DSP Software Toolkit

for

EE395 Introduction to Digital Signal Processing
EE545 Digital Signal Processing
EE589 Digital Speech Processing
EE594 Adaptive Signal Processing

Phillip L. De Leon
New Mexico State University
Klipsch School of Electrical and Computer Engineering
Box 30001, Dept. 3-O
Las Cruces, New Mexico 88003-8001
(575) 646-DSP1
pdeleon@nmsu.edu

Contents

Chapter 1

Introduction

The purpose of this laboratory book is to guide the student in the design and construction of a MATLAB®-based Digital Signal Processing (DSP) toolkit. The purpose of the toolkit is to aid in the practice and understanding of DSP (at both undergraduate and graduate levels), Adaptive Signal Processing, and Digital Speech Processing. We firmly believe in "learning by doing" and in DSP this means nothing short of writing software to analyze and process signals. Our choice in using the Mathwork's MATLAB environment is to make this "rite-of-passage" as pain-free as possible.

In the Klipsch School of Electrical & Computer Engineering at New Mexico State University, the full edition of MATLAB along with various licensed toolboxes is available in Thomas and Brown Room 206 (DSP laboratory).

Chapter 2

MATLAB

2.1 Getting Started

MATLAB is considered by many in the control, communications, and digital signal processing community to be the *de facto* numerical computing environment. Good tutorials on MATLAB can be downloaded from the following web pages

`http://www.math.ucsd.edu/~bdriver/21d-s99/matlab-primer.html`
`http://www.mathworks.com/help/pdf_doc/matlab/getstart.pdf`

Additional information on MATLAB as well as user-contributed signal processing codes for MATLAB can be found at

`http://www.mathworks.com`

As you will see many of the software tools developed in this text are already provided in MATLAB as built-in functions or in the various companion toolboxes. The purpose in writing the tools described in this workbook from scratch is not to re-invent the wheel but rather to:

- understand the inner workings of the tool at the algorithm level and see the "theory in action"
- prevent the "garbage in, garbage out" syndrome from misuse of tools
- see how DSP tools are built upon each other
- have better agreement and understanding with the textbook

In this workbook, we will note the equivalent tools in MATLAB for your benefit, however, your projects are to use your tools unless otherwise noted.

2.2 MATLAB Student Version

The MathWorks offers a Student Version of MATLAB at a significantly reduced price. Although, we do not require purchasing of the Student Version, some students may find it convenient to work from home. For further information see

`http://www.mathworks.com/academia/student_version/`

2.3 Octave

Octave is an open-source (free) software program that is quite similar to MATLAB and may be used to develop the DSP software toolkit. For further information see

`http://www.gnu.org/software/octave/`

2.4 Functions and Main Programs

The DSP toolkit will be written as a series of functions or "tools" called by a main program. Each tool should be a stand-alone piece of code acting like a black-box: you put something in and something comes out. Homework projects then involve nothing more then calling the tools from the main program. The following two .m files illustrate this approach. Here the `main` program calls `sampfunc` with two input values, `a` and **b**. `Sampfunc` adds these two inputs and returns the output to `main`.

Example

main.m

```
a = 1;
b = 2;
c = sampfunc(a,b);
```

sampfunc.m

```
function z = sampfunc(x,y)
z = x + y;
```

From the command window we type

```
>>main
```

to execute our code.

Chapter 3

Tips and Tricks

Listed below (and in no particular order) are some tips and tricks for use in building your DSP toolkit.

3.1 Main Program Header

Listed below is a sample header that you should include at the top of each main program you write in MATLAB.

```
%*******************************************************************************
%NAME:                                                              DATE:
%-------------------------------------------------------------------------------
%PURPOSE: (Describe what the code does)
%-------------------------------------------------------------------------------
%OUTLINE: (High-level, step-by-step outline of your algorithm.  Your commented
%         code should fill in the details.)
%-------------------------------------------------------------------------------
%FUNCTIONS: List all functions needed for this program to run
%NOTES: Any addition notes such as parameters to play with
%*******************************************************************************
```

You can find this header at

`http://www.ece.nmsu.edu/~pdeleon/Teaching/EE545/mainhead.m`

3.2 Sample Function Header

Listed below is a sample header that you should include at the top of each function you write in MATLAB.

```
function [output_variables] = function_name(input_variables)
% Description: This function ...
%
% Call Syntax: [output_variables] = function_name(input_variables)
%
% Input Arguments:
%       Name: x
%       Type: vector
%       Description: input signal
```

```
10  %
11  % Output Arguments:
12  %       Name: y
13  %       Type: vector
14  %       Description: output signal
15  %
16  % Creation Date:
17  % Last Revision:
18  %
19  % Required subroutines:
20  %
21  % Notes:
22  %
23  % References:
24  %*************************************************************************
25
26  %------------------
27  % Check valid input
28  %------------------
29  if (nargin ~= 2)
30     error('Error (function_name): must have 2 input arguments.');
31  end;
32
33  %-----------
34  % Initialize
35  %-----------
36
37  %-----
38  % Main
39  %-----
```

You can find this header at

`http://www.ece.nmsu.edu/~pdeleon/Teaching/EE545/funchead.m`

3.3 MATLAB PATH

Many students have problems telling MATLAB where their tools are located. To do this, go to the File menu and select Set Path. Type in the full path to where your tools are located.

Example

`c:\temp\DSPtools`

3.4 Preallocating Vectors

If you need to build a vector element-by-element, *always* preallocate memory to the vector. Otherwise as your vector grows, MATLAB will reallocate new memory and transfer the old vector to new memory for each new element. This results in very inefficient execution.

Example

```
for n = 1:10000; % no prealloc
   x(n) = n; % runs
end; % S-L-O-W-L-Y
```

Now add the following line before the `for` statement and compare run-times.

```
x = zeros(10000,1); % prealloc
```

As a side note, you should never initialize a vector with a for loop as in the above example–see the next section on avoiding loops.

3.5 Avoid Loops–Vectorize Code

MATLAB is designed to operate efficiently on vectors. Therefore any code you have should be "vectorized" so that it executes as quickly as possible. Vectorizing can often get you an order of magnitude ($10\times$) improvement in execution time. Below are two examples of this process.

Example

The following code segment creates a vector **x** with the first 10000 integers as in the previous example but without the use of a for loop and therefore, much more efficiently.

```
x = [1:10000]';
```

Example

The following two code segments each illustrate an element-by-element vector multiply for the two vectors **a** and **b**.

```
% Code segment 1
for n = 1:10000;  % non-vectorized example
   c(n) = a(n) * b(n);
end;
```

```
% Code segment 2
c = a .* b;  % vectorized example
```

3.6 Debugging

MATLAB allows you to set breakpoints in your code so that you can examine variables. Breakpoints are established by placing the `keyboard` command in your code. Once your code executes the `keyboard` command, control goes to the Command window where you can examine variables. Note that in keyboard mode, the prompt changes to `K>>`. You can exit this mode by typing `return`. You may wish to place a `pause` command after `keyboard` to allow a simple halting of the program with CTRL-C. To continue through the pause, hit any key.

Example

```
n = [0:10000]';  % vector of integers
keyboard;  % type return to continue
pause;  % hit any key to continue or CTRL-C to break
m = n.^2;  % square each element
```

3.7 Determining the Number of Inputs

MATLAB provides a simple way to determine how many inputs have been passed to a function. This is useful for specifying optional inputs. The `nargin` variable will provide the number of arguments that are input.

Example

```
function z = sampfunc(a,b,c)
if (nargin == 2)
   z = a + b;
elseif (nargin == 3)
   z = a + b + c;
else
   error('Error (sampfunc): must have 2 or 3 input arguments.');
end;
```

3.8 No Zero Indexing

One deficiency of MATLAB is the lack of a zero index, i.e. $x(0)$. Naturally one only needs to offset the indexing by 1 and base all code on this offset. There are two ways to do this–choose one and stick with it.

Example

```
% offset the loop counter
for n = 1:N
   h(n) = n-1;
end;

% offset the vector index
for n = 0:N-1
   h(n+1) = n;
end;
```

3.9 Column vs. Row Vectors

As a matter of convention, we will treat all signals as column vectors with the exception of several of the adaptive signal processing tools. For these tools only, signals will be stored as row vectors in order to stay consistent with the adaptive signal processing text. All software should reflect this convention. In MATLAB, $'$ denotes complex conjugate transpose (Hermitian) while $.'$ denotes transpose. When applied to a vector of real numbers, $'$ and $.'$ are equivalent.

Example

```
n = [0:N-1]';  % column index vector
```

3.10 Reverse Indexing

In order to create a descending list of values or index a vector in reverse order, we add a negative step value.

Example

```
n = [N-1:-1:0]';  % descending list of values
```

We use the same idea in order to reverse index a vector.

Example

```
x(n:-1:n-N+1);  % reverse indexing gives last N samples starting with x(n)
```

Note that when creating a list of values, the step value can be fractional.

Example

```
n = [0:0.5:10]';  % fractional step value
```

3.11 Inner Product

One of the fundamental mathematical operations in DSP is the inner (or dot) product

$$\begin{aligned} c &= \mathbf{a}^H \mathbf{b} \\ &= \sum_{n=0}^{N-1} \bar{a}(n) b(n) \end{aligned} \tag{3.1}$$

where ^{H} denotes the Hermitian (complex conjugate transpose) and $^-$ denotes complex conjugation. In MATLAB the inner product is computed as (assuming $\mathbf{a}$ and $\mathbf{b}$ are length N column vectors)

Example

```
c = a' * b
```

3.12 Number Format

MATLAB defaults to presenting numbers with four digits of precision. You can change this by using the `format` command.

Example

```
format long
```

3.13 Acronyms and Standard Variables

The following list contains notation used in this lab book as well as variable names reserved for the tools in this toolkit.

$\mathbf{a}$	vector of feedforward gains
A	amplitude
$\mathbf{b}$	vector of feedback gains
BPF	bandpass filter
d	desired signal
D	downsampling factor
f	frequency
f_s	sampling rate
^{H}	matrix/vector Hermitian (complex conjugate transpose)
$\mathbf{h}$	impulse response
H	frequency response
HP	highpass
HPF	highpass filter
J	mean-squared error vector
L	input signal length
LP	lowpass
LPF	lowpass filter
μ	step size
M	model order
n	index variable for vectors
N	length of Fourier transform, filter/vector order, or filter/vector length (choice will be clear from context)
ϕ	phase
p	model order
$\mathbf{p}$	cross-correlation vector
r	correlation vector
R, $\mathbf{R}$	correlation matrix
S	power spectral density or periodogram
^{T}	matrix/vector transpose
Ω, ω	frequency variable continuous-, discrete-time respectively
$u, \mathbf{u}$	random input signal
U	upsampling factor
v, $\mathbf{v}$	random output signal
w	adaptive filter coefficients
$\mathbf{w}$	window vector
$\mathbf{x}$	input signal
$\mathbf{X}$	Fourier transform of input signal (spectrum), $\mathbf{x}$
$\mathbf{y}$	output signal

Chapter 4

Signal Synthesis and Visualization

This chapter contains several tools used to synthesize (generate) common signals, read in signals from WAV files, and visualize or plot the signals. You may choose to use the supplied "freebie" MATLAB code or write your own but be sure to use the sample function header.

4.1 SINGEN.M

Purpose

This function returns the samples of a sinusoid.

Input

A, amplitude of sinusoid
f, frequency of sinusoid in Hertz
ϕ, phase of sinusoid in radians
f_s, sampling rate in Hertz
duration, duration of sinusoid in seconds

Output

x, vector of sinusoid samples

Algorithm

$$x(n) = A\sin\left(2\pi\frac{f}{f_s}n+\phi\right) \tag{4.1}$$

Code

```
n = [0:fs*duration]';  % index vector
x = A*sin(2*pi*f/fs*n+phi);
```

Example

```
>>x = singen(1,60,0,1000,10/1000)

x =
                  0
   0.36812455268468
   0.68454710592869
   0.90482705246602
   0.99802672842827
   0.95105651629515
   0.77051324277579
   0.48175367410172
   0.12533323356430
  -0.24868988716485
  -0.58778525229247
```

4.2 COSINGEN.M

Purpose

This function returns the samples of a cosinusoid.

Input

A, amplitude of sinusoid
f, frequency of sinusoid in Hertz
ϕ, phase of sinusoid in radians
f_s, sampling rate in Hertz
duration, duration of sinusoid in seconds

Output

x, vector of cosinusoid samples

Algorithm

$$x(n) = A\cos\left(2\pi\frac{f}{f_s}n + \phi\right) \tag{4.2}$$

Code

```
n = [0:fs*duration]';  % index vector
x = A*cos(2*pi*f/fs*n+phi);
```

Example

```
>>x = cosingen(1,60,0,1000,10/1000)

x =
   1.00000000000000
   0.92977648588825
   0.72896862742141
   0.42577929156507
   0.06279051952931
  -0.30901699437495
  -0.63742398974869
  -0.87630668004386
  -0.99211470131448
  -0.96858316112863
  -0.80901699437495
```

4.3 CEXPGEN.M

Purpose

This function returns the samples of a complex exponential.

Input

A, amplitude of complex exponential
f, frequency of sinusoid in Hertz
ϕ, phase of sinusoid in radians
f_s, sampling rate in Hertz
duration, duration of sinusoid in seconds

Output

x, vector of complex exponential samples

Algorithm

$$x(n) = Ae^{j(2\pi \frac{f}{f_s} n + \phi)} \tag{4.3}$$

Code

```
n = [0:fs*duration]';  % index vector
x = A*exp(j*(2*pi*f/fs*n+phi));
```

Example

```
>>x = cexpgen(1,0.25,0,1,10)

x =
  1.00000000000000
  0.00000000000000 + 1.00000000000000i
 -1.00000000000000 + 0.00000000000000i
 -0.00000000000000 - 1.00000000000000i
  1.00000000000000 - 0.00000000000000i
  0.00000000000000 + 1.00000000000000i
 -1.00000000000000 + 0.00000000000000i
 -0.00000000000000 - 1.00000000000000i
  1.00000000000000 - 0.00000000000000i
  0.00000000000000 + 1.00000000000000i
 -1.00000000000000 + 0.00000000000000i
```

4.4 PLOTDSIG.M

Purpose

This function will plot and label a discrete-time signal.

Input

x, vector of samples
option, optional argument with values as follows
0–Samples are not connected (default)
1–Samples are connected
2–Samples are connected and discrete values marked

Output

Plot of a discrete-time signal

Code

```
if ((nargin ~= 1) & (nargin ~= 2))
   error('Error (plotdsig): must have 1 or 2 input arguments.');
end;
if (nargin == 1)
   option = 0;  % default option
end;

n = [0:length(x)-1]';  % build index vector

if (option==0)     % Samples are not connected (default)
   stem(n,x);
elseif (option==1) % Samples are connected
   plot(n,x);
else               % Samples are connected and discrete values marked
   plot(n,x,'-o');
end;
ylabel('x[n]');
xlabel('n');
grid;
```

Example

```
>>x = singen(1,60,0,1000,10/1000);
>>plotdsig(x)
>>plotdsig(x,1)
>>plotdsig(x,2)
```

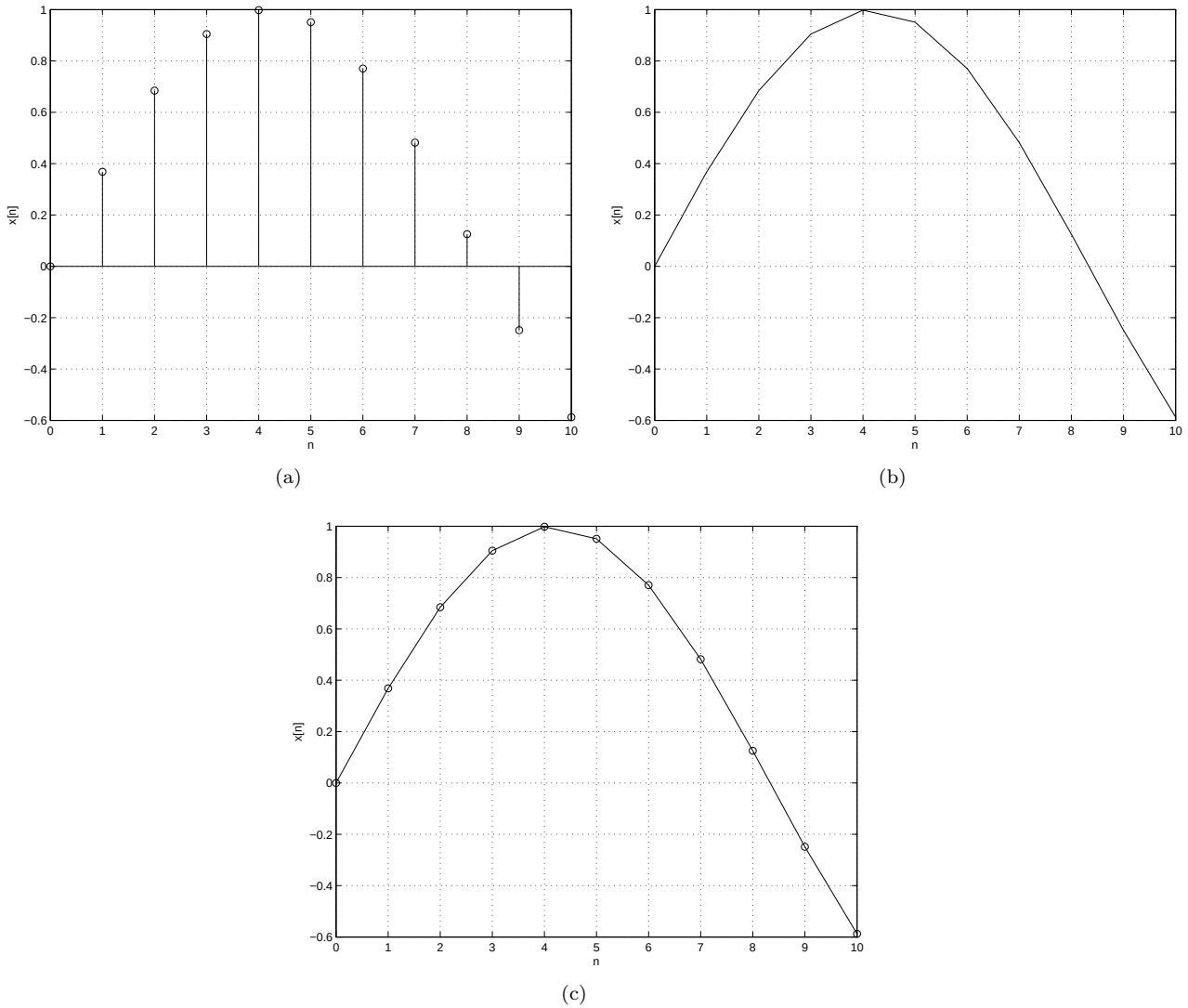


Figure 4.1: PLOTDSIG with (a) option = 0, (b) option = 1, and (c) option = 2.

4.5 PLOTCSIG.M

Purpose

This function will plot and label a continuous-time signal by connecting sample values.

Input

x, vector of samples
f_s, sampling rate in Hertz

Output

Plot of a continuous-time signal

Code

```
N = length(x)-1;
t = [0:1/fs:N/fs]';  % build time vector
plot(t,x);  % plot sequence vs. time vector
ylabel('x(t)');
xlabel('t (sec.)');
grid;
```

Example

```
>>x = cosingen(1,60,pi/2,10000,1000/10000);
>>plotcsig(x,10000)
```

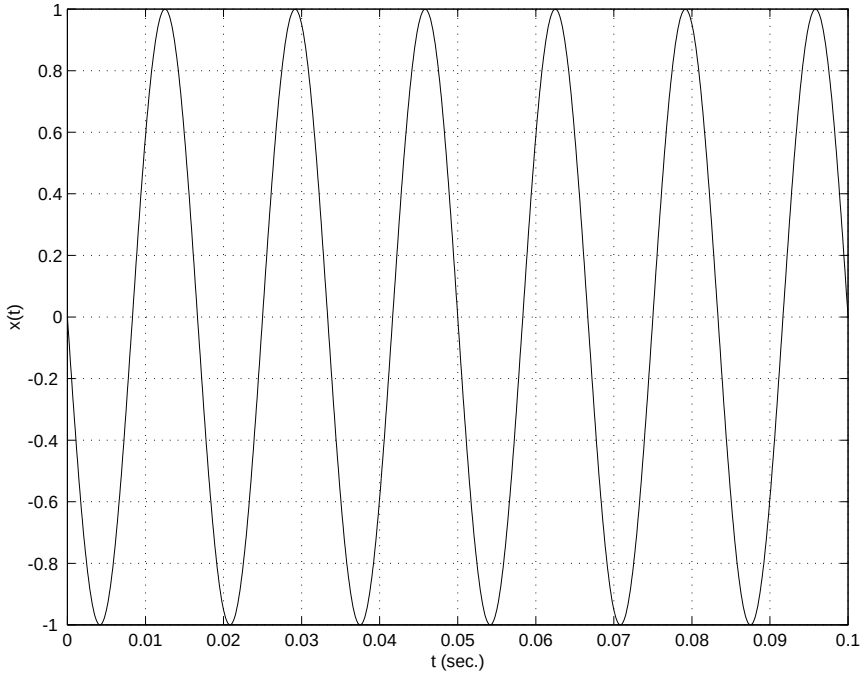


Figure 4.2: PLOTCSIG.

4.6 WAVREAD.M (Built in)

Purpose

This built in function will read in a signal from a Microsoft WAV file.

Input

$filename$, filename of Microsoft WAV file

Output

x, vector of samples
f_s, sampling rate in Hertz
$bits$, bits per sample

Example

```
>>[x,fs,bits] = wavread('song.wav');
```

4.7 WAVWRITE.M (Built in)

Purpose

This built in function will write the samples of a vector to a Microsoft WAV file.

Input

x, vector of samples
f_s, sampling rate in Hertz
bits, bits per sample
$filename$, filename of Microsoft WAV file

Output

Microsoft WAV file

Notes

In order to avoid clipping the signal, you should normalize the samples before wavwrite.

Example

```
>>x = x ./ max(abs(x));  % normalize samples
>>wavwrite(x,fs,bits,'song.wav');
```

4.8 ADSR_GEN.M

Purpose

This function returns an Attack, Decay, Sustain, and Release (ADSR) envelope. This envelope can be used to Amplitude Modulate a sinusoid and forms the basis for computer music synthesis.

Input

target, vector of attack, sustain, and release targets
gain, vector of attack, sustain, and release gains
duration, vector of attack, sustain, and release durations (in milliseconds)

Output

adsr, vector of envelope values. The vector has three stages corresponding to attach, sustain, and release portions of the envelope.

Algorithm

For each stage:
$adsr(n) = (1 - gain)adsr(n-1) + gain(target)$

Code

```
fs = 16000;
adsr = zeros(fs,1);  % malloc assuming a 1 second duration ADSR envelope
duration = round(duration./1000.*fs);  % envelope durations in samples

% Compute ADSR Values
% Attack phase
start = 2;
stop = duration(1);
for n = [start:stop]
   adsr(n) = (1.0 - gain(1))*adsr(n-1) + target(1)*gain(1);
end

% Sustain phase
start = stop + 1;
stop = start + duration(2);
for n = [start:stop]
   adsr(n) = (1.0 - gain(2))*adsr(n-1) + target(2)*gain(2);
end

% Release phase
start = stop + 1;
stop = fs;
for n = [start:stop]
   adsr(n) = (1.0 - gain(3))*adsr(n-1) + target(3)*gain(3);
end;
```

Notes

1. Assume $f_s = 16$kHz sample rate.
2. The complete envelope is 1 second or f_s samples long, i.e. sum(duration) = 1000ms.

Example

```
>>target = [0.99999;0.25;0];
>>gain = [0.005;0.0004;0.00075];
>>duration = [125;625;250];
>>adsr = adsr_gen(target,gain,duration);
>>plotcsig(adsr,16000);
```

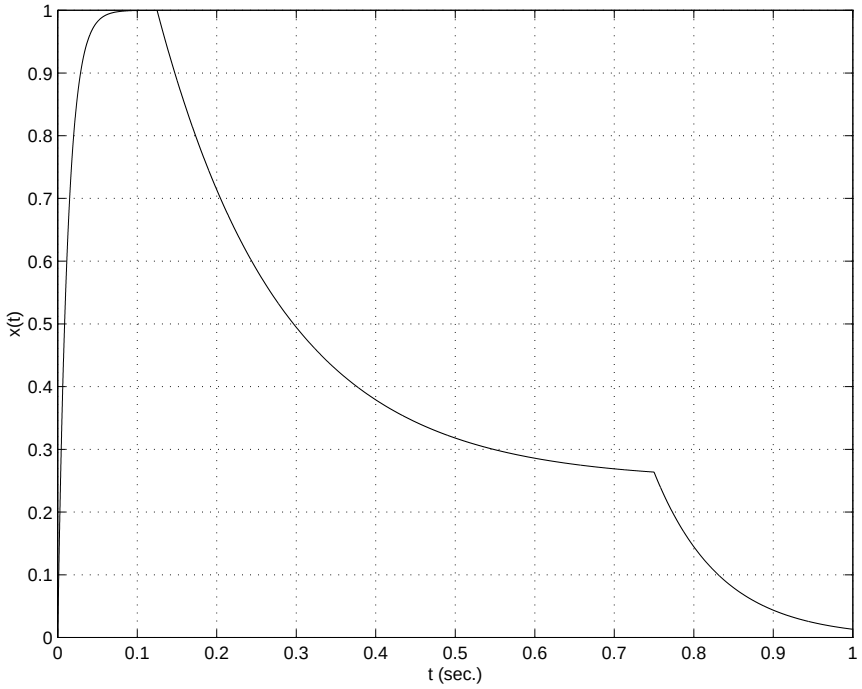


Figure 4.3: ADSR envelope.

Example

```
>>f0 = 440;fs = 16000;
>>x = singen(1,f0,0,fs,1-1/fs);
>>sound(x,fs); % Play tone
>>y = adsr .* x;  % Amplitude Modulate (AM)
>>for k = 1:5
>>  sound(y,fs); % Play AMed tone a few times
>>end;
```

4.9 KARPLUS.M

Purpose

This function implements the Karplus-Strong algorithm for generating plucked-string sounds.

Input

f_1, desired frequency (in Hz)
f_s, sample rate (in Hz)
duration, duration of sound (in seconds)

Output

x, vector of samples (length f_s)

Algorithm

Let

$$H(z) = \frac{1}{1 - z^{-D}G(z)} \tag{4.4}$$

where $D = f_s/f_1$ and $G(z) = \frac{1+z^{-1}}{2}$. Initialize the filter delays with random values and feed the filter a zero signal of specified duration.

Code

```
% Filter parameters
D = round(fs/f1);
b = 1;
a = [1;zeros(D-1,1);-0.5;-0.5];

% Initialize
o = zeros(duration*fs,1);  % zero input signal
x = filter(b,a,o,rand(D+1,1));  % delays are initialized with a random signal
```

Reference

R. Karplus and A. Strong, "Digital synthesis of plucked string and drum timbres," *Computer Music Journal*, vol. 7, no. 43, 1983.

S. Orphanidis, *Introduction to Signal Processing*, Prentice-Hall, 1996.

Example

```
>>x = karplus(50,8000,1);
>>sound(x,8000);
```

Chapter 5

Time Domain Tools

This chapter contains several tools used to process signals in the time-domain.

5.1 CONVLTN.M

Purpose

This function returns the convolution of two signals–one of which is typically an impulse response and the other an input signal.

Input

h, impulse response
x, input signal

Output

y, output signal

Algorithm

$$\begin{aligned} y(n) &= \sum_{m} h(m)x(n-m) \\ &= \sum_{m=\max(0,n-L+1)}^{\min(n,M)} h(m)x(n-m) \end{aligned} \tag{5.1}$$

where $0 \leq n \leq L+M-1$, length $(\mathbf{h}) = M+1$, and length $(\mathbf{x}) = L$.

Note

MATLAB has a built-in convolution function and when called as

```
y = conv(h,x)
```

produces the same output as our function.

Example

```
>>h = [1 2 3]';
>>x = [4 5 6 7]';
>>y = convltn(h,x)

y =
     4
    13
    28
    34
    32
    21

>>h = [1 2 3 4]';
>>x = [5 6 7]';
>>y = convltn(h,x)
```

```
y =
     5
    16
    34
    52
    45
    28
```

5.2 FLTR.M

Purpose

This function returns the output of a linear, shift-invariant system, i.e. digital filter described by the linear, constant-coefficient, difference equation

$$y(n) = -\sum_{i=1}^{M} a(i)y(n-i) + \sum_{i=0}^{L} b(i)x(n-i) \tag{5.2}$$

where $x(n)$ is the input signal and $y(n)$ is the output signal. Equivalently, the transfer function of the filter is given by $H(z) = B(z)/A(z)$ where

$$B(z) = b(0) + b(1)z^{-1} + \cdots + b(L)z^{-L} \tag{5.3}$$

and

$$A(z) = 1 + a(1)z^{-1} + \cdots + a(M)z^{-M}. \tag{5.4}$$

Input

b, vector of feedforward gains
a, vector of feedback gains
x, input signal

Output

y, output signal

Algorithm

$$y(n) = -\sum_{i=\max(1,n+1-N)}^{\min(n,M)} a(i)y(n-i) + \sum_{i=\max(0,n+1-N)}^{\min(n,L)} b(i)x(n-i), \quad 0 \le n \le N-1 \tag{5.5}$$

where N is the length of **x**.

Notes

1. The length of the output signal, **y** should be the same as the length of the input signal, **x**.
2. MATLAB has a built-in filter function and when called as

```
y = filter(b,a,x)
```

produces the same output as our function.

Example

```
>>b = [4 5 6 7]';
>>a = [1 2 3]';
>>x = [1 2 3 4 5 6 7 8 9]';
>>y = fltr(b,a,x)
```

```
y =
     4
     5
     6
    23
     8
     9
    74
   -37
    12
```

5.3 FREQSHFT.M

Purpose

This function shifts the frequency response of the filter, **h** by ω_0. The resulting impulse response is returned as **g**.

Input

h, impulse response of prototype filter
ω_0, frequency in radians/sample of shift

Output

g, impulse response of resulting filter

Algorithm

$$g(n) \quad = \quad h(n)e^{j\omega_0(n-M)} \tag{5.6}$$

where $M = (N-1)/2$ and length $(\mathbf{h}) = N$.

Notes

1. The above algorithm should be written without a for-loop.
2. If **h** is real-valued and $\omega_0 = \pi$, then **g** is real-valued. However, sometimes due to numerical roundoff error, **g** will have an imaginary part with magnitude on the order of the numerical precision of the computer. Eliminate the imaginary roundoff error for this case.

Example

```
>>h = [1 2 3 2 1]';
>>g = freqshft(h,pi/4)

g =
  0.000000000000000 - 1.000000000000000i
  1.414213562373095 - 1.414213562373095i
  3.000000000000000
  1.414213562373095 + 1.414213562373095i
  0.000000000000000 + 1.000000000000000i
```

5.4 MODULATE2.M

Purpose

This function modulates the impulse response **h** by ω_0. The resulting impulse response is returned as **g**.

Input

h, impulse response of prototype filter
ω_0, frequency in radians/sample for modulation

Output

g, impulse response of resulting filter

Algorithm

$$g(n) \quad = \quad h(n)\cos(\omega_0(n-M)) \tag{5.7}$$

where $M = (N-1)/2$ and length $(\mathbf{h}) = N$.

Notes

1. The above algorithm should be written without a for-loop.
2. The MATLAB Signal Processing Toolbox contains a modulate function and when called as

```
g = modulate(h,w0,2*pi,'am')
```

produces the same output as our function.

Example

```
>>h = [1 2 3 4 5]';
>>w0 = pi/4;
>>g = modulate2(h,w0)

g =
   0.000000000000000
   1.414213562373095
   3.000000000000000
   2.828427124746190
   0.000000000000000
```

Chapter 6

Frequency Domain Tools

This chapter contains several tools used to process and analyze signals in the frequency-domain.

6.1 DTFT.M

Purpose

This function returns the DTFT of a length L signal at desired frequency points.

Input

x, signal
ω, desired frequency point(s) in radians/sample

Output

X, DTFT of **x** at ω

Algorithm

$$X\left(e^{j\omega}\right) = \sum_{n=0}^{L-1} x(n)e^{-j\omega n} \tag{6.1}$$

Note

The MATLAB Signal Processing Toolbox contains a DTFT/DFT function and when called as

```
X = freqz(x,1,w,2*pi)
```

produces the same output as our function.

Example

```
>>x = singen(1,0.25,0,1,10);
>>w = [0;0.23;0.25;0.27;0.73;0.75;0.77;1]*2*pi;
>>X = dtft(x,w)

X =
  1.00000000000000
  2.75658333378326 - 3.79411146289144i
 -0.00000000000000 - 5.00000000000000i
 -2.75658333378327 - 3.79411146289143i
 -2.75658333378327 + 3.79411146289143i
 -0.00000000000001 + 5.00000000000000i
  2.75658333378326 + 3.79411146289144i
  1.00000000000000 + 0.00000000000000i
```

6.2 DFT.M

Purpose

This function returns the N-point DFT of a length L signal.

Input

$\mathbf{x}$, signal of length L
N, number of frequency evaluation points (optional)

Output

$\mathbf{X}$, N-pt DFT of $\mathbf{x}$

Algorithm

$$X(k) = \sum_{n=0}^{L-1} x(n)e^{-j\omega_k n} \tag{6.2}$$

where

$$\omega_k = \frac{2\pi k}{N} \tag{6.3}$$

and $k = 0, 1, \cdots, N-1$.

Notes

1. If N is not specified, assume $N = \text{length}(\mathbf{x})$.
2. MATLAB has a built-in DFT/FFT function and when called as

```
X = fft(x)
```

produces the same output as our function when N is not specified.

Example

```
>>x = singen(1,0.25,0,1,10);
>>X = dft(x,8)

X =
  1.00000000000000
  0.70710678118655 - 0.70710678118655i
 -0.00000000000000 - 5.00000000000000i
 -0.70710678118655 - 0.70710678118655i
 -1.00000000000000 - 0.00000000000000i
 -0.70710678118655 + 0.70710678118655i
 -0.00000000000001 + 5.00000000000000i
  0.70710678118655 + 0.70710678118654i

>>X = dft(x)
```

```
X =
  1.00000000000000
  1.09435114443714 + 0.32133048858417i
  1.70361562377557 + 1.09484728149484i
 -3.01333709166613 - 3.47757638588674i
 -0.26352111843337 - 0.57703076026650i
 -0.02110855811321 - 0.14681324646918i
 -0.02110855811321 + 0.14681324646918i
 -0.26352111843336 + 0.57703076026651i
 -3.01333709166614 + 3.47757638588673i
  1.70361562377557 - 1.09484728149484i
  1.09435114443714 - 0.32133048858416i
```

6.3 FREQRESP.M

Purpose

This function returns the frequency response values of

$$H\left(e^{j\omega_k}\right) = \frac{B\left(e^{j\omega_k}\right)}{A\left(e^{j\omega_k}\right)} \tag{6.4}$$

where $B\left(e^{j\omega_k}\right) = \sum_{m=0}^{L} b(m)e^{-j\omega_k m}$ and $A\left(e^{j\omega_k}\right) = \sum_{m=0}^{M} a(m)e^{-j\omega_k m}$. The kth DFT frequency, $\omega_k = 2\pi k/N$ (rads/sample) and N is the number of evaluation points. Here, $b(m)$ are the feedforward coefficients and $a(m)$ are the feedback coefficients of the filter defined by the difference equation

$$y(n) = -\sum_{i=1}^{M} a(i)y(n-i) + \sum_{i=0}^{L} b(i)x(n-i) \tag{6.5}$$

where $x(n)$ is the input signal and $y(n)$ is the output signal.

Input

b, vector of feedforward gains $[b(0),\ b(1),\ \cdots\ b(L)]^T$
a, vector of feedback gains $[1,\ a(1),\ \cdots\ a(M)]^T$
N, number of evaluation points

Output

H_mag, vector of magnitude response values
H_phase, vector of phase response values

Code

```
H_mag = zeros(N,1);  % prealloc for H_mag
H_phase = zeros(N,1);  % prealloc for H_phase

H_b = dft(b,N);
H_a = dft(a,N);
H_mag = abs(H_b./H_a);
H_phase = atan2(imag(H_b./H_a),real((H_b./H_a)));
```

Notes

1. Use `atan2()` instead of `atan()` so the principle angle is returned.
2. The MATLAB Signal Processing Toolbox contains a frequency response function and when called as

```
H = freqz(b,a,N,'whole');
H_mag = abs(H);H_phase = angle(H);
```

produces the same output as our function.

6.4 FR_PLOT.M

Purpose

This function plots the magnitude and phase responses of $H\left(e^{j\omega}\right)$.

Input

b, vector of feedforward gains
a, vector of feedback gains
f_s, sampling frequency (optional)
option, optional argument with values as follows
0–Magnitude response is plotted in dB (default)
1–Magnitude response is plotted in absolute units
2–Magnitude response is plotted in squared absolute units

Output

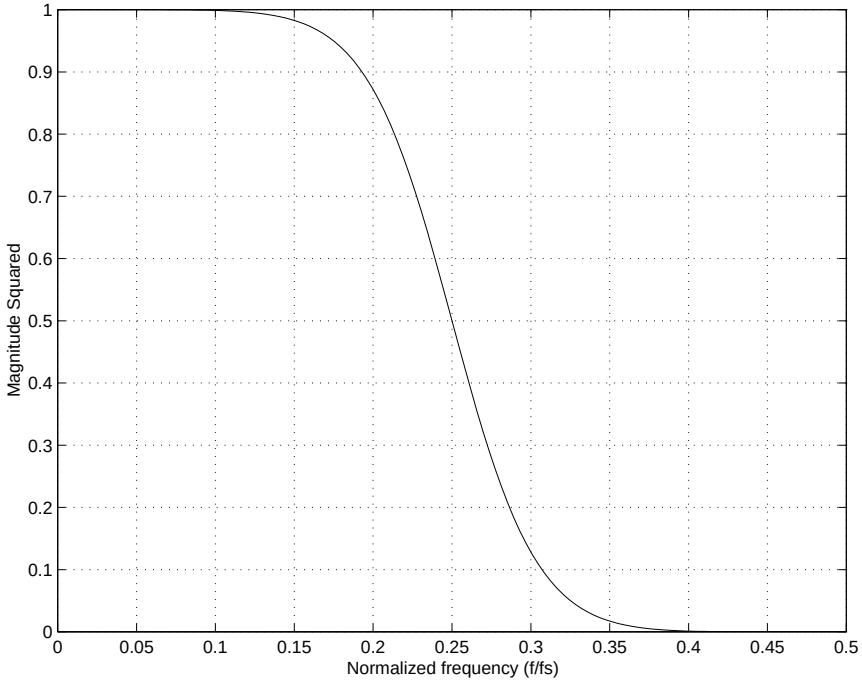
Plot with magnitude response and a plot with phase response

Notes

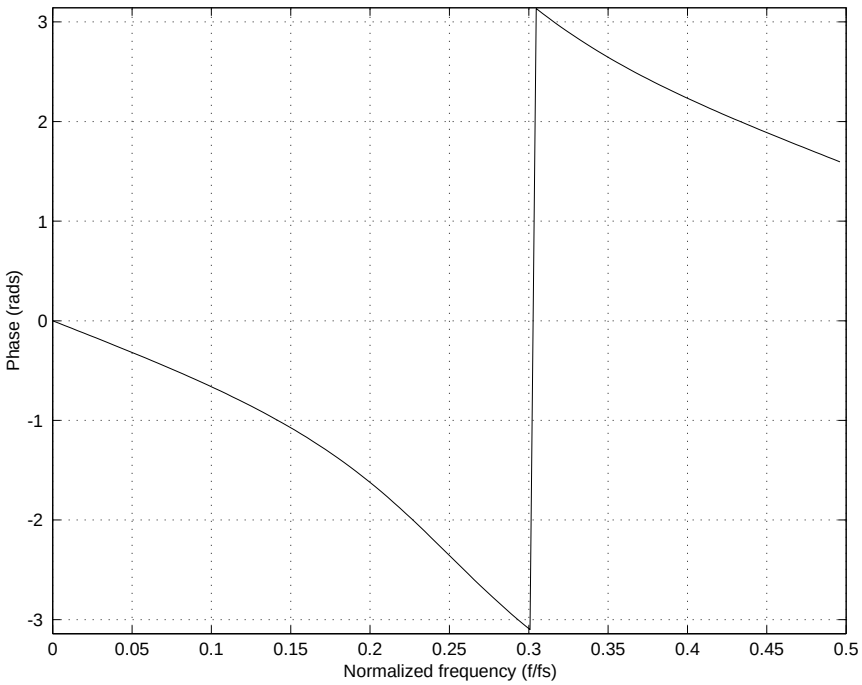
1. Magnitude and phase response plots should be on separate figures (do not use subplot).
2. Assume $N = 1024$ frequency evaluation points in your call to `freqresp.m`.
3. If **b** and **a** are both real-valued, plot only from zero to the foldover frequency.
4. A grid should be included to aid in reading the plots.
5. If f_s is specified one the following should happen:
$f_s = 2\pi$: Responses are plotted as a function of ω in radians/sample (default)
$f_s = 1$: Responses are plotted as a function of f/f_s (normalized frequency scale)
any other value of f_s: Responses are plotted as a function of f in Hz.

Example

```
>>[b a]

   0.16666666666667   1.00000000000000
   0.50000000000000  -0.00000000000000
   0.50000000000000   0.33333333333333
   0.16666666666667  -0.00000000000000

>>fr_plot(b,a,1,2)
```



(a)



(b)

Figure 6.1: FR_PLOT: (a) magnitude response plot and (b) phase response plot.

6.5 GRDPLOT.M

Purpose

This function plots the group delay of $H\left(e^{j\omega}\right)$ defined as

$$\text{grd}\left[H\left(e^{j\omega}\right)\right] \equiv -\frac{d}{d\omega}\left\{\arg\left[H\left(e^{j\omega}\right)\right]\right\}. \tag{6.6}$$

Input

b, vector of feedforward gains
a, vector of feedback gains
f_s, sampling frequency (optional)

Output

Plot of group delay

Algorithm

$$\begin{aligned} -\frac{d}{d\omega}\left\{\arg\left[H\left(e^{j\omega}\right)\right]\right\} &= Re\left[\frac{j\dfrac{dH\left(e^{j\omega}\right)}{d\omega}}{H\left(e^{j\omega}\right)}\right] \\ &= Re\left[\frac{j\dfrac{dB\left(e^{j\omega}\right)}{d\omega}}{B\left(e^{j\omega}\right)}\right] - Re\left[\frac{j\dfrac{dA\left(e^{j\omega}\right)}{d\omega}}{A\left(e^{j\omega}\right)}\right] \end{aligned} \tag{6.7}$$

where $H(e^{j\omega}) = B(e^{j\omega})/A(e^{j\omega})$ and we use the property that

$$nx(n) \leftrightarrow j\frac{dX\left(e^{j\omega}\right)}{d\omega}. \tag{6.8}$$

Notes

1. Assume $N = 1024$ frequency evaluation points.
2. If **b** and **a** are both real-valued, plot only from zero to the foldover frequency.
3. A grid should be included to aid in reading the plot.
4. If f_s is specified one the following happen:
   $f_s = 2\pi$: Responses are plotted as a function of ω in radians/sample (default)
   $f_s = 1$: Responses are plotted as a function of f/f_s
   any other value of f_s: Responses are plotted as a function of f in Hz.

5. The MATLAB Signal Processing Toolbox contains a group delay plotting function and when called as

```
grpdelay(b,a,1024,fs)
```

produces a group delay plot.

Example

```
>>[b a]

   0.16666666666667   1.00000000000000
   0.50000000000000  -0.00000000000000
   0.50000000000000   0.33333333333333
   0.16666666666667  -0.00000000000000

>>grdplot(b,a,1)
```

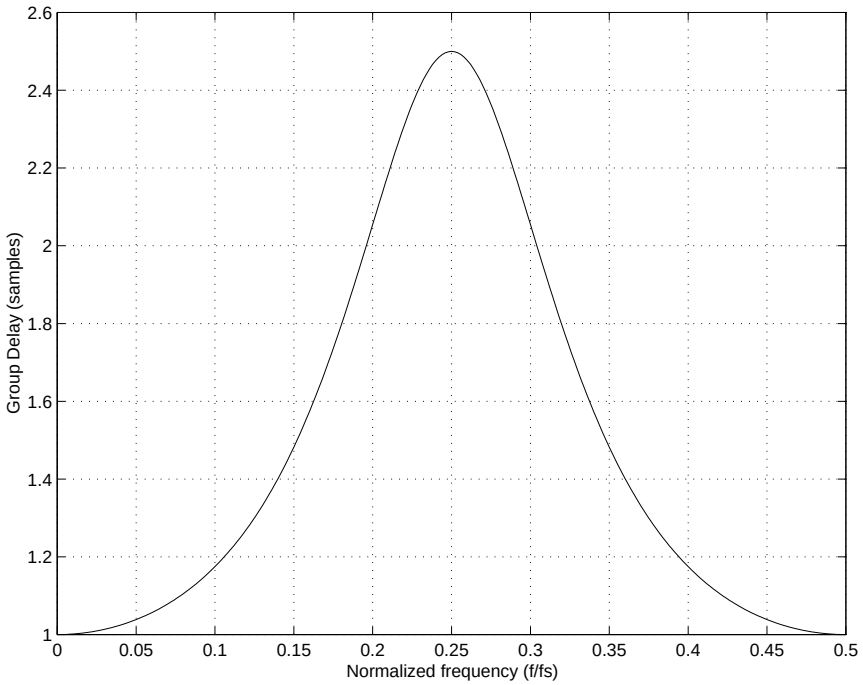


Figure 6.2: Group delay plot.

6.6 POLEZERO.M

Purpose

This function plots the pole-zero diagram of $H(z)$.

Input

b, vector of feedforward gains
a, vector of feedback gains

Output

Plot of pole-zero diagram including poles and zeros at the origin.

Code

```
M = length(b)-1; % poly order
N = length(a)-1; % poly order
if (M > N) % add zero coefs to account for pole(s) at the origin
   a = [a;zeros(M-N,1)];
end;
if (M < N) % add zero coefs to account for zero(s) at the origin
   b = [b;zeros(N-M,1)];
end;
p = roots(a);  % pole locations
z = roots(b);  % zero locations
w = [0:2*pi/100:2*pi]';  % a few phase angles...
unit_circle = exp(j*w);  % ...to compute points around the unit circle
plot(unit_circle);  % draw a unit circle
hold on;
plot([0 0],[-1 1],'--'),  % cross hairs
plot([-1 1],[0 0],'--'),  % cross hairs
plot(real(p),imag(p),'x'),  % draw the poles
plot(real(z),imag(z),'o'),  % draw the zeros
axis('square'),  % 1:1 aspect ratio
title('Pole/Zero Locations for Filter'),
ylabel('Imag(z)'),
xlabel('Real(z)'),
grid;
hold off;
```

Example

```
>>b =
1.000000000000000
0.000000000000000
1.000000000000000
>>a =
1.000000000000000
-0.500000000000000
```

```
>>polezero(b,a)
```

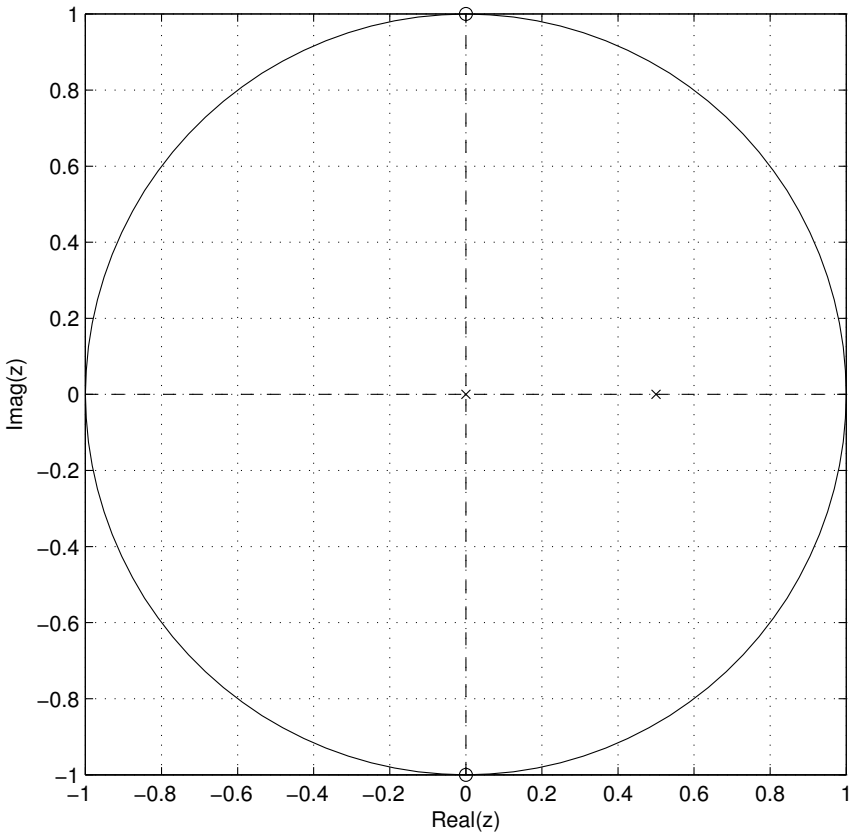


Figure 6.3: Pole-zero plot.

6.7 STFT.M

Purpose

This function returns a plot of the short-time Fourier transform (STFT).

Input

x, input signal
w, window vector
overlap, overlap factor ($0 \leq overlap < 1$)
f_s, sampling frequency (optional)

Output

X, STFT of **x**
Plot of STFT

Code

```
M = length(x); % signal length
L = length(w); % window length
LL = floor(overlap*L);  % number of samples to overlap
N = floor((M-LL)/(L-LL));  % number of FFTs to compute
X = zeros(L,N);

start = 1;
for n = 1:N
   tmp = x(start:start+L-1).* w;
   X(:,n) = fft(tmp);
   start = start + L - LL;
end;

if ((fs == 2*pi) | (fs == 1))  % create time and frequency vectors
   t = [0:M/N:M-1]';
else
   t = [0:M/N/fs:M/fs-1]';
end;
f = [0:fs/L:fs/2]';

imagesc(t,f,20*log10(abs(X(1:L/2+1,:)))); colormap(jet);
set(gca,'Ydir','normal');
if (fs == 2*pi)  % create time and frequency vectors
   xlabel('n');
   ylabel('Frequency (rads/s)');
elseif (fs == 1)
   xlabel('n');
   ylabel('Normalized Frequency (f/fs)');
else
   xlabel('t (sec)');
   ylabel('Frequency (Hz)');
end;
```

Notes

1. Use the FFT function in this computation.
2. Assume **x** is real-valued.
3. If f_s is specified one the following happen:
   $f_s = 2\pi$: Responses are plotted as a function of ω in radians/sample (default)
   $f_s = 1$: Responses are plotted as a function of f/f_s
   $f_s =$?: Responses are plotted as a function of f in Hz.

4. The MATLAB Signal Processing Toolbox contains a spectrogram function and when called as

```
[X,f,t] = spectrogram(x,256,128,256,1000,'yaxis');
surf(t,f,20*log10(abs(X)),'EdgeColor','none');
axis tight;view(0,90);
xlabel('Time');ylabel('Frequency (Hz)');
```

produces the same output as our function.

Example

```
>>w0 = 2*pi*7.5*10^(-6);
>>n = [0:28800]';
>>x = cos(w0*n.*n);  % linear chirp
>>X = stft(x,hamming(256),0.5,1000);
```

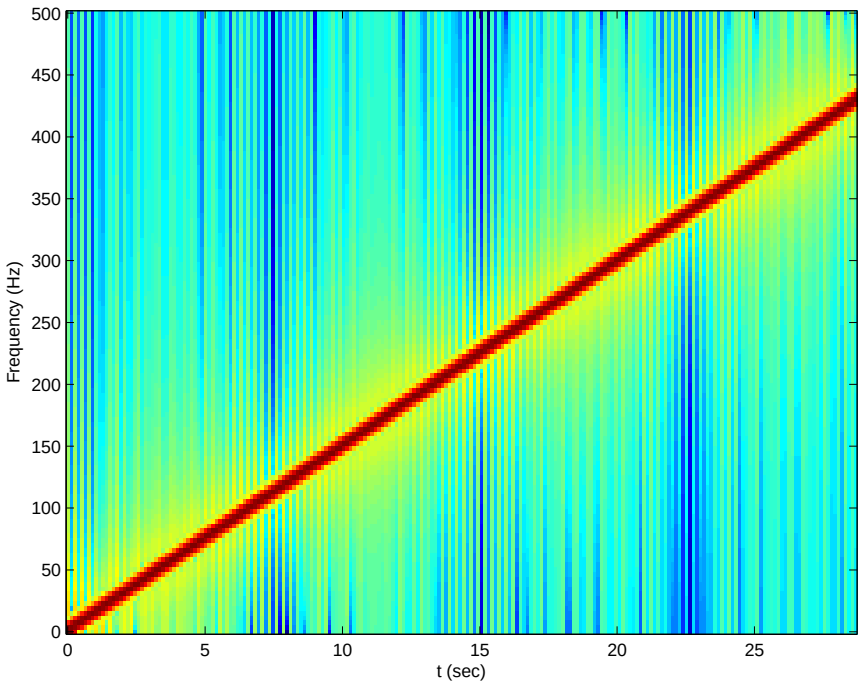


Figure 6.4: STFT plot.

Chapter 7

Filter Design Tools

This chapter contains several tools used to design digital filters. You may choose to use the supplied "freebie" MATLAB code or write your own.

7.1 RECTANGL.M

Purpose

This function returns the rectangular window of length N.

Input

N, length of window

Output

w, window of length N

Algorithm

$$w(n) = 1.0 \tag{7.1}$$

where $0 \leq n \leq N-1$.

Note

The MATLAB Signal Processing Toolbox contains a rectangular window function and when called as

```
w = rectwin(N)
```

produces the same output as our function.

Reference

Fredric J. Harris, "On the use of windows for harmonic analysis with the discrete fourier transform," *Proc. IEEE*, vol. 66, no. 1, Jan. 1978, pp. 51-83.

Example

```
>>w = rectwin(8)

w =
     1
     1
     1
     1
     1
     1
     1
     1
```

7.2 HAMMING2.M

Purpose

This function returns the Hamming window of length N.

Input

N, length of window

Output

w, window of length N

Algorithm

$$w(n) = 0.54 - 0.46\cos\left[\frac{2\pi}{(N-1)}n\right] \tag{7.2}$$

where $0 \leq n \leq N-1$.

Note

MATLAB Signal Processing Toolbox contains a hamming window function and when called as

```
w = hamming(N)
```

produces the same output as our function.

Reference

Fredric J. Harris, "On the use of windows for harmonic analysis with the discrete fourier transform," *Proc. IEEE*, vol. 66, no. 1, Jan. 1978, pp. 51-83.

Example

```
>>w = hamming2(8)

w =
   0.08000000000000
   0.25319469114498
   0.64235962961990
   0.95444567923511
   0.95444567923511
   0.64235962961990
   0.25319469114498
   0.08000000000000
```

7.3 HANNING2.M

Purpose

This function returns the Hanning window of length N.

Input

N, length of window

Output

w, window of length N

Algorithm

$$w(n) = 0.5 - 0.5\cos\left[\frac{2\pi}{(N-1)}n\right] \tag{7.3}$$

where $0 \leq n \leq N-1$.

Note

MATLAB Signal Processing Toolbox contains a hanning window function and when called as

```
w = hann(N)
```

produces the same output as our function.

Reference

Fredric J. Harris, "On the use of windows for harmonic analysis with the discrete fourier transform," *Proc. IEEE*, vol. 66, no. 1, Jan. 1978, pp. 51-83.

Example

```
>>w = hanning2(8)

w =
   0.000000000000000
   0.188255099070633
   0.611260466978157
   0.950484433951210
   0.950484433951210
   0.611260466978157
   0.188255099070633
   0.000000000000000
```

7.4 FIR_WIND.M

Purpose

This function returns the coefficients of a lowpass filter designed using the window method.

Input

ω_c, cutoff frequency in radians/sample
w, vector of N window samples
option, optional argument with values as follows
0–yields $\left|H\left(e^{j\omega_c}\right)\right| = 1/2$ (half-amplitude) (default)
1–yields $\left|H\left(e^{j\omega_c}\right)\right|^2 = 1/2$ (half-power or -3dB)

Output

h, impulse response of filter

Algorithm

$$h(n) = \frac{\omega_c}{\pi}\text{sinc}\left[\omega_c(n-M)\right]w(n) \tag{7.4}$$

where $0 \le n \le N-1$, $\text{sinc}(x) \equiv \sin(x)/x$, and $M = (N-1)/2$. If *option* $= 1$ employ a binary search for ω_c' such that when used in the above equation we have $|H\left(e^{j\omega_c}\right)|^2 = 1/2$.

Notes

1. The filter coefficients should be scaled so that the DC gain is 1, i.e. $\sum h(n) = 1$.
2. MATLAB defines $\text{sinc}(x) \equiv \sin(\pi x)/(\pi x)$. Therefore, if MATLAB's sinc function is used, be sure to divide the argument by π.
3. If *option* $= 1$, search should terminate when $\left|H\left(e^{j\omega_c}\right)\right|^2 = 1/2 \pm 0.001$.
4. The MATLAB Signal Processing Toolbox contains a window-based FIR filter design function and when called as

```
h = fir1(N-1,wc/pi)'
```

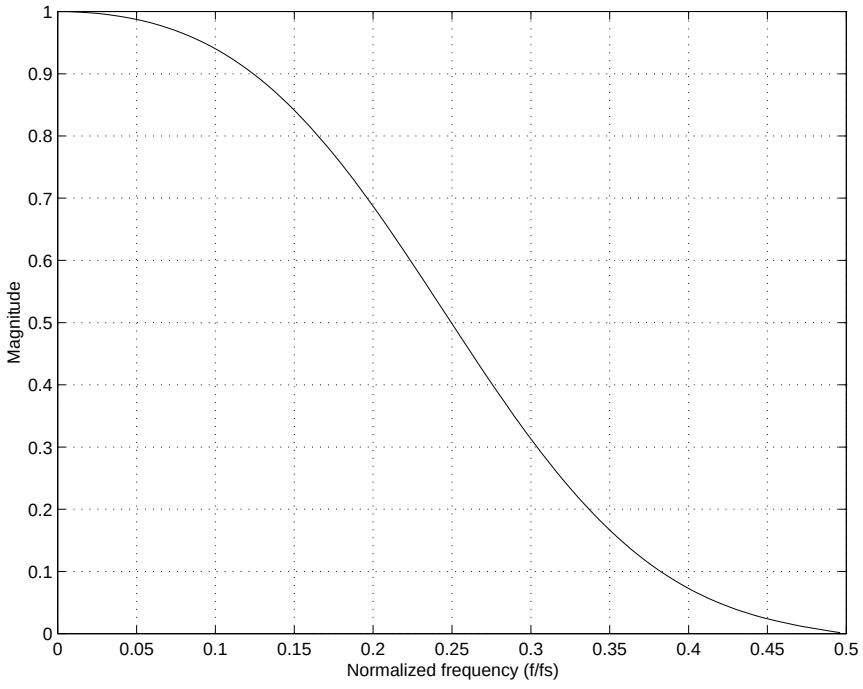
produces the same output as our function with *option* $= 0$.

Example

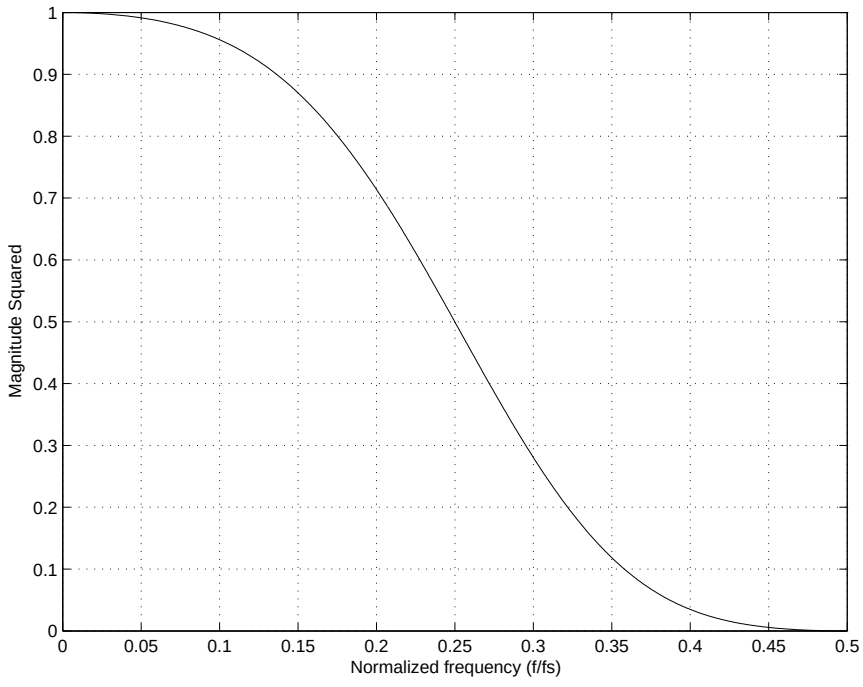
```
>>w = hamming2(8);
>>h = fir_wind(pi/2,w,0)

h =
  -0.005164298061200
  -0.022882524920256
   0.096755650536968
   0.431291172444487
   0.431291172444487
   0.096755650536968
  -0.022882524920256
```

```
  -0.005164298061200

>>h = fir_wind(pi/2,w,1)

h =
   0.00341173808217
  -0.03171816581314
   0.03184651731837
   0.49645991041260
   0.49645991041260
   0.03184651731837
  -0.03171816581314
   0.00341173808217
```

Magnitude responses for these filters are given in Figure 7.1.



(a)



(b)

Figure 7.1: FIR_WIND: (a) LPF designed with half-amplitude cutoff and (b) LPF designed with half-power cutoff.

7.5 BTTRWRTH.M (Orfanidis)

Purpose

This function returns the Nth order LP digital Butterworth filter coefficients in length $N+1$ vectors **b** and **a**. The cut-off frequency is ω_c in radians/sample.

Input

N, filter order
ω_c, cutoff frequency

Output

b, vector of feedforward gains
a, vector of feedback gains

Algorithm

$$H(z) = H_0(z)\prod_{i=1}^{K} H_i(z) \tag{7.5}$$

where

$$H_0(z) = \begin{cases} 1 & ,N=2K\ (N\text{even}) \\ \dfrac{G_0\left(1+z^{-1}\right)}{1+a_{01}z^{-1}} & ,N=2K+1\ (N\text{odd}) \end{cases} \tag{7.6}$$

and

$$H_i(z) = \frac{G_i\left(1+z^{-1}\right)^2}{1+a_{i1}z^{-1}+a_{i2}z^{-2}}, \tag{7.7}$$

with

$$G_i = \frac{\Omega_c^2}{1-2\Omega_c\cos\theta_i+\Omega_c^2}, \tag{7.8}$$

$$G_0 = \frac{\Omega_c}{\Omega_c+1}, \tag{7.9}$$

$$a_{i1} = \frac{2\left(\Omega_c^2-1\right)}{1-2\Omega_c\cos\theta_i+\Omega_c^2}, \tag{7.10}$$

$$a_{i2} = \frac{1+2\Omega_c\cos\theta_i+\Omega_c^2}{1-2\Omega_c\cos\theta_i+\Omega_c^2}, \tag{7.11}$$

$$a_{01} = \frac{\Omega_c-1}{\Omega_c+1}, \tag{7.12}$$

$$\theta_i = \frac{\pi}{2N}(N-1+2i), \tag{7.13}$$

and

$$\Omega_c = \tan\left(\frac{\omega_c}{2}\right). \tag{7.14}$$

Code

```
% Prewarp
Wc = tan(wc/2);

% Initialize first-order sections
if (rem(N,2))  % N is odd
   G = Wc/(Wc+1);
   b = [G G]';
   a1 = (Wc-1)/(Wc+1);
   a = [1 a1]';
   K = (N-1)/2;
else
   b = 1;
   a = 1;
   K = N/2;
end;

% Compute b and a
for i = 1:K
   theta = (pi/(2*N))*(N-1+2*i);
   G = (Wc*Wc)/(1-2*Wc*cos(theta)+Wc*Wc);
   a1 = 2*(Wc*Wc-1)/(1-2*Wc*cos(theta)+Wc*Wc);
   a2 = (1+2*Wc*cos(theta)+Wc*Wc)/(1-2*Wc*cos(theta)+Wc*Wc);
   b = convltn(b,[G 2*G G]');
   a = convltn(a,[1 a1 a2]');
end;
```

Note

MATLAB's Butterworth design tool is called `butter`.

Reference

S. Orfanidis, *Introduction to Signal Processing*, Sections 11.6.1 - 11.6.2, Prentice-Hall: Upper Saddle River, N.J., 1996.

Example

```
>>[b,a] = bttrwrth(3,pi/2)
>>[b a]

   0.16666666666667   1.00000000000000
   0.50000000000000  -0.00000000000000
   0.50000000000000   0.33333333333333
   0.16666666666667  -0.00000000000000
```

7.6 BTTRWRTH.M (Oppenheim)

Purpose

This function returns the Nth order LP digital Butterworth filter coefficients in length $N+1$ vectors **b** and **a**. The cut-off frequency is ω_c in radians/sample.

Input

N, filter order
ω_c, cutoff frequency

Output

b, vector of feedforward gains
a, vector of feedback gains

Algorithm

$$H(z) = \begin{cases} \dfrac{\left(\Omega_c + \Omega_c z^{-1}\right)^N}{\prod_{n=1}^{N/2}\left(A_n + B_n z^{-1} + C_n z^{-2}\right)}, & N \text{ even} \\ \dfrac{\left(\Omega_c + \Omega_c z^{-1}\right)^N}{g\prod_{n=1}^{(N-1)/2}\left(A_n + B_n z^{-1} + C_n z^{-2}\right)}, & N \text{ odd} \end{cases} \tag{7.15}$$

where

$$A_n = 1 - 2\Omega_c \cos\left[\frac{\pi}{2}\left(1 + \frac{2n-1}{N}\right)\right] + \Omega_c^2 \tag{7.16}$$

$$B_n = -2 + 2\Omega_c^2 \tag{7.17}$$

$$C_n = 1 + 2\Omega_c \cos\left[\frac{\pi}{2}\left(1 + \frac{2n-1}{N}\right)\right] + \Omega_c^2 \tag{7.18}$$

and $g = (1+\Omega_c) + (-1+\Omega_c)\, z^{-1}$.

Code

```
% Prewarp
Wc = tan(wc/2);

% Compute b coefficients
b = 1;
for n = 1:N
   b = convltn(b,[Wc;Wc]);
end;
```

```
% Compute a coefficients
if (rem(N,2))  % N is odd initialize with 1st order term
   a = [1+Wc;-1+Wc];
else % N is even initialize with 1
   a = 1;
end;
for n = 1:floor(N/2)
   costheta = cos(pi/2*(1+(2*n-1)/N));
   An = 1-2*Wc*costheta+Wc*Wc;
   Bn = -2+2*Wc*Wc;
   Cn = 1+2*Wc*costheta+Wc*Wc;
   a = convltn(a,[An;Bn;Cn]);
end;

% Scale b and a so that a(0) = 1, i.e. monic A(z);
b = b ./ a(1);
a = a ./ a(1);
```

Note

The MATLAB Signal Processing Toolbox contains a Butterworth low-pass filter design function and when called as

```
[b,a] = butter(N,wc/pi)
```

produces the same output as our function.

Reference

A. Oppenheim and R. Schafer, *Discrete-Time Signal Processing*, Section 7.1.2, Prentice-Hall: Englewood Cliffs, N.J., 1989.

Example

```
>>[b,a] = bttrwrth(3,pi/2)
>>[b a]

   0.16666666666667   1.00000000000000
   0.50000000000000  -0.00000000000000
   0.50000000000000   0.33333333333333
   0.16666666666667  -0.00000000000000
```

7.7 BTRWRTHP.M (Orfanidis)

Purpose

This function returns the Nth order HP digital Butterworth filter coefficients in length $N+1$ vectors **b** and **a**. The cut-off frequency is ω_c in radians/sample.

Input

N, filter order
ω_c, cutoff frequency

Output

b, vector of feedforward gains
a, vector of feedback gains

Algorithm

$$H(z) = H_0(z)\prod_{i=1}^{K} H_i(z) \tag{7.19}$$

where

$$H_0(z) = \begin{cases} 1 & , N = 2K \ (N \text{ even}) \\ \dfrac{G_0\left(1-z^{-1}\right)}{1+a_{01}z^{-1}} & , N = 2K+1 \ (N \text{ odd}) \end{cases} \tag{7.20}$$

and

$$H_i(z) = \frac{G_i\left(1-z^{-1}\right)^2}{1+a_{i1}z^{-1}+a_{i2}z^{-2}}, \tag{7.21}$$

with

$$G_i = \frac{\Omega_c^2}{1-2\Omega_c\cos\theta_i+\Omega_c^2}, \tag{7.22}$$

$$G_0 = \frac{\Omega_c}{\Omega_c+1}, \tag{7.23}$$

$$a_{i1} = \frac{2\left(\Omega_c^2-1\right)}{1-2\Omega_c\cos\theta_i+\Omega_c^2}, \tag{7.24}$$

$$a_{i2} = \frac{1+2\Omega_c\cos\theta_i+\Omega_c^2}{1-2\Omega_c\cos\theta_i+\Omega_c^2}, \tag{7.25}$$

$$a_{01} = -\frac{\Omega_c-1}{\Omega_c+1}, \tag{7.26}$$

$$\theta_i = \frac{\pi}{2N}(N-1+2i), \tag{7.27}$$

and

$$\Omega_c = \cot\left(\frac{\omega_c}{2}\right), \tag{7.28}$$

Note

MATLAB's Butterworth design tool is called `butter`.

Reference

S. Orfanidis, *Introduction to Signal Processing*, Sections 11.6.1 - 11.6.2, Prentice-Hall: Upper Saddle River, N.J., 1996.

Example

```
>>[b,a] = btrwrthp(3,pi/2)
>>[b a]

   0.16666666666667   1.00000000000000
  -0.50000000000000  -0.00000000000000
   0.50000000000000   0.33333333333333
  -0.16666666666667  -0.00000000000000
```

7.8 BTRWRTHP.M (Oppenheim)

Purpose

This function returns the Nth order HP digital Butterworth filter coefficients in length $N+1$ vectors **b** and **a**. The cut-off frequency is ω_c in radians/sample.

Input

N, filter order
ω_c, cutoff frequency

Output

b, vector of feedforward gains
a, vector of feedback gains

Algorithm

$$H(z) = \begin{cases} \dfrac{\left[(1-\alpha)-(1-\alpha)\,z^{-1}\right]^N}{\prod_{n=1}^{N/2}\left(A_n + B_n z^{-1} + C_n z^{-2}\right)}, & N \text{ even} \\ \dfrac{\left[(1-\alpha)-(1-\alpha)\,z^{-1}\right]^N}{2\left(1+\alpha z^{-1}\right)\prod_{n=1}^{(N-1)/2}\left(A_n + B_n z^{-1} + C_n z^{-2}\right)}, & N \text{ odd} \end{cases} \tag{7.29}$$

where

$$A_n = 2 - 2\cos\left[\frac{\pi}{2}\left(1+\frac{2n-1}{N}\right)\right] + 2\alpha^2 + 2\cos\left[\frac{\pi}{2}\left(1+\frac{2n-1}{N}\right)\right]\alpha^2 \tag{7.30}$$

$$B_n = 8\alpha \tag{7.31}$$

$$C_n = 2 + 2\cos\left[\frac{\pi}{2}\left(1+\frac{2n-1}{N}\right)\right] + 2\alpha^2 - 2\cos\left[\frac{\pi}{2}\left(1+\frac{2n-1}{N}\right)\right]\alpha^2 \tag{7.32}$$

and $\alpha = -\cos\left(\dfrac{\pi/2+\omega_c}{2}\right) / \cos\left(\dfrac{\pi/2-\omega_c}{2}\right)$.

Code

```
% Compute alpha
alpha = -cos((pi/2+wc)/2)/cos((pi/2- wc)/2);

% Compute b coefficients
b = 1;
for n = 1:N
   b = convltn(b,[1-alpha;alpha-1]);
```

```
end;

% Compute a coefficients
if (rem(N,2))   % N is odd initialize with 1st order term
   a = [2;2*alpha];
else % N is even initialize with 1
   a = 1;
end;
for n = 1:floor(N/2)
   costheta = cos((pi/2)*(1+((2*n-1)/N)));
   An = 2-2*costheta+2*alpha*alpha+2*costheta*alpha*alpha;
   Bn = 8*alpha;
   Cn = 2+2*costheta+2*alpha*alpha-2*costheta*alpha*alpha;
   a = convltn(a,[An;Bn;Cn]);
end;

% Scale b and a so that a(0) = 1, i.e. monic A(z);
b = b ./ a(1);
a = a ./ a(1);
```

Note

The MATLAB Signal Processing Toolbox contains a Butterworth high-pass filter design function and when called as

```
[b,a] = butter(N,wc/pi,'high')
```

produces the same output as our function.

Reference

A. Oppenheim and R. Schafer, *Discrete-Time Signal Processing*, Section 7.1.2, Prentice-Hall: Englewood Cliffs, N.J., 1989.

Example

```
>>[b,a] = btrwrthp(3,pi/2)
>>[b a]

   0.16666666666667   1.00000000000000
  -0.50000000000000  -0.00000000000000
   0.50000000000000   0.33333333333333
  -0.16666666666667  -0.00000000000000
```

7.9 CHEBYSHV.M (Orfanidis)

Purpose

This function returns the Nth order, Type I, LP digital Chebyshev filter coefficients in length $N+1$ vectors **b** and **a**. The 3-dB frequency is ω_c in radians/sample and passband ripple is $1/(1+\varepsilon^2)$ on a magnitude-squared plot.

Input

N, filter order
ε, passband ripple parameter
ω_c, cutoff frequency, i.e. $\left|H(e^{j\omega_c})\right|^2 = 0.5$ or $\left|H(e^{j\omega_c})\right| = -3$ dB

Output

b, vector of feedforward gains
a, vector of feedback gains

Algorithm

$$H(z) = H_0(z)\prod_{i=1}^{K} H_i(z) \tag{7.33}$$

where

$$H_0(z) = \begin{cases} \sqrt{\dfrac{1}{1+\epsilon^2}} & ,N = 2K \ (N \text{ even}) \\ \dfrac{G_0\left(1+z^{-1}\right)}{1+a_{01}z^{-1}} & ,N = 2K+1 \ (N \text{ odd}) \end{cases} \tag{7.34}$$

and

$$H_i(z) = \frac{G_i\left(1+z^{-1}\right)^2}{1+a_{i1}z^{-1}+a_{i2}z^{-2}}, \tag{7.35}$$

with

$$G_i = \frac{\Omega_0^2+\Omega_i^2}{1-2\Omega_0\cos\theta_i+\Omega_0^2+\Omega_i^2}, \tag{7.36}$$

$$G_0 = \frac{\Omega_0}{\Omega_0+1}, \tag{7.37}$$

$$a_{i1} = \frac{2\left(\Omega_0^2+\Omega_i^2-1\right)}{1-2\Omega_0\cos\theta_i+\Omega_c^2+\Omega_i^2}, \tag{7.38}$$

$$a_{i2} = \frac{1+2\Omega_0\cos\theta_i+\Omega_0^2+\Omega_i^2}{1-2\Omega_0\cos\theta_i+\Omega_c^2+\Omega_i^2}, \tag{7.39}$$

$$a_{01} = \frac{\Omega_0 - 1}{\Omega_0 + 1}, \tag{7.40}$$

$$\theta_i = \frac{\pi}{2N}(N - 1 + 2i), \tag{7.41}$$

$$\Omega_0 = \frac{\Omega_c}{\cosh\left[\frac{1}{N}\cosh^{-1}(1/\varepsilon)\right]}\sinh(\mu), \tag{7.42}$$

$$\Omega_i = \frac{\Omega_c}{\cosh\left[\frac{1}{N}\cosh^{-1}(1/\varepsilon)\right]}\sin(\theta_i), \tag{7.43}$$

and

$$\mu = \frac{1}{N}\ln\left(\frac{1}{\varepsilon} + \sqrt{\frac{1}{\varepsilon^2} + 1}\right), \tag{7.44}$$

Code

```
% Prewarp
Wc = tan(wc/2);

mu = log(1/epsilon + sqrt(1/(epsilon*epsilon)+1))/N;
W0 = Wc*sinh(mu)/cosh(acosh(1/epsilon)/N);

% Compute first-order sections
if (rem(N,2))  % N is odd
   G = W0/(W0+1);
   a1 = (W0-1)/(W0+1);
   b = [G G]';
   a = [1 a1]';
   K = (N-1)/2;
else
   b = sqrt(1/(1+epsilon*epsilon));
   a = 1;
   K = N/2;
end;

% Compute second-order sections
for i = 1:K
   theta = (pi/(2*N))*(N-1+2*i);
   Wi = Wc*sin(theta)/cosh(acosh(1/epsilon)/N);
   G = (W0*W0+Wi*Wi)/(1-2*W0*cos(theta)+W0*W0+Wi*Wi);
   a1 = (2*(W0*W0+Wi*Wi-1))/(1-2*W0*cos(theta)+W0*W0+Wi*Wi);
   a2 = (1+2*W0*cos(theta)+W0*W0+Wi*Wi)/(1-2*W0*cos(theta)+W0*W0+Wi*Wi);
   b = convltn(b,[G 2*G G]');
   a = convltn(a,[1 a1 a2]');
end;
```

Note

MATLAB's Chebyshev design tool is called `cheby1`.

Reference

S. Orfanidis, *Introduction to Signal Processing*, Sections 11.6.1 - 11.6.2, Prentice-Hall: Upper Saddle River, N.J., 1996.

Example

```
>>[b,a] = chebyshv(3,0.5,pi/2);
>>[b a]

   0.11360451857163   1.00000000000000
   0.34081355571488  -0.52333550882906
   0.34081355571488   0.65720251065607
   0.11360451857163  -0.22503085325401
```

7.10 CHEBYSHV.M (Oppenheim)

Purpose

This function returns the Nth order, Type I, LP digital Chebyshev filter coefficients in length $N+1$ vectors **b** and **a**. The cut-off frequency is ω_c in radians/sample and passband ripple is $1/(1+\varepsilon^2)$ on a magnitude-squared plot.

Input

N, filter order
ε, passband ripple parameter
ω_c, cutoff frequency, i.e. $\left|H(e^{j\omega_c})\right|^2 = 1/\left(1+\varepsilon^2\right)$

Output

b, vector of feedforward gains
a, vector of feedback gains

Algorithm

$$H(z) = \begin{cases} \dfrac{g\left(\Omega_c + \Omega_c z^{-1}\right)^N}{\prod_{n=1}^{N/2}\left(A_n + B_n z^{-1} + C_n z^{-2}\right)} & , N \text{ even} \\ \dfrac{g\left(\Omega_c + \Omega_c z^{-1}\right)^N}{\left\{1 + \Omega_c \sinh(\mu) + \left[1 - \Omega_c \sinh(\mu)\right] z^{-1}\right\} \prod_{n=1}^{(N-1)/2}\left(A_n + B_n z^{-1} + C_n z^{-2}\right)} & , N \text{ odd} \end{cases} \tag{7.45}$$

where

$$A_n = 1 + 2\Omega_c \sinh(\mu)\sin(\theta_n) + \Omega_c^2\left[\sinh^2(\mu) + \cos^2(\theta_n)\right], \tag{7.46}$$

$$B_n = -2 + 2\Omega_c^2\left[\sinh^2(\mu) + \cos^2(\theta_n)\right], \tag{7.47}$$

$$C_n = 1 - 2\Omega_c \sinh(\mu)\sin(\theta_n) + \Omega_c^2\left[\sinh^2(\mu) + \cos^2(\theta_n)\right], \tag{7.48}$$

$$\Omega_c = \tan\left(\frac{\omega_c}{2}\right), \tag{7.49}$$

and

$$g = \begin{cases} \dfrac{\sum_{n=0}^{N} a_n}{\sqrt{1+\varepsilon^2}\sum_{n=0}^{N} b_n} & , N \text{ even} \\ \dfrac{\sum_{n=0}^{N} a_n}{\sum_{n=0}^{N} b_n} & , N \text{ odd} \end{cases} \tag{7.50}$$

Code

```
% Prewarp
Wc = tan(wc/2);

% Initialize
mu = log((1+sqrt(1+epsilon*epsilon))/epsilon)/N;

% Compute b coefficients
b = 1;
for n = 1:N
   b = convltn(b,[Wc;Wc]);
end;

% Compute a coefficients
if (rem(N,2))  % N is odd initialize with 1st order term
   a = [1+Wc*sinh(mu);-1+Wc*sinh(mu)];
else % N is even initialize with 1
   a = 1;
end;
for n = 1:floor(N/2)
   theta = pi/2*(2*n-1)/N;
   An = 1+2*Wc*sinh(mu)*sin(theta)+Wc*Wc*(sinh(mu)^2+cos(theta)^2);
   Bn = -2+2*Wc*Wc*(sinh(mu)^2+cos(theta)^2);
   Cn = 1-2*Wc*sinh(mu)*sin(theta)+Wc*Wc*(sinh(mu)^2+cos(theta)^2);
   a = convltn(a,[An;Bn;Cn]);
end;

% Scale DC gain
if (rem(N,2))  % N is odd
   g = sum(a)/sum(b);
else
   g = sum(a)/(sqrt(1+epsilon*epsilon)*sum(b));
end;
b = b .* g;

% Scale b and a so that a(0) = 1, i.e. monic A(z);
```

```
b = b ./ a(1);
a = a ./ a(1);
```

Note

The MATLAB Signal Processing Toolbox contains a Chebyshev function that is called with

```
[b,a] = cheby1(N,R,Wp)
```

where W_p is the normalized frequency ($f_s = 1$) and R is the peak-to-peak passband ripple in decibels. However, cheby1 does not produce the same output as our function since ours sets the DC gain differently.

Reference

A. Oppenheim and R. Schafer, *Discrete-Time Signal Processing*, Section 7.2 and B.2, Prentice-Hall: Englewood Cliffs, N.J., 1989.

Example

```
>>[b,a] = chebyshv(3,0.5,pi/2);
>>[b a]

   0.13333333333333   1.00000000000000
   0.40000000000000  -0.33333333333333
   0.40000000000000   0.60000000000000
   0.13333333333333  -0.20000000000000
```

7.11 FREQTRAN.M

Purpose

This function returns the coefficients of a frequency transform

$$F(z) = \pm\frac{p(z)}{\tilde{p}(z)} \tag{7.51}$$

where

$$\begin{aligned} p(z) &= p_0 + p_1 z^{-1} + \cdots + p_n z^{-n} \\ \tilde{p}(z) &= z^{-n} p\left(z^{-1}\right) \end{aligned} \tag{7.52}$$

and $F(z)$ meets the requirements for a valid frequency transform. The goal is to construct $G(z)$ which has multiple passbands with bandedge frequencies

$$0 < \phi_1 < \phi_2 < \cdots \phi_n < \pi \tag{7.53}$$

from a prototype low pass filter $H(z)$ with cutoff frequency $\omega_c = \pi/2$ using the composition

$$G(z) = H(F(z)). \tag{7.54}$$

Input

phi - vector of ordered bandedge frequencies, ϕ_k satisfying (7.53)
option, optional argument with values as follows
1–assume a lowpass frequency transform [positive sign in (7.51)]
(-1)–assume a lowstop frequency transform [negative sign in (7.51)]

Output

p, vector of coefficients for $p(z)$
ptilde, vector of coefficients for $\tilde{p}(z)$

Code

```
%-----------
% Initialize
%-----------
n = length(phi); % number of bandedges
v = 1/2;
p = [1;zeros(n,1)];
q = zeros(n+1,1);

%-----
% Main
%-----
for k = 1:n
   v = -v;
   phiprime = (phi(k) - pi)*v;
   for j = 0:k
      alpha = 0;
```

```
      beta = 0;
      if (j > 0)
         alpha = alpha + p(j-1+1);
         beta = beta - p(k-j+1);
      end;
      if (j < k)
         alpha = alpha + p(j+1);
         beta = beta + p(k-j);
      end;
      q(j+1) = alpha*cos(phiprime)+beta*sin(phiprime);
   end;
   p = q;
end;
ptilde = flipud(p);
p = option*p;
```

Reference

R. Roberts and C. Mullis, *Digital Signal Processing*, Addison Wesley.

Example

```
>>[p,ptilde] = freqtran(pi/8.*[1:7]')

   1.41421356237309                  0
   0.00000000000000   0.28130456767205
   0.67912930243137   0.00000000000000
   0.00000000000000   0.45377969226968
   0.45377969226968   0.00000000000000
   0.00000000000000   0.67912930243137
   0.28130456767205   0.00000000000000
                  0   1.41421356237309

>>[p,ptilde] = freqtran(3*pi/4,-1)

  -1.30656296487638   0.54119610014620
  -0.54119610014620   1.30656296487638
```

7.12 COMPOSE.M

Purpose

This function returns the coefficients of the function composition

$$G(z) = H(F(z)) \tag{7.55}$$

where

$$\begin{aligned} H(z) &= \frac{B(z)}{A(z)} \\ &= \frac{b_0 + b_1 z^{-1} + \cdots + b_{m-1} z^{-m+1}}{a_0 + a_1 z^{-1} + \cdots + a_{m-1} z^{-m+1}} \end{aligned} \tag{7.56}$$

is a prototype low pass filter with cutoff frequency $\omega_c = \pi/2$ and the frequency transformation, $F(z)$ is given by (7.51) and is generated from `FREQTRAN.M`. Substituting (7.52) into (7.51) and the result into (7.55) yields

$$\begin{aligned} G(z) &= \frac{B_G(z)}{A_G(z)} \\ &= \frac{b_0 [p(z)]^{m-1} + b_1 \tilde{p}(z) [p(z)]^{m-2} + b_2 [\tilde{p}(z)]^2 [p(z)]^{m-3} + \cdots + b_{m-1} [\tilde{p}(z)]^{m-1}}{a_0 [p(z)]^{m-1} + a_1 \tilde{p}(z) [p(z)]^{m-2} + a_2 [\tilde{p}(z)]^2 [p(z)]^{m-3} + \cdots + a_{m-1} [\tilde{p}(z)]^{m-1}}. \end{aligned} \tag{7.57}$$

Input

b, vector of feedforward gains for $H(z)$
a, vector of feedback gains $H(z)$
p, vector of coefficients for $p(z)$
ptilde, vector of coefficients for $\tilde{p}(z)$

Output

bg, vector of feedforward gains for $B_G(z)$
ag, vector of feedback gains for $A_G(z)$

Algorithm

1. Compute and store the coefficient vectors for $[p(z)]^0$, $[p(z)]^1$, ..., $[p(z)]^{m-1}$
2. Repeat 1) but for $\tilde{p}(z)$
3. Build $B_G(z)$ and $A_G(z)$ according to (7.57) using vectors from steps 1) and 2).
4. Scale $G(z)$ so that $A_G(z)$ is monic.

Notes

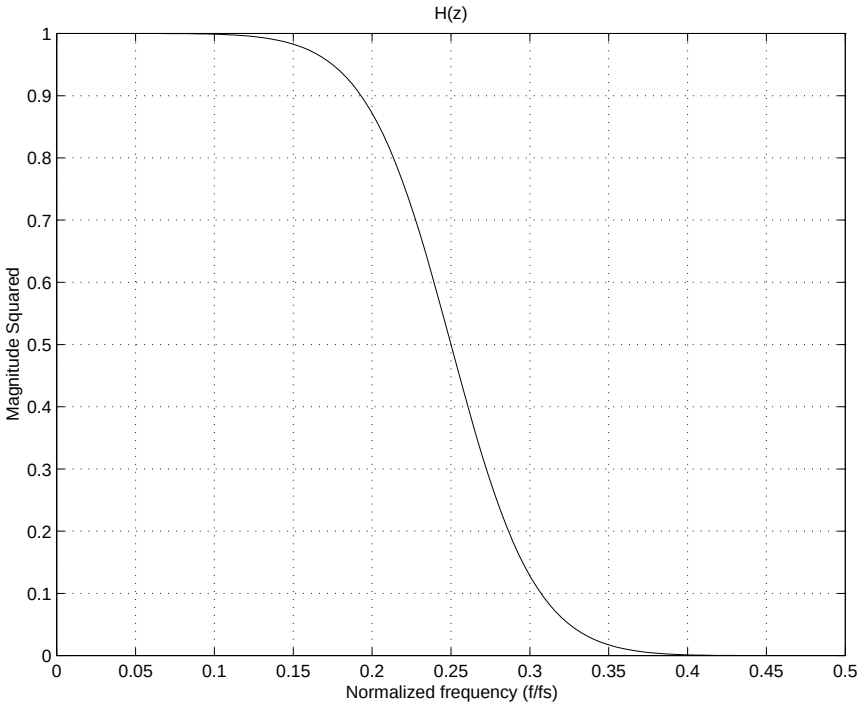
1. If length (**b**) $\neq$ length (**a**) you should zero pad the shorter vector to the length of the longer vector.
2. You can compute powers of polynomials by iteratively convolving coefficient vectors.

Reference

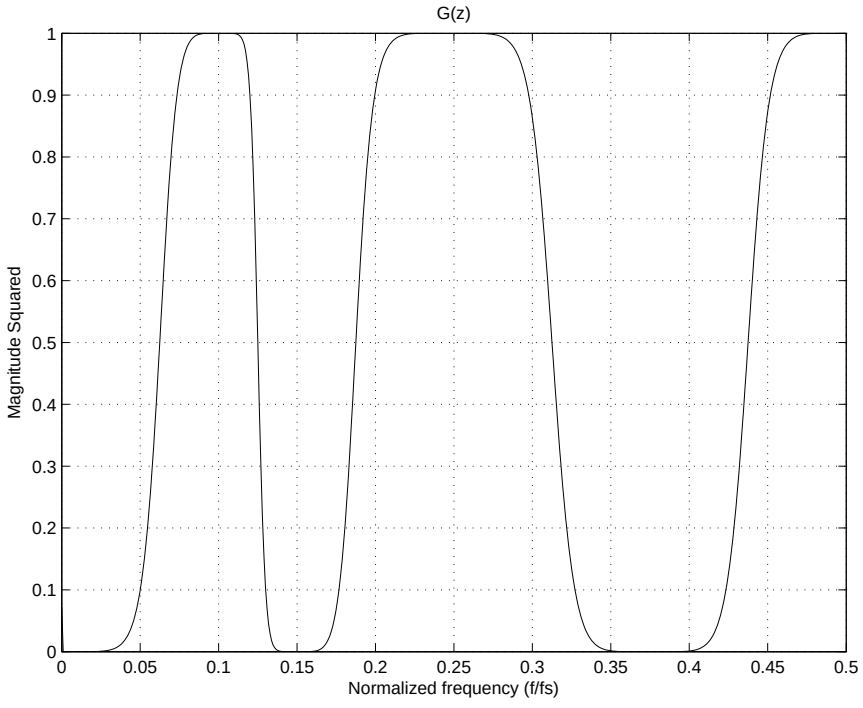
R. Roberts and C. Mullis, *Digital Signal Processing*, Addison Wesley.

Example

```
>>[b,a] = bttrwrth(3,pi/2); % lowpass prototype H(z)
>>[p,ptilde] = freqtran(pi/8.*[0;1;2;3;5;7]'); % F(z) = p(z)/p~(z)
>>[bg,ag] = compose(b,a,p,ptilde); % G(z) = H(F(z))
```



(a)



(b)

Figure 7.2: COMPOSE: (a) prototype filter and (b) filter resulting from composition.

Chapter 8

Multirate Signal Processing Tools

This chapter contains several tools used in multirate signal processing (processing signals with various sample rates).

8.1 UPSAMP.M

Purpose

This function returns an upsampled signal.

Input

x, input signal
U, upsampling factor

Output

y, upsampled signal

Algorithm

$$y(n) = \begin{cases} x(n/U), & n = 0, \pm U, \pm 2U, \cdots \\ 0, & o.w. \end{cases} \tag{8.1}$$

Notes

1. The above algorithm should be written without a for-loop.
2. The MATLAB Signal Processing Toolbox contains an upsample function and when called as

```
y = upsample(x,U)
```

produces the same output as our function.

Example

```
>>x = [1 2 3]';
>>y = upsamp(x,3)

y =
     1
     0
     0
     2
     0
     0
     3
     0
     0
```

8.2 DOWNSAMP.M

Purpose

This function returns a downsampled signal.

Input

x, input signal
D, downsampling factor

Output

y, downsampled signal

Algorithm

$$y(n) = x(Dn) \tag{8.2}$$

Notes

1. The above algorithm should be written without a for-loop.
2. The MATLAB Signal Processing Toolbox contains a downsample function and when called as

```
y = downsamp(x,D)
```

produces the same output as our function.

Example

```
>>x = [1 2 3 4 5 6 7 8 9]';
>>y = downsample(x,3)

y =
     1
     4
     7
```

Chapter 9

Adaptive Signal Processing Tools

In this chapter we develop software tools for adaptive and optimal signal processing. The tools can be grouped into four main areas:

1. Random signal synthesis and analysis
2. Optimal filters
3. Adaptive filter adjustment algorithms
4. Adaptive filter performance analysis

9.1 CORRELATION.M

Purpose

This function returns the first M correlations between the signal $\mathbf{x}$ and $\mathbf{y}$.

Input

$\mathbf{x}$, signal1
$\mathbf{y}$, signal2
M, maximum lag
option, optional argument with values as follows
0–Correlations are unbiased (default)
1–Correlations are biased

Output

$\mathbf{r}$, correlation vector

Algorithm

$$r(k) = E[x(n)\,y^*(n-k)] \text{ (unbiased)} \tag{9.1}$$

or

$$r(k) = \frac{1}{N}\sum_{n=0}^{N-1} x(n)\,y^*(n-k) \text{ (biased)} \tag{9.2}$$

Code

```
N = length(x);  % assume x and y same length
r = zeros(M+1,1);
if (option == 0)
   for k = 0:M % positive lags
      r(k+1) = mean(x(k+1:N).*conj(y(1:N-k))); % assume signals loaded forward in time.
   end;
else
   for k = 0:M % positive lags
      r(k+1) = sum(x(k+1:N).*conj(y(1:N-k))); % assume signals loaded forward in time.
   end;
   r = r ./ N;
end;
```

Notes

1. This routine produces an *estimate* of the correlation since we do not have an infinite number of realizations or (assuming ergodicity) an infinitely long signal.

2. This routine can be used to compute the auto-correlation of $\mathbf{x}$ with the function call

```
r = correlation(x,x,M);
```

3. Because of MATLAB's positive indexing we have either $r(1) = E[x(n)\,y^*(n)]$ or $r(1) = \frac{1}{N}\sum_{n=0}^{N} x(n)\,y^*(n)$.

4. MATLAB's built-in correlation function is called with

```
r = xcorr(x,y,'unbiased');
r = xcorr(x,y,'biased');
```

Example

```
>>randn('state',0); % reset random number generator to initial state
>>u = randn(10,1);
>>y = randn(10,1);
>>r_uy = correlation(u,y,3)
>>r_uy_biased = correlation(u,y,3,1)

r_uy =
   0.08079365325442
  -0.25940682153763
   0.45661040843919
   0.17692039770706

r_uy_biased =
   0.08079365325442
  -0.23346613938387
   0.36528832675135
   0.12384427839494
```

9.2 PERIODOGRAM.M

Purpose

This function returns the periodogram [estimate of power spectral density (PSD)] of the signal **u**.

Input

u, length N input signal

Output

$S(\omega_k)$ for $0 \leq k \leq N-1$ where $\omega_k = e^{j2\pi k/N}$.

Algorithm

Based on Haykin (3.2) and (3.12).

Code

```
N = length(u);
U = fft(u,N)
S = U.*conj(U)/N;
```

Notes

1. The periodogram will give us a "noisy" estimate of the true PSD. Better estimates can be made by averaging periodograms over short windows of the signal (see WELCH.M).
2. It is common practice to window **u** with a Hamming or other non-rectangular window prior to computing the periodogram.
3. A PSD tool exists in MATLAB's Signal Processing Toolbox and is based on the spectrogram.

Reference

S. Haykin, *Adaptive Filter Theory, 3rd Ed.*, Prentice-Hall: Upper Saddle River, N.J., 1996.

Example

```
>>randn('state',0); % reset random number generator to initial state
>>u = randn(12,1);
>>S=periodogram(u)

S =
   0.025378022701470
   0.959167568020567
   0.269232479462049
   0.607499902770735
   2.027209966952694
   0.051152739362579
   0.053301310493617
   0.051152739362579
```

```
2.027209966952694
0.607499902770735
0.269232479462049
0.959167568020567
```

9.3 WELCH2.M

Purpose

This function returns the periodogram [estimate of power spectral density (PSD)] of the signal **u** using Welch's method of averaging periodograms taken of overlapped windows of the signal.

Input

u, length N input signal
w, length L window
overlap, window overlap in samples

Output

$S(\omega_k)$ for $0 \leq k \leq L-1$ where $\omega_k = e^{j2\pi k/L}$.

Algorithm

Based on Hayes.

Code

```
N = length(u);
L = length(w);
U = sum(w.^2)/L; % window gain
K = floor((N-L)/(L-overlap))+1;  % number of windows
S = 0;  % clear accumulator

start = 1;stop = L;
for i = 1:K
   uu = u(start:stop).*w;  % window signal
   UU = fft(uu,L);  % compute periodogram
   S = S + (UU.*conj(UU));  % accumulate periodograms
   start = start + (L - overlap);  % advance window pointers
   stop = stop + (L - overlap);
end;
S = S / (K*L*U);  % average periodograms, divide out window and FFT gains
```

Notes

A PSD tool exists in MATLAB's Signal Processing Toolbox and is based on the spectrogram.

Reference

M. Hayes, *Statistical Digital Signal Processing*, John Wiley & Sons, 1996.

Example

```
>>randn('state',0); % reset random number generator to initial state
>>u = randn(26,1);
>>w = hamming(12);
```

```
>>S=welch2(u,w,8)

S =
   0.71612520758699
   0.59869794558873
   0.35369962883475
   0.53513519102011
   0.94113773433479
   1.09562216413306
   0.75190655017891
   1.09562216413306
   0.94113773433479
   0.53513519102011
   0.35369962883475
   0.59869794558873
```

9.4 PERIODOGRAM_PLOT.M

Purpose

This function plots periodogram data (PSD estimate).

Input

$S(\omega_k)$, periodogram data for $0 \leq \omega_k \leq 2\pi$
units, optional argument with values as follows
    0–plot in dB (default)
    1–plot in normal units
f_s, optional argument with values as follows
    2π–frequency is in radians/sample (default)
    other specified value–frequency is in Hertz

Output

Plot of $S(\omega_k)$

Notes

1. If $units = 0$ or is not given, plot in dB.
   If $units = 1$, plot in normal units.

2. If $f_s = 2\pi$ or is not given, frequency is in radians/sample.
   If $f_s \neq 2\pi$, frequency is in Hertz.

3. Code should should test for $S(\omega) = S(-\omega)$. If true, plot should only be over $0 \leq \omega \leq \pi$ or equivalent in Hertz.

Example

```
>>randn('state',0); % reset random number generator to initial state
>>u = randn(512,1);
>>w = hamming(128);
>>S=welch2(u,w,64)
>>periodogram_plot(S);
```

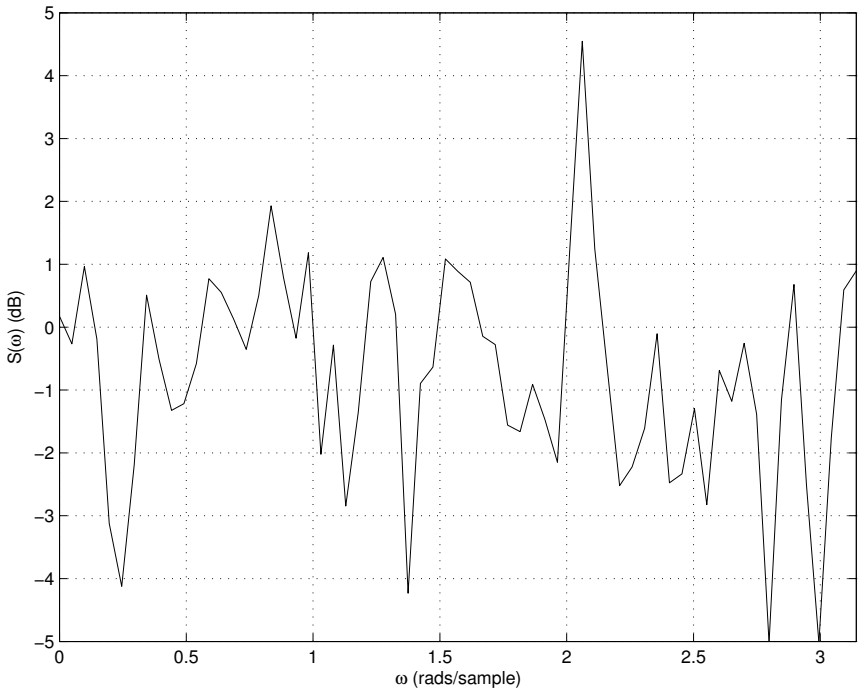


Figure 9.1: Periodogram plot.

9.5 AR_SYNTHESIZER.M

Purpose

This function returns a synthesized autoregressive (AR) process by filtering a zero mean, σ^2-variance white Gaussian noise through an all-pole filter defined by the AR coefficients.

Input

$\mathbf{a} = [1, a_1, \ldots, a_M]^T$, AR coefficients
N, signal length
σ^2, variance of input Gaussian noise

Output

$\mathbf{u}$, length N AR process

Code

```
v = randn(N,1)*sqrt(sigma);
u = filter(1,a,v);
```

Example

```
>>randn('state',0);  % reset random number generator to initial state
>>a = [1 -1 1/4]';
>>u = ar_synthesizer(a,10,2)

u =
  -0.61173902306857
  -2.96723104004965
  -2.63704963666230
  -1.48840598140388
  -2.45049890524423
  -0.39418860674107
   1.90016826143497
   1.94549392296543
   1.93331315397192
   1.69391671752913
```

9.6 AR_COEFFICIENT_ESTIMATOR.M

Purpose

This function returns the coefficients of an order M AR model of the signal $\mathbf{x}$ through the use of the Yule-Walker equations.

Input

$\mathbf{u}$, AR process
M, model order

Output

$\mathbf{a} = [1, a_1, \ldots, a_M]^T$, vector of AR coefficient estimates

Algorithm

$$\mathbf{w} = \mathbf{R}^{-1}\mathbf{r} \tag{9.3}$$

where

$$\mathbf{w} = \begin{bmatrix} w_1 & w_2 & \ldots & w_M \end{bmatrix}^T, \tag{9.4}$$

$$\mathbf{R} = \begin{bmatrix} r(0) & r(1) & \ldots & r(M-1) \\ r^*(1) & r(0) & \ldots & r(M-2) \\ \vdots & \vdots & \ddots & \vdots \\ r^*(M-1) & r^*(M-2) & \ldots & r^*(0) \end{bmatrix}^T, \tag{9.5}$$

and

$$\mathbf{r} = \begin{bmatrix} r_1^* & r_2^* & \ldots & r_M^* \end{bmatrix}^T. \tag{9.6}$$

Code

```
r = correlation(u,u,M);
R = toeplitz(r(1:M));
w = inv(R)*conj(r(2:M+1));
a = [1 (-1).*w']';
```

Example

```
>>randn('state',0);  % reset random number generator to initial state
>>a = [1 -1 1/4]';   % actual
>>u = ar_synthesizer(a,10,2);
>>a_est = ar_coef_est(u,2)

a_est =
   1.00000000000000
  -1.25612921674934
   0.54595749717382
```

9.7 ADD_NOISE.M

Purpose

This function adds the signal, s to the noise, v so that a desired SNR (in dB) is met. The resulting noisy signal, y is normalized, $-1 \leq y(n) \leq 1$ and returned.

Input

s, signal of interest
v, noise signal
SNR, desired signal-to-noise ratio in dB

Output

y, noisy signal with desired SNR

Example

```
>>randn('state',0);  % reset random number generator to initial state
>>s = randn(10,1);
>>v = randn(10,1);
>>y = add_noise(s,v,10)

y =
  -0.37762707167451
  -1.00000000000000
  -0.09360643410563
   0.60932471665082
  -0.84062992463817
   0.76441220904635
   0.96787370978648
  -0.06269084441913
   0.14623256337815
  -0.11328892402434
```

9.8 STEEPEST_DESCENT.M

Purpose

This function adjusts the length-M filter, w using the Steepest-Descent (SD) algorithm such that the output of w when driven by u is close (in the mean square sense) to d. The resulting squared error signal e^2 and w are returned.

Input

w_0, initial value of w, typically $w_0 = [0, \ldots, 0]^T$
R_u, correlation matrix of input signal
R_{du}, cross-correlation vector between desired and input signals
μ, step size
σ_d^2, variance of desired signal
N, number of iterations (updates) to perform
option, optional argument with values as follows
    0–return w_N (default)
    1–return a matrix W whose columns are $w_0, w_1, \ldots, w_N$

Output

J, vector of squared errors (squared error signal)
w, either w_N or a matrix, W whose columns are $w_0, w_1, \ldots, w_N$ according to how *option* is set

Algorithm

$J(0) = \sigma_d^2 - w_0^* R_{du} - R_{du}^* w_0 + w_0^* R_u w_0$
for $i = 1$ to N
    $w_i = w_{i-1} + \mu \left[R_{du} - R_u w_{i-1} \right]$
    $J(i) = \sigma_d^2 - w_i^* R_{du} - R_{du}^* w_i + w_i^* R_u w_i$
end

Notes

1. Large N and long w have been known to kill ordinary computers.

2. This code should check that

$$0 < \mu < \frac{2}{\lambda_{\max}} \tag{9.7}$$

    where $\lambda_{\max}$ is the largest eigenvalue of R_u and warn if it is not.

3. This tool is based on Sayed (Section 4.2).

Reference

A. Sayed, *Fundamentals of Adaptive Filtering*, John Wiley & Sons: Hoboken, N.J., 2003.

Example

```
>>r = [1;0.5;0.25];
>>R_u = toeplitz(r(1:2));
>>R_du = [r(2);r(3)];
>>[J W] = steepest_descent([0;0],R_u,R_du,1.0,r(1),6,1)

J =
   1.00000000000000
   0.81250000000000
   0.76562500000000
   0.75390625000000
   0.75097656250000
   0.75024414062500
   0.75006103515625

W =
    0    0.5000    0.3750    0.5000    0.4688    0.5000    0.4922
    0    0.2500         0    0.0625         0    0.0156         0
```

9.9 MSE_PLOT.M

Purpose

This function plots the mean squared-error (MSE) data, J.

Input

$J = \left[e^2(0), \ldots, e^2(N)\right]^T$
start (optional), index at which filter adjustment begins (see Notes for more details)
stop (optional), index at which filter adjustment ends (see Notes for more details)
option, optional argument with values as follows
0–plot is in normal units
1–plot is in dB (default)
D (optional), J is downsampled by a factor of D (see Notes for more details)

Output

Plot of squared-error data

Code

```
if (option)
   plot([0:D:stop-start],10*log10(J(start+1:D:stop+1)),'k');
   ylabel('MSE (dB)');
   axis([0 stop-start min(10*log10(J(start+1:stop+1))) max(10*log10(J(start+1:stop+1)))]);
else
   plot([0:D:stop-start],J(start+1:D:stop+1),'k');
   ylabel('MSE');
   axis([0 stop-start 0 ceil(max(J(start+1:stop+1)))]);
end;
xlabel('n');
grid
```

Notes

1. If *start* is not specified, assume $start = 0$; if *stop* is not specified, assume $stop = N$. Must always have $start \geq 0$ and $stop \geq start$.
2. The following pairs are to be plotted $[0, J(start)]$, $[1, J(start+1)]$, $\ldots$, $[stop - start, J(stop)]$.
3. If D is not specified, assume $D = 1$ (no downsampling). For $D > 1$ time scale on plot stays the same.

Example

```
>>J =[1.0000;0.8125;0.7656;0.7539;0.7510;0.7502;0.7501];
>>MSE_plot(J,0,6,0,1);
>>MSE_plot(J,0,6,0,2);
>>MSE_plot(J,0,6,1,1);
```

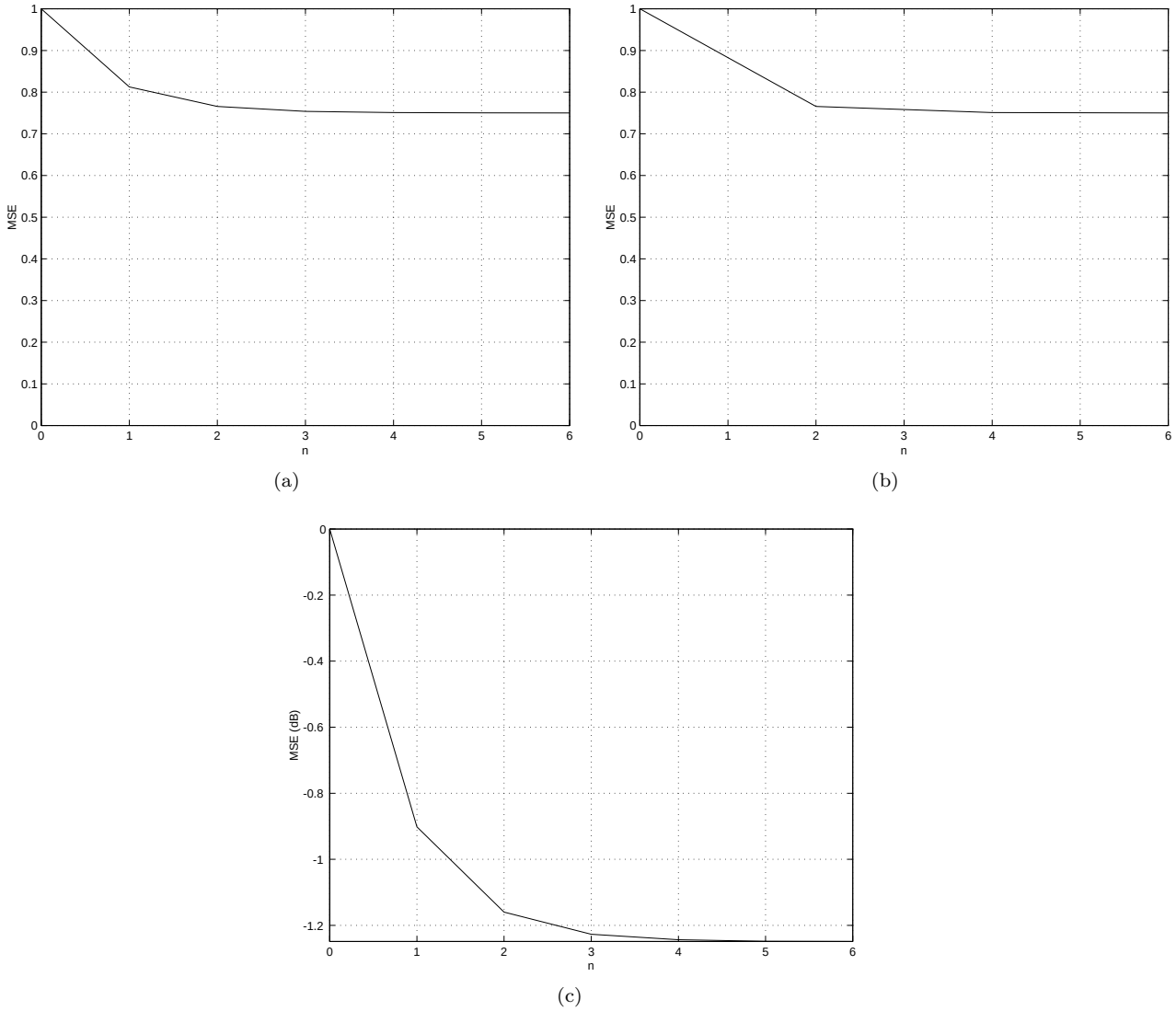


Figure 9.2: MSE_PLOT in (a) normal units, (b) normal units, data downsampled by 2, and (c) dB.

9.10 TRAJECTORY.M

Purpose

This function plots the trajectory of w superimposed onto contours of constant MSE.

Input

σ_d^2, variance of desired signal
r_u, vector of correlations of the input signal
R_{du}, cross-correlation vector between input and desired signals
W, matrix whose columns are $w_0, w_1, \ldots, w_N$. We assume a length two adaptive filter.
contours, vector of constant MSE values which are to be plotted
grid_data, 2×3 matrix of the form

$$\textbf{grid\_data} = \begin{bmatrix} w_{0,min} & w_{0,max} & w_{0,inc} \\ w_{1,min} & w_{1,max} & w_{1,inc} \end{bmatrix} \tag{9.8}$$

used to build grid points for contour evaluation where $w_{0,min}$, $w_{0,max}$, and $w_{0,inc}$ define the range of values and increment of w_0 and $w_{1,min}$, $w_{1,max}$, and $w_{1,inc}$ define the range of values and increment of w_1.

Output

Plot of the trajectory of w superimposed onto contours of constant MSE.

Code

(The motivated student is invited to improve this code. Please direct simpler, more efficient implementations to the author.)

```
%--------------
% Draw contours
%--------------
w0_min = grid_data(1,1);
w0_max = grid_data(1,2);
w0_inc = grid_data(1,3);
w1_min = grid_data(2,1);
w1_max = grid_data(2,2);
w1_inc = grid_data(2,3);

[W0,W1] = meshgrid([w0_min:w0_inc:w0_max],[w1_min:w1_inc:w1_max]);
J = sigma_d2 - 2.*R_du(1).*W0 - 2.*R_du(2).*W1 + r_u(1).*W0.^2 + 2.*r_u(2).*W0.*W1 + r_u(1).*W1.
contour(W0,W1,J,contours);

%----------------
% Draw trajectory
%----------------
hold on
plot(W(1,:),W(2,:),'k')
hold off

xlabel('w0'),
ylabel('w1'),
grid;
```

Notes

1. Formula for J is derived from (X.X) assuming $w = [w_0 \;\; w_1]^T$.
2. **contours** are typically $J(1), \ldots, J(6)$ from STEEPEST_DESCENT.M as in Sayed Figure X.X.
3. You should hand-label contour values on your figure as in Sayed Figure X.X.

Reference

A. Sayed, *Fundamentals of Adaptive Filtering*, John Wiley & Sons: Hoboken, N.J., 2003.

Example

```
>>r_u = [1;0.5;0.25];
>>R_u = toeplitz(r_u(1:2));
>>R_du = [r_u(2);r_u(3)];
>>[J W] = steepest_descent([0;0],R_u,R_du,1.0,r_u(1),6,1);
>>grid_data = [-0.5 1.5 0.01;-0.75 0.75 0.01];
>>trajectory(r_u(1),R_u,R_du,W,J(1:5),grid_data)
```

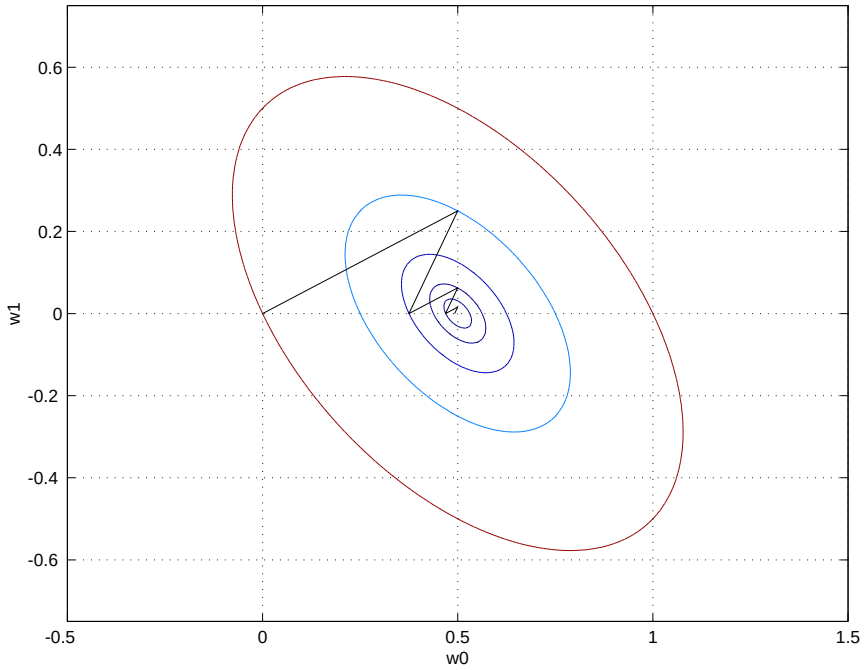


Figure 9.3: Trajectory plot of filter adjusted with steepest descent.

9.11 LMS.M

Purpose

This function adjusts the length-M filter, w using the Least-Mean-Square (LMS) algorithm such that the output of w when driven by the input signal, u is close (in the mean-square sense) to the desired signal, d. The resulting error signal e and w are returned.

Input

w_{-1}, initial value of w. Typically $w_{-1} = [0, \ldots, 0]^T$
u, input signal (row vector, length $N + 1$)
d, desired signal (length $N + 1$)
μ, step size
option, optional argument with values as follows
0–return w_N (default)
1–return a matrix W whose columns are $w_{-1}, w_0, w_1, \ldots, w_N$

Output

e, *a priori* error signal (length $N + 1$) defined as $e(i) = d(i) - u_i w_{i-1}$
w, either w_N or a matrix whose columns are $w_{-1}, w_0, w_1, \ldots, w_N$ according to how option is set

Algorithm

for $i = 0$ to N
$e(i) = d(i) - u_i w_{i-1}$
$w_i = w_{i-1} + \mu u_i^* e(i)$
end

Note

1. With *option* $= 1$, large M, and large N this function has been known to kill ordinary computers.

Example

```
>>w_init = zeros(2,1);
>>randn('state',0);
>>u = randn(1,6);
>>b = [1;2];a = [1];
>>d = filter(b,a,u.');
>>mu = 0.33;
>>[e,w] = lms(w_init,u,d,mu,1)

e =
  -0.43256481152822
  -2.42786905553292
  -2.80358560075979
  -0.06652722248221
   0.34720439371611
  -0.26409462616723
```

```
w =
   0    0.0617    1.3962    1.2803    1.2739    1.1426    1.0388
   0         0    0.3466    1.8875    1.8848    1.9177    2.0177
```

9.12 NLMS.M

Purpose

This function adjusts the length-M filter, w using the Normalized Least-Mean-Square (NMLS) algorithm such that the output of w when driven by the input signal, u is close (in the mean square sense) to the desired signal, d. The resulting error signal e and w are returned.

Input

w_{-1}, initial value of w, typically $w_{-1} = [0, \ldots, 0]^T$
u, input signal (row vector, length $N + 1$)
d, desired signal (length $N + 1$)
μ, step size
option, optional argument with values as follows
0–return w_N (default)
1–return a matrix W whose columns are $w_{-1}, w_0, w_1, \ldots, w_N$

Output

e, *a priori* error signal (length $N + 1$) defined as $e(i) = d(i) - u_i w_{i-1}$
w, either w_N or a matrix whose columns are $w_{-1}, w_0, w_1, \ldots, w_N$ according to how option is set

Algorithm

for $i = 0$ to N
$e(i) = d(i) - u_i w_{i-1}$
$w_i = w_{i-1} + \frac{\mu}{\epsilon + ||u_i||^2} u_i^* e(i)$
end

Notes

1. This code should check that

$$0 < \mu < 2 \tag{9.9}$$

and warn if it is not.

2. Assume in the algorithm the offset, $\epsilon = 0.01$.

3. With *option* $= 1$, large M, and large N this function has been known to kill ordinary computers.

Example

```
>>w_init = zeros(2,1);
>>randn('state',0);
>>u = randn(1,6);
>>b = [1;2];a = [1];
>>d = filter(b,a,u.');
>>mu = 1;
>>[e,w] = nlms(w_init,u,d,mu,1)
e =
  -0.43256481152822
```

```
  -0.94962887762260
  -3.16126295143432
  -0.10019156046895
   0.11313534057223
  -0.06341687326547

w =
   0    0.9493    1.4816    1.3401    1.0744    0.9822    0.9546
   0         0    0.1382    2.0188    1.9030    1.9262    1.9527
```

9.13 APA.M

Purpose

This function adjusts the length-M filter, w using the Affine Projection Algorithm (APA), also known as the Generalized Normalized Least-Mean-Square (GNLMS) algorithm such that the output of w when driven by the input signal, u is close (in the mean-square sense) to the desired signal, d. The resulting error signal e and w are returned.

Input

w_{-1}, initial value of w, typically $w_{-1} = [0, \cdots, 0]^T$
u, input signal (row vector, length $N+1$)
d, desired signal (length $N+1$)
μ, step size
K, projection order
option, optional argument with values as follows
0–return w_N (default)
1–return a matrix W whose columns are $w_{-1}, w_0, w_1, \ldots, w_N$

Output

e, *a priori* error signal (length $N+1$) defined as $e(i) = d(i) - u_i w_{i-1}$
w, either w_N or a matrix whose columns are $w_{-1}, w_0, w_1, \ldots, w_N$ according to how option is set

Algorithm

for $i = 0$ to N
$U_i = \left[u_i^T | u_{i-1}^T | \ldots | u_{i-K+1}^T\right]^T$
$e\_\text{vec} = d_i - U_i w_{i-1}$
$e(i) = e\_\text{vec}(1)$
$w_i = w_{i-1} + \mu U_i^* \left(\epsilon \mathrm{I} + U_i U_i^*\right)^{-1} e\_\text{vec}$
end

Notes

1. Assume in the algorithm the offset, $\epsilon = 10^{-4}$.
2. With *option* $= 1$, large M, large N, and large K this function has been known to kill ordinary computers.

Example

```
>>w_init = zeros(2,1);
>>randn('state',0);
>>u = randn(1,6);
>>b = [1;2];a = [1];
>>d = filter(b,a,u.');
>>mu = 1;
>>[e,w] = apa(w_init,u,d,mu,2,1)

e =
```

```
  -0.43256481152822
  -0.86601929976804
  -0.02843472135000
   0.00000000226432
   0.00000000028938
   0.00000000000009

w =
   0    0.9995    1.0041    1.0000    1.0000    1.0000    1.0000
   0         0    1.9832    2.0000    2.0000    2.0000    2.0000
```

9.14 SOAF.M

Purpose

This function adjusts the length-M transformed filter, $\bar{w}$ using the general transform-domain LMS algorithm a.k.a. self-orthogonalizing adaptive filter (SOAF) algorithm such that the output of w (untransformed filter) when driven by the input signal, u is close (in the mean square sense) to the desired signal, d. The resulting error signal e and $\bar{w}$ are returned.

Input

$\bar{w}_{-1}$, initial value of $\bar{w}$, typically $\bar{w}_{-1} = [0, \cdots, 0]^T$
u, input signal (row vector, length $N+1$)
d, desired signal (length $N+1$)
μ, step size
T, $M \times M$ matrix whose columns are the eigenvectors of the correlation matrix, R
D, $M \times M$ diagonal matrix whose elements are the eigenvalues of the correlation matrix, R

Output

e, *a priori* error signal (length $N+1$) defined as $e(i) = d(i) - \bar{u}_i \bar{w}_{i-1}$
$\bar{w}\,(N)$, final value of the transformed adaptive filter coefficients

Algorithm

for $i = 0$ to $N-1$
    $\bar{u}_i = u_i T$
    $e(i) = d(i) - \bar{u}_i \bar{w}_{i-1}$
    $\bar{w}_i = \bar{w}_{i-1} + \mu D^{-1} \bar{u}_i^* e(i)$
end

Notes

1. The computation of $\bar{u}_i$ can be in-place, i.e. only the current value need be stored.
2. D^{-1} may be precomputed outside the main loop.
3. D^{-1} may be computed with MATLAB's pseudo-inverse command, `pinv` in the case where one or more of the diagonal elements is zero, i.e. R has a zero eigenvalue (singularity).
4. $D^{-1}\bar{u}_i^*$ can be efficiently implemented in MATLAB as $\lambda^{-1}.* \bar{u}_i^*$ where λ^{-1} is a vector containing the inverses of the eigenvalues.
5. The untransformed filter coefficients can be obtained with $w_i = T\bar{w}_i$.

Example

```
>>randn('state',0);
>>w_init = zeros(2,1);
>>u = randn(1,6);
>>Q = eye(2,2);
>>D = diag(ones(2,1));
>>b = [1;2];a = [1];
```

```
>>d = filter(b,a,u');
>>mu = 1.0;
>>[e,w] = soaf(w_init,u,d,mu,Q,D)

e =
  -0.43256481152822
  -2.21906265050177
  -2.09374524299257
  -1.06063645702914
   1.98902585438116
   3.26726193172241

w =
   4.92628598842705
   1.14063941032866
```

9.15 DCT-LMS.M

Purpose

This function adjusts the length-M transformed filter, $\bar{w}$ using the Discrete Cosine Transform (DCT) Least-Mean-Square (LMS) algorithm such that the output of w (untransformed filter) when driven by the input signal, u is close (in the mean square sense) to the desired signal, d. The resulting error signal e and $\bar{w}$ are returned.

Input

$\bar{w}_{-1}$, initial value of $\bar{w}$, typically $\bar{w}_{-1} = [0, \cdots, 0]^T$
u, input signal (row vector, length $N+1$)
d, desired signal (length $N+1$)
μ, step size

Output

e, *a priori* error signal (length $N+1$) defined as $e(i) = d(i) - \bar{u}_i \bar{w}_{i-1}$
$\bar{w}\,(N)$, final value of the transformed adaptive filter coefficients

Algorithm

$\alpha(0) = 1/\sqrt{M}$, $\alpha(k) = \sqrt{2/M}$ for $k \neq 0$
$S = \text{diag}\{2\cos(k\pi/M)\}, \quad k = 0, 1, \ldots, M-1$
$\lambda_k(-1) = \epsilon$ (small number)
for $i = 0$ to N
  $a(k) = [u(i) - u(i-1)]\cos(\frac{k\pi}{2M}), \quad k = 0, 1, \ldots, M-1$
  $b(k) = (-1)^k[u(i-M) - u(i-M-1)]\cos(\frac{k\pi}{2M}), \quad k = 0, 1, \ldots, M-1$
  $\phi(k) = \alpha(k)[a(k) - b(k)], \quad k = 0, 1, \ldots, M-1$
  $\bar{u}_i = \bar{u}_{i-1}S - \bar{u}_{i-2} + [\phi(0)\ \phi(1)\ \cdots \phi(M-1)]$
  $\bar{u}_i(k) = k$-th entry of $\bar{u}_i$
  $\lambda_k(i) = \beta\lambda_k(i-1) + (1-\beta)|\bar{u}_i(k)|^2, \quad k = 0, 1, \ldots, M-1$
  $D_i = \text{diag}\{\lambda_k(i)\}$
  $e(i) = d(i) - \bar{u}_i \bar{w}_{i-1}$
  $\bar{w}_i = \bar{w}_{i-1} + \mu D_i^{-1} \bar{u}_i^* e(i)$
end

Notes

1. Choose $\beta = 0.99$ and $\epsilon = 10^{-6}$.
2. Precalculate $\cos(\frac{k\pi}{2M})$ for $0 \leq k \leq M-1$ and store in a lookup table.
3. In a MATLAB implementation, calculations for a, b, ϕ, and λ can be vectorized.
4. $D^{-1}\bar{u}_i^*$ can be efficiently implemented in MATLAB as $\lambda^{-1}.*\bar{u}_i^*$ where λ^{-1} is a vector containing the inverses of the eigenvalue estimates.
5. The untransformed filter coefficients can be obtained with $w_i = T\bar{w}_i$ where $T = C^T$ and C is the DCT matrix.

Reference

A. Sayed, *Fundamentals of Adaptive Filtering*, John Wiley & Sons: Hoboken, N.J., 2003.

Example

```
>>randn('state',0);
>>w_init = zeros(2,1);
>>u = randn(1,6);
>>b = [1;2];a = [1];
>>d = filter(b,a,u');
>>mu = 0.02;
>>[e,w] = dct_lms(w_init,u,d,mu)

e =
  -0.43256481152822
   4.12458095866599
   1.16150171104083
   1.99084118510753
  -6.61603673778776
  -0.55030247469508

w =
  -0.82771300864532
  -0.61460815137789
```

9.16 RLS.M

Purpose

This function adjusts the length-M filter, w using the Recursive Least Squares (RLS) algorithm such that the sum of the squared errors is minimized. The resulting a priori estimation error signal e and w are returned.

Input

w_{-1}, initial value of w, typically $w_{-1} = [0, \cdots, 0]^T$
u, input signal (row vector, length $N+1$)
d, desired signal (length $N+1$)
λ, forgetting factor
option, optional argument with values as follows
0–return w_N (default)
1–return a matrix W whose columns are $w_{-1}, w_0, w_1, \ldots, w_N$

Output

e, *a priori* error signal (length $N+1$) defined as $e(i) = d(i) - u_i w_{i-1}$
w, either w_N or a matrix whose columns are $w_{-1}, w_0, w_1, \ldots, w_N$ according to how option is set

Algorithm

ϵ = small positive constant
$P_{-1} = \epsilon^{-1}\mathrm{I}$
for $i = 0$ to N
$P_i = \lambda^{-1}\left(P_{i-1} - \frac{\lambda^{-1}P_{i-1}u_i^* u_i P_{i-1}}{1+\lambda^{-1}u_i P_{i-1}u_i^*}\right)$
$e(i) = d(i) - u_i w_{i-1}$
$w_i = w_{i-1} + P_i u_i^* e(i)$
end

Notes

1. Unless otherwise specified, set ϵ to the machine epsilon (`eps` in MATLAB).
2. Precompute the constant, λ^{-1} for use in the main loop.
3. Only the most recent P need to be stored.
4. Depending on λ, you may wish to begin the loop at $i = M-1$ instead of $i = 0$ so that initial zeros in the regressor do not interfere in the adaptation.

Example

```
>>randn('state',0);
>>w_init = zeros(2,1);
>>u = randn(1,6);
>>b = [1;2];a = [1];
>>d = filter(b,a,u');
>>lambda = 1.0;
>>[e,w] = rls(w_init,u,d,lambda,1)
```

```
e =
  -0.43256481152822
  -0.81663157193429
  -0.12489437440169
  -0.00330305701062
   0.01241099256073
  -0.00689625709134

w =
   0    1.0291    1.0291    1.0112    1.0109    1.0076    1.0064
   0         0    1.9272    2.0005    2.0005    2.0019    2.0034
```

9.17 KALMAN.M

Purpose

This function implements the Discrete Kalman Filter which returns an estimate of the state vector, $\hat{\boldsymbol{x}}_{i+1|i}$ and an updated estimation error covariance matrix, $\mathbf{P}_{i+1|i}$ given $\hat{\boldsymbol{x}}_{i|i-1}$; $\mathbf{P}_{i|i-1}$; current observation or measurement data, $\boldsymbol{y}_i$; and other state equation data.

Input

$\hat{\boldsymbol{x}}_{i|i-1}$, state vector estimate for time i given observations up through time $i-1$
$\hat{\boldsymbol{y}}_i$, measurement vector for time i
F_i, state transition matrix for time i
G_i, process noise matrix for time i
H_i, observation or measurement matrix for time i
$P_{i|i-1}$, estimation error covariance matrix for time i given observations up through time $i-1$
Q_i, process noise covariance matrix for time i
R_i, measurement noise covariance matrix for time n
S_i cross-covariance matrix between process and measurement noise for time i

Output

$\hat{\boldsymbol{x}}_{i+1|i}$, state vector estimate for time $i+1$ given observations up through time i
$P_{i+1|i}$, estimation error covariance matrix for time $i+1$ given observations up through time i

Algorithm

$R_{e,i} = R_i + H_i P_{i|i-1} H_i^*$
$K_{p,i} = \left(F_i P_{i|i-1} H_i^* + G_i S_i\right) R_{e,i}^{-1}$
$\boldsymbol{e}_i = \boldsymbol{y}_i - H_i \hat{\boldsymbol{x}}_{i|i-1}$
$\hat{\boldsymbol{x}}_{i+1|i} = F_i \hat{\boldsymbol{x}}_{i|i-1} + K_{p,i} \boldsymbol{e}_i$
$P_{i+1|i} = F_i P_{i|i-1} F_i^* + G_i Q_i G_i^* - K_{p,i} R_{e,i} K_{p,i}^*$

Reference

A. Sayed, *Fundamentals of Adaptive Filtering*, John Wiley & Sons: Hoboken, N.J., 2003.

Example

Example 7.4.2 in M. Hayes, *Statistical Digital Signal processing and Modeling*, Wiley: New York, N.Y., 1996.

```
>> randn('state',0);
>> N = 10;  % number of estimates
>> x_est = zeros(N,1);  % malloc and init for state estimates
>> F = 0.8;  % state transition matrix
>> G = 1.0; % process noise matrix
>> H = 1.0;  % measurement matrix
>> P = [1;zeros(N-1,1)];  % malloc and init for error covariance
>> Q = 0.36;  % assume process noise is white, 0.36 variance
>> R = 1;  % assume measurement noise is white, unit variance
```

```
>> S = 0; % assume zero cross-correlation between process and measurement noises
>> x = ar_synthesizer([1 -0.8],N,Q);  % actual states
>> v = randn(N,1);  % measurement noise
>> y = H.*x + v;  % get measurements
>> for i = 1:N-1
>>    [x_est(i+1),P(i+1)] = kalman(x_est(i),y(i),F,G,H,P(i),Q,R,S);
>> end;
>> [x x_est P]  % display info

ans =
  -0.25953888691693                   0   1.00000000000000
  -1.20698173647640  -0.17849898583935   0.68000000000000
  -0.89038600529622  -0.24081380657327   0.61904761904762
  -0.53970295202185  -0.57129936626056   0.60470588235294
  -1.11964517202636   0.21064265800152   0.60117302052786
  -0.18116685823529  -0.27202823431448   0.60029304029304
   0.56856503440303  -0.15616603009643   0.60007324665812
   0.43227206156643   0.41255796347047   0.60001831082628
   0.54219306609834   0.35374549038018   0.60000457765418
   0.53853793857123   0.31083627612490   0.60000114441027
```

Chapter 10

Digital Speech Processing Tools

In this chapter we develop software tools for digital speech processing. The tools can be grouped into four main areas:

1. Visualization
2. Analysis
3. Synthesis
4. Application-specific (coding and speaker identification)

10.1 PLOTCSIG2.M

Purpose

This function will plot a continuous-time signal by connecting sample values. The input sequence will be downsampled so as to avoid an "ink blob."

Input

x, vector of samples
f_s, sampling rate in Hertz

Output

Plot of a continuous-time signal

Code

```
N = length(x);
D = max(round(N/500),1);  % avoid D < 1
x = x(1:D:N);  % downsample signal to 500 samples
ND = length(x);  % downsampled signal length
t = [0:D/fs:ND*D/fs-D/fs]';  % build time vector
plot(t,x);  % plot sequence vs. time vector
ylabel('x(t)');
xlabel('t (sec.)');
grid;
```

Example

```
>>[x,fs,bits] = wavread('name.wav');
>>plotcsig2(x,fs)
```

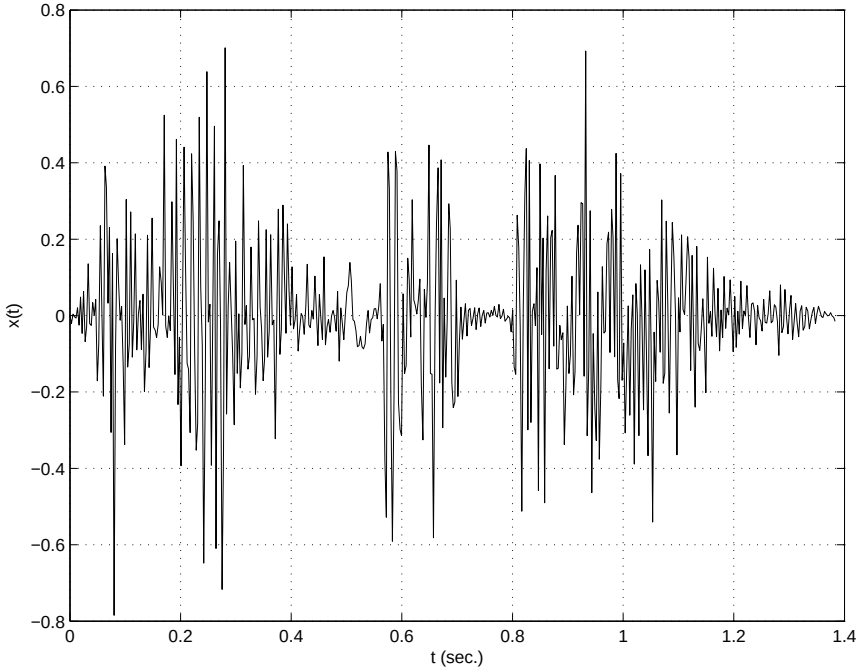


Figure 10.1: PLOTCSIG2.

10.2 LEVINSON2.M

Purpose

This function computes the model parameters, $[\alpha_1, \cdots, \alpha_p]^T$ and gain A for a pth order linear predictor of the form

$$H(z) = \frac{A}{1 - \sum_{k=1}^{p} \alpha_k z^{-k}} \tag{10.1}$$

such that the squared-prediction error is minimized. The implementation uses the Levinson recursion.

Input

s, signal
p, model order

Output

α, model parameter vector (length p)
A, gain
k, PARCOR coefficients

Algorithm

$\alpha_0^{(0)} = 0$
$E = r[0]$
for $i = 1$ to p
  $k_i = \frac{1}{E}\left\{r[i] - \sum_{j=1}^{i-1} \alpha_j^{(i-1)} r\,[i-j]\right\}$
  $\alpha_i^{(i)} = k_i$;
  for $j = 1$ to $i-1$
    $\alpha_j^{(i)} = \alpha_j^{(i-1)} - k_i \alpha_{i-j}^{(i-1)}$
  end
  $E \leftarrow (1 - k_i^2)E$
end
$\alpha = [\alpha_1^{(p)}, \cdots, \alpha_p^{(p)}]^T$
$A = \sqrt{E}$

Notes

1. The MATLAB Signal Processing Toolbox also has a Levinson function and is called with

   ```
   r = correlation(s,s,p,1);
   [alpha,E,k] = levinson(r,p);
   ```

   If you negate MATLAB's α and prepend 1.0 to the vector [see (10.1)], levinson and levinson2 produce the same LPC coefficients. If you negate MATLAB's k, levinson and levinson2 produce the same PARCORs. Finally, if you take the square root of MATLAB's prediction error output E, levinson and levinson2 produce the same gain value. MATLAB also has an LPC function which takes care of the above steps.

2. The autocorrelations, $r[0], ..., r[p]$ should be calculated using the CORRELATION.M tool with the 'biased' option.

3. The first step of the outer loop should be implemented with an inner product.

4. In the MSE update (last step of the outer loop), you may need to add a small value, ϵ to prevent a potential division by zero problem in the first step of the outer loop.

Reference

T. Quatieri, *Discrete-Time Speech Signal Processing*, Prentice-Hall: Upper Saddle River, N.J., 2001.

Example

```
>>randn('state',0);
>>p = exp(j*pi*[0.25;0.5;0.75]);
>>p = [p;conj(p)];  % spread some poles around
>>a = poly(p); % transfer function, H(z) = 1/A(z)
>>s = ar_synthesizer(a,100,1);
>>[alpha,A,k] = levinson2(s,6)

alpha =
  -0.01602780015730
  -0.98448479803672
   0.04711718658819
  -0.91629048420903
  -0.01749090304817
  -0.88364886161545

A =
   1.08354132916812

k =
  -0.01997688386198
  -0.66010686694396
   0.03873933044581
  -0.21150129563659
  -0.01518472567185
  -0.88364886161545
```

10.3 REQUANTIZE.M

Purpose

This function requantizes input samples to floats which are represented with B bits per sample.

Input

x, signal
B, desired bits per sample

Output

$\hat{\mathbf{x}}$, requantized signal

Algorithm

Scale input signal so that $-1.0 \leq x < +1.0$, i.e. $2.0 - \epsilon$ peak-to-peak range
Calculate quantization step size, $\Delta = 2.0/2^B$
Quantize, $\hat{x} = \lfloor (\frac{x}{\Delta}) \rfloor \times \Delta + \frac{\Delta}{2}$ (mid-tread) where $\lfloor\ \rfloor$ denotes the floor operation

Code

```
if (max(abs(x))>1.0)
   disp('Warning: signal amplitude exceeds +/- 1.0.  Will scale.')
   x = x ./ max(abs(x));
end;

% Do not allow any +1.0
i = find(x == 1.0);
x(i) = 1.0 - eps;

% Calculate step size = [1-(-1)]/[2^B]
delta = 2.0 / (2.0 ^ B);

% Quantize
x_hat = floor(x./delta).*delta + delta/2;  % mid-tread
```

Example

```
>>rand('state',0); % reset random number generator to initial state
>>x = [2*(rand(10,1)-0.5);1.0]; % create random samples between -1.0 and +1.0
>>x_hat = requantize(x,3); % 2^3 levels are -0.875, -0.625, -0.375 -0.125, 0.125, 0.625, 0.875
>> [x x_hat]

   0.90025857029435   0.87500000000000
  -0.53772297285142  -0.62500000000000
   0.21368516708357   0.12500000000000
  -0.02803506258140  -0.12500000000000
   0.78259793229780   0.87500000000000
   0.52419366605479   0.62500000000000
  -0.08706466966332  -0.12500000000000
```

```
-0.96299271350355  -0.87500000000000
 0.64281432859051   0.62500000000000
-0.11059327129361  -0.12500000000000
 1.00000000000000   0.87500000000000
```

10.4 PDF.M

Purpose

This function will plot the probability density function (pdf) os an input signal.

Input

x, vector of samples
option, optional argument with values as follows
0–Linear plot (default)
1–Semi-logarithm plot

Output

Plot of a pdf

Code

```
if ((nargin ~= 1) & (nargin ~= 2))
   error('Must have 1 or 2 input arguments.');
end;
if (nargin == 1)  % no options are specified, use defaults
   option = 0;
end;

x_max = max(abs(x));
delta = 2*x_max/50;  % assume 51 bins (change as desired)
bins = [-x_max:delta:x_max];  % histogram bin centers
n = hist(x,bins);  % compute histogram of x
A = sum(n)*delta;
n = n ./ A;  % normalize histogram/pdf to 1

if (option == 1);
   semilogy(bins,n);
   hold on
   semilogy(bins,n,'*');
   hold off;
else
   plot(bins,n);
   hold on
   plot(bins,n,'*');
   hold off;
end;
ylabel('p_x(x)');
xlabel('x');  % very pretty plots...
axis([-x_max x_max min(n) max(n)]);  % pretty plots...
grid;
```

Example

```
>>randn('state',0); % reset random number generator to initial state
```

```
>>x = randn(10^6,1);
>>pdf(x);
```

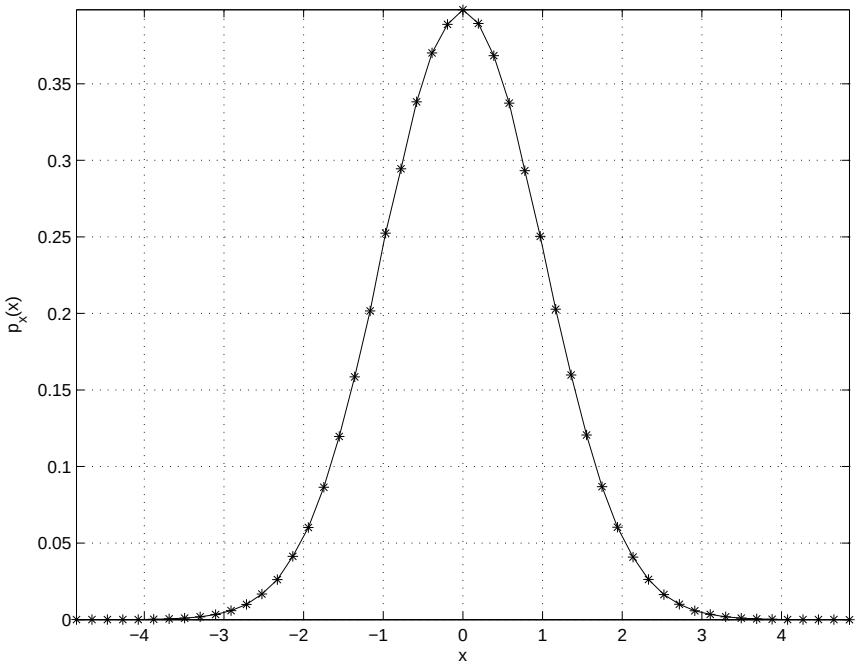


Figure 10.2: PDF.

10.5 UNI2MU.M

Purpose

This function transforms a uniform-quantized signal to a mu-law quantized signal. This has the effect of companding the signal.

Input

x, vector of samples (linear or uniform quantized)
μ, design variable

Output

y, vector of samples (μ-law quantized)

Algorithm

Input samples are transformed according to the rule

$$y[n] \quad = \quad \text{sgn}(x[n]) \frac{\ln(1+\mu|x[n]|)}{\ln(1+\mu)}. \tag{10.2}$$

Notes

Equation (10.2) assumes the input samples, $x[n]$ are amplitude normalized.

Example

```
>>rand('state',0);
>>x = 2*rand(10,1)-1;
>>y = uni2mu(x,100)

y =
   0.977470380865374
  -0.867406284956519
   0.673363665105204
  -0.289466620848227
   0.947479261493471
   0.861986883379261
  -0.492466524467013
  -0.991911583623390
   0.905438470046498
  -0.539495856515863
```

10.6 MU2UNI.M

Purpose

This function inverse transforms a mu-law quantized signal back to uniform quantized signal.

Input

y, vector of samples (μ-law quantized)
μ, design variable

Output

x, vector of samples (linear or uniform quantized)

Algorithm

Input samples are inverse transformed according to the rule

$$x[n] \quad = \quad \frac{\text{sgn}(y[n])}{\mu}\left[(1+\mu)^{|y[n]|}-1\right] \tag{10.3}$$

where (10.3) is the inverse transform of (10.2).

Notes

After the inverse transform, we normalize the amplitude of the resulting signal to 1.0.

Example

```
>>rand('state',0);
>>y = 2*rand(10,1)-1;
>>x = mu2uni(y,100)

x =
   0.627393300475795
  -0.109610594142161
   0.016809810708616
  -0.001381284382929
   0.360320507359548
   0.102370566130692
  -0.004945332388168
  -0.841425941922215
   0.184270339067681
  -0.006659595693813
```

10.7 CENTER_CLIPPER.M

Purpose

This function will center clip the input signal according to the rule

$$y = \begin{cases} +1.0, & x > C_L \\ 0.0, & C_L \leq x \leq C_L \\ -1.0, & x < C_L \end{cases} \tag{10.4}$$

Input

x, vector of samples
C_L, clipping level

Output

y, vector of center clipped samples

Example

```
>>n = [0:10]';
>>x = sin(2*pi*n/10);
>>y = center_clipper(x,sqrt(2)/2)

y =
     0
     0
     1
     1
     0
     0
     0
    -1
    -1
     0
     0
```

10.8 OLAISTFT.M

Purpose

This function synthesizes a real, time-domain signal from a short-time Fourier transform (STFT) using the overlap-add (OLA) method. As part of this method, we determine the window gain function for each sample so that perfect-reconstruction is attainable.

Input

X, STFT data
win, vector of window values
noverlap, number of overlapped samples

Output

x, synthesized signal

Algorithm

Determine window gains. Apply an inverse DFT to each column of **X** to obtain the windowed frames. Overlap and add the resulting windowed frames. Divide the signal by the window gains.

Code

```
[N,L] = size(X);  % N = FFT size, L = number of frames
nadvance = N-noverlap;

% overlap add the windows to determine the window gains
repeated_windows = win(:,ones(L,1)); % build a matrix whose columns are the window values
window_gains = overlapadd(repeated_windows',rectwin(N),nadvance);

windowed_frames = real(ifft(X)); % invert each column of the STFT to get windowed frames
x = overlapadd(windowed_frames',rectwin(N),nadvance); % overlap add the windowed frames
x = x./window_gains; % divide out the window gains
```

Notes

1. The above code requires the Voicebox toolbox.

2. Due to numerical issues, there may be a small imaginary part in **x**. In this case, simply return real(x).

3. The STFT is described in Section 6.7.

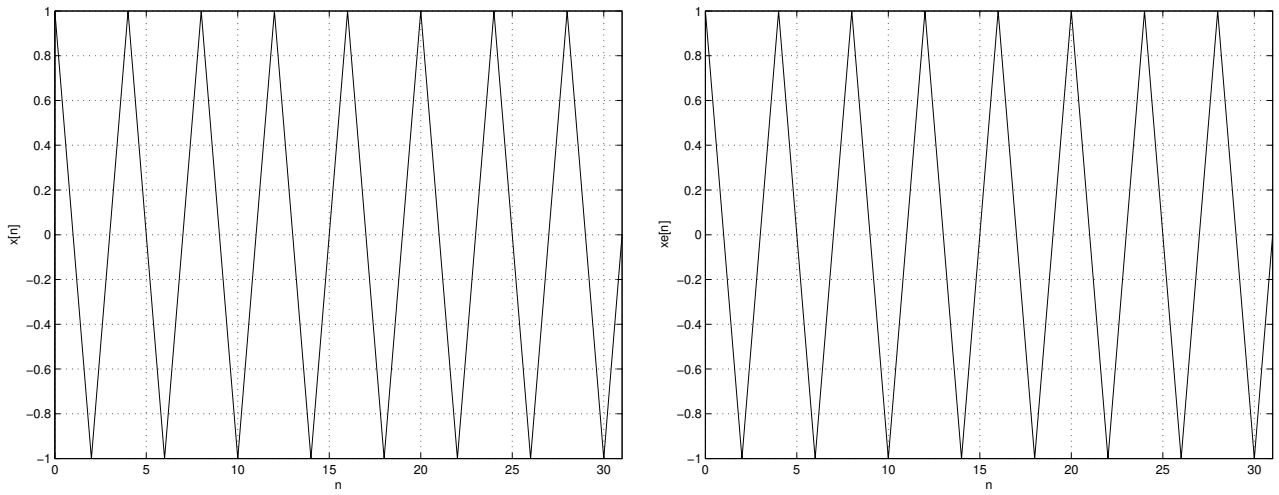
4. If the window is Bartlett (triangular) and the overlap is 50%, the window gains are all 1.0 as expected.

Reference

T. Quatieri, *Discrete-Time Speech Signal Processing*, Prentice-Hall: Upper Saddle River, N.J., 2001.

Example

```
>>N = 32;
>>window_length = N/4; % must be integer
>>noverlap = 3*window_length/4; % must be integer
>>win = hamming(window_length);
>>x(1) = x(1) + eps*i; % add a tiny imaginary part to force a two-sided STFT
>>X = spectrogram(x,win,noverlap); % use MATLAB to compute STFT
>>xe = olaistft(X,win,noverlap);
```



(a) (b)

Figure 10.3: (a) Signal used to generate STFT data and (b) signal estimate from OLA-based inverse STFT.

10.9 LSEISTFT.M

Purpose

This function synthesizes a real, time-domain signal from a short-time Fourier transform (STFT) using the least-squares estimate (LSE) method.

Input

X, STFT data computed with overlapped windows
w, window
L, window overlap (in samples)

Output

$\mathbf{x}_e$, estimate of inverse STFT of **X**

Algorithm

Each sample $x_e[n]$ is computed with

$$x_e[n] = \frac{\sum_{m=-\infty}^{\infty} w[mL-n]x[mL,n]}{\sum_{m=-\infty}^{\infty} w^2[mL-n]} \tag{10.5}$$

where $w[n]$ is the window value and $x[mL,n]$ corresponds to the inverse DFT of the spectrum $X[mL,k]$. This formula may be better understood by examining Figure 10.4. In this figure, we have three overlapping windows originally used for calculating the STFT, **X**. The inverse DFT of the individual windows, **x** are shown in the rectangular boxes. Consider recovery of the sample at time n. Since there is a 2/3rd window overlap, the inverse DFT corresponding to three window positions will contain the sample corresponding to time n as shown in the shaded boxes. Let $x_1,\ x_2,$ and x_3 correspond to the values at the shaded locations and $w_1,\ w_2,$ and w_3 be the window weights for these samples. Then according to (10.5),

$$x_e[n] = \frac{w_1x_1 + w_2x_2 + w_3x_3}{w_1^2 + w_2^2 + w_3^2}. \tag{10.6}$$

Equation (10.5) can be thought of as a weighted mean of different inverse DFTs corresponding to time n.

Notes

1. In order to simplify programming, the first and last window of samples do not have to be reconstructed.
2. Due to numerical issues, there may be a small imaginary part in $\mathbf{x}_e$. In this case, simply return real(xe).
3. The STFT is described in Section 6.7.

Reference

T. Quatieri, *Discrete-Time Speech Signal Processing*, Prentice-Hall: Upper Saddle River, N.J., 2001.

Example

```
>>N = 32;
>>x = cos(pi/2.*[0:N-1]');
>>x(1) = x(1) + j*10^(-10); % add small imag. to get complete spectrum
>>win_len = N/4;
>>overlap = win_len/2;
>>w = hamming(win_len);
>>X = spectrogram(x,w,overlap,win_len,2*pi); % generate STFT data from signal
>>xe = istft(X,w,overlap); % estimate signal from STFT data
```

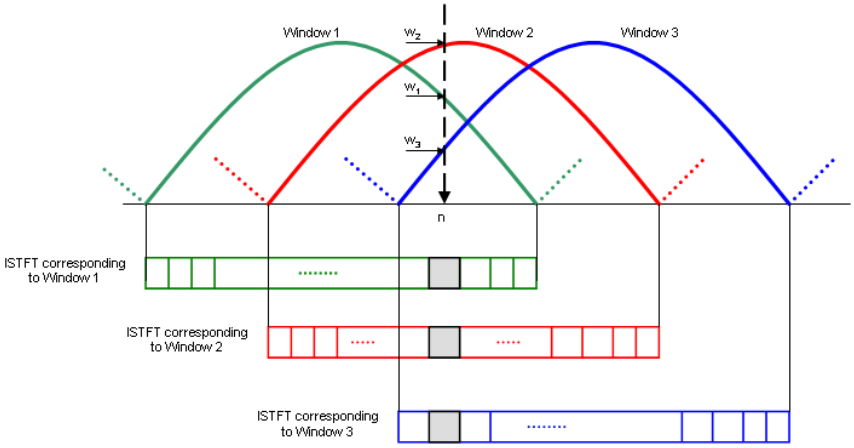
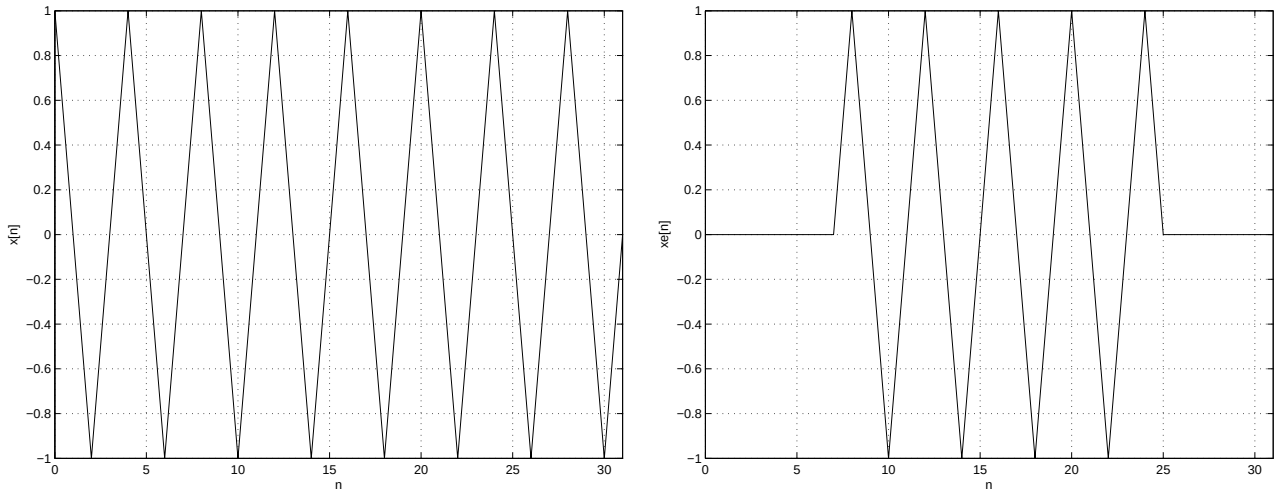


Figure 10.4: LSE-based inverse STFT.



(a) (b)

Figure 10.5: (a) Signal used to generate STFT data and (b) signal estimate from inverse STFT. For simplification, LSEISTFT.M does not reconstruct first and last window of samples.

10.10 OVERSPECSUB.M

Purpose

This function enhances noisy speech signals by implementing the oversubtraction form of spectral subtraction as proposed by Berouti.

Input

$\mathbf{Y}$, noisy signal STFT computed with overlapped windows
α_0, value of attenuation parameter at 0 dB SNR
β, spectral floor parameter
τ, smoothing parameter for noise power spectrum estimate

Output

$\hat{\mathbf{X}}$, enhanced signal STFT

Algorithm

1: Estimate the noise power spectrum, $S_d(\omega)$ using an average of $|Y(\omega)|^2$ from the first few windows of $\mathbf{Y}$ which are assumed to be noise only
2: Set the silence threshold in dB as $\theta = 10\log_{10}\left[1.5 \cdot \sum_{\omega} S_d(\omega)\right]$
3: **for** each window of $\mathbf{Y}$ **do**
4: Calculate the *a posteriori* SNR in dB

$$\gamma = 10\log_{10}\left[\sum_{\omega} |Y(\omega)|^2 / \sum_{\omega} S_d(\omega)\right] \tag{10.7}$$

5: Calculate the attenuation parameter, $\alpha = \alpha_0 - \frac{3}{20}\gamma$
6: For each frequency, ω compute the enhanced signal power spectrum

$$|\hat{X}(\omega)|^2 = \begin{cases} |Y(\omega)|^2 - \alpha S_d(\omega), & \text{if } |Y(\omega)|^2 > (\alpha+\beta)S_d(\omega) \\ \beta S_d(\omega), & \text{otherwise} \end{cases} \tag{10.8}$$

7: Compute the enhanced spectrum using $|\hat{X}(\omega)|^2$ and the noisy phase, $e^{j\phi_y} = Y(\omega)/|Y(\omega)|$

$$\hat{X}(\omega) = |\hat{X}(\omega)|e^{j\phi_y} \tag{10.9}$$

8: **if** $\gamma < \theta$ **then**
9: Update the estimate of the noise power spectrum

$$S_d(\omega) = \tau S_d(\omega) + (1-\tau)|Y(\omega)|^2 \tag{10.10}$$

10: **end if**
11: **end for**

Notes

1. Window length used in computing the input STFT is typically 20–40 ms.
2. It is suggested that $3 \leq \alpha_0 \leq 6$ and $\tau = 0.99$.
3. For high noise levels (SNR ≤ 0 dB), it is suggested $0.02 \leq \beta \leq 0.06$; for low levels of noise (SNR > 0) it is suggested that $0.005 \leq \beta \leq 0.02$.
4. If the noise is stationary, the update of the estimate of the noise power spectrum is not required.
5. The output STFT can be inverted using the `OLAISTFT.M` or `LSEISTFT.M` (preferred).

References

P. Loizou, *Speech Enhancement Theory and Practice*, CRC Press: Boca Raton, Fl., 2007.

T. Quatieri, *Discrete-Time Speech Signal Processing*, Prentice-Hall: Upper Saddle River, N.J., 2001.

10.11 WIENER.M

Purpose

This function enhances noisy speech signals by implementing the iterative version of the frequency-domain, generalized Wiener filter

$$H(\omega) \quad = \quad \left[\frac{S_x(\omega)}{S_x(\omega) + kS_d(\omega)}\right]^{1/2} \tag{10.11}$$

where $S_x(\omega)$ is the estimated power spectrum of the clean signal, $S_d(\omega)$ is the estimated power spectrum of the noise, k is the noise scale parameter, and a is the power exponent.

Input

$\mathbf{Y}$, noisy signal STFT computed with overlapped windows
k, noise scale parameter
τ, smoothing parameter for noise power spectrum estimate
num_iterations, number of algorithm iterations

Output

$\hat{\mathbf{X}}$, enhanced signal STFT (filter output)

Algorithm

1: Estimate the noise power spectrum, $S_d(\omega)$ using an average of $|Y(\omega)|^2$ from the first few windows of $\mathbf{Y}$ which are assumed to be noise only
2: Set the noise energy threshold, $\theta = 1.5 \cdot \sum_{\omega} S_d(\omega)$
3: Initialize the Wiener filter, $H(\omega)$ (see Notes below)
4: **for** $i = 1$ to *num_iterations* **do**
5: **for** each window of $\mathbf{Y}$ **do**
6: Compute the filter output

$$\hat{X}(\omega) \quad = \quad H(\omega)Y(\omega). \tag{10.12}$$

7: Compute a smoothed estimate

$$S_x \quad = \quad \tau S_x + (1-\tau)|\hat{X}(\omega)|^2. \tag{10.13}$$

8: **if** signal energy, $\sum_{\omega} |Y(\omega)|^2 < \theta$ **then**
9: Update the estimated noise power spectrum

$$S_d \quad = \quad \tau S_d + (1-\tau)|Y(\omega)|^2 \tag{10.14}$$

10: **end if**
11: Use S_x from (10.13) and S_d to update the Wiener filter using (10.11)
12: **end for**
13: **end for**

Notes

1. Window length used in computing the input STFT is typically 20–40 ms.
2. The Wiener filter can be initialized as $H_{\text{opt}}(\omega) = 1$ or random.
3. It is suggested that $0.5 \leq k \leq 20$ and $\tau = 0.99$.
4. Iterating through the algorithm 2-4 times is sufficient.
5. The output STFT can be inverted using the `OLAISTFT.M` or `LSEISTFT.M` (preferred).

Reference

T. Quatieri, *Discrete-Time Speech Signal Processing*, Prentice-Hall: Upper Saddle River, N.J., 2001.