling ratio. `OSR` is only needed when optimized NTF zeros are requested.
`opt`	A flag used to request optimized NTF zeros. `opt=0` puts all NTF zeros at band-center. `opt=1` optimizes the NTF zeros according to the high-OSR limit. `opt=2` puts at least one zero at band-center, but optimizes the rest. `opt=3` uses the Optimization toolbox to optimize the zeros.
`H_inf`	The maximum out-of-band gain of the NTF. Lee's rule states that `H_inf<2` should yield a stable modulator with a binary quantizer. Reducing `H_inf` increases the likelihood of success, but reduces the attenuation provided by the NTF and thus the theoretical resolution of the modulator.
`f0`	The center frequency of the modulator. $f_0 \neq 0$ yields a bandpass modulator; `f0=0.25` puts the center frequency at $f_s/4$.

Output

`ntf`	The modulator NTF, given as an LTI object in zero-pole form.

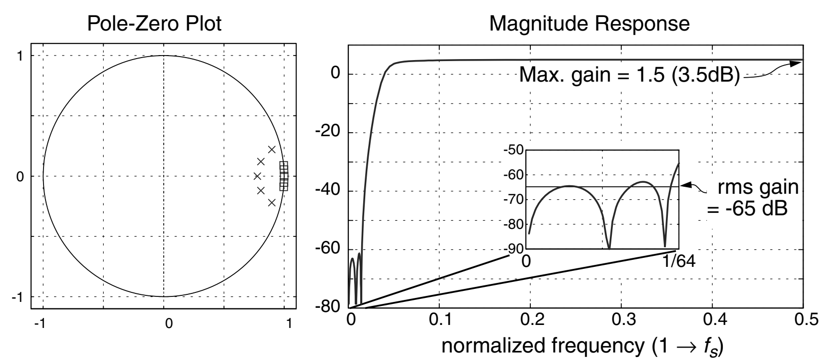
Bugs

If `OSR` or `H_inf` are low, the NTF is not optimal. Use `synthesizeChebyshevNTF` instead.

Example

Fifth-order lowpass modulator; zeros optimized for an oversampling ratio of 32.

```
» H = synthesizeNTF(5,32,1)
Zero/pole/gain:
(z-1) (z^2 - 1.997z + 1) (z^2 - 1.992z + 1)
---------------------------------------
(z-0.7778) (z^2 - 1.613z + 0.6649) (z^2 - 1.796z + 0.8549)
Sampling time:  1
```



clans

Synopsis: `ntf = clans(order=4,OSR=64,Q=5,rmax=0.95,opt=0)`
Synthesize a lowpass NTF using the CLANS (Closed-loop analysis of noise-shaper) methodology [1]. This function requires the Optimization toolbox.

[1] J. G. Kenney and L. R. Carley, "Design of multibit noise-shaping data converters," *Analog Integrated Circuits Signal Processing Journal*, vol. 3, pp. 259-272, 1993.

Input

`order`	The order of the NTF.
`OSR`	The oversampling ratio.
`Q`	The maximum number of quantization levels used by the fed-back quantization noise. (Mathematically, $Q = \|\|h\|\|_1 - 1$, i.e. the sum of the absolute values of the impulse response samples minus 1.) The maximum stable input of a ΔΣ modulator is guaranteed to be at least $(n_{lev} - Q)$.
`rmax`	The maximum radius for the NTF poles.
`opt`	A flag used to request optimized NTF zeros.

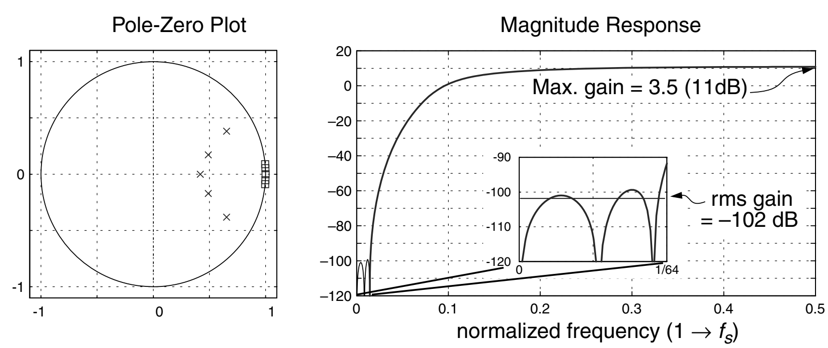
Output

`ntf`	The modulator NTF, given as an LTI object in zero-pole form.

Example

5th-order lowpass modulator; time-domain noise gain of 5, zeros optimized for $OSR = 32$.

```
» H= clans(5,32,5,.95,1)
```



synthesizeChebyshevNTF

Synopsis: `ntf = synthesizeChebyshevNTF(order,OSR,opt,H_inf,f0)`
Obtain a noise transfer function (NTF) in which has equiripple magnitude in the passband. `synthesizeChebyshevNTF` creates NTFs which are no better than `synthesizeNTF`, except when `OSR` or `H_inf` are low.

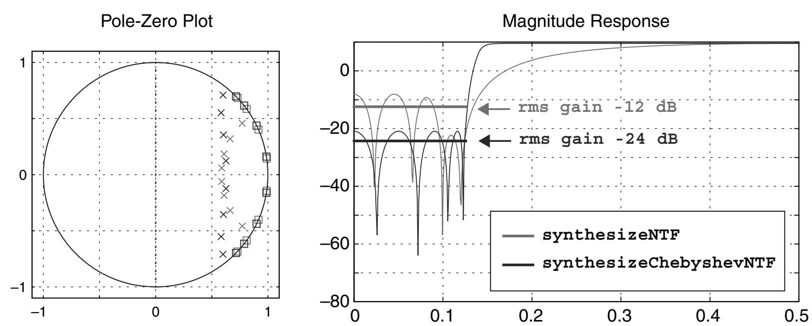
Input and Output

Same as `ssynthesizeNTF`, except that the `opt` argument is not supported yet.

Examples

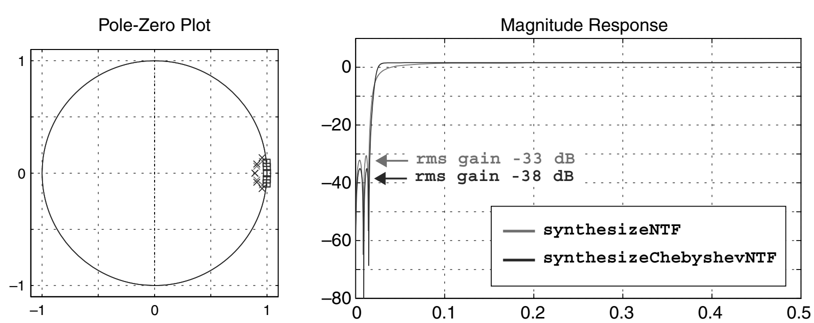
Compare the NTFs created by `synthesizeNTF` and `synthesizeChebyshevNTF` when OSR is low:

```
» OSR = 4; order = 8; H_inf = 3;
» H1 = synthesizeNTF(order,OSR,1,H_inf);
» H3 = synthesizeChebyshevNTF(order,OSR,1,H_inf);
```



Repeat for H_inf low:

```
» OSR = 32; order = 5; H_inf = 1.2;
» H1 = synthesizeNTF(order,OSR,1,H_inf);
» H3 = synthesizeChebyshevNTF(order,OSR,1,H_inf);
```



simulateDSM

Synopsis: `[v,xn,xmax,y] = simulateDSM(u,ABCD|ntf,nlev=2,x0=0)`
Simulate a delta-sigma modulator with a given input. For maximum speed, make sure that the compiled mex file is on your search path by typing `which simulateDSM` at the MATLAB™ prompt.

Input

`u`	The input sequence to the modulator, given as a $m \times N$ matrix, where m is the number of inputs (usually 1). Note that full-scale corresponds to an input of magnitude `nlev`−1.
`ABCD`	A state-space description of the modulator loop-filter.
`ntf`	The modulator NTF, given in zero-pole form. The modulator STF is assumed to be unity.
`nlev`	The number of levels in the quantizer. Multiple quantizers are indicated by making `nlev` a column vector.
`x0`	The initial state of the modulator.

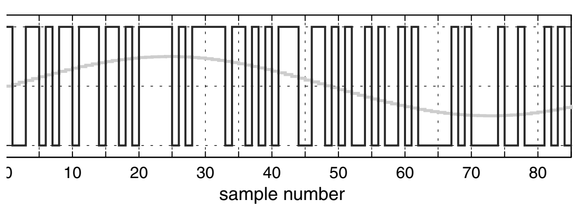
Output

`v`	The samples of the output of the modulator, one for each input sample.
`xn`	The internal states of the modulator, one for each input sample, given as an $n \times N$ matrix.
`xmax`	The maximum absolute values of each state variable.
`y`	The samples of the quantizer input, one per input sample.

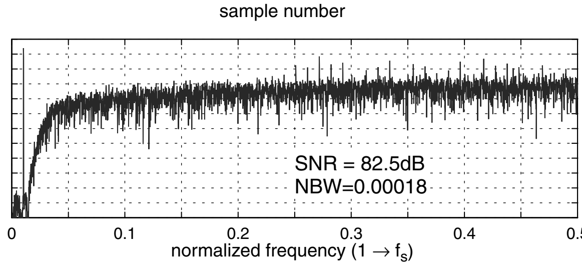
Example

Simulate a 5th-order binary lowpass modulator with a half-scale sine-wave input and plot its output in the time and frequency domains.

```
>> OSR = 32; H = synthesizeNTF(5,OSR,1)}
>> N = 8192; fB = ceil(N/(2*OSR));}
>> f=85; u = 0.5*sin(2*pi*f/N*[0:N-1]);}
>> v = simulateDSM(u,H);
```




```
t = 0:85;
stairs(t, u(t+1),'g');
hold on;
stairs(t,v(t+1),'b');
axis([0 85 -1.2 1.2]);
ylabel('u, v');
```




```
spec=fft(v.*ds_hann(N))/(N/4)
plot(linspace(0,0.5,N/2+1), .
  dbv(spec(1:N/2+1)));
axis([0 0.5 -120 0]);
grid on;
ylabel('dBFS/NBW')
snr=calculateSNR(spec(1:fB),f
s=sprintf('SNR = %4.1fdB\n',s
text(0.25,-90,s);
s=sprintf('NBW=%7.5f',1.5/N);
text(0.25, -110, s);
```

simulateSNR

Synopsis: `[snr,amp] = simulateSNR(ntf|ABCD|function,osr,amp,f0=0, nlev=2,f=1/(4*OSR),k=13,quadrature=0)`

Simulate a delta-sigma modulator with sine wave inputs of various amplitudes and calculate the signal-to-noise ratio (SNR) in dB for each input.

Input

`ntf`	The modulator NTF, given in zero-pole form.
`ABCD`	A state-space description of the modulator loop-filter, or the name of a function taking the input signal as its sole argument.
`osr`	The oversampling ratio.
`amp`	A row vector listing the amplitudes to use. Defaults to [-120 -110...-20 -15 -10 -9 -8 ... 0] dB, where 0 dB means a full-scale (peak value = `n_lev`−1) sine wave.
`f0`	The center frequency of the modulator.
`nlev`	The number of levels in the quantizer. Multiple quantizers are indicated by making `nlev` a vector.
`f`	The test frequency, adjusted to be an FFT bin.
`k`	The number of time points used for the FFT is 2^k.
`quadrature`	A flag indicating that the system being simulated is quadrature. This flag is set automatically if either `ntf` or `ABCD` are complex.

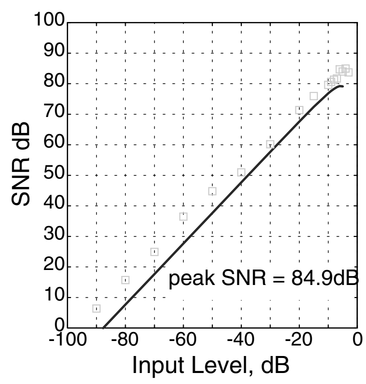
Output

`snr`	A row vector containing the SNR values calculated from the simulations.
`amp`	A row vector listing the amplitudes used.

Example

Compare the SNR versus input amplitude curve determined by the describing function method of Ardalan and Paulos with that determined by simulation for a 5th-order modulator.

```
» OSR = 32; H = synthesizeNTF(5,OSR,1)
» [snr_pred,amp] = predictSNR(H,OSR);
» [snr,amp] = simulateSNR(H,OSR);
```




```
plot(amp,snr_pred,'b',amp,snr,'gs');
grid on;
figureMagic([-100 0], 10, 2, ...
   [0 100], 10, 1);
xlabel('Input Level, dB');
ylabel('SNR dB');
s=sprintf('peak SNR = %4.1fdB\n',...
   max(snr));
text(-65,15,s);
```

realizeNTF

Synopsis: `[a,g,b,c] = realizeNTF(ntf,form='CRFB',stf=1)`
Convert an NTF into a set of coefficients for a particular modulator topology.

Input

`ntf`	The modulator NTF, given in zero-pole form (i.e. a zpk object).
`form`	A string specifying the modulator topology. `CRFB` Cascade-of-resonators, feedback form. `CRFF` Cascade-of-resonators, feedforward form. `CIFB` Cascade-of-integrators, feedback form. `CIFF` Cascade-of-integrators, feedforward form. `---D` Any of the above, but the quantizer is delaying. Structures are described in "Modulator Model Details" on page 533.
`stf`	The modulator STF, specified as a zpk object. Note that the poles of the STF must match those of the NTF in order to guarantee that the STF can be realized without the addition of extra state variables.

Output

`a`	Feedback/feedforward coefficients from/to the quantizer $(1 \times n)$.
`g`	Resonator coefficients $(1 \times \lfloor n/2 \rfloor)$.
`b`	Feed-in coefficients from the modulator input to each integrator $(1 \times (n+1))$.
`c`	Integrator inter-stage coefficients. $(1 \times n)$. In unscaled modulators, c is all ones.

Example
Determine the coefficients for a 5th-order modulator with the cascade-of-resonators structure, feedback (CRFB) form.

```
>> H = synthesizeNTF(5,32,1);}
>> [a,g,b,c] = realizeNTF(H,'CRFB')}
 a = 0.0007    0.0084    0.0550    0.2443    0.5579}
 g = 0.0028    0.0079}
 b = 0.0007    0.0084    0.0550    0.2443    0.5579    1.0000}
 c = 1     1     1     1     1
```

See Also
Use `realizeNTF_ct` (page 516) to realize an NTF with a continuous-time loop-filter.

stuffABCD

Synopsis: `ABCD = stuffABCD(a,g,b,c,form='CRFB')`
Calculate the ABCD matrix given the parameters of a specified modulator topology.

Input

`a`	Feedback/feedforward coefficients from/to the quantizer.
`g`	Resonator coefficients.
`b`	Feed-in coefficients from the modulator input to each integrator.
`c`	Integrator inter-stage coefficients.
`form`	See `realizeNTF` on page 510 for a list of supported forms and "Supported Modulator Topologies" on page 534 for block diagrams of them.

Output

`ABCD`	A state-space description of the loop-filter.

mapABCD

Synopsis: `[a,g,b,c] = mapABCD(ABCD,form='CRFB')`
Calculate the parameters for a specified modulator topology, assuming ABCD fits that topology.

Input

`ABCD`	A state-space description of the modulator loop-filter.
`form`	See `realizeNTF` on page 510 for a list of supported structures.

Output

`a`	Feedback/feedforward coefficients from/to the quantizer.
`g`	Resonator coefficients.
`b`	Feed-in coefficients from the modulator input to each integrator.
`c`	Integrator inter-stage coefficients.

scaleABCD

Synopsis: `[ABCDs,umax]=scaleABCD(ABCD,nlev=2,f=0,xlim=1,ymax=nlev+5,umax,N=1e5)`

Scale the ABCD matrix so that the state maxima are less than a specified limit. The maximum stable input is determined as a side-effect of this process.

Input

`ABCD`	A state-space description of the modulator loop-filter.
`nlev`	The number of levels in the quantizer.
`f`	The normalized frequency of the test sinusoid.
`xlim`	The limit on the states. May be given as a vector.
`ymax`	The threshold for judging modulator stability. If the quantizer input exceeds `ymax`, the modulator is considered to be unstable.

Output

`ABCDs`	The scaled state-space description of the modulator loop-filter.
`umax`	The maximum stable input. Input sinusoids with amplitudes below this value should not cause the modulator states to exceed their specified limits.

calculateTF

Synopsis: `[ntf,stf] = calculateTF(ABCD,k=1)`
Calculate the NTF and STF of a delta-sigma modulator.

Input

`ABCD`	A state-space description of the modulator's loop-filter.
`k`	The quantizer gain to assume.

Output

`ntf`	The modulator NTF, given as an LTI system in zero-pole form.
`stf`	The modulator STF, given as an LTI system in zero-pole form.

Example
Realize a 5th-order modulator with the cascade-of-resonators structure, feedback form. Calculate the ABCD matrix of the loop-filter and verify that the NTF and STF are correct.

```
>> H = synthesizeNTF(5,32,1)
Zero/pole/gain:
(z-1) (z^2 - 1.997z + 1) (z^2 - 1.992z + 1)
-----------------------------------------------------------
(z-0.7778) (z^2 - 1.613z + 0.6649) (z^2 - 1.796z + 0.8549)
Sampling time: 1

>> [a,g,b,c] = realizeNTF(H)
a = 0.0007    0.0084    0.0550    0.2443    0.5579
g = 0.0028    0.0079
b = 0.0007    0.0084    0.0550    0.2443    0.5579    1.0000
c = 1     1     1     1     1

>> ABCD = stuffABCD(a,g,b,c)
ABCD =
1.0000         0         0         0         0    0.0007   -0.0007
1.0000    1.0000   -0.0028         0         0    0.0084   -0.0084
1.0000    1.0000    0.9972         0         0    0.0633   -0.0633
     0         0    1.0000    1.0000   -0.0079    0.2443   -0.2443
     0         0    1.0000    1.0000    0.9921    0.8023   -0.8023
     0         0         0         0    1.0000    1.0000         0

>> [ntf,stf] = calculateTF(ABCD)
Zero/pole/gain:
(z-1) (z^2 - 1.997z + 1) (z^2 - 1.992z + 1)
-----------------------------------------------------------
(z-0.7778) (z^2 - 1.613z + 0.6649) (z^2 - 1.796z + 0.8549)
Sampling time: 1

Zero/pole/gain:
1
Static gain.
```

simulateMS

Synopsis: `[sv,sx,sigma_se,max_sx,max_sy]`
`= simulateMS(v,M=16,mtf,d=0,dw=[1,1,...],sx0=[0-])`

Simulate the element selection logic of a mismatch-shaping DAC.

Input

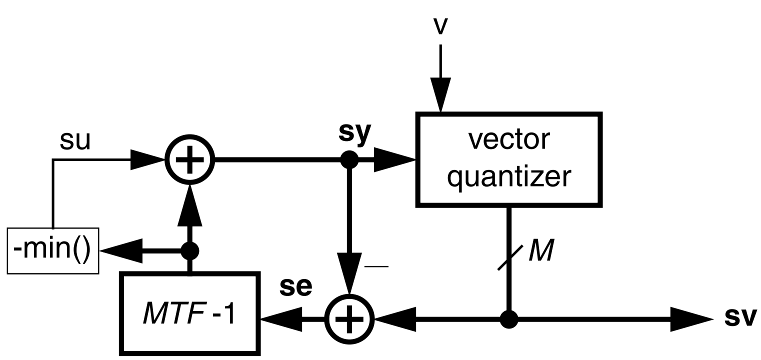
`v`	The DAC input. `v` must be in `-M:2:M` if $dw = [1, 1, \ldots]$. For other dw, v must be in the range $[-\sum_i^M dw(i), \sum_i^M dw(i)]$.
`M`	The number of DAC elements.
`mtf`	The mismatch-shaping transfer function, given in zero-pole form.
`d`	Dither uniformly distributed in $[-d, d]$ is added to the sy input of the vector quantizer.
`dw`	A vector containing the nominal weight associated with each element.
`sx0`	An $n \times M$ matrix containing the initial state of the element selection logic. n is the order of `mtf`.

Output

`sv`	The selection vector: a vector of zeros and ones indicating which elements to enable.
`sx`	An $n \times M$ matrix containing the final state of the element selection logic.
`sigma_se`	The rms value of the selection error, $se = sv - sy$. `sigma_se` may be used to analytically estimate the power of in-band noise caused by element mismatch.
`max_sx`	The maximum value attained by any state in the ESL.
`max_sy`	The maximum value attained by any component of the (un-normalized) "desired usage" vector sy.

See Also

`simulateTSMS`, `simulateBiDWA`, `simulateXS` and `simulateMXS`.

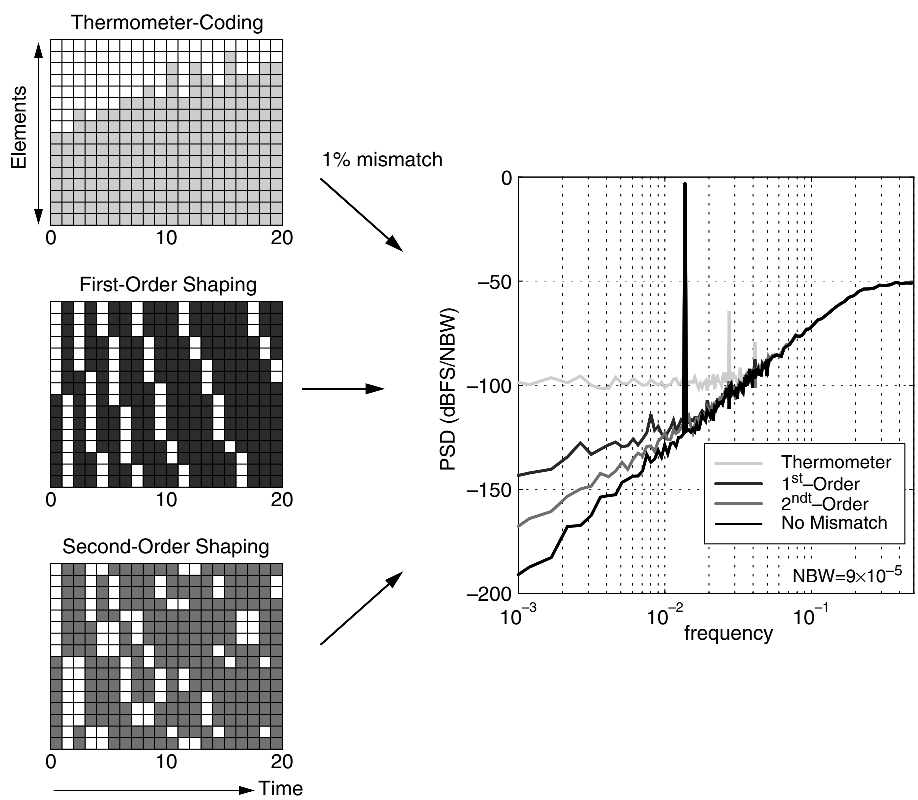


Block diagram of the Element Selection Logic

Compare the usage patterns and example spectra for a 16-element DAC driven with thermometer-coded, 1st-order and 2nd-order mismatch-shaped data generated by a 3rd-order modulator.

```
ntf = synthesizeNTF(3,[],[],4);
```

```
M = 16;
N = 2^14;
fin = round(0.33*N/(2*12));
u = M/sqrt(2)*sin((2*pi/N)*fin*[0:N-1]);
v = simulateDSM(u,ntf,M+1);
sv0 = ds_therm(v,M);
mtf1 = zpk(1,0,1,1);                        % First-order shaping
sv1 = simulateMS(v,mtf1,M);
mtf2 = zpk([ 1 1 ], [ 0 0 ], 1, 1);         % Second-order shaping
sv2 = simulateMS(v,mtf2,M);
ue = 1 + 0.01*randn(M,1);                   % 1% mismatch
dv0 = ue' * sv0;
spec0 = fft(dv0.*ds_hann(N))/(M*N/8);
plotSpectrum(spec0,fin,'g');
```



realizeNTF_ct

Synopsis: `[ABCDc,tdac2] = realizeNTF_ct(ntf,form='FB' ,tdac=[0 1], ordering=[1:n],bp=zeros(-),ABCDc)`

Realize a noise transfer function (NTF) with a continuous-time loop-filter.

Input

`ntf`	The modulator NTF, specified as an LTI object in zero-pole form.
`form`	A string specifying the modulator topology. `FB` Feedback form. `FF` Feedforward form.
`tdac`	The timing for the feedback DAC(s). If `tdac(1)` ≥ 1, direct feedback terms are added to the quantizer. Multiple timings (one or more per integrator) for the FB topology can be specified by making tdac a cell array, e.g. `tdac = {[1,2]; [1 2]; [0.5 1],[1 1.5]; [];}`
`ordering`	A vector specifying which NTF zero-pair to use in each resonator. Default is for the zero-pairs to be used in the order specified in the NTF.
`bp`	A vector specifying which resonator sections are bandpass. The default (zeros(...)) is for all sections to be lowpass.
`ABCDc`	The loop-filter structure, in state-space form. If this argument is omitted, ABCDc is constructed according to `form`.

Output

`ABCDc`	A state-space description of the CT loop-filter.
`tdac2`	A matrix with the DAC timings, one for each input, including ones that were automatically added.

Example

Realize the NTF with a CT system (cf. the example on page 517).

```
>> ntf = zpk([1 1],[0 0],1,1);
>> [ABCDc,tdac2] = realizeNTF_ct(ntf,'FB')

ABCDc =
     0         0    1.0000   -1.0000
1.0000         0         0   -1.5000
     0    1.0000         0    0.0000

tdac2 =
-1    -1
 0     1
```

mapCtoD

Synopsis: `[sys, Gp] = mapCtoD(sys_c,t=[0 1],f0=0)`
Map a MIMO continuous-time system to a SIMO discrete-time equivalent. The criterion for equivalence is that the sampled pulse response of the CT system must be identical to the impulse response of the DT system. I.e. if y_c is the output of the CT system with an input v_c taken from a set of DACs fed with a single DT input v, then y, the output of the equivalent DT system with input v satisfies $y[n] = y_c(n^-)$ for integer n. The DACs are characterized by rectangular impulse responses with edge times specified in the `t` matrix.

Input

`sys_c`	The LTI description of the CT system.
`t`	The edge times of the DAC pulse used to make CT waveforms from DT inputs. Each row corresponds to one of the system inputs; `[-1 -1]` denotes a CT input. The default is `[0 1]` for all inputs except the first, which is assumed to be a CT input.
`f0`	The frequency for which the `Gp` filters' gains are to be set to unity. Default 0 (DC).

Output

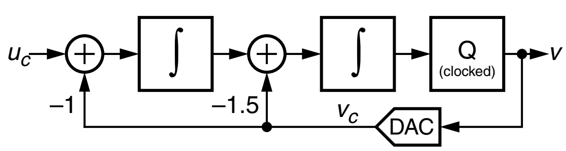
`sys`	The LTI description for the DT equivalent.
`Gp`	The mixed CT/DT prefilters which form the samples fed to each state for the CT inputs.

Reference

R. Schreier and B. Zhang, "Delta-sigma modulators employing continuous-time circuitry," *IEEE Transactions on Circuits and Systems I*, vol. 43, no. 4, pp. 324-332, April 1996.

Example

Map the standard second-order CT modulator shown below to its DT equivalent and verify that the NTF is $(1 - z^{-1})^2$.



$$\begin{bmatrix} \dot{x}_{1c} \\ \dot{x}_{2c} \\ y_c \end{bmatrix} = \left[\begin{array}{cc|cc} 0 & 0 & 1 & -1 \\ 1 & 0 & 0 & -1.5 \\ \hline 0 & 1 & 0 & 0 \end{array}\right] \begin{bmatrix} x_{1c} \\ x_{2c} \\ u_c \\ v_c \end{bmatrix}$$

```
>> LFc = ss([0 0;1 0], [1 -1;0 -1.5], [0 1], [0 0]);
>> tdac = [0 1];
>> [LF,Gp] = mapCtoD(LFc,tdac);
>> ABCD = [LF.a LF.b; LF.c LF.d];
>> H = calculateTF(ABCD)

Zero/pole/gain:
(z-1)^2
-------
  z^2
Sampling time: 1
```

evalTFP

Synopsis: `H = evalTFP(Hs,Hz,f)`
Use this function to evaluate the signal transfer function of a continuous-time (CT) system. In this context `Hs` is the open-loop response of the loop-filter from the *u* input and `Hz` is the closed-loop noise transfer function.

Input

`Hs`	A continuous-time transfer function in zpk form.
`Hz`	A discrete-time transfer function in zpk form.
`f`	A vector of frequencies.

Output

`H`	The value of $H_s(j2\pi f)H_z(e^{j2\pi f})$.

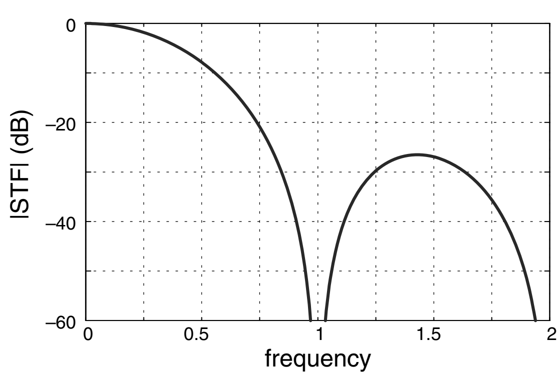
See Also

`evalMixedTF` is a more advanced version of this function which is used to evaluate the individual feed-in transfer functions of a CT modulator.

Example

Plot the STF of the 2nd-order CT system depicted on page 517.

```
Ac = [0 0; 1 0];
Bc = [1 -1; 0 -1.5];
Cc = [0 1];
Dc = [0 0];
LFc = ss(Ac, Bc, Cc, Dc);
L0c = zpk(ss(Ac,Bc(:,1),Cc,Dc(1)));
tdac = [0 1];
[LF,Gp] = mapCtoD(LFc,tdac);
ABCD = [LF.a LF.b; LF.c LF.d];
H = calculateTF(ABCD);
% Yields H=(1-z^-1)^2
f = linspace(0,2,300);
STF = evalTFP(L0c,H,f);
plot(f,dbv(STF));
```



synthesizeQNTF

Synopsis: `ntf = synthesizeQNTF(order=3,OSR=64,f0=0,f0=-60,ING=-20, n_im=order/3)`

Synthesize a noise transfer function (NTF) for a quadrature delta-sigma modulator.

Input

`order`	The order of the NTF.
`OSR`	The oversampling ratio.
`f0`	The center frequency of the modulator.
`NG`	The rms in-band noise gain (dB).
`ING`	The rms image-band noise gain (dB).
`n_im`	Number of image-band zeros.

Output

`ntf`	The modulator NTF, given as an LTI object in zero-pole form.

Bugs

ALPHA VERSION. This function uses an experimental ad hoc method that is neither optimal nor robust.

Example

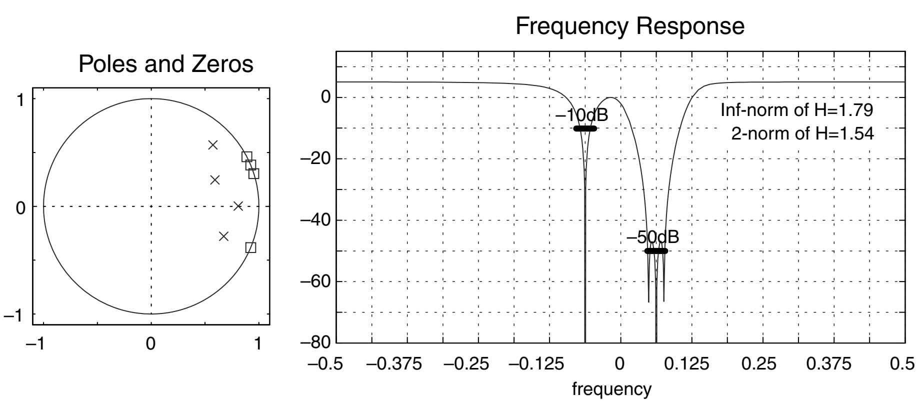
Fourth-order, $OSR = 32$, $f_0 = 1/16$, bandpass NTF with an rms in-band noise gain of -50 dB and an image-band noise gain of -10 dB.

```
>> ntf = synthesizeQNTF(4,32,1/16,-50,-10);

Zero/pole/gain:

      (z-(0.953+0.303i)) (z^2 - 1.85z + 1) (z-(0.888+0.460i))
---------------------------------------------------------------------------
(z-(0.809+0.003i)) (z-(0.591+0.245i)) (z-(0.673-0.279i)) (z-(0.574+0.570i))

Sampling time: 1
```



simulateQDSM

Synopsis: `[v,xn,xmax,y] = simulateQDSM(u,ABCD|ntf,nlev=2,x0=0)`
Simulate a quadrature delta-sigma modulator with a given input. For improved simulation speed, use `simulateDSM` with a 2-input/2-output `ABCDr` argument as indicated in the example in `mapQtoR` on page 522.

Input

`u`	The input sequence to the modulator, given as a $1 \times N$ row vector. Full-scale corresponds to an input of magnitude $nlev - 1$.
`ABCD`	A state-space description of the modulator's loop-filter.
`ntf`	The modulator NTF, given in zero-pole form.
`nlev`	The number of levels in the quantizer. Multiple quantizers are indicated by making `nlev` a column vector.
`x0`	The initial state of the modulator.

Output

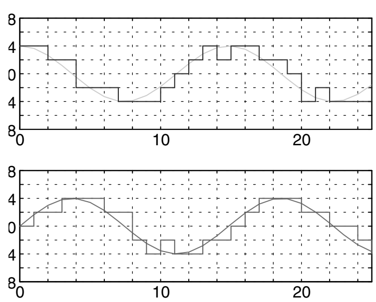
`v`	The samples of the output of the modulator, one for each input sample.
`xn`	The internal states of the modulator, one for each input sample, given as an $n \times N$ matrix.
`xmax`	The maximum absolute values of each state variable.
`y`	The samples of the quantizer input, one per input sample.

Example

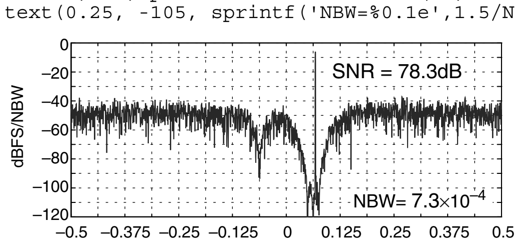
Simulate a 4th-order 9-level quadrature modulator with a half-scale sine-wave input and plot its output in the time and frequency domains.

```
nlev = 9; f0 = 1/16; osr = 32; M = nlev-1;
ntf = synthesizeQNTF(4,osr,f0,-50,-10);
N = 64*osr; f = round((f0+0.3*0.5/osr)*N)/N;
u = 0.5*M*exp(2i*pi*f*[0:N-1]);
v = simulateQDSM(u,ntf,nlev);
```

```
t = 0:25;
subplot(211)
plot(t, real(u(t+1)),'g');
hold on;
stairs(t,real(v(t+1)),'b');
figureMagic(…)
ylabel('real');
```




```
spec = fft(v.*ds_hann(N))/(M*N/2);
spec = [fftshift(spec) spec(N/2+1)];
plot(linspace(-0.5,0.5,N+1), dbv(spec))
figureMagic([-0.5 0.5],1/16,2, [-120 0],10
ylabel('dBFS/NBW')
[f1 f2] = ds_f1f2(osr,f0,1);
fb1 = round(f1*N); fb2 = round(f2*N);
fb = round(f*N)-fb1;
snr = calculateSNR(spec(N/2+1+[fb1:fb2]),f
text(f,-10,sprintf(' SNR = %4.1fdB\n',snr)
text(0.25, -105, sprintf('NBW=%0.1e',1.5/N
```



realizeQNTF

Synopsis: `ABCD = realizeQNTF(ntf,form='FB',rot=0,bn)`
Convert a quadrature NTF into an ABCD matrix for the specified structure.

Input

`ntf`	A zpk object specifying the modulator's NTF.
`form`	A string specifying the modulator topology. `FB` Feedback `PFB` Parallel feedback `FF` Feedforward `PFF` Parallel feedforward
`rot`	`rot=1` means rotate states to make as many coefficients as possible real.
`bn`	The coefficient of the auxiliary DAC for `form = 'FF'`.

Output

`ABCD`	State-space description of the loop-filter.

Example
Determine coefficients for the parallel feedback (PFB) structure.

```
>> ntf = synthesizeQNTF(5,32,1/16,-50,-10);
>> ABCD = realizeQNTF(ntf,'PFB',1)
ABCD =
Columns 1 through 4
 0.8854+0.4648i       0                 0                 0
 0.0065+1.0000i   0.9547+0.2974i       0                 0
     0            0.9715+0.2370i   0.9088+0.4171i       0
     0                0            0.8797+0.4755i   0.9376+0.3477i
     0                0                 0                 0
     0                0                 0            -0.9916-0.1294i
Columns 5 through 7
     0            0.0025           0.0025+0.0000i
     0                0            0.0262+0.0000i
     0                0            0.1791+0.0000i
     0                0            0.6341+0.0000i
 0.9239-0.3827i       0            0.1743+0.0000i
-0.9312-0.3645i       0                 0
```

mapQtoR

Synopsis: `ABCDr = mapQtoR(ABCD)`
Convert a quadrature matrix into its real (IQ) equivalent.

Input

`ABCD`	A complex matrix describing a quadrature system.

Output

`ABCDr`	A real matrix corresponding to ABCD. Each element z in `ABCD` is replaced by a 2×2 matrix to make `ABCDr`. Specifically

$$z \rightarrow \begin{bmatrix} x & -y \\ y & x \end{bmatrix} \text{ where } x = Re(z) \text{ and } y = Im(z).$$

Example
Replace a call to `simulateQDSM` with a faster code block using `simulateDSM`.

```
% v = simulateQDSM(u,ntf,nlev);
ABCD = realizeQNTF(ntf,'FF');
ABCDr = mapQtoR(ABCD);
ur = [real(u); imag(u)];
vr=simulateDSM(ur,ABCDr,nlev*[1;1]);
v = vr(1,:) + 1i*vr(2,:);
```

mapRtoQ

Synopsis: `[ABCDq ABCDp] = mapR2Q(ABCDr)`
Map a real `ABCDr` to a quadrature `ABCD`. `ABCDr` has its states paired (real, imaginary) as indicated above in `mapQtoR`.

Input

`ABCDr`	A real matrix describing a quadrature system.

Output

`ABCDq`	The quadrature (complex) version of ABCDr.
`ABCDp`	The mirror-image system matrix. `ABCDp` is zero if `ABCDr` has no quadrature errors.

calculateQTF

Synopsis: `[ntf stf intf istf] = calculateQTF(ABCDr)`
Calculate the noise and signal transfer functions for a quadrature modulator.

Input

`ABCDr`	A real state-space description of the modulator's loop-filter. I/Q asymmetries may be included in the description. These asymmetries result in non-zero image transfer functions.

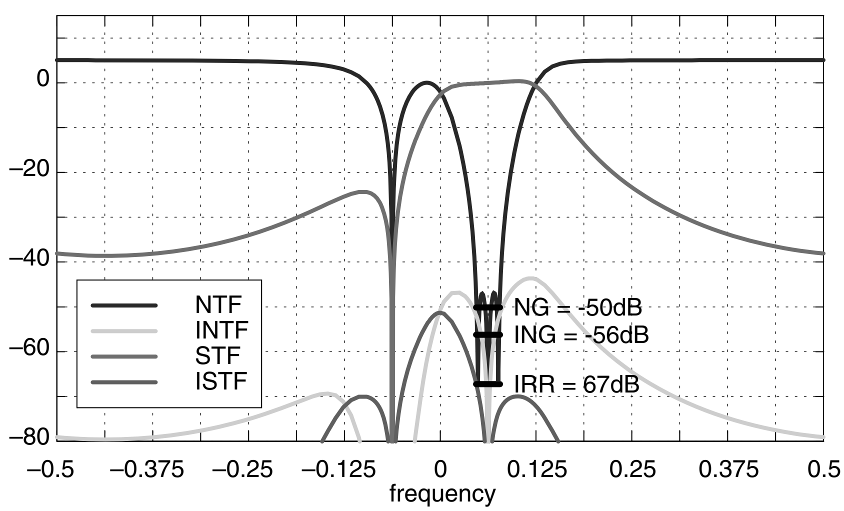
Output

`ntf, stf`	The noise and signal transfer functions.
`intf, istf`	The image noise and image signal transfer functions.

All transfer functions are returned as LTI systems in zero-pole form.

Example
Examine the effect of mismatch in the first feedback.

```
>> ABCDr = mapQtoR(ABCD);
>> ABCDr(2,end) = 1.01*ABCDr(2,end);  % 0.1% mismatch in first feedback
>> [H G HI GI] = calculateQTF(ABCDr);
```



simulateQESL

Synopsis: `[sv,sx,sigma_se,max_sx,max_sy]`
`= simulateQESL(v,mtf,M=16,sx0=[0-])`

Simulate the element selection logic (ESL) of a quadrature differential DAC.

Input

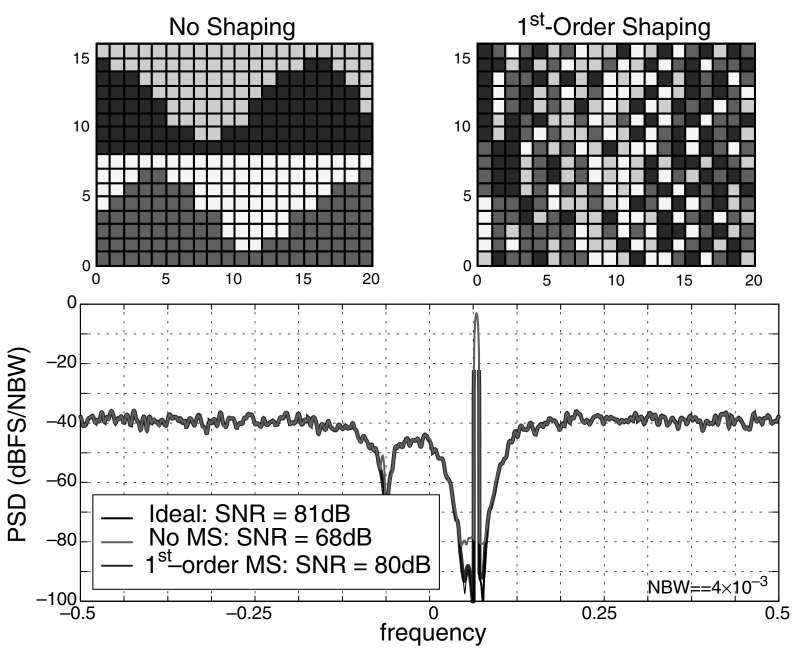
`v`	A vector the digital input values.
`mtf`	The mismatch-shaping transfer function, given in zero-pole form.
`M`	The number of elements. There is a total $2M$ elements.
`sx0`	An $n \times M$ matrix whose columns are the initial state of the ESL.

Output

`sv`	The selection vector: a vector of zeros and ones indicating which elements to enable.
`sx`	An $n \times M$ matrix containing the final state of the ESL.
`sigma_se`	The rms value of the selection error, $se = sv = sy$. `sigma_se` may be used to estimate the power of in-band noise caused by element mismatch.
`max_sx`	The maximum absolute value attained by any state in the ESL.
`max_sy`	The maximum absolute value attained by any input to the VQ.

Example

```
>> mtf1 = zpk(exp(2i*pi*f0),0,1,1);
% First-order complex shaping
>> sv1 = simulateQESL(v,mtf1,M);
```



designHBF

Synopsis: `[f1,f2,info]=designHBF(fp=0.2,delta=1e-5,debug=0)`
Design a hardware-efficient linear-phase half-band filter for use in the decimation or interpolation filter associated with a delta-sigma modulator. This function is based on the procedure described by Saramäki [1]. Note that since the algorithm uses a non-deterministic search procedure, successive calls may yield different designs.

[1] T. Saramäki, "Design of FIR filters as a tapped cascaded interconnection of identical subfilters," *IEEE Transactions on Circuits and Systems*, vol. 34, pp. 1011-1029, 1987.

Input

`fp`	Normalized passband cutoff frequency.
`delta`	Passband and stopband ripple in absolute value.

Output

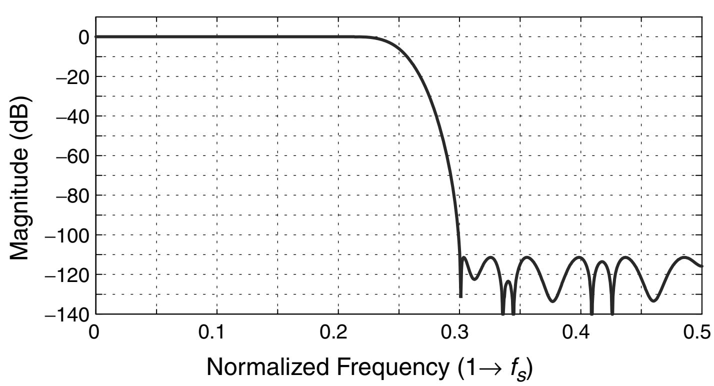
`f1,f2`	Prototype filter and subfilter coefficients and their canonical-signed digit (csd) representation.
`info`	A vector containing the following information data (only set when `debug=1`):

`complexity`	The number of additions per output sample.
`n1,n2`	The length of the `f1` and `f2` vectors.
`sbr`	The achieved stop-band attenuation in dB.
`phi`	The scaling factor for the F2 filter.

Example
Design of a lowpass half-band filter with a cut-off frequency of $0.2f_s$, a passband ripple of less than 10^{-5} and a stopband gain less than 10^{-5} (-100 dB).

```
>> [f1,f2] = designHBF(0.2,1e-5);
>> f = linspace(0,0.5,1024);
>> plot(f, dbv(frespHBF(f,f1,f2)))
```

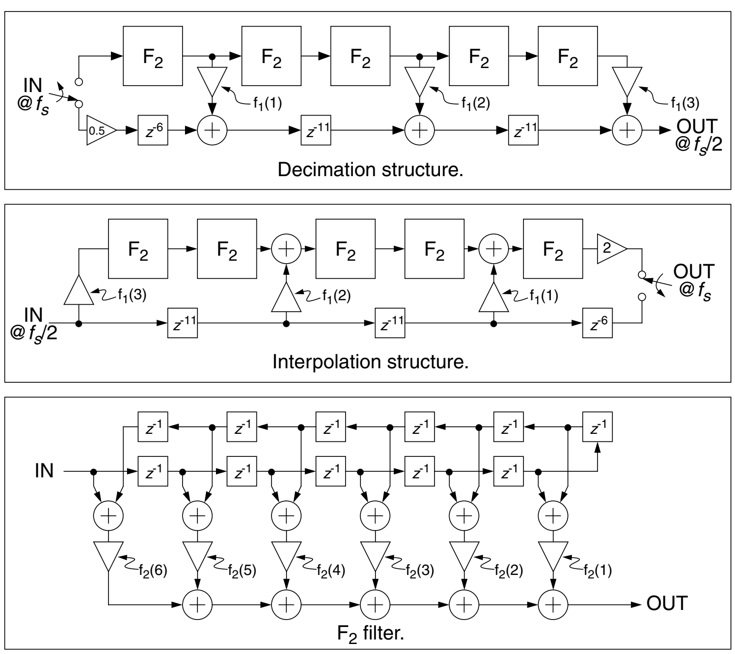
A plot of the filter response is shown below. The filter achieves 109 dB of attenuation in the stopband and uses only 124 additions (no true multiplications) to produce each output sample.



The structure of this filter as a decimation or interpolation filter is shown below. The coefficients and their canonical signed-digit (csd) decompositions are

```
[f1.val]' =     [f2.val]' =     >> f1.csd        >> f2.csd
    0.9453          0.6211      ans =            ans =
   -0.6406         -0.1895          0 -4 -7          -1 -3 -8
    0.1953          0.0957          1 -1  1           1  1 -1
                   -0.0508      ans =            ans =
                    0.0269         -1 -3 -6          -2 -4 -9
                   -0.0142         -1 -1 -1          -1  1 -1
                                ans =            ans =
                                   -2 -4 -7          -3 -5 -9
                                    1 -1  1           1 -1  1
                                                 ans =
                                                     -4 -7 -8
                                                     -1  1  1
                                                 ans =
                                                     -5 -8 -11
                                                      1 -1 -1
                                                 ans =
                                                     -6 -9 -11
                                                     -1  1 -1
```

In the csd expansions, the first row contains the powers of two while the second row gives their signs. For example, $f_1(1) = 0.9453 = 2^0 - 2^{-4} + 2^{-7}$. Since the filter coefficients use only 3 csd terms, each multiply-accumulate operation shown in the diagram below needs only 3 additions. An implementation of this 110$^{\text{th}}$-order FIR filter therefore needs only $3 \times 3 + 5 \times (3 \times 6 + 6 - 1) = 124$ additions at the low $(f_s/2)$ rate.



Decimation structure.

Interpolation structure.

F_2 filter.

simulateHBF

Synopsis: `y = simulateHBF(x,f1,f2,mode=0)`
Simulate a Saramäki half-band filter (see `designHBF` on page 525) in the time domain.

Input

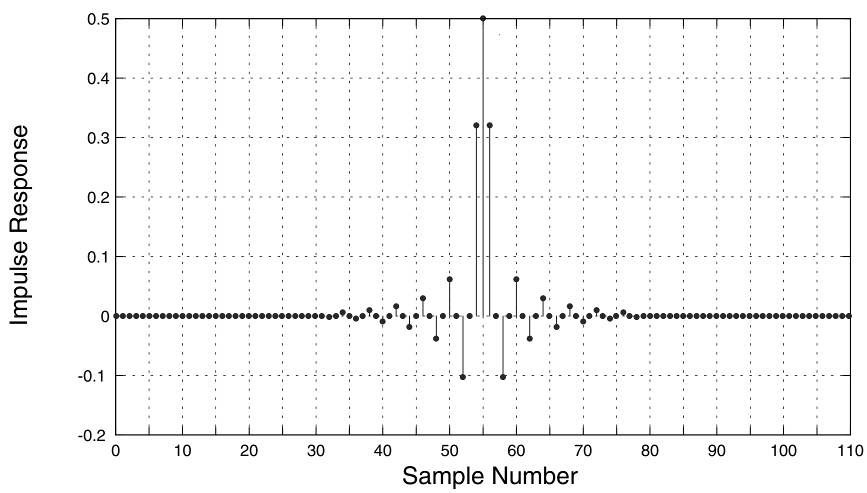
`x`	The input data.
`f1,f2`	Filter coefficients. `f1` and `f2` can be vectors of values or struct arrays like those returned from `designHBF`.
`mode`	This flag determines whether the input is filtered, interpolated, or decimated according to the following: `0` Plain filtering, no interpolation or decimation. `1` The input is interpolated. `2` The output is decimated, even samples are taken. `3` The output is decimated, odd samples are taken.

Output

`y`	The output data.

Example
Plot the impulse response of the HBF designed on the previous page.

```
>> N = (2*length(f1)-1)*2*(2*length(f2)-1)+1;
>> y = simulateHBF([1 zeros(1,N-1)],f1,f2);
>> stem([0:N-1],y);
>> figureMagic([0 N-1],5,2, [-0.2 0.5],0.1,1)
```



designPBF

Synopsis: `[C, e, x0] = designPBF(N,M,pb,pbr,sbr,ncd,np,ns,fmax)`
Design a symmetric polynomial-based filter (PBF) according to Hunter's method [1]. `designPBF` requires the Optimization toolbox.

[1] M. T. Hunter, "Design of polynomial-based filters for continuously variable sample rate conversion with applications in synthetic instrumentation and software defined radio," Ph.D. thesis, University of Florida, 2008.

Input

`N=10`	Number of polynomial pieces.
`M=5`	Order of the polynomial pieces.
`pb=0.25`	Passband width. Relative to the input sample rate, the passband is $[0, \mathtt{pb}]$ and the stopband is $[1-\mathtt{pb}, \infty)$. Use `pb = 0.5/OSR` where `OSR` is the oversampling ratio of the input.
`pbr=0.1`	Passband ripple in dB.
`sbr=-100`	Stobpand ripple in dB.
`ncd=0`	Number of continuous derivatives. To allow the impulse response itself to be discontinuous, use `ncd = -1`.
`np=100`	Number of points in the passband.
`ns=1000`	Number of points in the stopband.
`fmax=5`	Maximum frequency checked in the stopband.

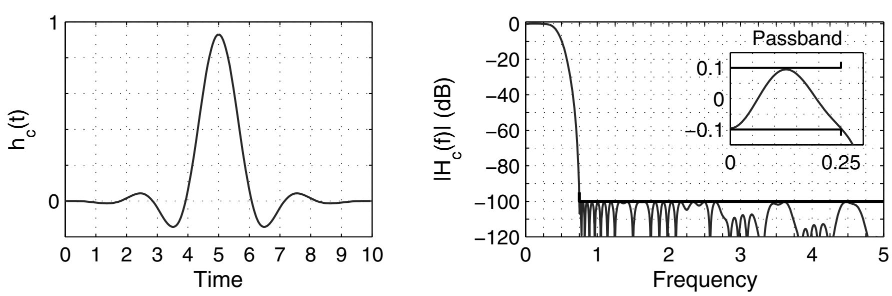
Output

`C`	$\mathtt{N} \times (\mathtt{M}+1)$ matrix containing the coefficients of the polynomial pieces. Piece i is $p_i(x) = C(i,1) + C(i,2)x + C(i,3)x^2 + ... + C(i, M+1)x^M$.
`e`	The maximum weighted error. $\mathtt{e} \le 1$ indicates the specs were met.
`x0=-0.5`	Offset on the polynomial argument, i.e. $x = \mu + \mathtt{x0}$, where $\mu \in [0,1]$.

Example

Construct a 10-segment PBF using polynomials of order 5 for interpolating signals with an input OSR of 2. Aim for a passband ripple of 0.1 dB and a stopband ripple of −100 dB.

```
[C, e, x0] = designPBF(10, 5, 0.5/2, 0.1, -100);
[hc, t] = impulsePBF(C,20,x0);
subplot(121); plot(t, hc, 'Linewidth', 1);
f = linspace(0,5,1000);
Hc = frespPBF(f,C,x0);
subplot(122); plot(f, dbv(Hc), 'Linewidth', 1);
```



predictSNR

Synopsis: `[snr,amp,k0,k1,sigma_e2] = predictSNR(ntf,OSR=64,amp=...,f0=0)`
Use the describing function method of Ardalan and Paulos [1] to predict the signal-to-noise ratio (SNR) in dB for various input amplitudes. This method is only applicable to binary modulators.

[1] S. H. Ardalan and J. J. Paulos, "Analysis of nonlinear behavior in delta-sigma modulators," *IEEE Transactions on Circuits and Systems*, vol. 34, pp. 593-603, June 1987.

Input

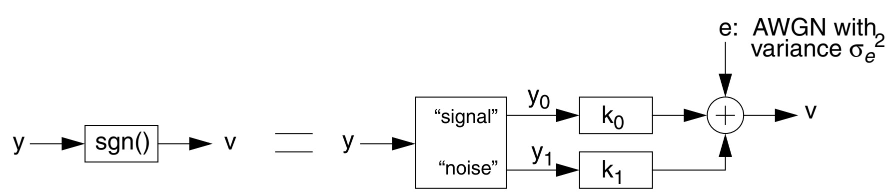
`ntf`	The modulator NTF, given in zero-pole form.
`OSR`	The oversampling ratio.
`amp`	A row vector listing the amplitudes to use. `amp` defaults to $[-120-110...-20-15-10-9-8...0]$ dB, where 0 dB means a full-scale (peak value = 1) sine wave.
`f0`	The center frequency of the modulator.

Output

`snr`	A row vector containing the predicted SNRs
`amp`	A row vector listing the amplitudes used.
`k0`	A row vector containing the signal gain of the quantizer model.
`k1`	A row vector containing the noise gain of the quantizer model.
`sigma_e2`	A row vector containing the mean square value of the noise in the quantizer model.

Example
See the example on page 509.

The Quantizer Model
The binary quantizer is modeled as a pair of linear gains and a noise source, as shown in the figure below. The input to the quantizer is divided into signal and noise components which are processed by signal-dependent gains k_0 and k_1. These components are added to a noise source, which is assumed to be white and to have a Gaussian distribution to produce the quantizer output. The variance σ_e^2 of the noise source is also signal-dependent.



findPIS, find2dPIS (in the PosInvSet subdirectory)

Synopsis:
```
[s,e,n,o,Sc] = findPIS(u,ABCD,nlev=2,options)
[s,e,n,o,Sc] = findPIS(u,ABCD,nlev=2,options)
options = [dbg=0 itnLimit=2000 expFactor=0.005 N=1000 skip=100]
```

Find a convex positively-invariant set for a delta-sigma modulator. `findPIS` requires compilation of the `qhull` mex file; `find2dPIS` does not but is limited to second-order systems.

This function is an implementation of the method described in [1][1]

Input

`u`	The input to the modulator. If `u` is a scalar, the input to the modulator is constant. If `u` is a 2×1 vector, the input to the modulator may be any sequence whose samples lie in the range $[u(1), u(2)]$.
`ABCD`	A state-space description of the modulator loop-filter.
`nlev`	The number of quantizer levels.
`dbg`	Set `dbg=1` to see a graphical display of the iterations.
`itnLimit`	The maximum number of iterations.
`expFactor`	The expansion factor applied to the hull before every mapping operation. Increasing `expFactor` decreases the number of iterations but results in sets which are inflated.
`N`	The number of points to use when constructing the initial guess.
`skip`	The number of time steps to run the modulator before observing the state. This handles the possibility of transients in the modulator.
`qhullArgA`	The 'A' argument to the `qhull` program. Adjacent facets are merged if the cosine of the angle between their normals is greater than the absolute value of this parameter. Negative values imply that the merge operation is performed during hull construction, rather than as a post-processing step.
`qhullArgC`	The 'C' argument to the `qhull` program. A facet is merged into its neighbor if the distance between the facet's centrum (the average of the facet's vertices) and the neighboring hyperplane is less than the absolute value of this parameter. As with the above argument, negative values imply pre-merging while positive values imply post-merging.

Output

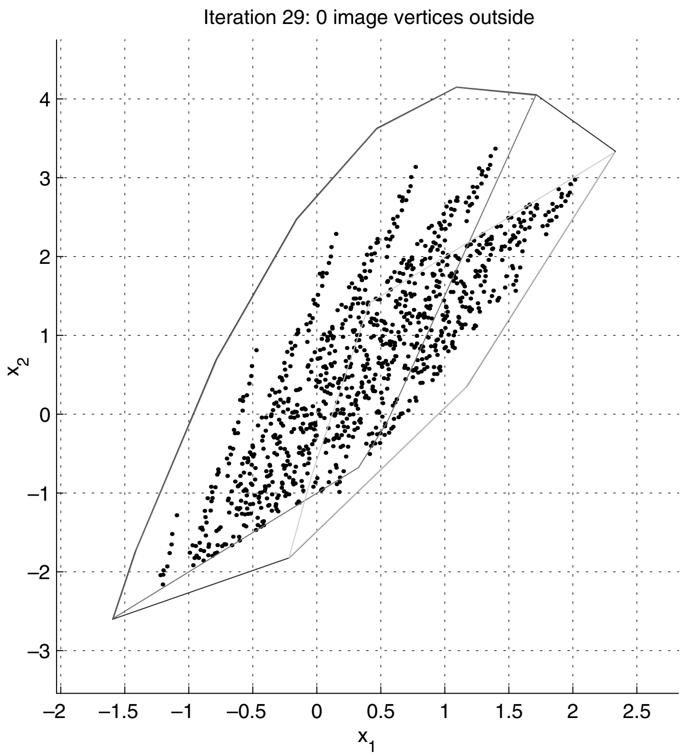
`s`	The vertices of the set ($dim \times n_v$).
`e`	The edges of the set, listed as pairs of vertex indices ($2 \times n_e$).
`n`	The normals for the facets of the set ($dim \times n_f$).
`o`	The offsets for the facets of the set ($1 \times n_f$).
`Sc`	The scaling matrix which was used internally to round out the set.

Find a positively-invariant set for the second-order modulator with an input of $1/\sqrt{7}$.

```
>> ABCD = [
1     0     1    -1
1     1     1    -2
0     1     0     0];
>> s = find2dPIS(sqrt(1/7),ABCD,1)
```

[1] [1] R. Schreier, M. Goodson and B. Zhang "An algorithm for computing convex positively invariant sets for delta-sigma modulators," *IEEE Transactions on Circuits and Systems I*, vol. 44, no. 1, pp. 38-44, January 1997.

```
s =
Columns 1 through 7
-1.5954   -0.2150    1.1700    2.3324    1.7129    1.0904    0.4672
-2.6019   -1.8209    0.3498    3.3359    4.0550    4.1511    3.6277
Columns 8 through 11
-0.1582   -0.7865   -1.4205   -1.5954
2.4785    0.6954   -1.7462   -2.6019
```



findPattern

Synopsis: `[data, snr] = findPattern(N=1024,OSR=64,ntf,ftest,Atest, f0=0,nlev=2,quadrature=0,dbg=0)`

Use delta-sigma modulation to create a length-N data-stream that has good spectral properties when repeated.

Input

`N`	The length of the data record.
`OSR`	The oversampling ratio.
`NTF`	The modulator NTF.
`ftest`	The signal frequency. `ftest` may be a vector.
`Atest`	The target output level as a fraction of full-scale.
`f0`	The center frequency.
`nlev`	The number of levels in the output data.
`quadrature`	A flag which indicates to use quadrature modulation.
`dbg`	A flag which enables showing the progress of the iterations.

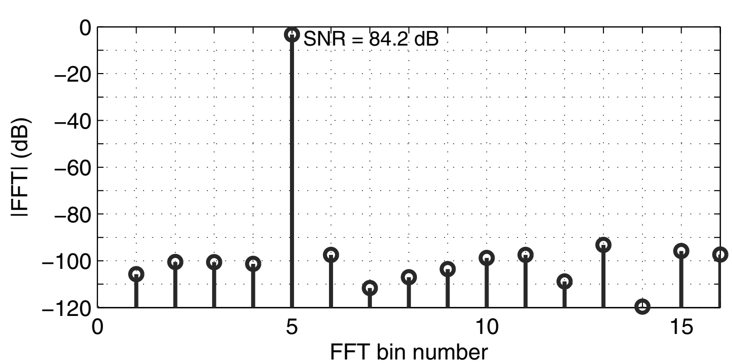
Output

`data`	$1 \times N$ data record.
`snr`	The in-band signal-to-noise ratio, in dB.

Example

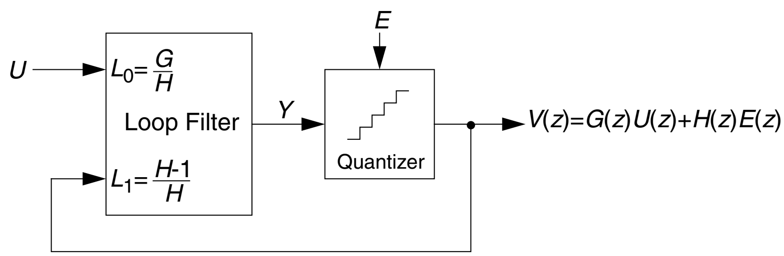
Length-1024 data record containing a −3-dBFS, 5-cycle sine wave with low in-band noise for an oversampling ratio of 32.

```
N = 1024;
osr = 32;
ntf = synthesizeNTF(5,osr,1,1.5);
ftest = 5/N;
Atest = undbv(-3);
[data snr] = findPattern(N,osr,ntf,ftest,Atest);
spec = fft(data)/(N/2);
inband = 0:ceil(N/(2*osr));
lollipop(inband,dbv(spec(inband+1)),'b',2,-120);
```



Modulator Model

A delta-sigma modulator with a single quantizer is assumed to consist of quantizer connected to a loop-filter as shown in the diagram below.



The Loop Filter

The loop-filter is described by an *ABCD matrix*. For single-quantizer systems, the loop-filter is a two-input, one-output linear system and ABCD is an $(n+1) \times (n+2)$ matrix, partitioned into A $(n \times n)$, B $(n \times 2)$, C $(1 \times n)$ and D (1×2) sub-matrices as shown below:

$$ABCD = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array} \right] \tag{B.1}$$

The equations for updating the state and computing the output of the loop-filter are

$$\begin{aligned} x[n+1] &= Ax[n] + B \begin{bmatrix} u[n] \\ v[n] \end{bmatrix} \\ y[n] &= Cx[n] + D \begin{bmatrix} u[n] \\ v[n] \end{bmatrix} \end{aligned} \tag{B.2}$$

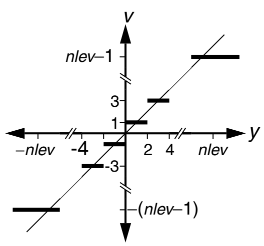
This formulation is sufficiently general to encompass all single-quantizer modulators that employ linear loop-filters. The toolbox currently supports translation to/from an ABCD description and coefficients for the following topologies:

CIFB	Cascade-of-integrators, feedback form.
CIFF	Cascade-of-integrators, feedforward form.
CRFB	Cascade-of-resonators, feedback form.
CRFF	Cascade-of-resonators, feedforward form.
CRFBD	Cascade-of-resonators, feedback form, delaying quantizer.
CRFFD	Cascade-of-resonators, feedforward form, delaying quantizer
Stratos	A CIFF-like structure supporting NTF zeros on the unit circle (Jeff Gealow)
DSFB	Double-sampled, feedback (Dan Senderowicz)

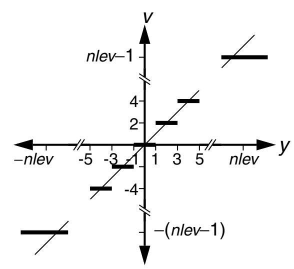
Multi-input and multi-quantizer systems can also be described with an ABCD matrix and Eq. (B.2) will still apply. For an n_i-input, n_o-output modulator, the dimensions of the sub-matrices are $A : n \times n$, $B : n \times (n_i + n_o)$, $C : n_o \times n$ and $D : n_o \times (n_i + n_o)$.

The Quantizer

The quantizer is ideal, producing integer outputs centered about zero. Quantizers with an even number of levels are of the mid-rise type and produce outputs which are odd integers. Quantizers with an odd number of levels are of the mid-tread type and produce outputs which are even integers.



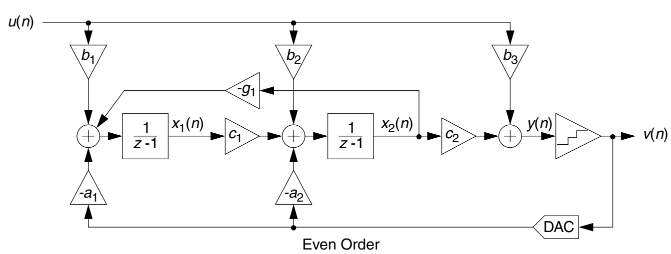
Transfer curve of a quantizer with an even number of levels.



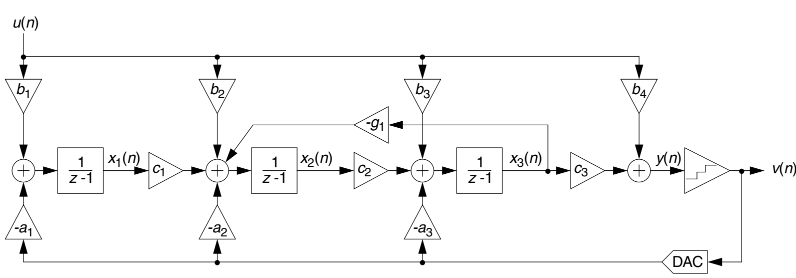
Transfer curve of a quantizer with an odd number of levels.

Supported Modulator Topologies

CIFB Structure

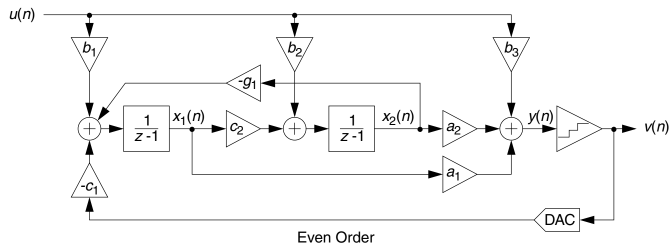


Even Order

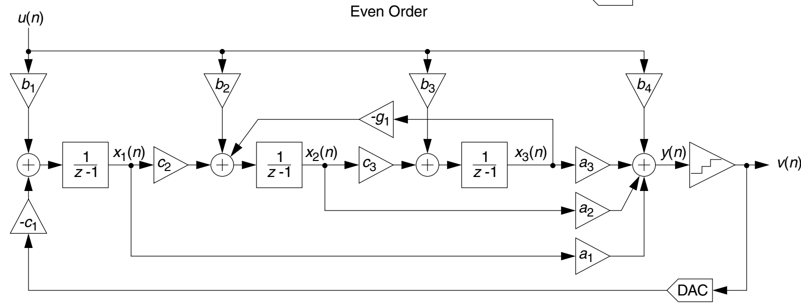


Odd Order

CIFF Structure

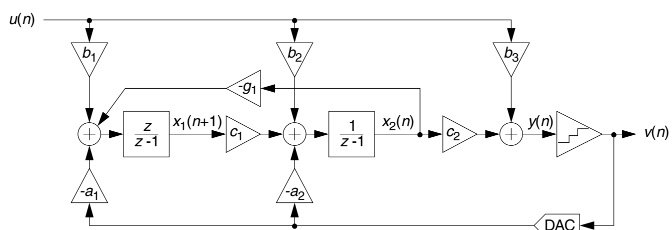


Even Order



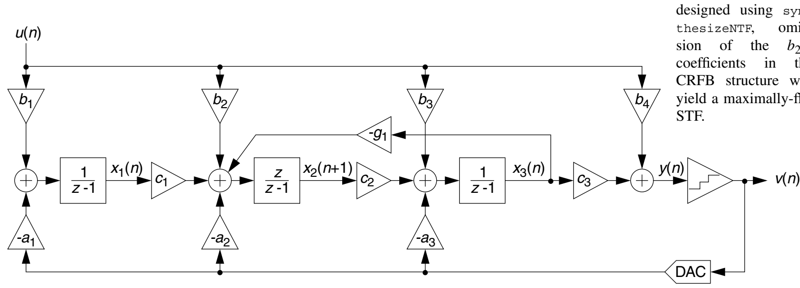
Odd Order

CRFB Structure



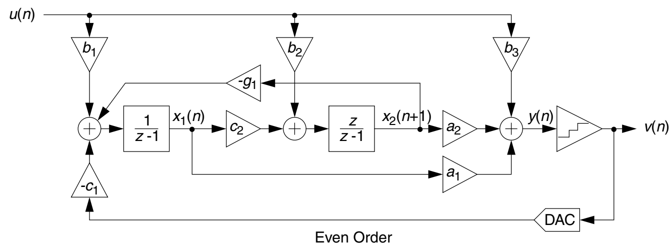
Even Order

Note that with NTFs designed using `synthesizeNTF`, omission of the $b_{2...}$ coefficients in the CRFB structure will yield a maximally-flat STF.

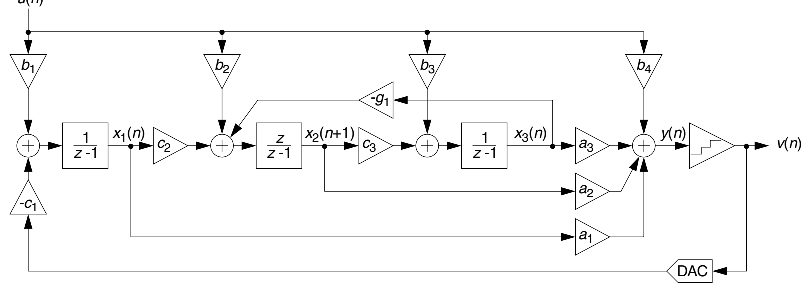


Odd Order

CRFF Structure

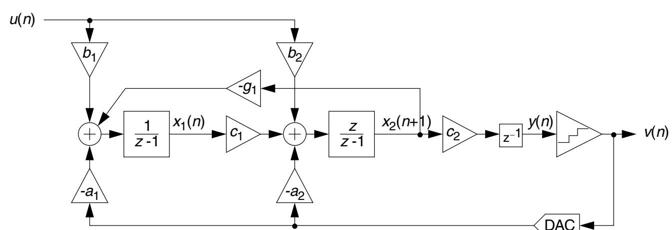


Even Order



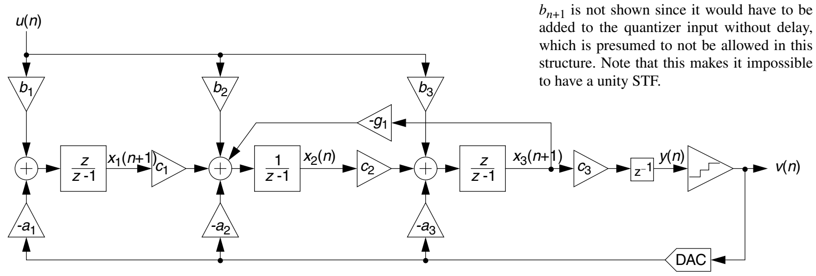
Odd Order

CRFBD Structure



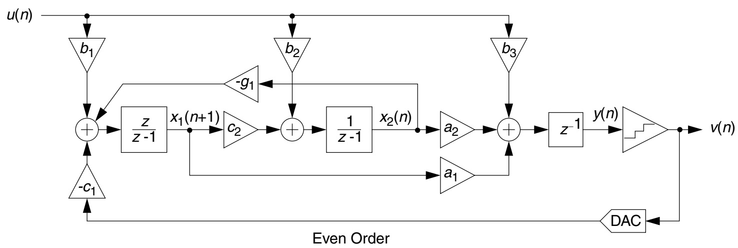
Even Order

b_{n+1} is not shown since it would have to be added to the quantizer input without delay, which is presumed to not be allowed in this structure. Note that this makes it impossible to have a unity STF.

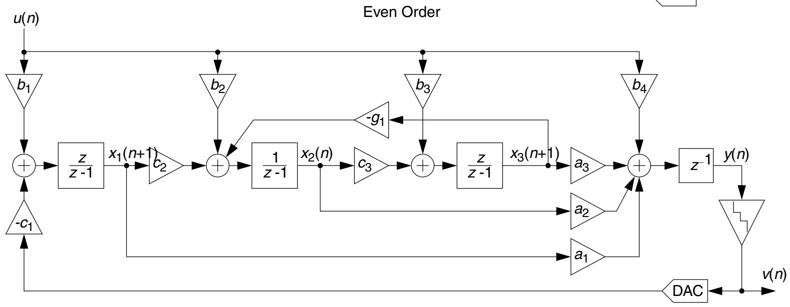


Odd Order

CRFFD Structure

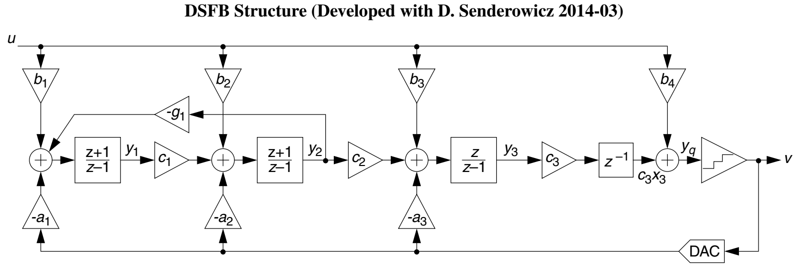


Even Order

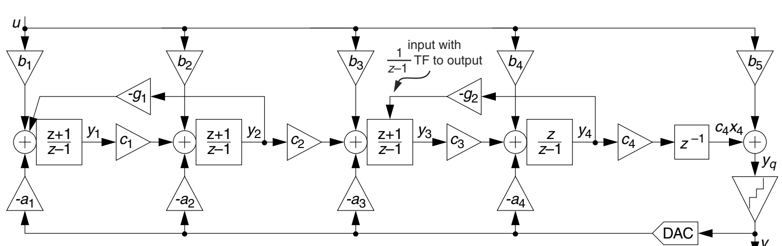


Odd Order

DSFB Structure (Developed with D. Senderowicz 2014-03)



3rd Order



4th Order