Let's look at a homework problem...

Last session

- Moving average filters
- Differentiation
- Fourier series & transform

This session

- Complexity and big-O
- The FFT
- Fourier transform pairs

Concept: “Time Complexity”

- Question: How does program execution time scale with size of input?
- Examples:
  - Dot product: $\vec{u} \cdot \vec{v}$ Execution time grows as length of vector. (Explanation on board)
  - Matrix multiplication: $A\,B$
    Execution time grows as length of vector cubed. (Explanation on board.)
- Finding “performant” algorithms is a main goal of numerical analysis.

Big-O notation

- Consider growth of execution time as input size goes to infinity.
- Only keep the fastest growing part.
  Example: $O(N^2 + N) = O(N^2)$
- Ignore any constant multiplicative factors – we are only interested in *scaling*.
- $O(N)$ – Execution time grows linearly with input data size
- $O(\log(N))$ – Execution time grows as log of input data size
- $O(N^2)$ – Execution time grows as square of input data size.

Examples of different algorithms

- O(1) – Return first element of vector.
- O(N) -- Sum elements of a vector. Search an unordered list.
- $O(N^2)$ – Multiply two N digit numbers. Bubble sort.
- O(log N) – Binary search on ordered list. Example: finding a name in a phone book of N names.
- $O(N^3)$ – matrix multiply (Naive).
- O(N!) -- Computing determinant of matrix using permutations (what you learned as undergraduate).

Example O(1)

- Function which computes a number and returns.
- Independent of input data size
  - This fcn works only on scalars.

```
function y = derivative( f, x, h )
  % This fcn returns the forward-difference
  % derivative of f at x using step h
  y = (f(x+h) - f(x)) / (h);
end
```

Example $O(N^3)$ – matrix multiplication

```
function z = mymatmul(x, y)
  % Matrix multiplication the naive way, using loops.
  % This algorithm is O(n^3)
  % z = x*y
  % size(x) = [n, m]
  % size(y) = [m, p]
  % size(z) = [n, p]

  [n, m] = size(x);
  [m, p] = size(y);

  z = zeros(n, p);
  for row = 1:n
    for col = 1:p
      for idx = 1:m
        z(row, col) = z(row, col) + x(row, idx)*y(idx, col);
      end
    end
  end

end
```

Three nested loops

The main point

- Performant algorithms have low-order complexity.
  - $O(N)$ is good.
  - $O(N \log N)$ is good.
  - $O(N^p)$ where $p >> 1$ not so good.
- Consider (naive) matrix multiply -- $O(N^3)$
  - Suppose N=10 takes 1 sec.
  - Then N = 100 takes 1000 sec.
  - N = 1000 takes 1e6 seconds.

Numerical algorithm: Fast Fourier Transform

- Algorithm implementing Fourier Transform for sampled signal.
- “The most important numerical algorithm of our lifetime”, Gilbert Strang, American Scientist 82 (3): 253 (1994).
- Operates on sampled (discrete) signal.
- ***Complexity: O(N log N)***
- Matlab: fft(), ifft()

Fourier transform

$$Y(\omega) = \int_{-\infty}^{\infty} dt\, y(t)\, e^{-i\omega t}$$

$$y(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} d\omega\, Y(\omega)\, e^{i\omega t}$$

Computation of FFT

- Yf = fft(y) takes vector of length N and returns vector of length N.

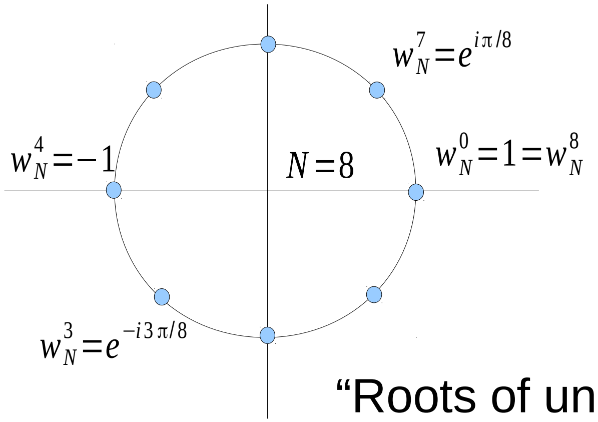
$$Y_k = \sum_{n=0}^{N-1} y_n e^{-i 2\pi n k / N}$$

“Roots of unity”

- Define: $w_N^k = e^{-i 2\pi k / N}$

- So:

$$Y_k = \sum_{n=0}^{N-1} y_n \left(w_N^k\right)^n$$



“Roots of unity” in complex plane

Consider 4 point example

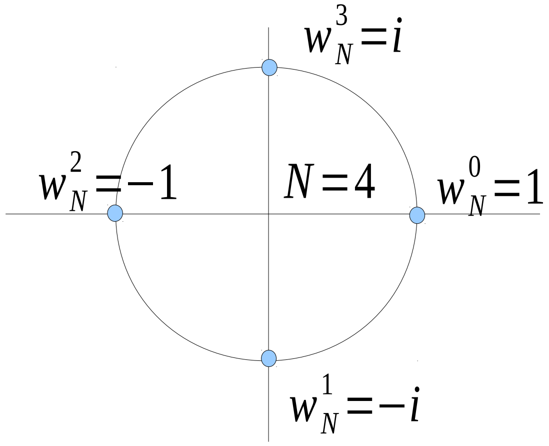
$$Y_k = \sum_{n=0}^{N-1} y_n \left(w_N^k\right)^n$$

$$Y_0 = y_0\left(w_N^0\right)^0 + y_1\left(w_N^0\right)^1 + y_2\left(w_N^0\right)^2 + y_3\left(w_N^0\right)^3$$

$$Y_1 = y_0\left(w_N^1\right)^0 + y_1\left(w_N^1\right)^1 + y_2\left(w_N^1\right)^2 + y_3\left(w_N^1\right)^3$$

$$Y_2 = y_0\left(w_N^2\right)^0 + y_1\left(w_N^2\right)^1 + y_2\left(w_N^2\right)^2 + y_3\left(w_N^2\right)^3$$

$$Y_3 = y_0\left(w_N^3\right)^0 + y_1\left(w_N^3\right)^1 + y_2\left(w_N^3\right)^2 + y_3\left(w_N^3\right)^3$$



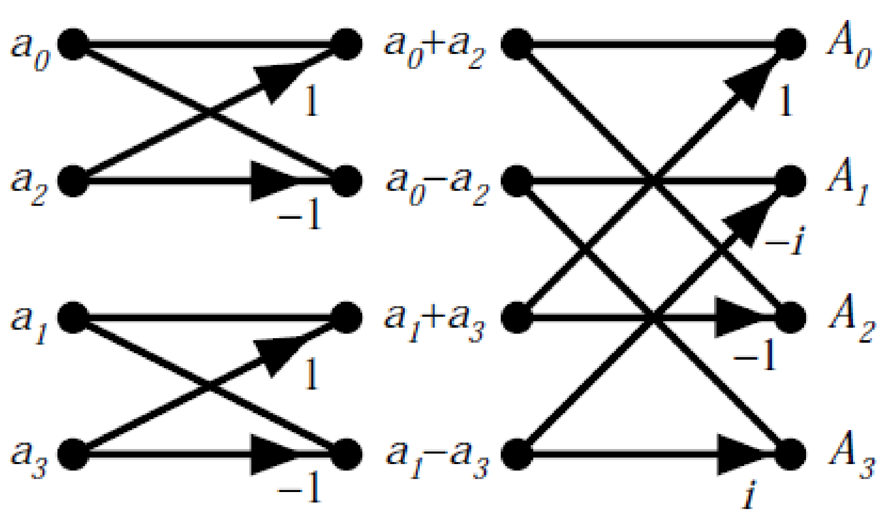
Evaluate w coefficients

$$Y_0 = y_0 (w_N^0)^0 + y_1 (w_N^0)^1 + y_2 (w_N^0)^2 + y_3 (w_N^0)^3$$
$$Y_1 = y_0 (w_N^1)^0 + y_1 (w_N^1)^1 + y_2 (w_N^1)^2 + y_3 (w_N^1)^3$$
$$Y_2 = y_0 (w_N^2)^0 + y_1 (w_N^2)^1 + y_2 (w_N^2)^2 + y_3 (w_N^2)^3$$
$$Y_3 = y_0 (w_N^3)^0 + y_1 (w_N^3)^1 + y_2 (w_N^3)^2 + y_3 (w_N^3)^3$$

$$Y_0 = y_0 + y_1 + y_2 y_3$$
$$Y_1 = y_0 - i y_1 - y_2 + i y_3$$
$$Y_2 = y_0 - y_1 + y_2 - y_3$$
$$Y_3 = y_0 + i y_1 - y_2 - i y_3$$

$$Y_0 = (y_0 + y_2) + (y_1 + y_3)$$
$$Y_1 = (y_0 - y_2) - i (y_1 - y_3)$$
$$Y_2 = (y_0 + y_2) - (y_1 + y_3)$$
$$Y_3 = (y_0 - y_2) + i (y_1 - y_3)$$

Think of this as two step computation.....

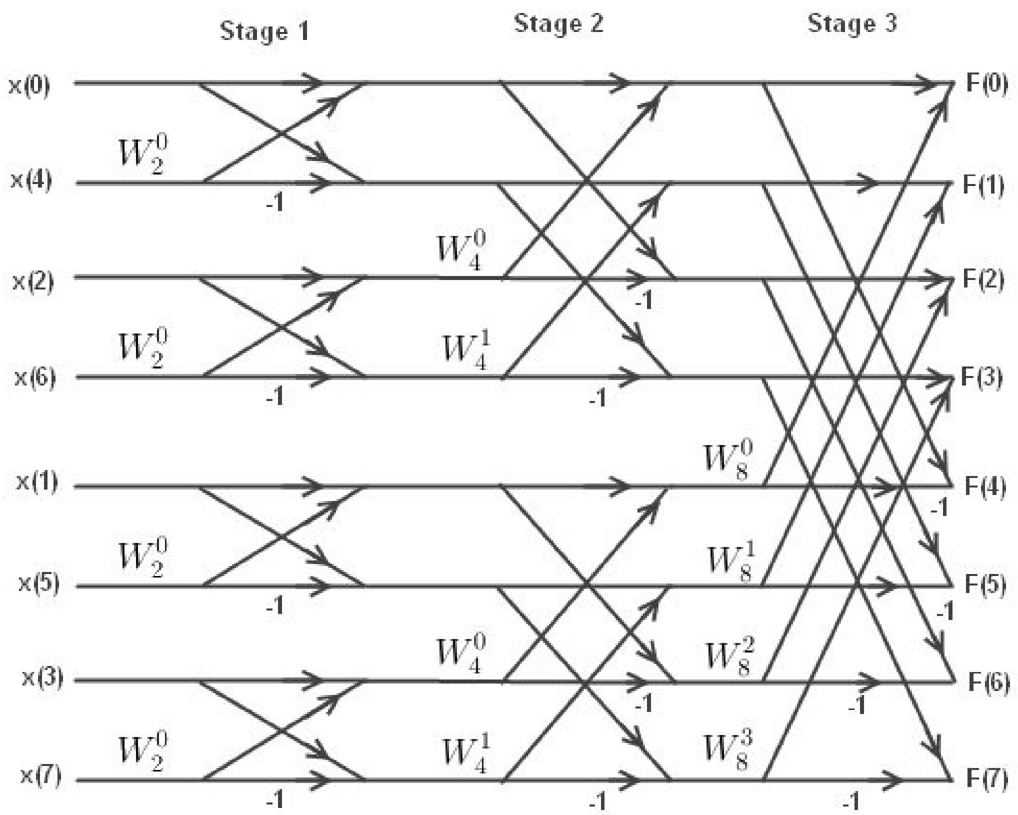


“Butterfly diagram”

*I have computed 4 output values in 4*2 multiplications*

“Butterfly diagram”

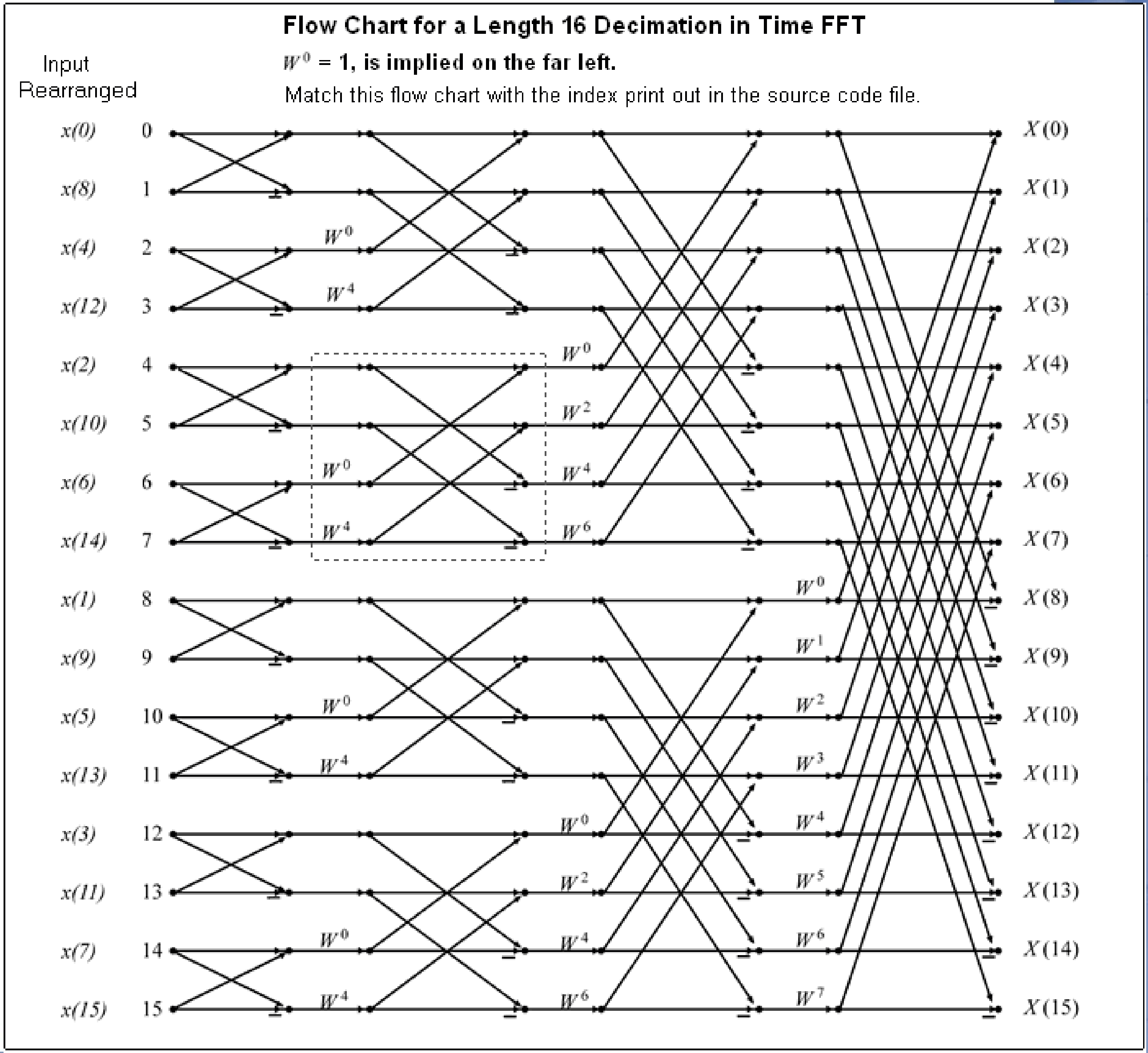
An 8 Input Butterfly. Note, you double a 4 input butterfly, extend output lines, then connect the upper and lower butterflies together with diagonal lines.



- N rows of values, 3 layers of multiplication.
- $\text{Log}_2(8) = 3$
- Number of multiplications = n log n

FFT is O(n log n)

- 4 input points
  - 4*2 operations
- 8 input points
  - 8*3 operations
- 16 input points
  - 16*4 operations



Burrus, C. Fast Fourier Transforms, Connexions Web site. http://cnx.org/content/col10550/1.22/.

Different versions of FFT

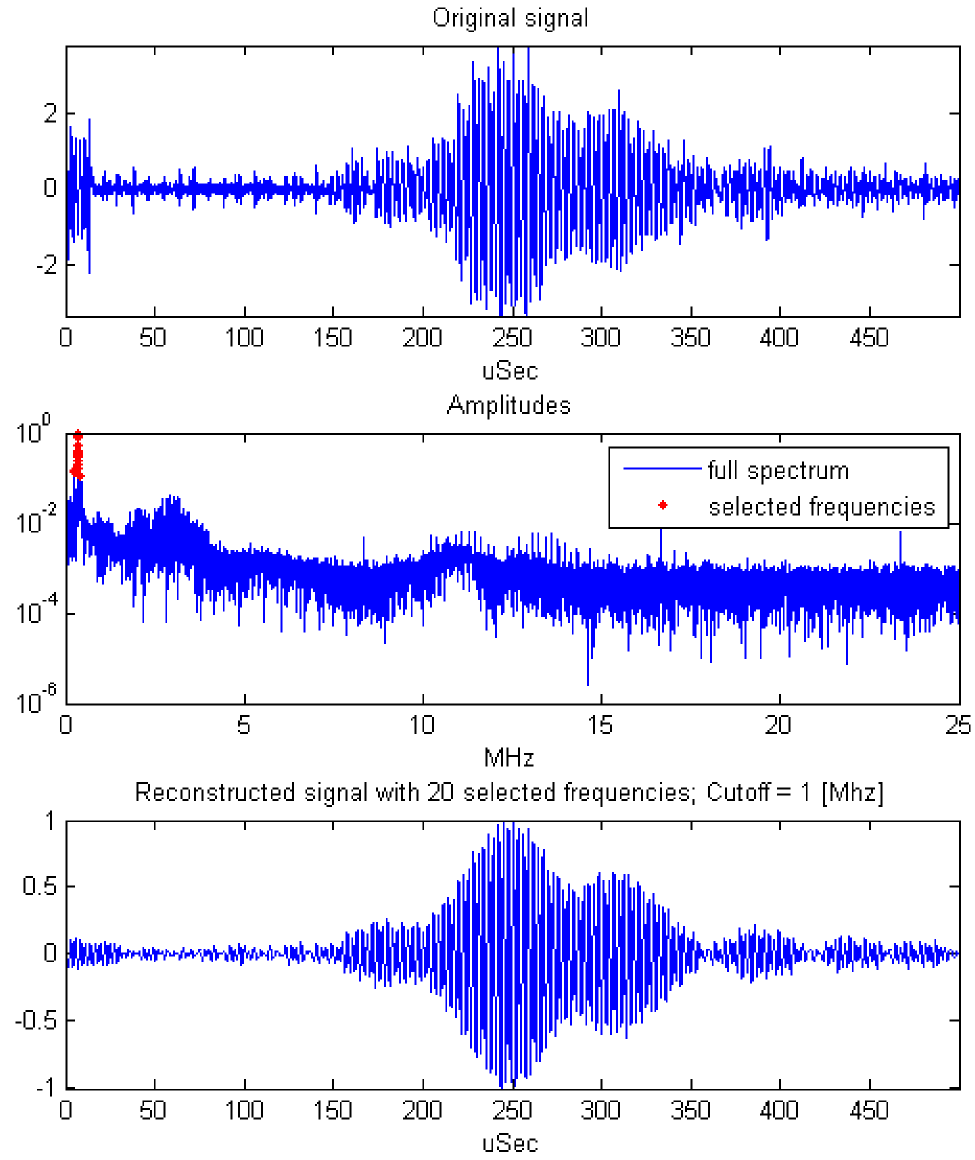
- FFT (fast Fourier transform)
  - exp(-iwt) basis functions.
  - Input & output complex
- DCT (discrete cosine transform)
  - cos(wt) basis functions
  - Input & output real.
- DST (discrete sine transform)
  - sin(wt) basis functions
  - Input & output real

FFTW

- Matlab uses FFTW to compute FFTs.

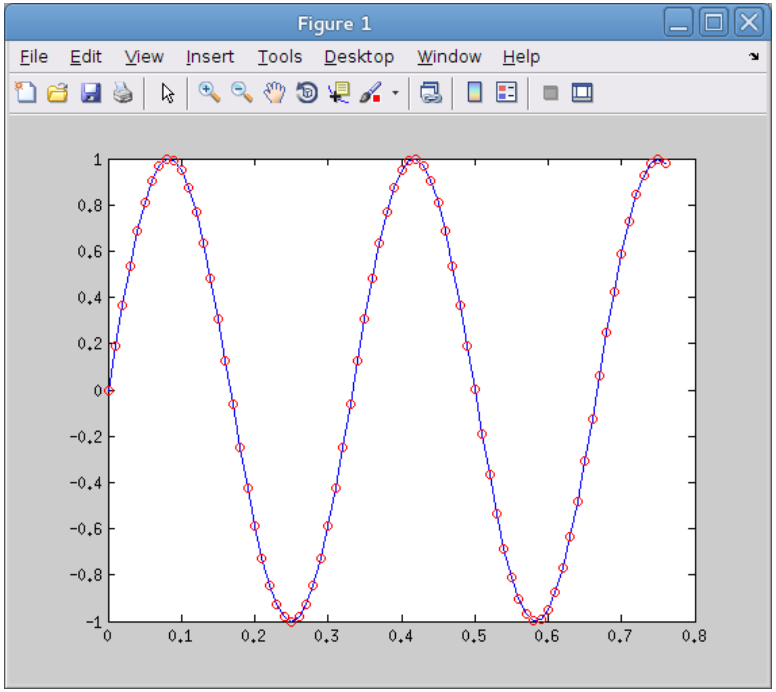
Our goal: filter signals using FFT

1. FFT your input time-domain signal.
2. Select frequencies you want to keep.
3. Inverse FFT to get filtered time-domain signal



time <=> frequency

Input signal



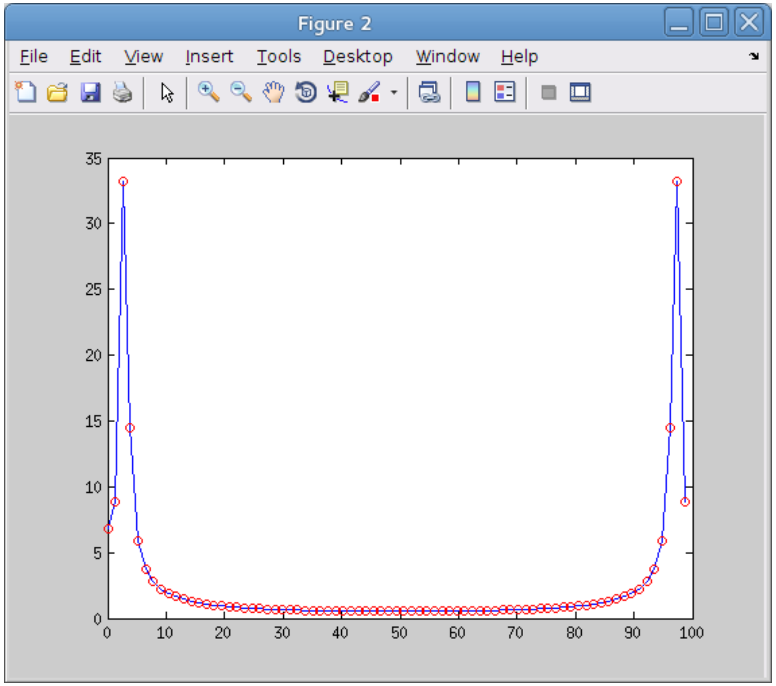
Number of samples N

Sample period Δt

Sample length $T=(N-1)\Delta t$

Sample frequency $f_s=\dfrac{1}{\Delta t}$

Output FFT

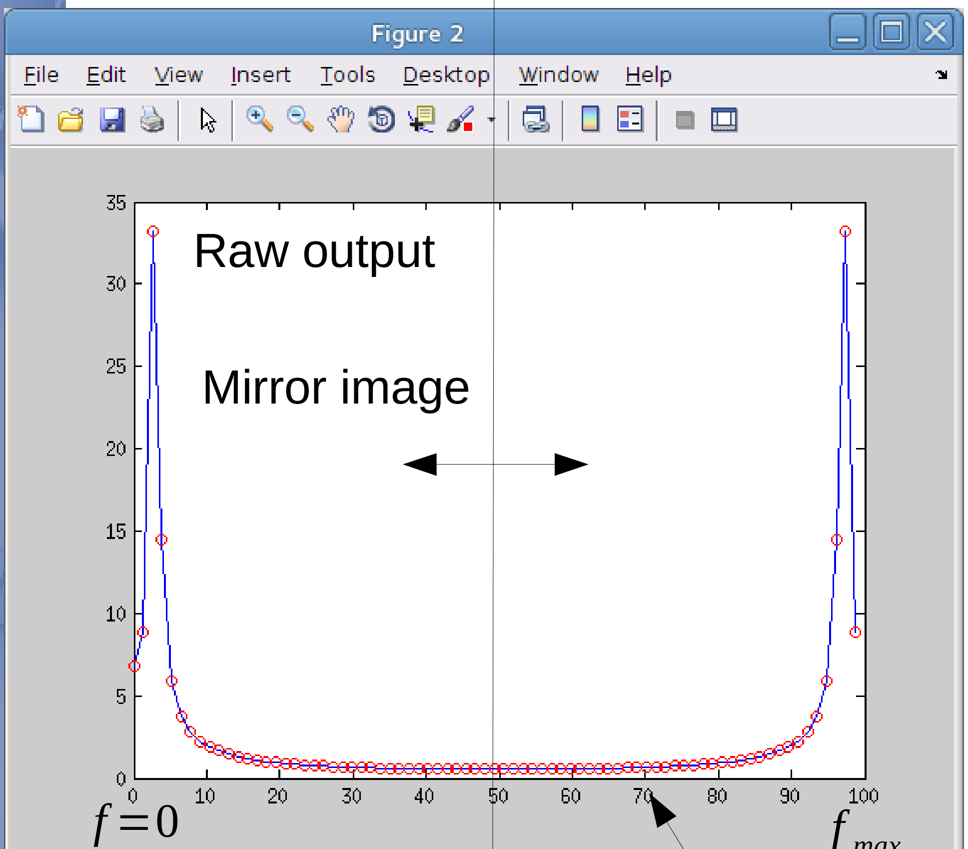


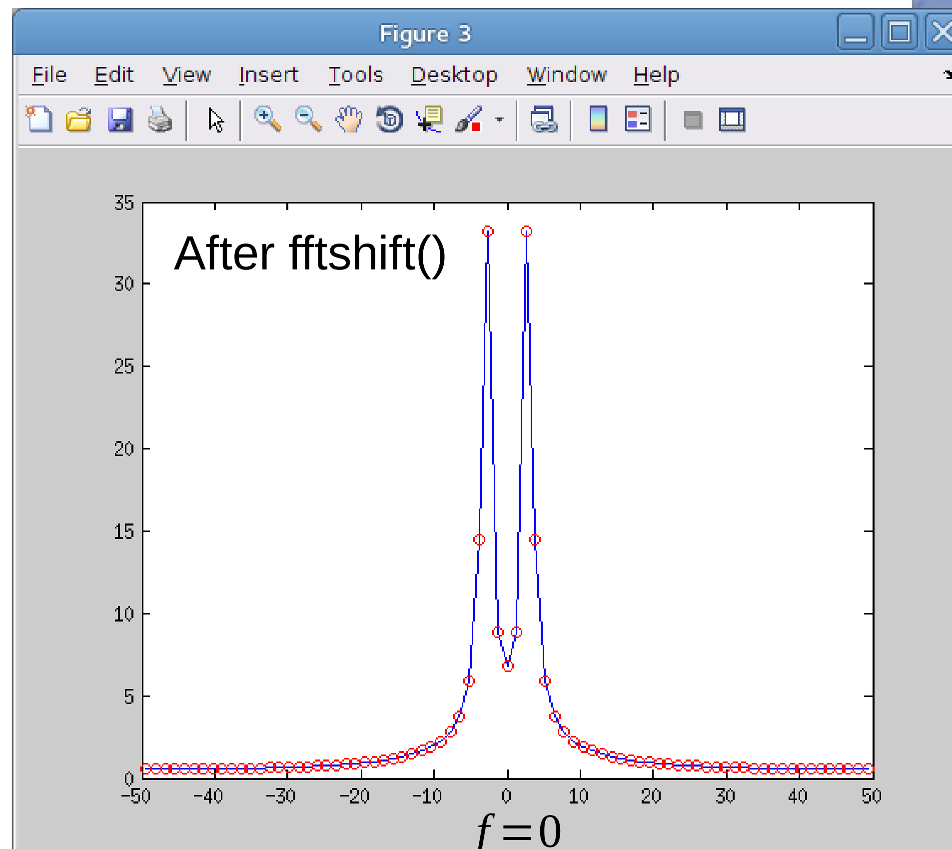
Number of samples $M=N$

Maximum frequency $\dfrac{M-1}{M}f_s$

Frequency step $\Delta f=\dfrac{f_s}{M}$

Raw output of FFT





$f=\frac{f_{max}}{2}$

No new information in upper 1/2 of spectrum

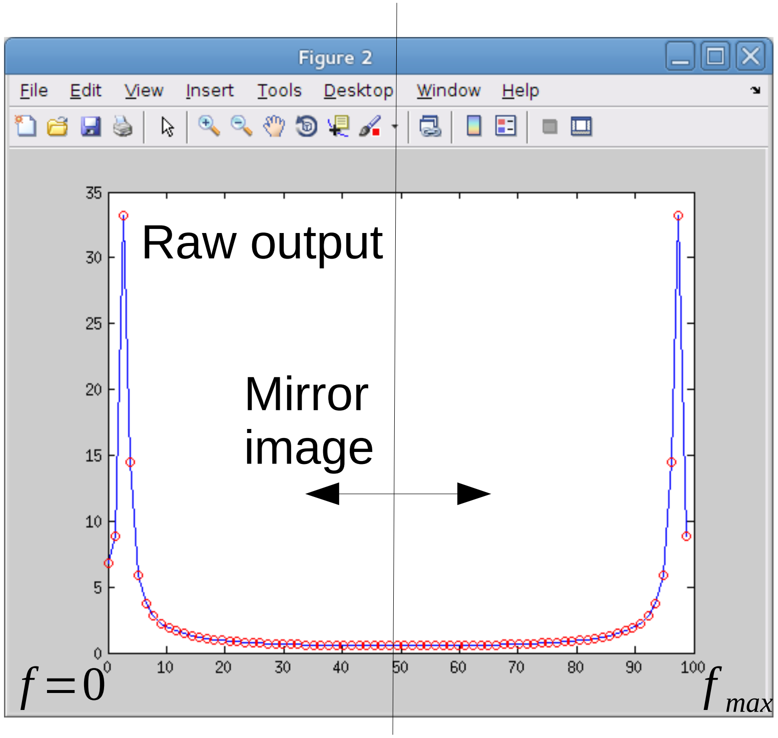
$f=-\frac{f_{max}}{2}$

$f=\frac{f_{max}}{2}$

fftshift() folds upper 1/2 of spectrum down to give two-sided spectrum

Nyquist frequency

- Upper 1/2 of spectrum from FFT is redundant (no new information).
- Nyquist frequency is maximum frequency of signal which can be correctly captured and reconstructed using FFT.



$$f_{Nyquist} = \frac{f_{max}}{2}$$

Sampling theorem

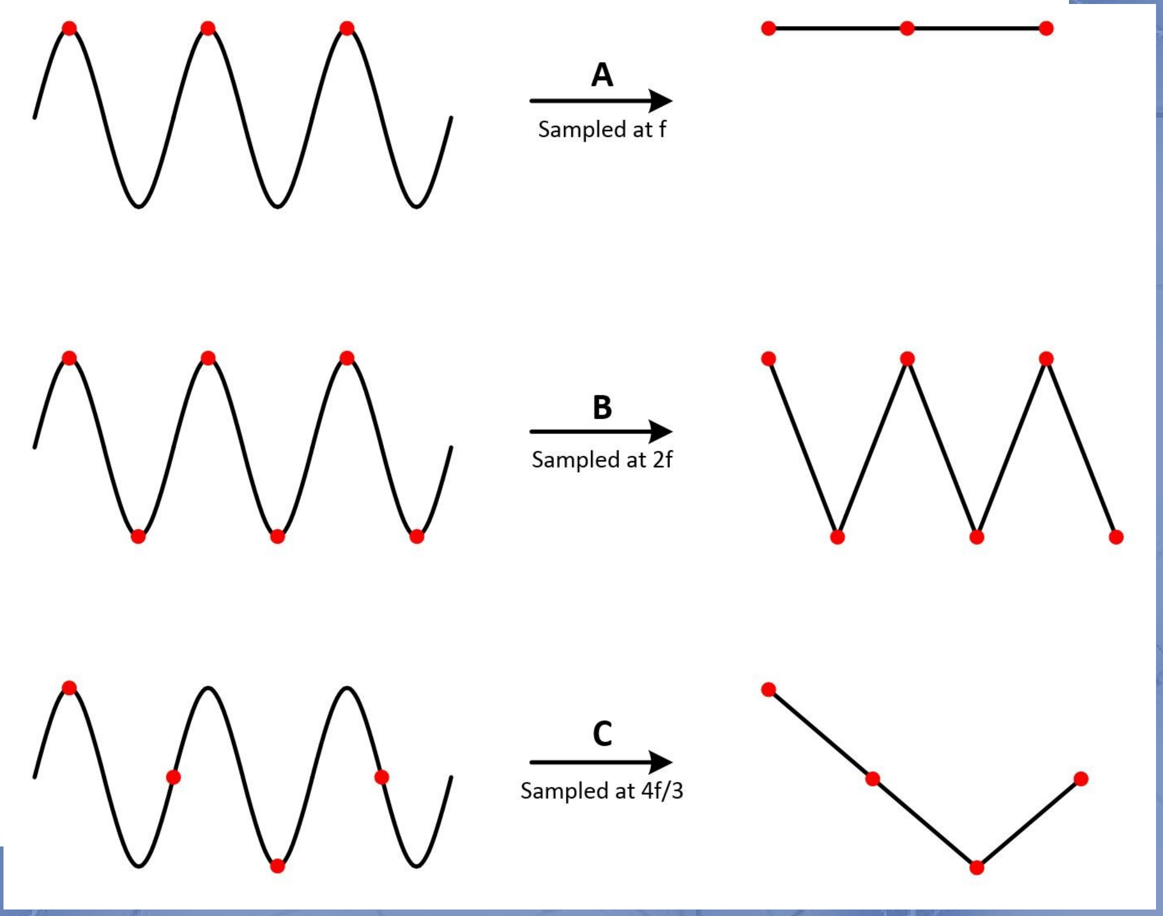
- Our usual goal in using the FFT:
  - Capture some time-domain signal
  - FFT the signal
  - Do some operations in frequency domain (filter the signal)
  - Inverse FFT to reconstruct time-domain signal.
- Sampling theorem: You must sample the signal at f_s >= 2* maximum frequency in signal to avoid distortion in reconstructed signal

Aliasing

- If signal is not sampled fast enough, the sampled signal does not represent the actual (continuous) signal.
- For correct reconstruction of the actual signal, you must have

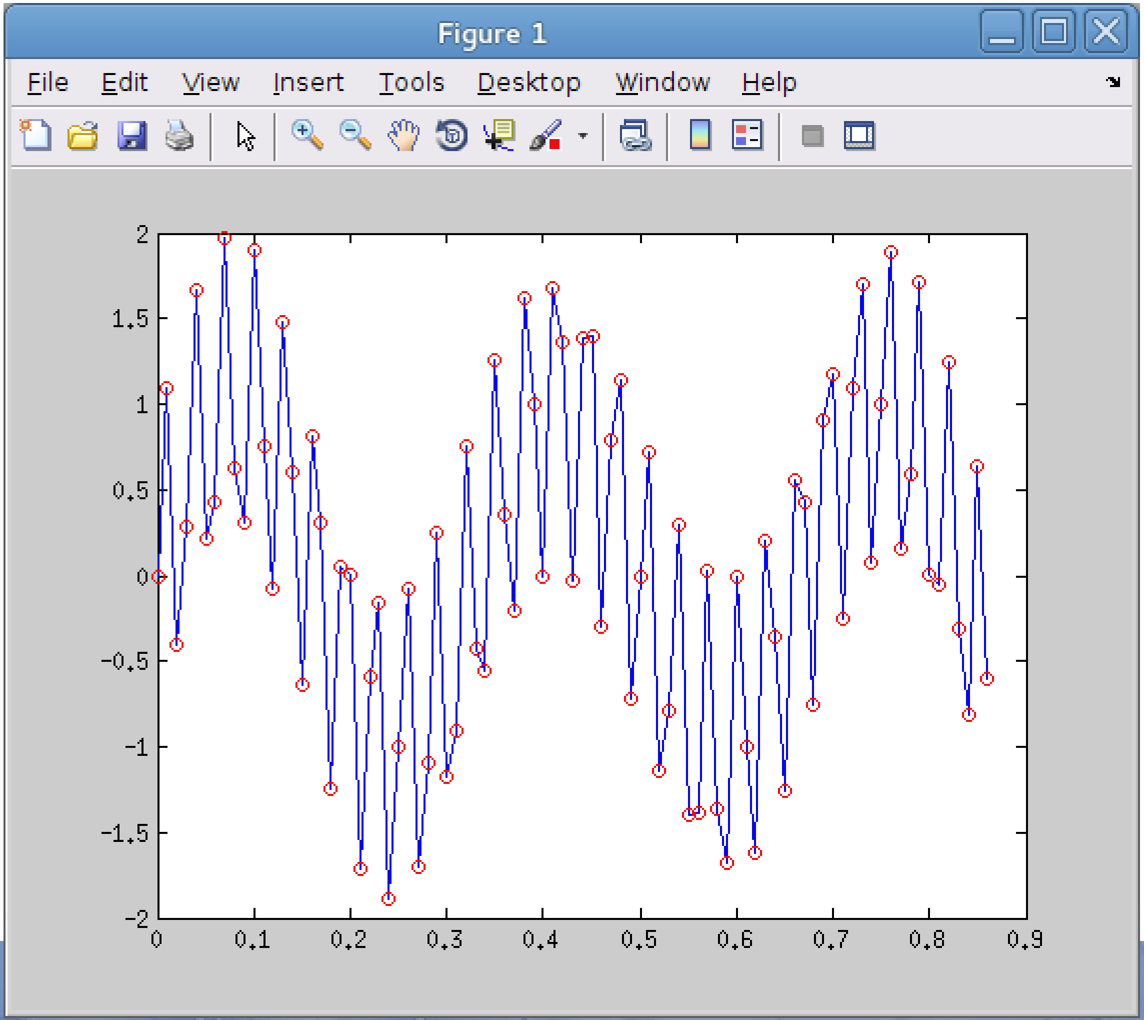
$f_s \geq$ Maximum frequency component in signal

- If you don't, the effect is called "aliasing".



Example: low-pass filter

- Two sine waves: 3Hz and 32Hz.
- Want to cut out high-frequency sine wave.



Look in
fft_playpen
for code

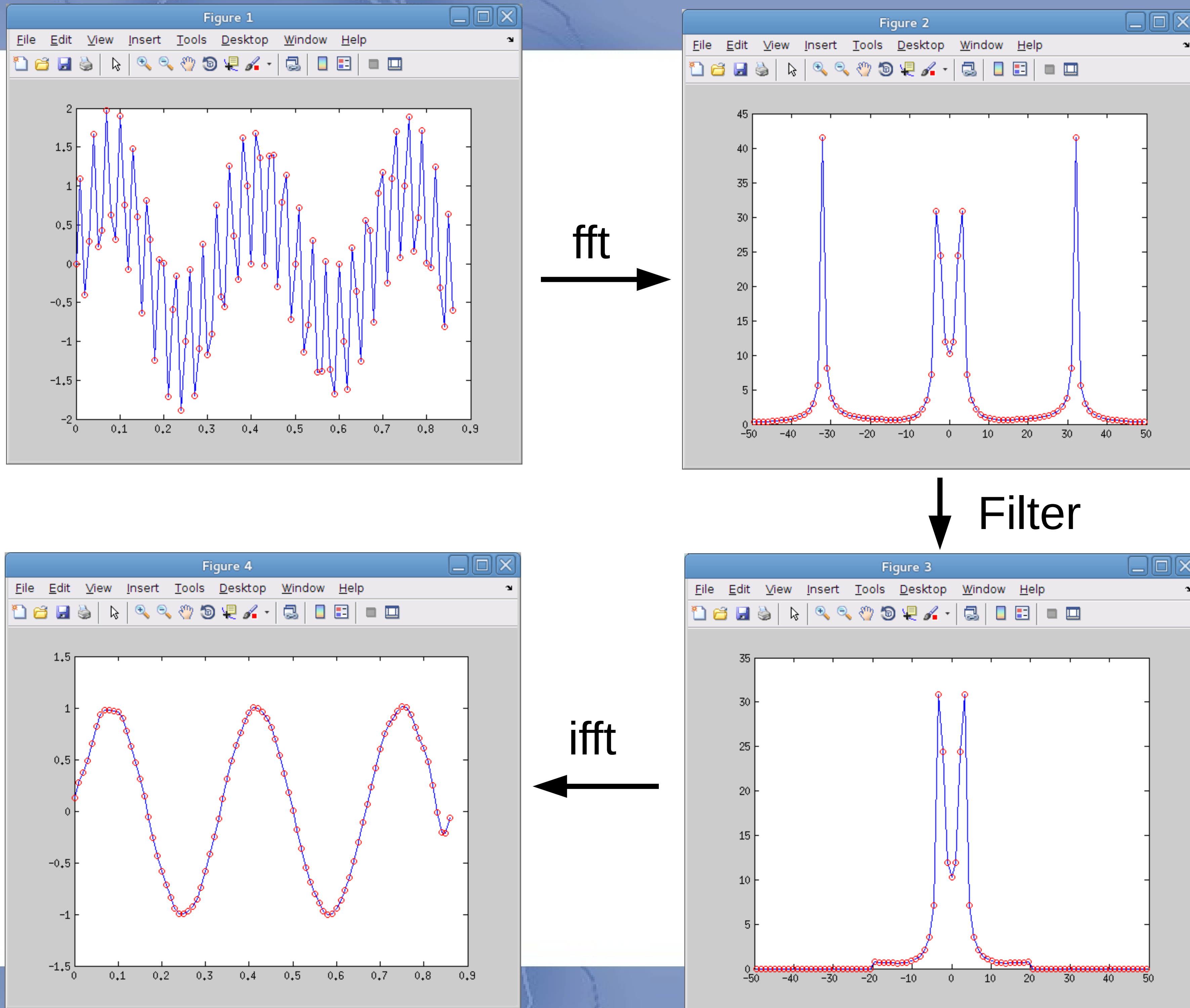


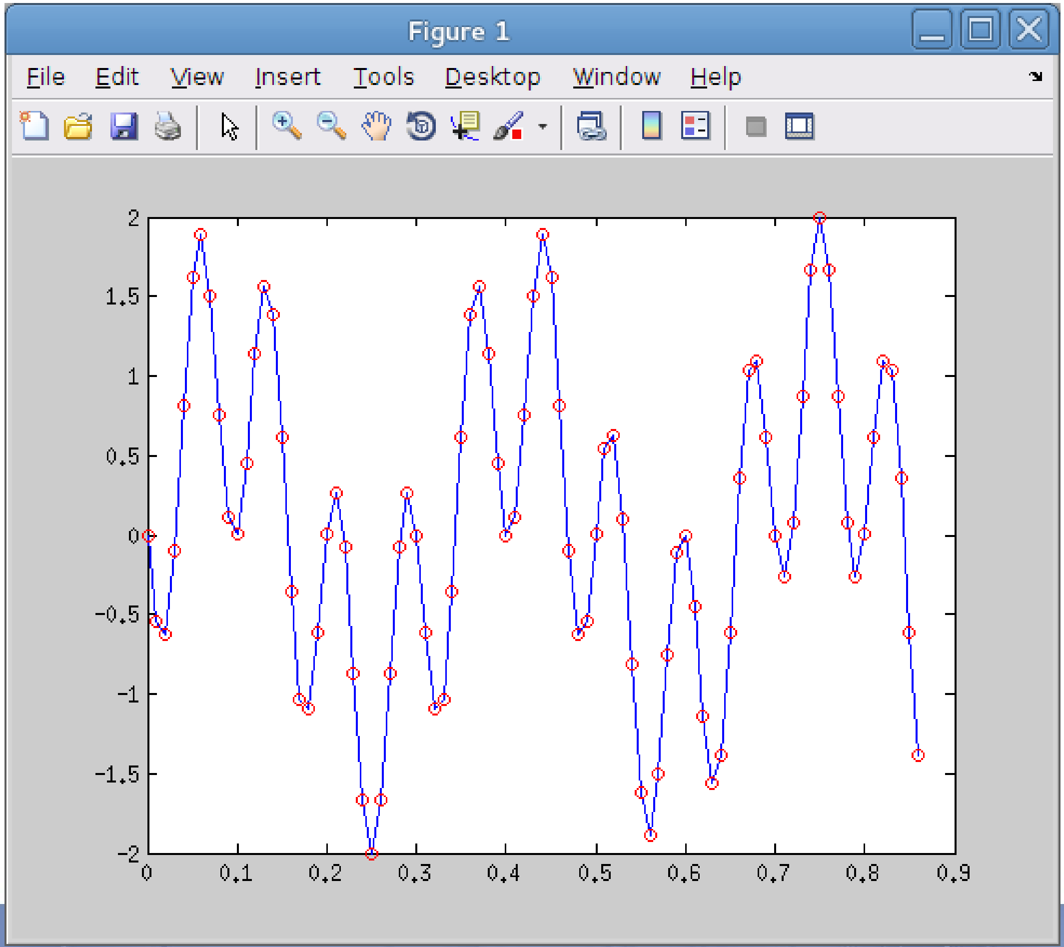
Figure 1
File Edit View Insert Tools Desktop Window Help
fft
Figure 2
File Edit View Insert Tools Desktop Window Help
Filter
Figure 3
File Edit View Insert Tools Desktop Window Help
ifft
Figure 4
File Edit View Insert Tools Desktop Window Help

```
function filter_sines()
  % This fcn creates two sine waves of different frequencies.
  % It then FFTs the time-domain signal and zeros out
  % the high frequency stuff (low-pass filter), then
  % does ifft.

  M = 87;      % Number of sample points
  dt = .01;     % 10mS sample period.
  fs = 1/dt;    % Sample freq
  t = linspace(0, (M-1)*dt, M);    % Vector of timestamps

  % Create sine waves at 3 and 32 Hz
  w1 = 3;       % 3 Hz signal
  w2 = 32;
  x = sin(2*pi*w1*t) + sin(2*pi*w2*t);      samples
  figure(1)
  plot(t, x)
  hold on
  plot(t, x, 'ro')

  % Now do FFT of input signal
  Xf = fft(x); % Fast Fourier Transform
  w = linspace(0, (M-1)*(fs/M), M);
```

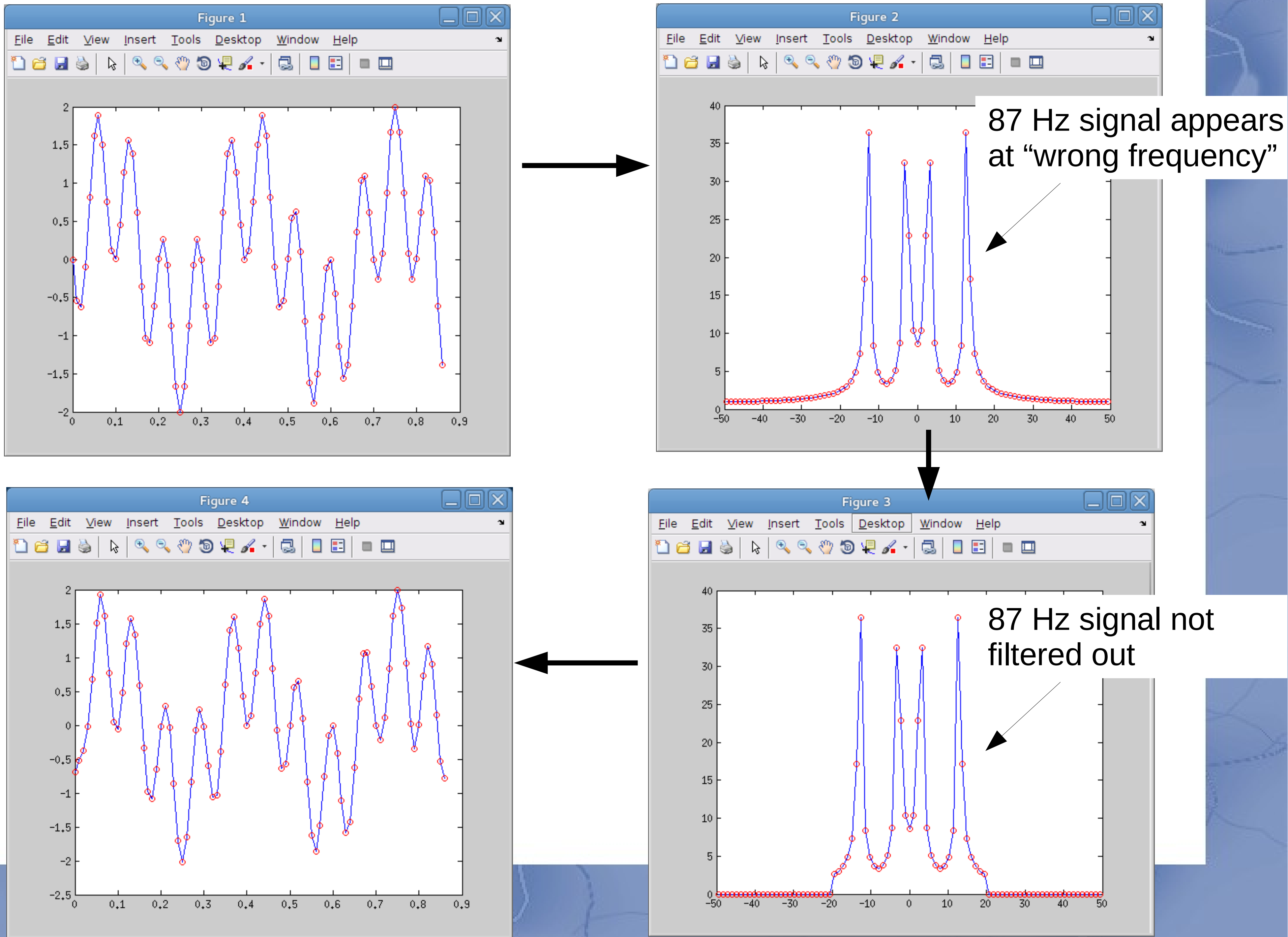
```
  % Shift over negative freqs on frequency axis in prep for fftshift
  w1 = w;
  w1(w >= fs/2) = w(w >= fs/2) - fs;
  ws = fftshift(w1);
  Xfs = fftshift(Xf);
  figure(2)
  plot(ws, abs(Xfs))
  hold on
  plot(ws, abs(Xfs), 'ro')

  % Now zero out all frequency components above 20 Hz.
  idx = find(abs(ws) > 20);
  Xfs(idx) = 0;
  figure(3)
  plot(ws, abs(Xfs))
  hold on
  plot(ws, abs(Xfs), 'ro')

  % Now ifft signal back to time doman
  Xf = ifftshift(Xfs);
  xnew = ifft(Xf);
  figure(4)
  plot(t, real(xnew))
  hold on
  plot(t, real(xnew), 'ro')

end
```

Effect of undersampling

- Two sines: 3Hz and 87Hz
- Sample frequency: 100Hz (0.01sec sample period)
- This input signal has frequency components larger than Nyquist frequency.



Effect of undersampling



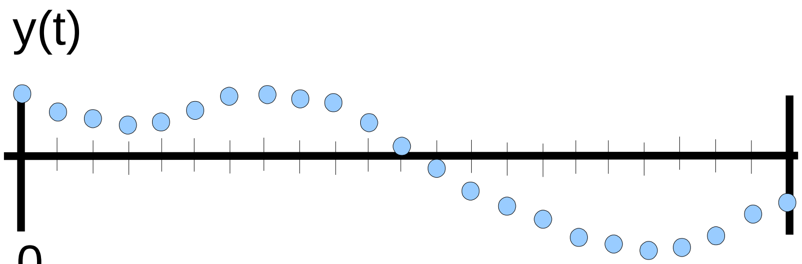
Takeaway points (so far)

- Algorithm to compute FFT is fast (O(N log N)).
- Must pay close attention to time and frequency axis.
- Must sample signal with f_s >= 2*highest frequency component of signal.
- If you don't, aliasing will introduce spurious components into your signal.

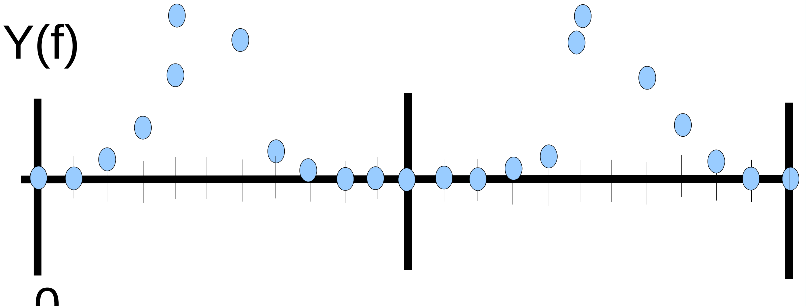
Next topic: Fourier transform pairs

- Fourier transform pair: Every time-domain function has a frequency-domain dual.

$$y(t)=\frac{1}{2\pi}\int_{-\infty}^{\infty} d\omega\, Y(\omega)\, e^{i\omega t} \quad \longleftrightarrow \quad Y(\omega)=\int_{-\infty}^{\infty} dt\, y(t)\, e^{-i\omega t}$$

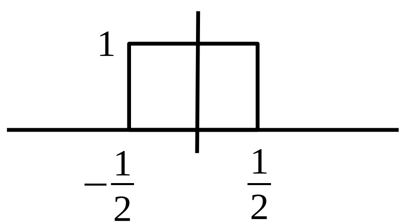


Time domain



Frequency domain

Fourier transform pairs

Time domain			Frequency domain	
	$y(t)=\frac{1}{2\pi}\int_{-\infty}^{\infty} d\omega\, Y(\omega)\, e^{i\omega t}$	$\leftrightarrow$	$Y(\omega)=\int_{-\infty}^{\infty} dt\, y(t)\, e^{-i\omega t}$	
Gaussian	$e^{-t^2/(2\sigma^2)}$	$\leftrightarrow$	$\sigma\sqrt{2\pi}\, e^{-\sigma^2\omega^2/2}$	Gaussian
Box function		$\leftrightarrow$	$\frac{1}{\sqrt{2\pi}}\frac{\sin(\omega/2)}{\omega/2}$	sinc
Delta function	$\delta(t)$	$\leftrightarrow$	1	Constant over all frequencies
Sine	$\sin(\omega_0 t)$	$\leftrightarrow$	$-i\pi\left(\delta(\omega-\omega_0)-\delta(\omega-\omega_0)\right)$	

Two delta functions at +/- input frequency

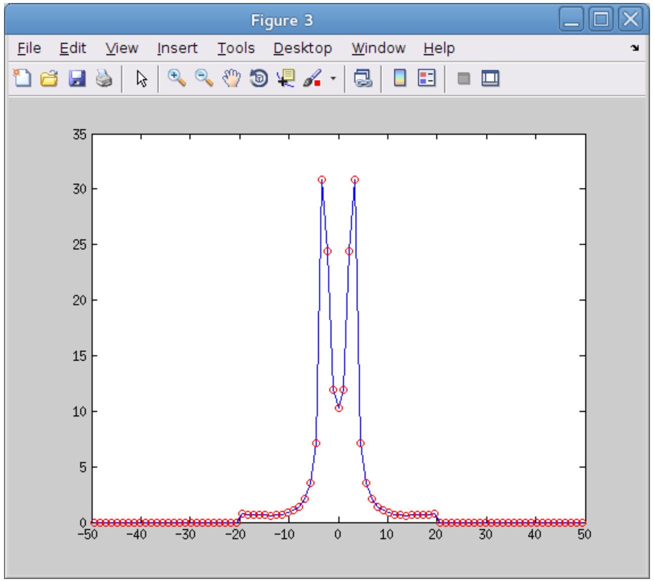
More Fourier transform pairs...

Shift	$y(t-t_0)$	$\leftrightarrow$	$Y(\omega)e^{-i\omega t_0}$	
Time derivative	$\left(\frac{d}{dt}\right)y(t)$	$\leftrightarrow$	$(-i\omega)Y(\omega)$	
Convolution	$\int_{-\infty}^{\infty} d\tau\, y(\tau)z(t-\tau)$	$\leftrightarrow$	$Y(\omega)Z(\omega)$	Multiplication

- Note that these pairs apply to operations, not just to functions

Convolution and filtering

- Recall our low-pass filter from earlier this session.
- That filter was equivalent to multiplying by a box function in frequency domain. $Y(\omega)Z(\omega)$



- This is equivalent to convolution by sinc() in time domain.

$$\frac{\sin(\omega/2)}{\omega/2}$$

- Recall this filter kernel from session 1?

Filtering and convolution

- Recall this weighted-average filter from Session 1?

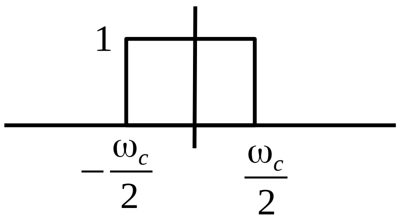
$$y_n = \sum_{i=-m}^{m} w_{n-i} x_i$$

- That filter was performing a convolution in the time domain.

$$\int_{-\infty}^{\infty} d\tau \, y(\tau) z(t-\tau)$$

- Hard-wall filter in frequency is same as convolution with sinc() in time domain.

$$\frac{\sin(t/2)}{t/2}$$

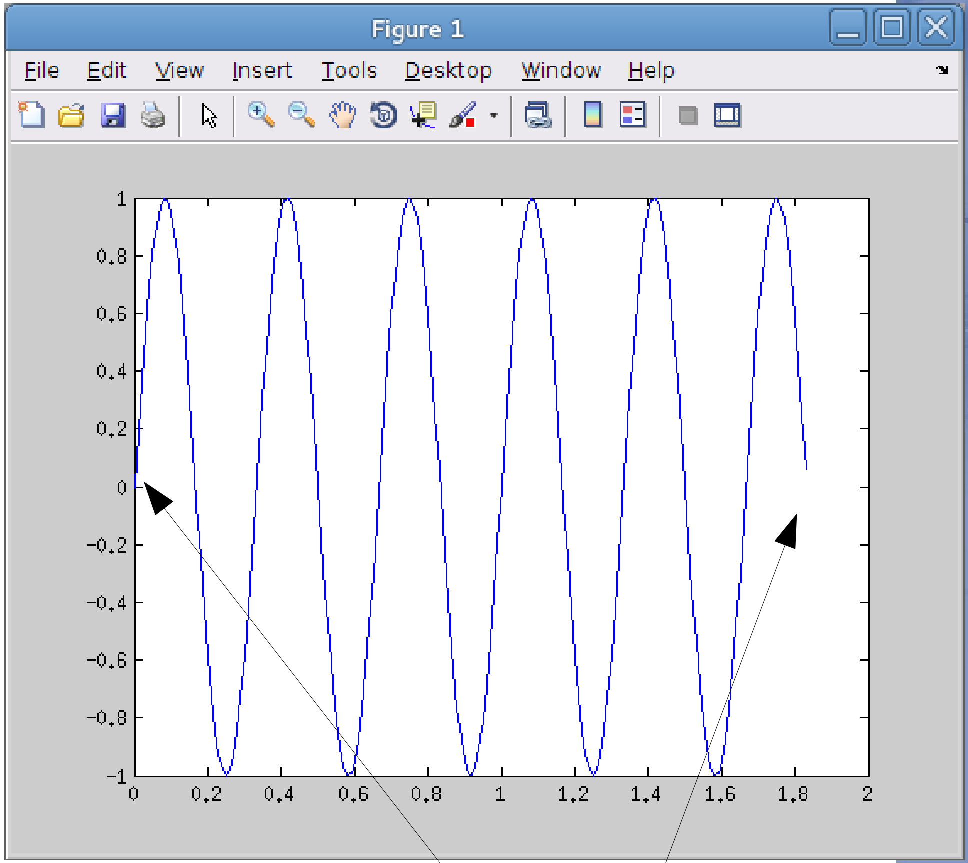


Remarks

- Fourier transforms allow you to create a desired frequency response for your filter.
- Time-domain filter kernel may be analyzed in frequency domain.
- Two ways to filter your signal:
  - FFT into frequency domain and manipulate its spectrum.
  - Create desired filter kernel in frequency domain, then inverse FFT into time domain and use convolution to filter your signal

Final topic: Windowing

- FFT is sensitive to discontinuities at ends of signal.
- FFT “thinks” signal is periodic.
- Therefore, if the ends don't match, the FFT senses a discontinuity.
    - This causes effects in the frequency spectrum.



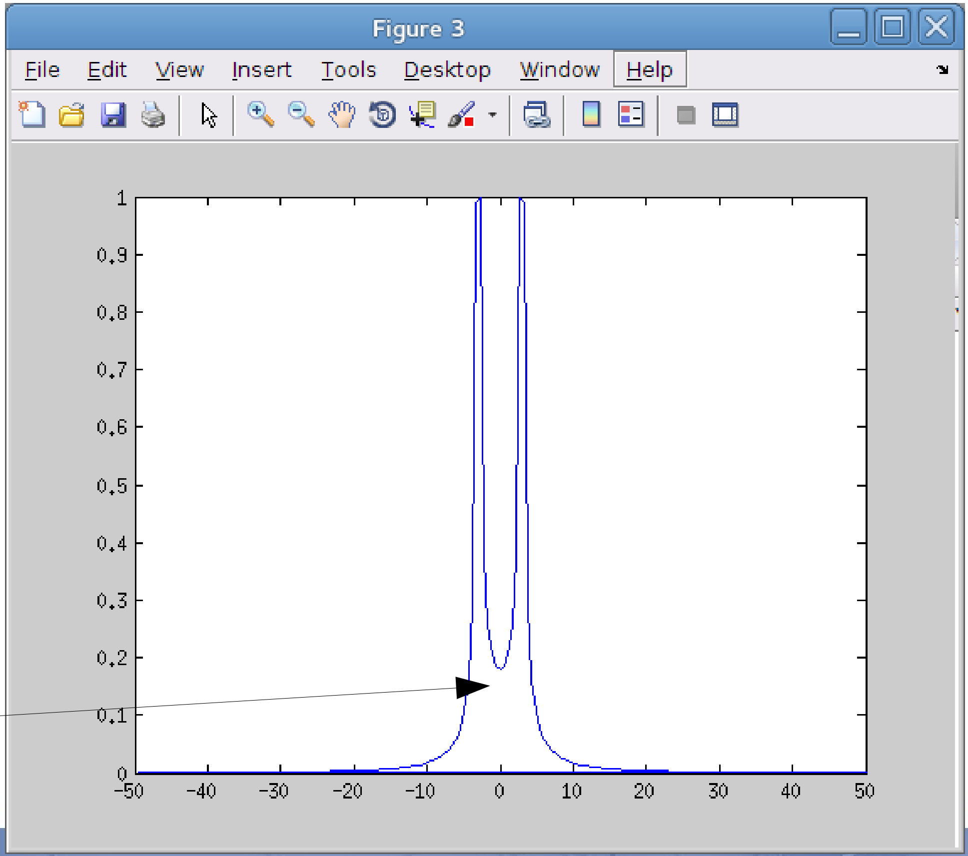
Abrupt change of signal at ends because FFT thinks signal is periodic.

Effect on computed spectrum

- Recall Fourier transform pair:

$$\sin(\omega_0 t) \longleftrightarrow -i\pi\left(\delta(\omega-\omega_0)-\delta(\omega-\omega_0)\right)$$

- The FFT should give two delta functions at +/- 3Hz.
- But the deltas sit on a pedestal!

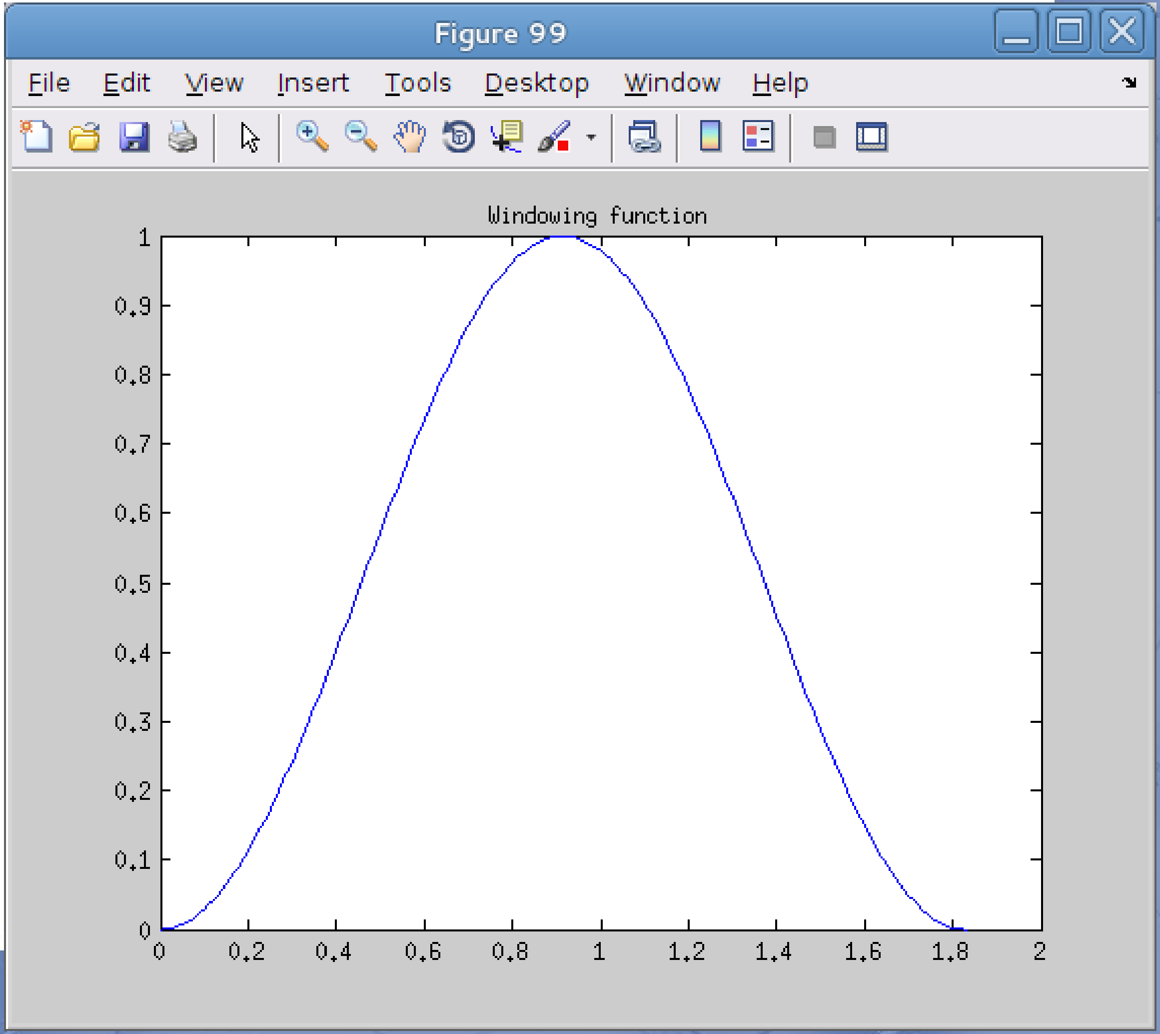


Pedestal

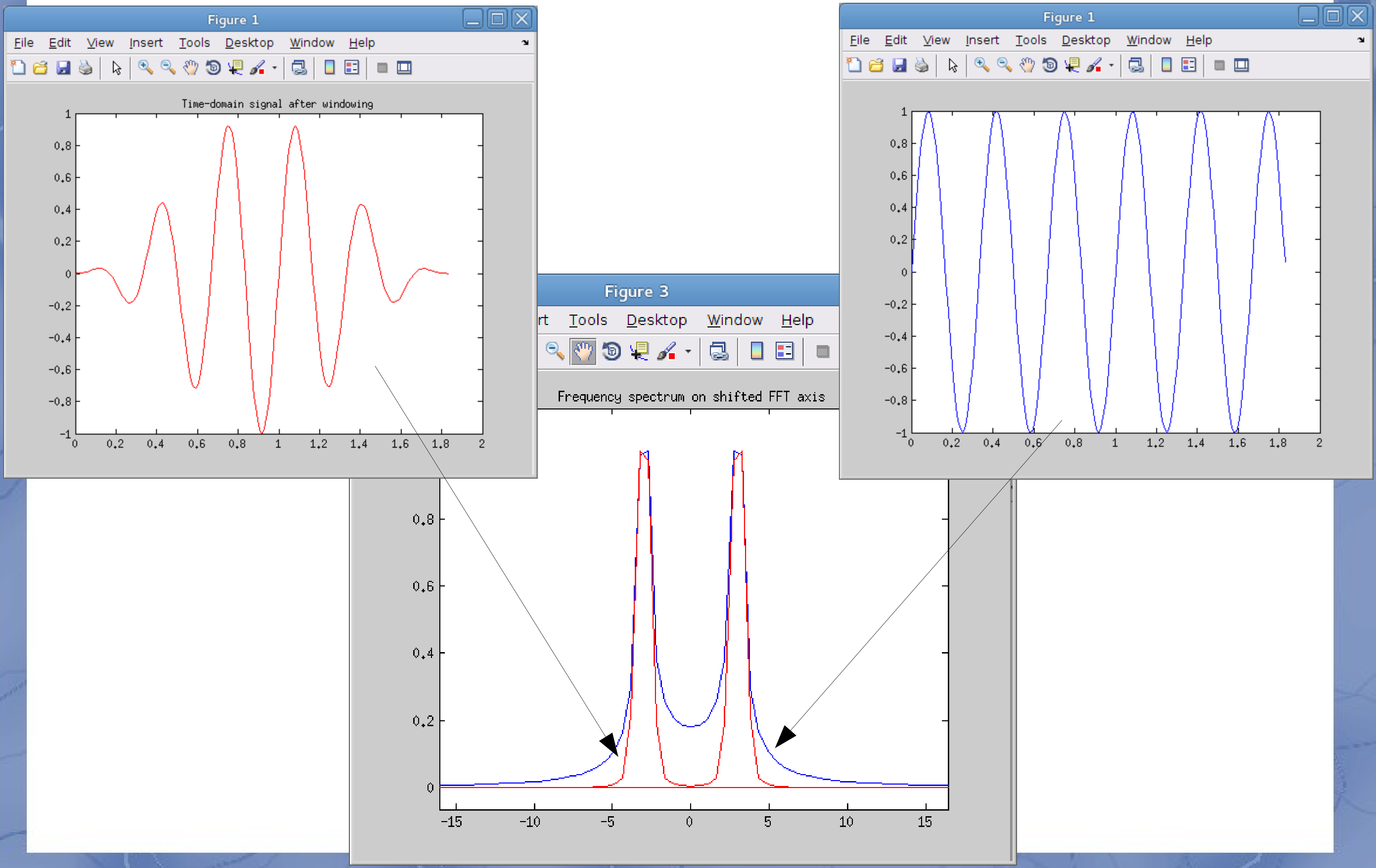
Solution: Use window

- Multiply input signal by smooth function which goes to zero at ends.
- Example:

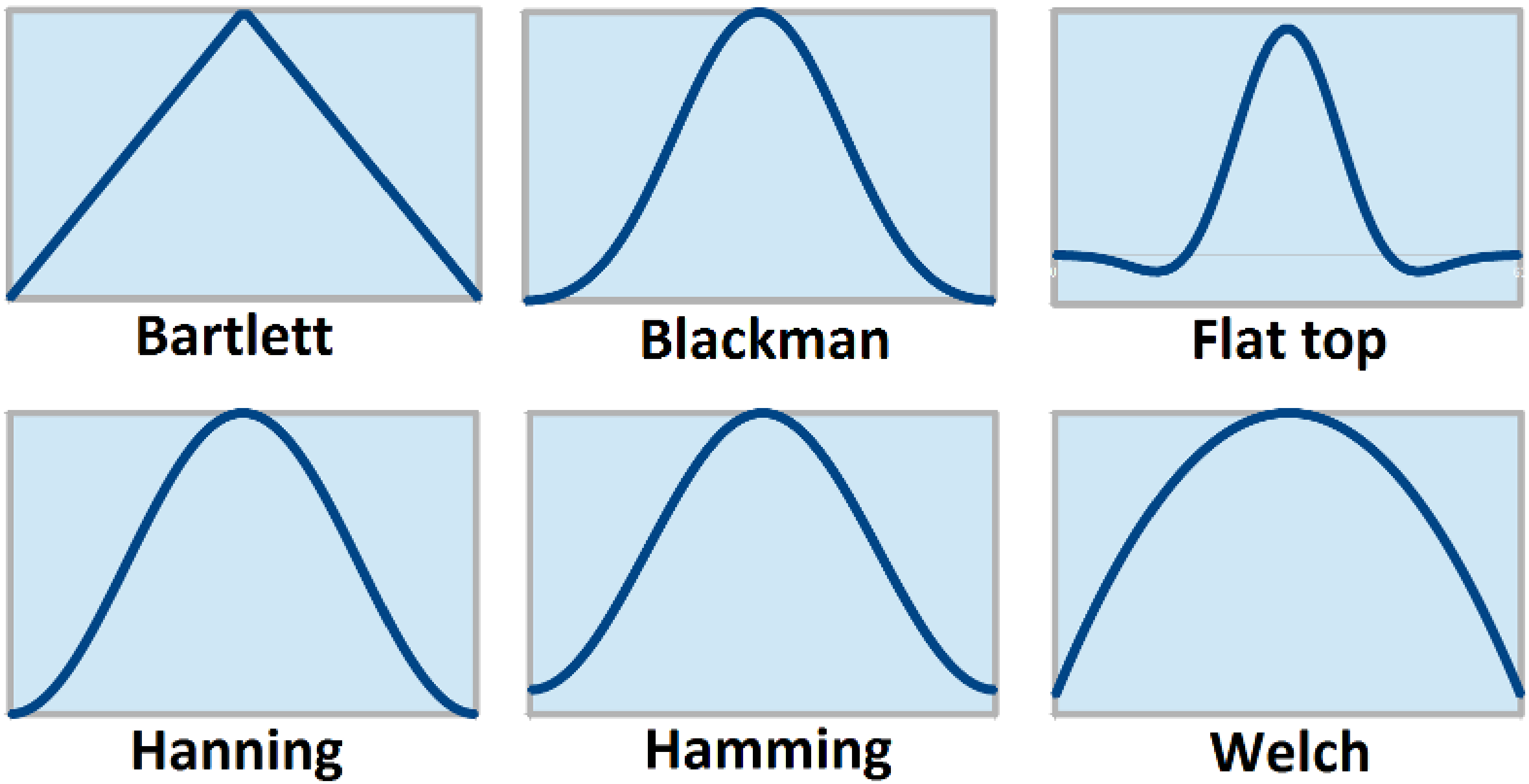
$$u(t) = \frac{1 - \cos(2\pi t / T)}{2}$$



FFT: Window vs. No window



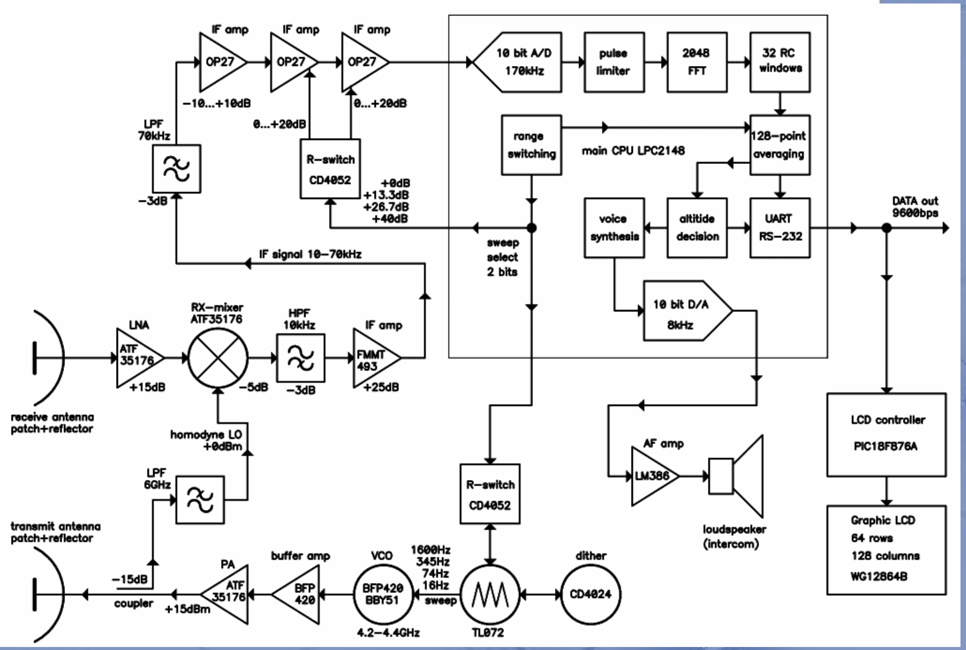
Common window functions



- Window to use depends upon details of your signal, your application, etc.

Last slide: DSP

- DSP = Digital signal processing
- Very important, very large branch of electrical engineering, very math intensive.
  - Radar
  - Sonar
  - Audio processing
  - etc.



Session summary

- Complexity and big-O.
- The FFT – why it is important and why it is fast.
- Filtering using the FFT.
- Nyquist frequency and aliasing.
- Filtering and convolution.
- Windowing.