Table of Contents

%%

linSysTutorial.m

David Heeger, Eero Simoncelli, and Patrick Teo 6/96.

Based on OBVIUS tutorial by David Heeger and Eero Simoncelli.

%%

```
% Discrete-time sequences are represented as a sequence of numbers
% f[n], for integer values of n.  There are several important, basic
% sequences.  Examples of these basic sequences will be plotted
% below.  We now plot several of the important sequences,
% from n=1 to n=32.

nRange=0:31;
```

```
% Impulse sequence, delta[n], is 1 only when n=0. Here we plot
% delta[n-16]:
figure(1), clf; cax = gca;
impulse = zeros(1,32);
impulse(17)=1;
plot(nRange,impulse, 'g--', nRange,impulse, 'o', 'MarkerEdgeColor' , 'g', 'MarkerF
set(cax, 'xLim', [0 31], 'yLim' , [-0.2 1.2])
figure(1)

% Step sequence, u[n], is 0 when n<0.  Here we plot u[n-16]:
step = zeros(1,32);
step(17:32)=ones(1,16);
figure(1), plot(nRange,step, 'g--', nRange,step, 'o', 'MarkerEdgeColor' , 'g', 'Ma
set(cax, 'xLim', [0 31], 'yLim' , [-0.2 1.2])
figure(1)

% Next, we plot a sinusoidal sequence with period 8:
amplitude = 1; phase_off = 0; period = 8; freq = 2*pi/period;
sinusoid = amplitude .* sin(freq .* nRange + phase_off);
figure(1), subplot(2, 1, 1)
plot(nRange,sinusoid, 'g--', nRange,sinusoid, 'o', 'MarkerEdgeColor' , 'g', 'Marke
set(gca, 'xLim', [0 31], 'yLim' , [-1.2 1.2])
figure(1)

% Notice that for discrete sinusoids, unlike continous sinusoids,
% adding 2*pi to the frequency gives the same sinusoid:
freq = 2*pi/period + 2*pi;
sinusoid_shift_2pi = amplitude .* sin(freq .* nRange + phase_off);
figure(1), subplot(2, 1, 2)
plot(nRange,sinusoid_shift_2pi , 'r--', nRange, sinusoid_shift_2pi , 'o', ...
  'MarkerEdgeColor' , 'r', 'MarkerFaceColor', 'r');
set(gca, 'xLim', [0 31], 'yLim' , [-1.2 1.2])
figure(1)

% The importance of this is that we need only consider frequencies in
% a frequency interval of length 2*pi such as -pi to pi.  Also notice
% that although continuous sinusoids with frequency w are periodic
% with period 2*pi/w, this is not necessarily true of discrete
% sinusoids.  For example, a discrete sinusoid with frequency w=1 is
% NOT periodic:
period = 2*pi; freq = 2*pi/period;
non_periodic_sinusoid = amplitude .* sin(freq .* nRange + phase_off);
figure(1), clf
plot(nRange,non_periodic_sinusoid,'r-',nRange,non_periodic_sinusoid,'go');
axis([0 31 -1.2 1.2]);

% Why isn't this sequence periodic?  Is it because we've plotted
% only 32 samples?  If we were to plot more samples, would it
% ever repeat?

% For a finite length sequence, we have an even more stringent
% requirement.  By a periodic finite length sequence, we mean
% circularly periodic.  When you go off the end you start back at
% the beginning.  To be periodic, the sequence length must be a
% multiple of the period.

% Altogether, there are only N distinguishable frequencies that
% are circularly periodic with period (sequence length) N.  These
% frequencies are:
%       2 pi k/N for k=0,1,...,N-1.
% In our examples, N=32, so the frequencies are:
%       0, pi/16, 2pi/16, 3pi/16,..., 31pi/16.

% This set of discrete (co-)sinusoids can also be indexed in
% another way:
%       2 pi k/N for k=-N/2,...,-1,0,1,...,N/2.
% In our examples, these periods are:
%       -16pi/16,...,-pi/16,0,pi/16,...,15pi/16.
% Take a look at some of these sinusoids and cosinusoids to see
% that these frequencies are all distinct.

% Are the sinusoids and cosinusoids with frequencies 0 and pi
% distinct?  How about with frequencies -pi and pi?

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Linear systems:

```
% A discrete time system is defined as a transformation or operator
% that maps an input sequence to an output sequence:
% f[x] -> g[x]  or g[x] = T{f[x]}

% Linear systems are defined by the principle of superposition.
% Superposition has two parts: additivity and homogeneity.

% Additivity:
% T{f1[x] + f2[x]} = T{f1[x]} + T{f2[x]}

% Homogeneity (scaling):
% T{a f[x]} = a T{f[x]}

% A linear system can be expressed as a matrix multiplication:
%                   g[x] = M f[x]
% where M is an m-by-n matrix, g[x] is a sequence of length m and
% f[x] is a sequence of length n.

% A time-invariant system is one for which a shift or delay of
% the input sequence causes a corresponding shift in the output
% sequence.  An example of a linear time-VARIANT system is
% subsampling.  We'll get to that later.

% For a linear time-invariant system, the rows of the M matrix
% are all shifted copies of one another.  Such a matrix is called
% a Toeplitz matrix.  The output of a linear time-invariant
% system can also be computed using convolution.  Convolution is
% equivalent to matrix-multiplication when using a Toeplitz
% matrix.
% First, let's create a Toeplitz matrix and display it as an
% image.

Tmatrix = zeros(32,32);
one_row = [-.004 -.031 -.047 .031 .102 .031 -.047 -.031 -.004];
for row = 1:32
  k = 1;
  for col = row-4:row+4
    Tmatrix(row,rem(col-1+32,32)+1) = one_row(k);
    k = k + 1;
  end
end
figure(1), clf
showIm(Tmatrix);

% Now, let's take our impulse signal and multiply it through the
% matrix:
impulse_response = Tmatrix * impulse';
plot(nRange,impulse_response,'r-',nRange,impulse_response,'go');
axis([0 31 -.06 .12]); figure(1)

% Matrix multiplication is an inefficient way of doing the above
% computation because most of the entries in the matrix are
% zeros.  Matlab provides facilities for efficient linear
% filtering.  A linear filter has only the interesting (non-zero)
% entries of the Toeplitz matrix.  Then the output is computed
% using convolution, shifting the filter over the input signal.
% The efficiency of convolution (over matrix multiplication) will
% be critical when we get to 2D linear transforms on images, and
% 3D (space-time) transforms on image sequences.  The matrices
% would be huge and very sparse (lots of zeros).  Let's do the
% above transform again, using convolution.
filter = [-.004 -.031 -.047 .031 .102 .031 -.047 -.031 -.004];
new_impulse_response = cconv2(impulse, filter);
plot(nRange,new_impulse_response,'r-',nRange,new_impulse_response,'go');
axis([0 31 -.06 .12]); figure(1);

% A linear time-invariant system is completely characterized by
% its impulse response, that is, its response to an impulse
% input.  The response to an impulse is the corresponding column
% in the Toeplitz matrix.  Given the impulse response, we can
% compute the response to any input.  Any input can be expressed
% as the sum of a bunch of scaled impulses.  Since the system is
% linear, the output is the sum of a bunch of scaled copies of
% the impulse response.  Example:

% Sum of two impulses:
sa

% Filter responses to individual impulses:
impulse_response_10 = cconv2(impulse_10,filter);
plot(nRange,impulse_response_10,'g-',nRange,impulse_response_10,'go');
axis([0 31 -.06 .12]); figure(1);
```

```
impulse_response_15 = cconv2(impulse_15,filter);
plot(nRange,impulse_response_15,'g-',nRange,impulse_response_15,'go');
axis([0 31 -.06 .12]); figure(1);

% Filter response to sum of impulses:
signal_response = cconv2(signal,filter);
plot(nRange,signal_response,'r-',nRange,signal_response,'ro');
axis([0 31 -.75 1.5]); figure(1);

% Sum of filter responses to individual impulses:
sum_of_impulse_responses = ...
    5.*impulse_response_10 + 13.*impulse_response_15;
plot(nRange,sum_of_impulse_responses,'r-',nRange,sum_of_impulse_responses,'go');
axis([0 31 -.75 1.5]); figure(1);

% The difference (or mean-squared error) between them should
% return 0:
mean((signal_response-sum_of_impulse_responses).^2)

%%%%%%%%%%%%%%%%%%%%

% As another example, consider the filter response to the step:
step_response = cconv2(step,filter);
plot(nRange,step_response,'r-',nRange,step_response,'go');
axis([0 31 -.1 .1]); figure(1);

% Represent filter to response to step as sum of filter responses
% to impulses:
sum_of_impulse_responses = zeros(1,32);
for n = 17:32
  impulse_n = zeros(1,32);
  impulse_n(n) = 1;
  sum_of_impulse_responses = sum_of_impulse_responses + ...
      cconv2(impulse_n,filter);
end
plot(nRange,sum_of_impulse_responses,'r-',nRange,sum_of_impulse_responses,'go');
axis([0 31 -.1 .1]); figure(1);

% This should return 0:
mean((step_response-sum_of_impulse_responses).^2)

% Note that cconv2 performs circular convolution.  Circular
% convolution wraps around from the end the sequence back to the
% beginning again, as if the sequence was a full period of a
% longer periodic sequence.  The consequence of circular
% convolution is evident in the step-response.  Obviously, there
% are many alternative ways to handle the endpoints of a
% sequence.  For example, one way is to repeat the end sample
% values (at both ends) rather than wrapping.  The ISEP toolbox
% includes convolution routines that offer a variety of edge
% handling options:

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Properties of linear time-invariant systems:

```
% Convolution is commutative (i.e. the order of two consecutive
% convolution operations is irrelavant):

filter1 = [-.004 -.031 -.047 .031 .102 .031 -.047 -.031 -.004];
filter2 = [1 4 6 4 1] ./ 16.0;

response_1_2 = cconv2(cconv2(step,filter1),filter2);
plot(nRange,response_1_2,'r-',nRange,response_1_2,'go');
axis([0 31 -.06 .06]); figure(1);

response_2_1 = cconv2(cconv2(step,filter2),filter1);
plot(nRange,response_2_1,'r-',nRange,response_2_1,'go');
axis([0 31 -.06 .06]); figure(1);

% Should be zero:
mean((response_1_2-response_2_1).^2)

% Even though convolution (linear shift-invariant linear systems)
% commute, not all linear systems commute.  For example, matrix
% multiplication is not, in general, commutative.

% Convolution also follows the distributive property (i.e., the
% sum of convolutions with two filters equals the convolution
% with the sum of the filters).

filter1 = [-.004 -.031 -.047 .031 .102 .031 -.047 -.031 -.004];
filter2 = [0 0 1 4 6 4 1 0 0] ./ 16;
response1 = cconv2(step,filter1);
response2 = cconv2(step,filter2);
sum_of_responses = response1 + response2;
plot(nRange,sum_of_responses,'r-',nRange,sum_of_responses,'go');
axis([0 31 -.2 1.2]); figure(1);

sum_of_filters = filter1 + filter2;
response_to_sum_of_filters = cconv2(step,sum_of_filters);
plot(nRange,response_to_sum_of_filters,'r-',nRange,response_to_sum_of_filters,'go'
axis([0 31 -.2 1.2]); figure(1);

% Should be zero:
mean((response_to_sum_of_filters - sum_of_responses).^2)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Invertible linear systems:

```
% Here's a simple example of a linear filter that shifts the
% sequence by 2 samples.  This operation can, of course, be
% inverted by shifting in the other direction.

shift_filter = [0 0 0 0 1];
```

```
unshift_filter = [1 0 0 0 0];

shifted_sinusoid = cconv2(sinusoid,shift_filter);
plot(nRange,shifted_sinusoid,'r-',nRange,shifted_sinusoid,'go');
axis([0 31 -1.2 1.2]); figure(1);

unshifted_sinusoid = cconv2(shifted_sinusoid,unshift_filter);
plot(nRange,unshifted_sinusoid,'r-',nRange,unshifted_sinusoid,'go');
axis([0 31 -1.2 1.2]); figure(1);

% Another way to think of inverting a linear transform is in
% terms of inverting the corresponding transform matrix.  The
% matrix for shifting operation looks like this:

Smatrix = zeros(32,32);
one_row = [1 0 0 0 0];
for row = 1:32
  k = 1;
  for col = row-2:row+2
    Smatrix(row,rem(col-1+32,32)+1) = one_row(k);
    k = k + 1;
  end
end
clf; showIm(Smatrix); figure(1);

% Now, lets' recompute the shifted sequence, using the matrix
% (instead of convolution):
new_shifted_sequence = Smatrix * sinusoid';
plot(nRange,new_shifted_sequence,'r-',nRange,new_shifted_sequence,'go');
axis([0 31 -1.2 1.2]); figure(1);

% To invert the transform, we just invert the matrix:
Smatrix_inv = inv(Smatrix);
clf; showIm(Smatrix_inv); figure(1);

% And, recompute the unshifted sequence:
new_unshifted_sequence = Smatrix_inv * new_shifted_sequence;
plot(nRange,new_unshifted_sequence,'r-',nRange,new_unshifted_sequence,'go');
axis([0 31 -1.2 1.2]); figure(1);

%%%%%%%%%%%%%%%%%%%%

% Here's another example of inverting a linear transform.  In
% this example, we split a signal into two bands, a high
% frequency (or "highpass") band and a low frequency (or
% "lowpass") band.  Each band has the same number of samples as
% the original sequence, so the entire transform has twice as
% many samples as the original.  The lowpass and highpass filters
% are carefully chosen so that summing the low and high bands
% reconstructs the original signal.

lo_filter = [.008 .031 .094 .219 .297 .219 .094 .031 .008];
plot([-4:1:4], lo_filter);
```

```
hi_filter = [-.008 -.031 -.094 -.219 .703 -.219 -.094 -.031 -.008];
plot([-4:1:4], hi_filter);

lo_impulse_response = cconv2(impulse,lo_filter);
plot(nRange,lo_impulse_response,'r-',nRange,lo_impulse_response,'go');
axis([0 31 -.02 .32]); figure(1);

hi_impulse_response = cconv2(impulse,hi_filter);
plot(nRange,hi_impulse_response,'r-',nRange,hi_impulse_response,'go');
axis([0 31 -.25 .75]); figure(1);

reconstruct_impulse = lo_impulse_response + hi_impulse_response;
plot(nRange,reconstruct_impulse,'r-',nRange,reconstruct_impulse,'go');
axis([0 31 -.2 1.2]); figure(1);

% Should be zero:
mean((reconstruct_impulse-impulse).^2)

% Let's compute the filter responses to a sinusoid:
lo_sin_response = cconv2(sinusoid,lo_filter);
plot(nRange,lo_sin_response,'r-',nRange,lo_sin_response,'go');
axis([0 31 -.65 .65]); figure(1);

hi_sin_response = cconv2(sinusoid,hi_filter);
plot(nRange,hi_sin_response,'r-',nRange,hi_sin_response,'go');
axis([0 31 -.65 .65]); figure(1);

reconstruct_sin = lo_sin_response + hi_sin_response;
plot(nRange,reconstruct_sin,'r-',nRange,reconstruct_sin,'go');
axis([0 31 -1.2 1.2]); figure(1);

% Should be zero:
mean((reconstruct_sin-sinusoid).^2)

%%%%%%%%%%%%%%%%%%%%

% Now, let's see what the matrix looks like for this transform:

LHmatrix = zeros(64,32);
lo_row = [.008 .031 .094 .219 .297 .219 .094 .031 .008];
hi_row = [-.008 -.031 -.094 -.219 .703 -.219 -.094 -.031 -.008];
for row = 1:32
  k = 1;
  for col = row-4:row+4
    LHmatrix(row,rem(col-1+32,32)+1) = lo_row(k);
    LHmatrix(row+32,rem(col-1+32,32)+1) = hi_row(k);
    k = k + 1;
  end
end
clf; showIm(LHmatrix); figure(1);

% The top half of the rectangular matrix represents the shifted
% filter coefficients for the low-pass filter.  The bottom half
% represents those of the hi-pass filter.
% To invert the transform, we can use the pseudo-inverse,
% (Mt M)^(-1) Mt, where M is the matrix, Mt is the matrix
% transpose, and the -1 indicates matrix inverse.

LHmatrix_inv = inv(LHmatrix' * LHmatrix) * LHmatrix';
clf; showIm(LHmatrix_inv); figure(1);

% Let's check that this really is the inverse
% (should be identity):
clf; showIm(LHmatrix_inv * LHmatrix); figure(1);

% Now, recompute the transform (the lo-pass and hi-pass transform
% coefficients are displayed in one double-length sequence, next
% to each other):
impulse_transform = LHmatrix * impulse';
plot([1:64],impulse_transform,'r-');
axis([1 64 -.25 .75]); figure(1);

% Now, invert the transform:
new_reconstruct_impulse = LHmatrix_inv * impulse_transform;
plot(nRange,new_reconstruct_impulse,'r-',nRange,new_reconstruct_impulse,'go');
axis([0 31 -.2 1.2]); figure(1);

% Should be zero:
mean((impulse - new_reconstruct_impulse').^2)

% The inverse that we just used is different from just adding
% together the coefficients of the two bands (used above).  There
% is more than one way to invert an overcomplete transform.
% (By "overcomplete", we mean a representation that has more
% parameters than the signal itself, i.e., is a redundant code.)
% Let's construct the matrix that corresponds to adding the
% coefficients from the two bands, and make sure that it also
% inverts the transform:

new_LHmatrix_inv = [diag(ones(1,32)) diag(ones(1,32))];
clf; showIm(new_LHmatrix_inv);  figure(1);

% Check that this is also an inverse:
clf; showIm(new_LHmatrix_inv * LHmatrix);  figure(1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Sinusoidal sequences:

```
% Sinusoidal and cosinusoidal sequences play a particularly important
% role in representing signals, because complex exponential sequences
% (including sines and cosines) are the "eigenfunctions" of
% finite-dimensional linear time-invariant systems.  That is, they are
% the functions which, when put through the system, are left unscathed
% but for a scalar multiplication (which, for a complex scalar, results
% in both an amplification and a phase shift). That is, a
% sinusoidal sequence convolved with a linear filter gives another
% sinusoidal sequence of the same frequency.  Only the phase and
% amplitude of the output sinusoid will be different.

plot(nRange,sinusoid,'r-',nRange,sinusoid,'go');
axis([0 31 -1.2 1.2]);
hold on; figure(1);
% Filter response to a sinusoid:
filter = [-.004 -.031 -.047 .031 .102 .031 -.047 -.031 -.004];
result = cconv2(sinusoid,filter);
plot(nRange,result,'r-',nRange,result,'go');
axis([0 31 -1.2 1.2]);

hold on; figure(1);

% That filter changes only the amplitude, not the phase.
% Here's one that also changes the phase (via shift/delay):
shift_filter = [0 0 -.004 -.031 -.047 .031 .102 .031 -.047 -.031 -.004];
result = cconv2(sinusoid,shift_filter);
plot(nRange,result,'r-',nRange,result,'go');
axis([0 31 -1.2 1.2]); figure(1);


% Notice that for discrete sinusoids, unlike continous sinusoids,
% adding 2*pi to the frequency gives the same sinusoid:
freq = 2*pi/period + 2*pi;
sinusoid_shift_2pi = amplitude .* sin(freq .* nRange + phase_off);
subplot(2,1,2); lplot(sinusoid_shift_2pi); figure(1);

% The importance of this is that we need only consider frequencies
% in a frequency interval of length 2*pi such as -pi to pi.

% Let's try some really weird (in fact random) filters, just to
% demonstrate that no matter what you use, you still get a
% sinusoid of the same frequency:
results = zeros(10,32);
for index = 1:10
  filter = rand(1,5);
  results(index,:) = cconv2(sinusoid,filter);
end

% Display it:
min_res = min(min(results)); max_res = max(max(results)); figure(1);
for index = 1:10
  plot(nRange,results(index,:),'r-',nRange,results(index,:),'go');
  axis([0 31 min_res max_res]);
  pause(0.5);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Fourier Series Representation:

```
% Any signal can be expressed as a (weighted) linear sum of
% impulses.  Likewise, a signal can be expressed as a (weighted)
% linear sum of sines and cosines.  Example, make a gaussian as a
% sum of cosines:

% Gaussian:
clf;
gaussian = exp(-((nRange-16).^2)./(2*4^2));
plot(nRange,gaussian,'r-',nRange,gaussian,'go');
axis([0 31 -.2 1.2]); figure(1);

% Fourier series approximation of Gaussian:
gaussian_series = ...
    (0.333 * ones(1,32)) + ...
    (0.470 * cos(2*pi*1/32*(nRange-16))) + ...
    (0.166 * cos(2*pi*2/32*(nRange-16))) + ...
    (0.029 * cos(2*pi*3/32*(nRange-16))) + ...
    (0.0025 * cos(2*pi*4/32*(nRange-16)));
plot(nRange,gaussian_series,'r-',nRange,gaussian_series,'go');
axis([0 31 -.2 1.2]); figure(1);

% Should be small:
mean((gaussian - gaussian_series).^2)

% The weights in the summation (.470, .166, 029, and .0025) are
% called the Fourier coefficients.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Fourier Transform:

```
% The Fourier transform is a particular linear transform that is
% used to compute the Fourier coefficients of a signal.  The
% transform matrix looks like this:

Fourier_mat = zeros(64,32);
for k = 0:31
  Fourier_mat(k+1,:) = cos(2*pi*k/32*nRange)/sqrt(32);
  Fourier_mat(k+32+1,:) = sin(-2*pi*k/32*nRange)/sqrt(32);
end
clf; showIm(Fourier_mat); figure(1);

% (Note, also discussed later, the scale factor of sqrt(32)
% makes all rows have length 1.)
% Each row is a sine or cosine.  Rows 2 and 6, for example:
plot(nRange,Fourier_mat(2,:),'r-',nRange,Fourier_mat(2,:),'go');
axis([0 31 -.2 .2]); figure(1);

plot(nRange,Fourier_mat(6,:),'r-',nRange,Fourier_mat(6,:),'go');
axis([0 31 -.2 .2]); figure(1);
```

```
% The top half of the matrix contains cosines of various
% frequencies, and the bottom half contains sines of various
% frequencies.  The very top row is a constant that pulls out the
% average (dc) component of a signal.  The transform is
% self-inverting, that is, multiplying the matrix by its
% transpose gives the identity transform.

Fourier_inv_mat = Fourier_mat';
clf; showIm(Fourier_inv_mat); figure(1);

clf; showIm(Fourier_inv_mat * Fourier_mat);  figure(1);

% Let's look at the rows of the inverse (transpose) matrix.  The
% first half of each row is a cosine, and the second half of each
% row is a sine.  These sines and cosines are the same as the
% rows of the the Fourier matrix.

plot(nRange,Fourier_inv_mat(2,[1:32]),'r-');
axis([0 31 -.2 .2]); figure(1);

plot(nRange,Fourier_inv_mat(2,[33:64]),'r-');
axis([0 31 -.2 .2]); figure(1);


% Let's take the Fourier transform of a cosinusoid.

num_cycles = 4;
cosinusoid = cos(2*pi*num_cycles/32*nRange);
plot(nRange,cosinusoid,'r-',nRange,cosinusoid,'go');
axis([0 31 -1.2 1.2]); figure(1);

ft_cosinusoid = Fourier_mat * cosinusoid';
plot(0:63,ft_cosinusoid,'r-');
axis([0 63 -.2 3]); figure(1);

% We get a pair of impulses in the transform.  One of the
% impulses corresponds to the frequency of the signal (4 cycles
% per image) at position 4 in the transform.  Why is there a
% second impulse?  The Fourier transform is really set up to
% analyze complex signals.  For real signals, the transform has
% certain symmetry properties.  We will go into those in more
% detail below.

% Try a different frequency:
num_cycles = 8;
cos_8 = cos(2*pi*num_cycles/32*nRange);
plot(nRange,cos_8,'r-',nRange,cos_8,'go');
axis([0 31 -1.2 1.2]); figure(1);

ft_cos_8 = Fourier_mat * cos_8';
plot(0:63,ft_cos_8,'r-');
axis([0 63 -.2 3]); figure(1);
```

```
% Again, the location of the impulses in the transform
% corresponds to the frequency of the signal.  For example, a
% frequency of 8 cycles per image gives an impulse at positions 8
% and 32-8=24 in the transform domain.

% For a sinusoid, we get impulses in the second half (positions >
% 32) of the output because the sinusoids are in the bottom half
% of the system matrix.

transform = Fourier_mat * sinusoid';
plot(0:63,transform,'r-');
axis([0 63 -3 3]); figure(1);

% The Fourier transform is inverted by using the transpose of the
% system matrix:

invert_transform = Fourier_mat' * transform;
plot(nRange,invert_transform,'r-',nRange,invert_transform,'go');
axis([0 31 -1.2 1.2]); figure(1);

% Should be zero:
mean((invert_transform' - sinusoid).^2)

%%%%%%%%%%%%%%%%%%%%

% The FFT algorithm is an efficient way of computing the Fourier
% transform, without bothering with the matrix multiplication.

ft_sinusoid = fft(sinusoid);

% Cosinusoidal (real) part:
realpart=real(ft_sinusoid);
plot(nRange,realpart,'r-',nRange,realpart,'go');
axis([0 31 -18 18]); figure(1);

% Sinusoidal (imaginary) part:
imagpart=imag(ft_sinusoid);
plot(nRange,imagpart,'r-',nRange,imagpart,'go');
axis([0 31 -18 18]); figure(1);

% FFT returns complex numbers. The imaginary part contains the
% sine components and the real-part contains the cosine
% components.  The first sample corresponds to the "DC" or
% constant coefficient.

% Often, people represent Fourier coefficients in terms of
% magnitude and phase, rather than real and imaginary:
mag_sin = abs(ft_sinusoid);
plot(nRange,mag_sin,'r-',nRange,mag_sin,'go');
axis([0 31 -0.2 18]); figure(1);

% Here's a whole series of Fourier transforms for different
% frequency sinusoids.
for freq = 0:31
```

```
  sin_freq = sin(2*pi*freq/32*nRange);
  imag_ft_sin_freq = imag(fft(sin_freq));
  plot(nRange,imag_ft_sin_freq,'r-',nRange,imag_ft_sin_freq,'go');
  axis([0 31 -18 18]); figure(1);
  pause(0.25);
end

% Now, let's compute Fourier transforms of some other signals.

% Fourier transform of a constant function:
constant = ones(32);
mag_constant = abs(fft(constant));
plot(nRange,mag_constant,'r-',nRange,mag_constant,'go');
axis([0 31 0 32]); figure(1);

% fftshift shifts the frequency axis to put the DC component
% in the middle:
mag_constant = abs(fftshift(fft(constant)));
plot(-16:15,mag_constant,'r-',-16:15,mag_constant,'go');
axis([-16 15 0 32]); figure(1);

% Fourier transform of a Gaussian function: the imaginary
% part is zero and the real part is itself a Gaussian. Again,
% fftshift puts the center of the Gaussian at the place
% Matlab considers to be the origin
gaussian = exp(-((nRange-16)./6).^2);
plot(-16:15,gaussian,'r-',-16:15,gaussian,'go');
axis([-16 15 -0.2 1.2]); figure(1);

ft_gaussian = fftshift(fft(fftshift(gaussian)));
imag_ft_gaussian = imag(ft_gaussian);
real_ft_gaussian = real(ft_gaussian);
plot(-16:15,real_ft_gaussian,'r-',-16:15,real_ft_gaussian,'go');
axis([-16 15 -12 12]); figure(1);

plot(-16:15,imag_ft_gaussian,'r-',-16:15,imag_ft_gaussian,'go');
axis([-16 15 -12 12]); figure(1);

% Making the Gaussian smaller in one domain makes it larger in
% the other domain:
little_gauss = exp(-(((nRange-16).^2)./2));
plot(-16:15,little_gauss,'r-',-16:15,little_gauss,'go');
axis([-16 15 -0.2 1.2]); figure(1);

ft_little_gauss = fftshift(fft(fftshift(little_gauss)));
real_ft_little_gauss = real(ft_little_gauss);
plot(-16:15,real_ft_little_gauss,'r-',-16:15,real_ft_little_gauss,'go');
axis([-16 15 -.2 3]); figure(1);

% This is an example of what is known as the uncertainty
% principle.  When a signal is more localized in one domain, it
% is less well localized in the other domain.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Symmetry properties of Fourier Transform:

```
% For any real-valued, antisymmetric (odd) function, in which
% f(x) = -f(-x), the real part of the FT is zero, and the
% imaginary part of the FT is antisymmetric (odd).  For any
% real-valued, symmetric (even) function, in which f(x) = f(-x),
% the imaginary part of the FT is zero and the real part is
% symmetric (even).

random_signal = 0.5-rand(1,64);

% Decompose random_signal into even and odd components:
%   even_signal(x) = 0.5*(random_signal(x)+random_signal(-x))
%   odd_signal(x)  = 0.5*(random_signal(x)-random_signal(-x))
%
% One can easily verify that
%   random_signal(x) = even_signal(x)+odd_signal(x).
%
% Note that the origin (i.e. x=0) is at location 33
% (i.e. random_signal(33)).

even_signal = zeros(1,64);
even_signal(1) = random_signal(1);
for index = -31:31
  even_signal(index+32+1) = ...
      0.5*(random_signal(index+32+1)+random_signal(-index+32+1));
end
plot(-32:31,even_signal,'r-',-32:31,even_signal,'go');
axis([-32 31 -0.5 0.5]); figure(1);

odd_signal = zeros(1,64);
odd_signal(1) = random_signal(1);
for index = -31:31
  odd_signal(index+32+1) = ...
      0.5*(random_signal(index+32+1)-random_signal(-index+32+1));
end
plot(-32:31,odd_signal,'r-',-32:31,odd_signal,'go');
axis([-32 31 -0.5 0.5]); figure(1);

% Real part of Fourier transform of an even signal is
% even-symmetric; imaginary part is zero.
ft_even_signal = fftshift(fft(fftshift(even_signal)));
real_ft_even = real(ft_even_signal);
plot(-32:31,real_ft_even,'r-',-32:31,real_ft_even,'go');
axis([-32 31 -8 8]); figure(1);

imag_ft_even = imag(ft_even_signal);
plot(-32:31,imag_ft_even,'r-',-32:31,imag_ft_even,'go');
axis([-32 31 -8 8]); figure(1);

% Imaginary part of Fourier transform of an odd signal
% is odd-symmetric; real part is zero.
% *** busted (real part isn't quite zero)
```

```
ft_odd_signal = fftshift(fft(fftshift(odd_signal)));
real_ft_odd = real(ft_odd_signal);
plot(-32:31,real_ft_odd,'r-',-32:31,real_ft_odd,'go');
axis([-32 31 -8 8]); figure(1);

imag_ft_odd = imag(ft_odd_signal);
plot(-32:31,imag_ft_odd,'r-',-32:31,imag_ft_odd,'go');
axis([-32 31 -8 8]); figure(1);

% For any real-valued signal, the real part of the FT is
% even-symmetric and the imaginary part of the FT is
% odd-symmetric.
ft_random_signal = fftshift(fft(random_signal));
real_ft = real(ft_random_signal);
plot(-32:31,real_ft,'r-',-32:31,real_ft,'go');
axis([-32 31 -5 5]); figure(1);

imag_ft = imag(ft_random_signal);
plot(-32:31,imag_ft,'r-',-32:31,imag_ft,'go');
axis([-32 31 -5 5]); figure(1);

% Taken together, these symmetry properties mean that there is
% quite a lot of redundancy in the FT of a real signal.  A simple
% way to count the amount of redundancy is to compare the number
% of samples.  Take a real-valued input signal with 64 samples.
% Computing its fft gives a total of 128 samples (half in the
% real part and half in the imaginary part), a factor of 2
% redundant.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Parseval's Theorem: sum of squared values over space domain

equals sum of squared values over frequency domain.

```
WARNING: You should compile the MEX version of "range2.c",
         found in the MEX subdirectory of matlabPyrTools, and put it in your matla

Undefined function or variable 'sa'.

Error in linSysTutorial (line 170)
sa
```

These should be the equal.

```
sum(sum(gaussian.^2))
sum(sum(abs(ft_gaussian).^2))/32
```

Ignore the factor of 32 for now. It is explained below.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Circular Shifting:

```
% If we shift the signal as if it were periodic (i.e., translate
% the signal, wrapping around at the edges), this does not affect
% the Fourier transform magnitude:

fft_gauss = fftshift(fft(ifftshift(gaussian)));
fft_shift_gauss = fftshift(fft(ifftshift(shift(gaussian,[3,0]))));

% Magnitudes are the same:
mag_fft_gauss = abs(fft_gauss);
plot(-16:15,mag_fft_gauss,'r-',-16:15,mag_fft_gauss,'go');
axis([-16 15 -.2 12]); figure(1);

mag_fft_shift_gauss = abs(fft_shift_gauss);
plot(-16:15,mag_fft_shift_gauss,'r-',-16:15,mag_fft_shift_gauss,'go');
axis([-16 15 -.2 12]); figure(1);

% Should be zero:
mean((mag_fft_shift_gauss - mag_fft_gauss).^2)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Differentiation:

```
% Taking a derivative of a signal in time is the same as multiplying
% by an imaginary ramp in frequency. In particular,
%                   Fourier{d/dx f(x)} = -i w Fourier{f(x)}
% where i = sqrt(-1) and w is normalized frequency.
% For an intuition for the derivative property, recall that the
% d/dx[cos(wx)] = -w sin(wx).

% For example, let's consider a Gaussian and the first derivative
% of a Gaussian.

gaussian = exp(-(((nRange-16).^2)./(2*4^2)));
plot(-16:15,gaussian,'r-',-16:15,gaussian,'go');
axis([-16 15 -.2 1.2]); figure(1);

gaussian_deriv = -2/(2*4^2).*(nRange-16).*gaussian;
plot(-16:15,gaussian_deriv,'r-',-16:15,gaussian_deriv,'go');
axis([-16 15 -.2 .2]); figure(1);

ft_gaussian = fftshift(fft(fftshift(gaussian)));
real_ft_gaussian = real(ft_gaussian);
plot(-16:15,real_ft_gaussian,'r-',-16:15,real_ft_gaussian,'go');
```

```
axis([-16 15 -.2 12]); figure(1);

ft_gaussian_deriv = fftshift(fft(fftshift(gaussian_deriv)));
imag_ft_gaussian_deriv = imag(ft_gaussian_deriv);
plot(-16:15,imag_ft_gaussian_deriv,'r-',-16:15,imag_ft_gaussian_deriv,'go');
axis([-16 15 -2 2]); figure(1);

% ramp := i w :
ramp = 2*pi/32*sqrt(-1).*(nRange-16);

% Compute the Fourier transform of the derivative of a gaussian
% by multiplying the Fourier transform of a gaussian by the ramp.
ft_gaussian_mul_ramp = ramp .* ft_gaussian;
imag_ft_gaussian_mul_ramp = imag(ft_gaussian_mul_ramp);
plot(-16:15,imag_ft_gaussian_mul_ramp,'r-',-16:15,imag_ft_gaussian_mul_ramp,'go');
axis([-16 15 -2 2]); figure(1);

% Should be zero:
mean((ft_gaussian_deriv - ft_gaussian_mul_ramp).^2)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Modulation Theorem:

```
% Multiplication in time domain is the same as convolution in the
% frequency domain, up to a known scale factor. For example, a
% Gabor function is a sinusoid multiplied by a Gaussian window.
% Thus, the FT of a Gabor is the convolution of the FT of a
% Gaussian with the FT of a sinusoid.  This is an easy way to
% gain an intuition for the filtering properties of a Gabor
% filter.

gabor = gaussian .* sinusoid;
plot(-16:15,gabor,'r-',-16:15,gabor,'go');
axis([-16 15 -1 1]); figure(1);

ft_gabor = fftshift(fft(ifftshift(gabor)));
imag_ft_gabor = imag(ft_gabor);
plot(-16:15,imag_ft_gabor,'r-',-16:15,imag_ft_gabor,'go');
axis([-16 15 -6 6]); figure(1);

ft_sinusoid = fftshift(fft(ifftshift(sinusoid)));
imag_ft_sinusoid = imag(ft_sinusoid);
plot(-16:15,imag_ft_sinusoid,'r-',-16:15,imag_ft_sinusoid,'go');
axis([-16 15 -18 18]); figure(1);

ft_gaussian = fftshift(fft(ifftshift(gaussian)));
real_ft_gaussian = real(ft_gaussian);
plot(-16:15,real_ft_gaussian,'r-',-16:15,real_ft_gaussian,'go');
axis([-16 15 -1 12]); figure(1);

imag_conv_of_fts = cconv2(real(ft_gaussian),imag(ft_sinusoid))/32;
plot(-16:15,imag_conv_of_fts,'r-',-16:15,imag_conv_of_fts,'go');
```

```
axis([-16 15 -6 6]); figure(1);

% Should be zero:
mean((imag_ft_gabor - imag_conv_of_fts).^2)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Convolution Theorem:

```
% Convolution in the time domain is the same as multiplication in
% the frequency domain, up to a known scale factor.  This theorem
% is extremely useful.  Sometimes, you have a filter that is
% simple to characterize in the frequency domain, but complicated
% in the time domain.  For example, it may be very compact in the
% frequency domain (i.e., zero nearly everywhere), but very big
% (i.e., lots of samples needed) in the time domain.  In such
% cases, you can do the filtering by Fourier transforming the
% signal, multiplying in the frequency domain, and then Fourier
% transforming back.

gabor_filter = gabor;
impulse_signal = zeros(1,128);
impulse_signal(65) = 1;
random_signal = 0.5-rand(1,128);

impulse_response = cconv2(impulse_signal,gabor_filter);
plot([0:127], impulse_response, 'r-');
axis([0 127 -1.0 1.0]); figure(1);

filtered_signal = cconv2(random_signal,gabor_filter);
plot([0:127], filtered_signal, 'r-');
axis([0 127 -1.0 1.0]); figure(1);

ft_filtered_signal = fftshift(fft(ifftshift(filtered_signal)));
plot([-64:63], abs(ft_filtered_signal), 'r-');
axis([-64 63 -1 20]); figure(1);

frequency_response = fftshift(fft(ifftshift(impulse_response)));
plot([-64:63], abs(frequency_response), 'r-');
axis([-64 63 -1 6]); figure(1);

ft_random_signal = fftshift(fft(ifftshift(random_signal)));
product_of_fts = frequency_response .* ft_random_signal;
plot([-64:63], abs(product_of_fts), 'r-');
axis([-64 63 -1 20]); figure(1);

% Should be zero:
mean((product_of_fts - ft_filtered_signal).^2)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Frequency Response:

```
% Since the Convolution Theorem is so useful, the Fourier
% transform of the impulse response of a time-invariant linear
% system has a special name.  It is called the Frequency Response
% of the linear system.

% Remember that for a sinusoidal input, the output of a
% time-invariant linear system is sinusoidal with the same
% frequency.  Only the amplitude and phase will be changed by
% filtering.  The frequency response of a filter can be used to
% "read off" the amplitude attenuation and the phase shift, for
% each frequency.  For a complicated signal, that can be
% expressed as the sum of a number of sinusoids, the frequency
% response can be used to "read off" the attenuation and phase
% shift for each component.

% As another example, let's compute the frequency response of the
% 1 sample delay system:

delay_filter = [0 0 1];
impulse_response = cconv2(impulse_signal, delay_filter);
frequency_response = fftshift(fft(fftshift(impulse_response)));
plot([-64:63], imag(frequency_response), 'r-');
axis([-64 63 -1.2 1.2]); figure(1);

plot([-64:63], real(frequency_response), 'r-');
axis([-64 63 -1.2 1.2]); figure(1);

mag_frequency_response = abs(frequency_response);
plot([-64:63], mag_frequency_response, 'r-');
axis([-64 63 0.0 1.2]); figure(1);

phase_frequency_response = angle(frequency_response);
plot([-64:63], phase_frequency_response, 'r-');
axis([-64 63 -3.2 3.2]); figure(1);

% For the delay system, the magnitude of the frequency response
% is constant (1 for all frequencies), and the phase is -w.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% You may have noticed in a few places, we multiplied by the
% square root of the number of samples (e.g., 32).  This scale
% factor is needed given the way that the fft is implemented in
% Matlab.  In some textbooks (e.g., Oppenheim and Schafer), the
% discrete Fourier transform (DFT) is defined so that you divide
% by the number of samples (N) when doing the inverse transform
% (from the frequency domain back into the space/time domain).
% In other texts, the DFT is defined so that you divided by N
% when doing the forward transform.  In still other texts, you
% divided by sqrt-N when doing both the forward and the inverse
% transforms, so that the Fourier transform is an orthonormal
% transform.  The implementation in Matlab follows the first
% (divide by N in the inverse tranform) of these conventions.

% If you lose track of which convention is being used, it is,
% unfortunately, easy to get confused.  For example, using the
% Matlab convenction, we write Parseval's theorem as follows
% (with a factor of 1/N):
%         sum |x[n]|^2 = 1/N sum |X[w]|^2
% Using the orthonormal (divide by root-N for the forward and
% inverse transforms) convention, Parseval's theorem has no scale
% factor:
%         sum |x[n]|^2 = sum |X[w]|^2

% Using the Matlab convention the convolution theorem has no
% scale factor:
%         F{x1[n] * x2[n]} = X1[k] X2[k]
% Using the orthonormal convention, the convolution theorem has a
% factor of sqrt-N:
%         F{x1[n] * x2[n]} = sqrt-N X1[k] X2[k]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Discrete Cosine Transform:

```
% The DCT is another linear transform that is closely related
% to the DFT. The rows of the DCT transform matrix are cosines:
%
%       c(k) 1/sqrt(N) cos[(pi/2N) k (2n+1)]
%
% where c(k)=1 for k=0 and c(k)=root-2 otherwise.  Here, k
% indexes the row and n indexes the column.  In other words,
% n indexes the sample position of the original signal and k
% indexes the transform coefficients.

DCT_mat = zeros(16,16);
nnRange = [0:15];
for k = 0:15
  if (k==0) c=1; else c=sqrt(2); end
  DCT_mat(k+1,:) = c/sqrt(16).*cos(pi/(2*16)*k*(2.*nnRange+1));
end
clf; showIm(DCT_mat);  figure(1);

% The DCT matrix is square and orthonormal:
clf; showIm(DCT_mat * DCT_mat');  figure(1);
clf; showIm(DCT_mat' * DCT_mat);  figure(1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Local Variables:

buffer-read-only: t

End:

Published with MATLAB® R2014b