Udemy DSP Notes

Philip Tracton

December 1, 2017

Contents

1 Introduction

1.1 Course Index

1.1.1 References

- Openheimer and Schafer Digital Signal Processing from Prentice Hall 1975
- Smith The scientist and Engineer's Guide to Digital Signal Processing 1997, see http://www.dspguide.com
- Hayes Schaum's Outline of Digital Signal Processing

1.1.2 Preliminary Concpets

- A signal is a carrier of information. An sort of physical or computable quantity which we give a meaning
- An analog signal can have any value at any time.
- A digital signal is a signal whose values are discrete numbers. Only exist at finite points of time and values.

1. Systems vs. Signals
   - A system is anything that can modify a signal, physically or electrically.
   - Often represented as a function or transform

$$Y = F(x) \tag{1}$$

2. Linear Shift Invariant Systems (LSI)
   - Linear – super position applies

$$F(x) = g \tag{2}$$

$$F(y) = h \tag{3}$$

$$F(ax + by) = ag + bh \tag{4}$$

   - Shift invariant - shape of system output does not vary with starting point. You want this.

- associative and commutative
- Analog to Linear Time Invariant system in time domain

$$if x(n) \rightarrow y(n) then x(n - n_0) \rightarrow y(n - n_0) \tag{5}$$

3. Sequence Properties

   - Periodicity

   A sequence is periodic if there exist a period T such that for all n and k

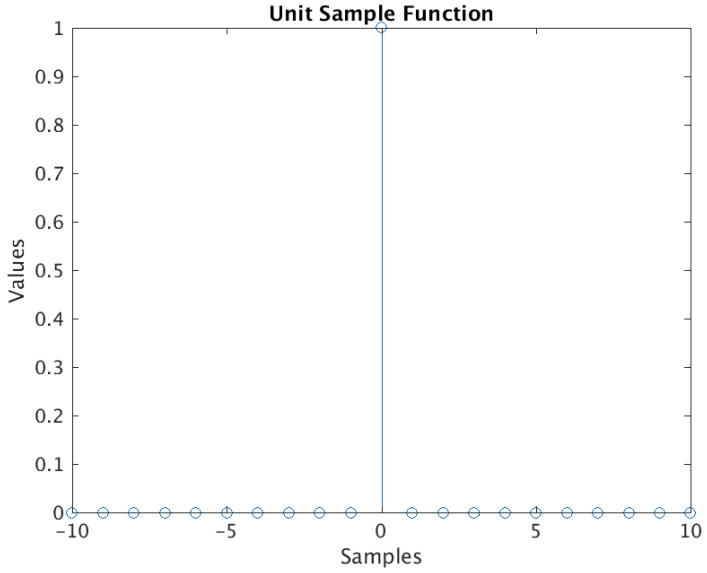
$$x(n + Tk) = x(n) \tag{6}$$

   - Symmetry
     - A sequence is even if for every n $x(n) = x(-n)$ Mirrored in y axis
     - A sequence is odd if for every n, $x(n) = -x(-n)$ Mirrored and x and y axis
     - A sequnce is *conjugate symmetric* if for all n x(n) = x^*(-n) * is complex conjugate
     - A sequnce is *conjugate asymmetric* if for all n x(n) = -x^*(-n)

4. Fundamental Sequences

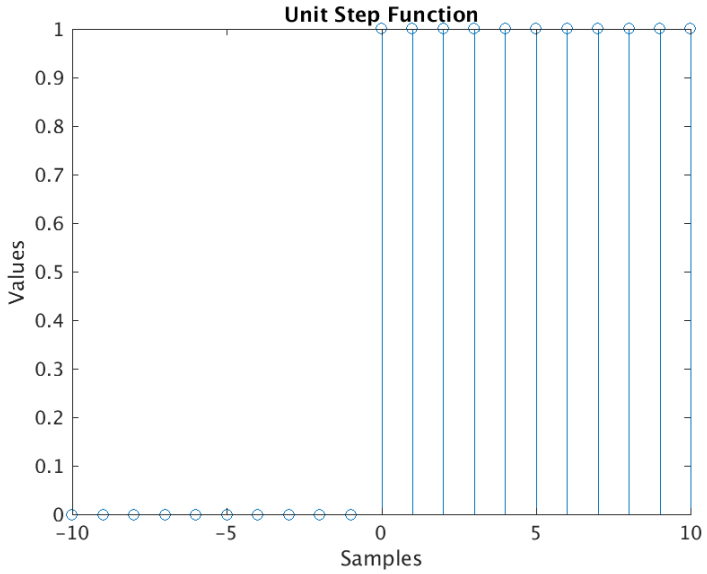
   - Can shift these signals
   - Unit Sample, digital equivalent of Dirac Delta function.

$$\delta(n) = \begin{cases} 1 & n = 0 \\ 0 & otherwise \end{cases} \tag{7}$$



- Unit step, digital equivalent of the Heaviside Step Function

$$u(n) = \begin{cases} 1 & n \geq 0 \\ 0 & otherwise \end{cases} \tag{8}$$



5. Definitions

   - A system is *stable* is the output is bounded for all bounded inputs.
     - $Y = F(x)$ is finite if x is finite
   - A system is *causal* if the current output depends only on the current input, previous inputs or previous outputs. It does **not** need to look into future to work.

6. System Representaion

   - Three common methods of representing a system
     - Difference Equation
     - Block Diagram
     - Signal Flow Graph
   - Difference Equation
     - Digital counter part to a differential equation
     - Represent the output as a combination of previous inputs and outputs

$$y_k = \sum_{n=-\infty}^{k-1} a_n x_n + \sum_{n=-\infty}^{k-1} b_n y_n \tag{9}$$

   - When output depends only on previous input and not previous output the system is known to have *Finite Impulse Response* and we categorize these as FIR. No feedback in system. b coefficients are 0.
   - The converse is also true and those systems are known as *Infinite Impulse Response* or IIR. There is feedback in the system. b coefficients are non zero.

7. Example Problems

   - $y_k = \alpha y_{k-1} + x_k$ where input $x_k = b^k$ for $k \geq 0$
   - Step 1 Homogeneous Solution
     - set equation to be equal to 0
     - $y_k - \alpha y_{k-1} = 0$ which yields the characteristic equation $r - \alpha = 0$
     - Thus $y_{k_h} = c_1 \alpha^k + c_2$
   - Step 2 Particular Solution

  - Undetermined Coefficients, assume $y_{k_p} = c_3 b^k$ due to form of input
  - Plug into difference equations to find coefficient.
    * $c_3 b^k = \alpha c_3 b^{k-1} + b^k$
    * $c_3 = \frac{1}{1-\frac{a}{b}}$
- Step 3 Combine
  - Yields solution $y_k = c_1 \alpha^k + \frac{1}{1-\frac{a}{b}} b^k + c_2$
  - Need to know initial conditions to solve for c_1 and c_2

8. Block Diagrams
   - Represent the flow of information in a visual manner
   - Basic components are unit delay (often z^{-1}), sum and gains/multiply
   - Useful in implementation
9. Signal Flow Graph
   - Similar to block diagram, different appearance
   - Each node is considered summation

2 Preliminaries

2.1 Convolution

2.1.1 The Heart of the Matter

- *Convolution* is a function derived from two given functions by integration which expresses how the shape of one modifies the shape of the other one.
  - Essentially a method of combining two functions.
  - in the continuous domain

$$h(t) = f(t) * g(t) \tag{10}$$

$$h(t) = \int_{-\infty}^{\infty} f(s)g(t-s)ds \tag{11}$$

  - in the discrete/digital domain

$$h(n) = f(n) * g(n) \tag{12}$$

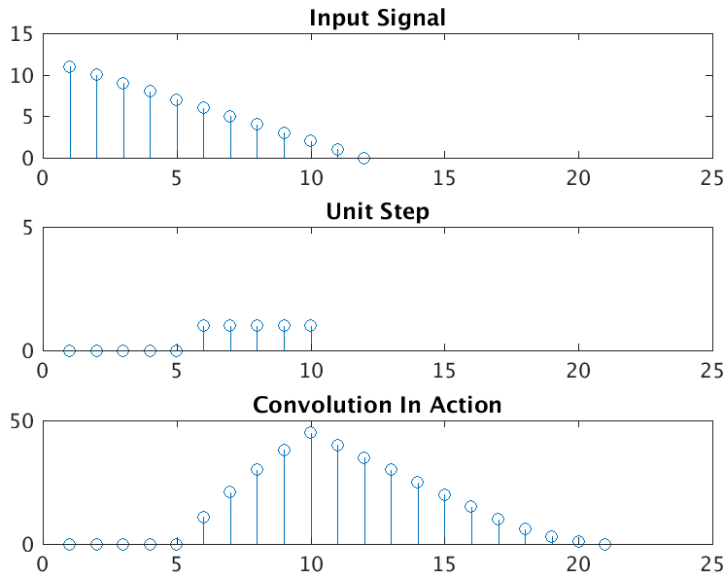
$$h(n) = \sum_{k=-\infty}^{\infty} f(k)g(n-k) \tag{13}$$

- In the case of LSI (linear shift invariant) systems the relationship between input and output can be described as the convolution of the input with the system's impulse response h

$$y(n) = h(n) * x(n) \tag{14}$$

- It is of interest to discover the effects of our design choices which affect h on the output y through convolution with the input
- Finite Disctete systems
  - "Flip and Shift"
  - Take one input sequence, and flip it left right. Starting at first element of the other input sequence, multiply and sum all overlapping elements. This is the value of the output at that point in time.

- The **FUNDAMENTAL** operation of DSP. Without it we cannot afect change in a system via any of the classic methods
- It is a *linear* operator. Order of convolution does not matter. It is associative and distributive with other operation
- It is computationally *expensive* as sequences get large.



1. Convolution Example
   - Is a linear operator which makes it both associative and commutative. Does not matter which sequence if flipped or shifted.
     - F(n) = [0,1,2] and g(n) = [1 1 1 1]
     - Flip F(n) $\rightarrow$ [2,1,0]
     - line up so that the last element of F overlaps the first element of g
     - Process the data $H(n) = \sum_{k=0}^{5} F(k)g(n-k)$
     - Sum up wherever there is an overlap of numbers
     - Keep shifting to the right by 1 for each iteration

     2 1 0
       1 1 1 1

- H(0) = 1* 0 = 0

  2 1 0 $\rightarrow by1$

  1 1 1 1

- H(1) = (0 * 1) + (1 * 1) = 1

  2 1 0

  1 1 1 1

- H(2) = (0 * 1) + (1 * 1) + (2 * 1) = 3

  2 1 0

  1 1 1 1

- H(3) = (0 * 1) + (1 * 1) + (2 * 1) = 3

  2 1

  1 1 1 1

- H(4) = (2 * 1) + (1 * 1) = 3

  2

  1 1 1 1

- H(5) = 2 * 1 = 2
- H(n) = [0 1 3 3 3 2]

```
f = [0 1 2 ]
g = [1 1 1 1]
h = conv(f, g)
```

```
f =

   0   1   2
g =

   1   1   1   1
h =

   0   1   3   3   3   2
```

2.2 Sampling

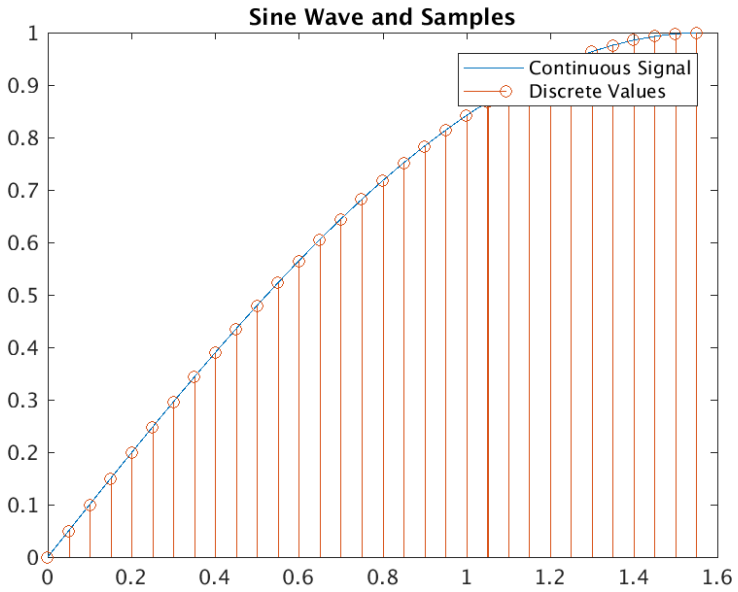
- In nearly all DSP applictions we will concern ourselves with signals that originate in the physical world.
  - Audio signals

  - Video signals
  - pressure, acceleration, attitude sensors

- The physical world is analog. Signals are continuous in both time and value. Must convert to digital format for DSP to be applied to it.
  - This is achieved by a *sampler* which translates a signal value to a discrete number.

2.2.1 ADCs and DACs

- Samplers are usually ADC (analog to digital converter)
  - Many different types with successive approximation being the most common but all function similarly.
- The opposite is a DAC (digital to analog converter)



2.2.2 Sampling Theory

- We can treat each sample as a numeric representation of the input waveform at a particular instant in time.

- We need enough of these samples to encode information about the waveform
- How much is enough?

2.2.3 Shannon-Nyquist Sampling Theorem

- How much is enough is a simple question.
- *The input sequence must be sampled **twice** as fast as the highest frequency input signal*
  - If we habe a 10KHz input signal we must sample at a minimum of 20KHz.
  - The frequency limit (10KHz in this case) is the *Nyquist Frequency*. We can find this by dividing the sampling frequency in half.
  - Any signals above the Nyquist will not be properly samples and even be aliased into the domain of interest.

2.3 Aliasing

2.3.1 How not to sample

- Unfortunate side affect of sampling is *aliasing*.
  - Signal higher than the Nyquist Frequency get reflected back into the sampled frequency band.
    * We sample a signal at 20KHz. It has an 8KHz and 15 KHz components.
    * The 15 KHz is above the Nyquist Frequency (20/2=10KHz) and aliased into the sampled band. The aliased signal is just the original frequency (15) minus the Nyquist (10). 15-10 = 5KHz signal in our samples that does not really exist.

2.3.2 How to sample

- The inclusion of *spurious* signals in our samples is undesirable especially given their unpredictability.
  - We may have a signal of interest at 10Khz and sample at 20KHz as per Nyquist but their might be a 4GHz component that we can't see or hear.

- Solution is to *always include an anti-aliasing filter* in our sampled system designs.
  - Most common technique is a passive network low pass filter with corner frequency around the nyquist rate.
  - **You can NOT do this in the digital domain**
    - Why not? The damage has already been done!

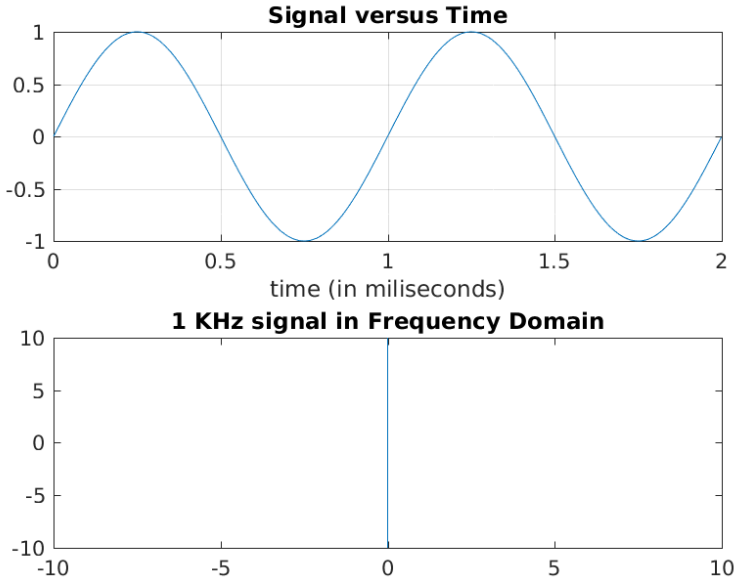
3 Introduction to Matlab

3.1 Using Matlab

4 Transforms

4.1 Time and Frequency Domain

- Signal Domains Time vs. Frequency
  - Up to now we have only considered the time domain.
    - This is where the signal is represented by a sequence of numbers. Each representing the amplitude of the signal at a given point in time.
- Frequency Domain
  - Rather than a sequnce of numbers representing amplitude vs. time we have amplitude vs. frequency.
  - Why is this useful?
    - Signal Inspection
      - Viewing a signal in the frequency domain allows the analyst to very easily discern the which frequencies have the most power.
      - This is useful in filter design where we would like to manipulate the frequency content of a signal to suit our purposes. Can't manipulate what you don't know.
    - Convolution
      - Convolution and Multiplication are duals of each other in different domains
      - convolution in time is multiplication in frequency
      - convolution in frequency is multiplication in time
      - convolution is computationally efficient in frequency domain!



4.2 Convolution Example

- Time to process n data points via convolution in time domain grows at n^2 This is bad!

```
close all;
clear all;
clc;

a = [1 2 3 4 5 6];
b = [1 1 1 1];

z = conv(a,b)  % <-- This is a LOT of steps

fft_a = fft(a)
fft_b = fft(b)

C = fft_a .* fft_b'

ifft(C)
```

```
close all;
clear all;
clc;
```

```
a = [1 2 3 4 5 6];
b = [1 1 1 1];

z = conv(a,b)  % <-- This is a LOT of steps

z =

     1     3     6    10    14    18    15    11     6

fft_a = fft(a)

fft_a =

  Columns 1 through 4

  21.0000 + 0.0000i  -3.0000 + 5.1962i  -3.0000 + 1.7321i  -3.0000 + 0.0000i

  Columns 5 through 6

  -3.0000 - 1.7321i  -3.0000 - 5.1962i
fft_b = fft(b)

fft_b =

     4     0     0     0

C = fft_a .* fft_b'

C =

  Columns 1 through 4

  84.0000 + 0.0000i -12.0000 +20.7846i -12.0000 + 6.9282i -12.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i

  Columns 5 through 6

 -12.0000 - 6.9282i -12.0000 -20.7846i
```

```
   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i

ifft(C)

ans =

  Columns 1 through 4

  21.0000 + 0.0000i  -3.0000 + 5.1962i  -3.0000 + 1.7321i  -3.0000 + 0.0000i
  21.0000 + 0.0000i  -3.0000 + 5.1962i  -3.0000 + 1.7321i  -3.0000 + 0.0000i
  21.0000 + 0.0000i  -3.0000 + 5.1962i  -3.0000 + 1.7321i  -3.0000 + 0.0000i
  21.0000 + 0.0000i  -3.0000 + 5.1962i  -3.0000 + 1.7321i  -3.0000 + 0.0000i

  Columns 5 through 6

  -3.0000 - 1.7321i  -3.0000 - 5.1962i
  -3.0000 - 1.7321i  -3.0000 - 5.1962i
  -3.0000 - 1.7321i  -3.0000 - 5.1962i
  -3.0000 - 1.7321i  -3.0000 - 5.1962i
'org_babel_eoe'

ans =

    'org_babel_eoe'
```

4.3 Enter the DFT

- Convolution and time/from time/frequency domains can be computationally expensive
- Frequency complexity grows slower than time domain especially once n $\geq$ 100 which is not large at all.
- Comparison
  - Convolution is porportional to n^2 and written as $O(n^2)$
  - The transform method is computationally $O(n * log_2(n))$
  - Cross over at 128 points

4.3.1 Fourier Transform

- Originated by Joseph Fourier in 1822
- Analog Fourier Transform

$$F(s) = \int_{-\infty}^{\infty} e^{-2\pi jxs} dx \tag{15}$$

- Converts an analog signal to a sum of coefficients representing its amplitude and phase at each frequency in the frequency domain.
- Transforms the function f(x) from a function in independent variable x to one in independent variable s.
- Discrete Fourier Transform

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi ikn}{N}} \tag{16}$$

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{2\pi ikn}{N}} \tag{17}$$

- Converts a discrete sequence of values to another sequence of values (in general complex) representing amplitude and phase at each frequency up to the Nyquist rate for the sequence.
  - Also transforms from time(n) to frequency(k)

4.4 DFT Computation Example

- After using the DFT on a signal we can get the frequency response of the signal

$$|X_k| = \sqrt{Re(X_k)^2 + Im(X_k)^2} \tag{18}$$

$$\theta_k = tan^{-1} \frac{Im(X_k)}{Re(X_k)} \tag{19}$$

- Notice the reflection of the response due to aliasing above the Nyquist rate.

4.4.1 Fast Fourier Transform

- Cooley and Tukey in 1965
- Reduces complexity via symmetry
  - Original DFT $O(n^2)$
  - FFT $O(nlog(n))$
- Reference Free C Implementation

4.4.2 Application to Convolution

- As mentioned previously the DFT cna be used to perform convolution.
  - Side Effect: convolution is circular and not linear
    - This can be the same answer in certain conditions
    - We generally want linear convolution since circular does not give proper results when filtering
  - In order to do it linearly there must be enough room (*padding*) in the 2 signals
    - Choose a period size N for both sequences that is equal to or greater than L + M -1 where L and M are the sizes of the 2 sequences.
    - Power of 2 is good, FFT is fastest this way.
    - Extend each input to N samples and set those extra samples to 0

4.5 The Z Transform

- Z Transform is a generalization of the DFT
  - Z plane is digital complex plain
  - S plane is the analog complex plain
  - Much like Laplace is to the analog Fourier Transform
  - The DFT is equivalent to the Z Transform evaluated on a unit circle
- What is it?

$$X(z) = \sum_n x[n]z^{-n} \tag{20}$$

- Powers of z are often interpreted as delays in the system where the power represents the number of times the signal is delayed.
  - z^{-2} is a two clock cycle delay
- Offers a powerful method of system description and analysis
  - Can obtain a rational *transfer function* of a system that will allow us to find *poles* and *zeros* of a system and determine its behavior.

4.5.1 Common Z Tranform Pairs

1. **TODO** Find the list and put it here

4.5.2 Transfer Functions

- Consider a system with impulse response h
  - The output of such a system can be obtained from the input such that

$$y(n) = x(n) * h(n) \tag{21}$$

- If we take the Z transform of the system we can obtain a *transfer function*

$$\frac{Y(z)}{X(z)} = H(z) \tag{22}$$

- When we form H(z) as a rational polynomial function (a ratio of 2 polynomial functions) we can easily discern the *poles* and *zeros* of the system.

4.5.3 Poles and Zeros

- Definitions
  - *Zero* of a system is a value of the complex frequency Z that will for the transfer function to a 0
  - *Pole* is a value of z which will cause the transfer function to become unbounded, i.e. infinite
- Finding them can be accomplished via analysis of the transfer function
  - Given a rational transfer function

$$H(z) = \frac{b_m z^m + ...b_1 Z^1 + b_0}{a_n z^n + ... + a_1 z^1 + a_0} \tag{23}$$

- We can consider the numerator and denominator as separate equations
  - The roots of the numerators are the *zeros*
  - The roots of the denominator are the *poles*, the characteristic equation
- We can factor the equation to arrive at a different form

$$H(z) = \frac{(z - z_1)(z - z_2)}{.}..(z - z_m)(z - p_1)(z - p_2)...(z - p_n) \tag{24}$$

- Here z_1 through z_m are the *zeros* and p_1 through p_n are the *poles*

4.5.4 Pole Placement and Response

- Poles outside the unit circle grow exponentially
- Poles inside the unit circle decay
- Poles right on unit circle are *marginally stable* and oscillate
- The characteristic equation is so called dueo the polse major effect on the behavior of the system
  - The figure shows how systems with different pole locations react to the exact same input (an impulse)
  - It should be noted that while pole placement dominates the characterization of the system resposne zeros also influence it.
  - Many filters are designed specifically to *not* use poles. This has certain performance advantages in real time systems.

5 FFT Convolution

5.1 FFT Convolution

5.1.1 Manual Steps of doing FFT and getting the example code working

```
% n = 16 so we get linear convolution output is 12 samples
% Need tp pad to get to 16 samples lone in each
a = [1 1 1 0 0 0 1 1 1];
b = [1 0 1 1];

x = [1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0];% manually added 0s, not good
                                      % for large sequences

clear x;

x = [a 0 0 0 0 0 0 0];
x = [a zeros(1,7)];
y = [b, zeros(1,12)];

A = fft(a, 16);
B = fft(b, 16);
X = fft(x);
Y = fft(y);

% . operations do element by element
C = A .* B;
Z = X .* Y;
Zi = ifft(Z)
Ci = ifft(C,16)
```


```
% n = 16 so we get linear convolution output is 12 samples
% Need tp pad to get to 16 samples lone in each
a = [1 1 1 0 0 0 1 1 1];
b = [1 0 1 1];

x = [1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0];% manually added 0s, not good
% for large sequences

clear x;

x = [a 0 0 0 0 0 0 0];
x = [a zeros(1,7)];
y = [b, zeros(1,12)];

A = fft(a, 16);
```

```
B = fft(b, 16);
X = fft(x);
Y = fft(y);

% . operations do element by element
C = A .* B;
Z = X .* Y;
Zi = ifft(Z)

Zi =

  Columns 1 through 7

    1.0000    1.0000    2.0000    2.0000    2.0000    1.0000    1.0000

  Columns 8 through 14

    1.0000    2.0000    2.0000    2.0000    1.0000         0         0

  Columns 15 through 16

         0    0.0000
Ci = ifft(C,16)

Ci =

  Columns 1 through 7

    1.0000    1.0000    2.0000    2.0000    2.0000    1.0000    1.0000

  Columns 8 through 14

    1.0000    2.0000    2.0000    2.0000    1.0000         0         0

  Columns 15 through 16

         0    0.0000
'org_babel_eoe'

ans =
```

```
'org_babel_eoe'
```

6 Digital Filters

6.1 Ideal Filters and Specifications

6.1.1 What we know so far

- A 'filter' for us will be
  - Linear Shift Invariant (LSI)
  - Discrete
  - Describable by difference equation and a z-domain transfer function
- It will also have a specific purpose
  - Let some portion of input pass and blocking others
  - Most often interested in filtering portions of a signals bandwidth

6.1.2 Definitions

- **Passband** is the group of frequencies which a filter allows to pass through with minimal alteration
- **Stopband** is the group of frequencies the filter blocks
- **Filter Order** is the number of delays necessary to implement the filter

6.1.3 Filter Types

- **Lowpass** is when the frequencies below the "corner" frequency are allowed to pass
- **Highpass** is when the frequencies above the "corner" frequency are allowed to pass
- **Bandpass** is when the frequencies in between the 2 "corner" frequencies are allows to pass
- **Bandstop or Notch** is when the frequencies outside of the 2 "corner" frequencies are allows to pass
- **Transition Band** the window between the corner frequency and the stop band.
- **Stop Band** the range of frequencies that are blocked

6.1.4 Ideal Filters

- No such thing in practice!
- An ideal low pass filter would be one with a frequency response as shown
  - A completely "flat" response in the passband and stop band
  - Instant transition from pass to stop (sharpness)
  - Zero phase/group delay
    - Signal does not get delayed going through filter

1. **TODO** Create image

6.1.5 Filter Specification

- Must specify exactly what they do
- Typically Include the following
  - **Stopband Attenuation** is how little of the unwanted signal should the filter allow through usually in dB
  - **Corner or Cutoff Frequency** is the frequency at which the passband ends and the stopband begins
  - **Passband Ripple** is how much the amplitude of passed frequencies should vary in the passband
  - **Phase Lag or Group Delay** is how much an input signal should be slowed down by passing through the filter.

1. **TODO** Create image

6.2 IIR Design

6.2.1 Digital Frontier

- In digital filters there are 2 classifications of filters
  - Infinite Impulse Response (IIR)
    - Present and previous values needed to process, this is the pole!
    - Refers to a system/filter that has poles in its transfer function

        * All analog filter fall into this category as it is not possible to synthethize an analog filter without poles
        * Can **not** be realized by FFT (due to *infinite* impulse response)
            · Can be done if choose a finite portion of the infinite response.
        * Easier to make and work better
    - Finite Impulse Response (FIR)
        * No feedback, only current input used
        * A system/filter without polse
        * Generally better time domain performance (fewer calculations, less memory) than IIR. Lower effective cost.

6.2.2 Techniques for IIR Design

- By far the most common technique for IIR digital filter design is to perofrm the design in the analog domain
    - The amount of literature and number of techniques available for analog filters makes it redundant to attempt to duplicate in digital domain
    - Relative easy method of transforming an analog filter's transfer function to a digital transfer function via Tustin's Bilinear Transformation, makes IIR simple when starting with an analog filter.
- Our design process is to find a digital transfer functions whose frequency response is as close a possible to the analog prototype.

6.2.3 Bilinear Transform

- Defined as

$$S = \frac{2}{T_s}\frac{1 - z^{-1}}{1 + z^{-1}} \tag{25}$$

- T_s is the sampling period
- Implement a filter in the LaPlace S-domain and plug this equation in for s to make it digital

- This gives us a direct method of substitution to obtain a digital transfer function from an analog one (which would be a function of the complex frequency s)
- Bilinear transform has the propert that it maps the entire imaginary axis of the s-plane to the unit circle of the z-plan.
  - This means we have an *infinitely long line* mapped to a finite length curve.
  - As such the imaginary axis mist be shortened to fit
  - The result is called *frequency warping*. The further we move around the unit circle after using the transform, the bigger the difference in the analog and digital frequencies.

6.2.4 Frequency Warping

- The relationship between analog frequency and digital under the bilinear transform is

$$\omega_d = tan^{-1}(\frac{\omega_a T_s}{2}) \tag{26}$$

- We can see the distortion gets more pronounced the close the analog frequency gets to the Nyquist rate ($f_N = \frac{2}{T_s}$).
- How to get around this?
  - We can *prewarp* our frequency constraints before we design the analog filter so that we end up with the desired performance in the digital filter.
    - Prewarping is accomplished by inverting the warping relation ship in 26

$$\omega_a T_s = 2tan(\frac{\omega_d}{2}) \tag{27}$$

- NOTE: Because we are design the filter after we prewarp we can choose an arbitrary sampling period here which is indepdent of the actual sampling period of the implementation of the filter.
- Thus with a good choice of sampling period the prewarp equation becomes

$$\omega_a = tan(\frac{\omega_d}{2}) \tag{28}$$

6.2.5 The Recipe for Filter Success

- With the bilinear transform and prewarping formulas in hand we are ready to design an IIR
  - Specify the filter performance criteria as necessary
  - Prewarp corner frequencies
  - Design an analog filter which meets the performance criteria at the *prewarp frequencies*
    - Lots of material about this online, not covered in this course
  - Substitute s in the transfer functions of the analog filter using the bilinear transformation
  - Use resulting digital transfer functions to implement the filter in software/hardware

6.3 IIR Design Example

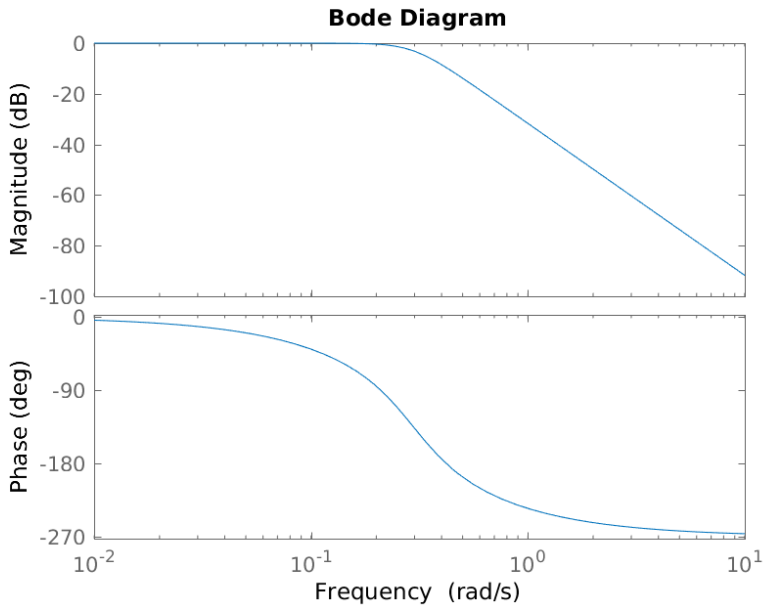
6.3.1 TODO The video is a repeat of the previous lesson. This has been true for over 6 months

```
%% Startup and Globals
clear all;
clc;

%% Filter Parameters
fs = 40000;

wc = 2000;
ws = 12000;
ripple = 0.1;
atten = 40;

%% Intermediate Calculations for Butterworth Filter Design
del = 10^(ripple/20) - 1;
A = 10^(atten/20);
eps = sqrt((1/(1-del))^2 - 1);

wcd = wc/fs*2*pi;
wsd = ws/fs*2*pi;

wc_warp = tan(wcd/2);
ws_warp = tan(wsd/2);

d = eps/sqrt(A^2 - 1);
k = wc_warp/ws_warp;
del_s = sqrt(d^2/((1-del)^(-2) - 1));

N = ceil(log(d)/log(k));
```

```
29
30  % Pre-warping of frequencies
31  wc_min = wc_warp*((1-del)^(-2) - 1)^(-1/(2*N));
32  wc_max = ws_warp*((del_s)^(-2) - 1)^(-1/(2*N));
33
34  wn = (wc_max + wc_min)/2;
35
36  %% Actual Filter Design
37
38  % Hey! Look at that, MATLAB has a function to design Butterworth Filters!
39  [b,a] = butter(N,wn,'s');
40
41  % This uses the coefficients to create a continuous time system object
42  sys = tf(b,a);
43  bode(sys);
44
45  % And this handy function converts that continuous time systems to
46  % a digital one for us, using the bilinear transformation.
47  d_sys = c2d(sys,2,'zoh')
48
49  saveas(gcf, '../Notes/images/iir_filter.png')
```

```
%% Startup and Globals
clear all;
clc;

%% Filter Parameters
fs = 40000;

wc = 2000;
ws = 12000;
ripple = 0.1;
atten = 40;

%% Intermediate Calculations for Butterworth Filter Design
del = 10^(ripple/20) - 1;
A = 10^(atten/20);
eps = sqrt((1/(1-del))^2 - 1);

wcd = wc/fs*2*pi;
wsd = ws/fs*2*pi;

wc_warp = tan(wcd/2);
ws_warp = tan(wsd/2);

d = eps/sqrt(A^2 - 1);
```

```
k = wc_warp/ws_warp;
del_s = sqrt(d^2/((1-del)^(-2) - 1));

N = ceil(log(d)/log(k));

% Pre-warping of frequencies
wc_min = wc_warp*((1-del)^(-2) - 1)^(-1/(2*N));
wc_max = ws_warp*((del_s)^(-2) - 1)^(-1/(2*N));

wn = (wc_max + wc_min)/2;

%% Actual Filter Design

% Hey! Look at that, MATLAB has a function to design Butterworth Filters!
[b,a] = butter(N,wn,'s');

% This uses the coefficients to create a continuous time system object
sys = tf(b,a);
bode(sys);

% And this handy function converts that continuous time systems to
% a digital one for us, using the bilinear transformation.
d_sys = c2d(sys,2,'zoh')

d_sys =

  0.02554 z^2 + 0.07528 z + 0.01412
  ---------------------------------
  z^3 - 1.849 z^2 + 1.27 z - 0.3058

Sample time: 2 seconds
Discrete-time transfer function.

saveas(gcf, '../Notes/images/iir_filter.png')
'org_babel_eoe'

ans =

    'org_babel_eoe'
```



https://www.youtube.com/watch?v=3yyp5JRqNXs butter

6.4 FIR Design

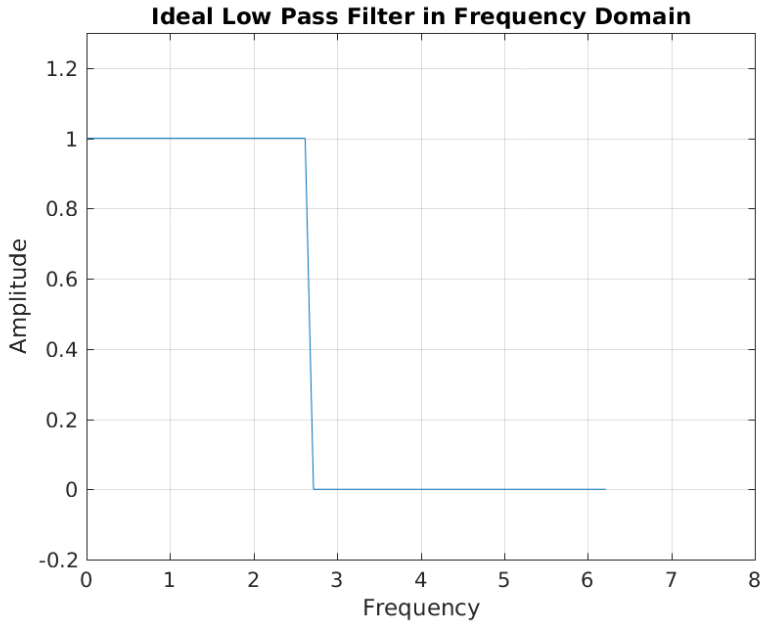
- There is no analog counterpart
- The FIR Filter has a transfer function with *no poles*. This equates to *no feedback* in the filter and certain advantages
  - Guaranteed to be stable due to lack of feedback
  - Simple to make the phase response linear
  - Efficient to compute in real time via FFT
- Several methods available to FIR design
  - Window/Truncation of an ideal filter
  - Frequency Domain sampling and an iFFT
  - Least Squares approximation
- Only discussing windowing method - simpler

6.4.1 More on our Ideals

- The ideal low pass filter frequecy response and it's time domain impulse response are shown
  - Ideally phase is 0
  - Note that while we could sample and use time impulse response it is both infinite and non-causal meaning it can not be used for real time 'on-line' filtering

```
x = -2*pi:.1:2*pi;
y = [ones(1,90),zeros(1,36)];
plot(x,y)
grid on
title('Ideal Low Pass Filter in Frequency Domain')
axis ([0,8 -0.2 1.3])
xlabel('Frequency')
ylabel('Amplitude')
saveas(gcf, '../Notes/images/ideal_low_pass_time_frequency.png')
```

```
org_babel_eoe
```

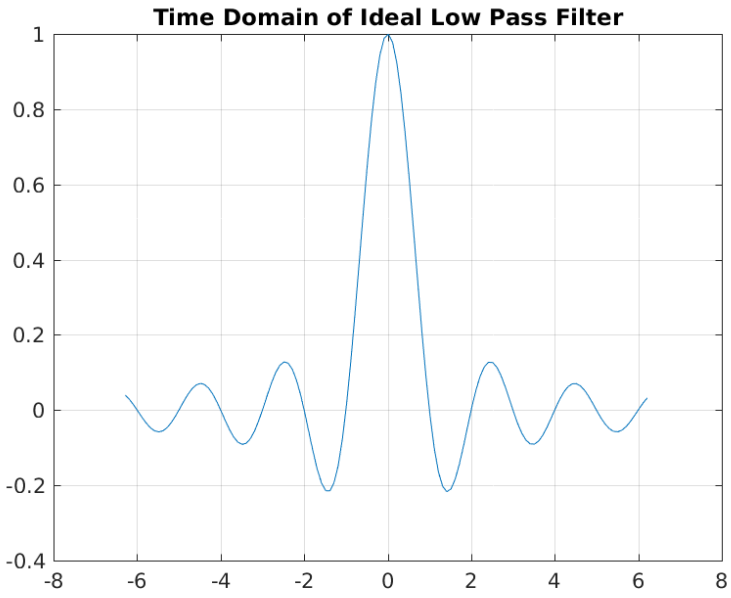


- Sinc funcion is infinite and can not fit in a computer's memory

- Sinc function is non-causal. There are samples that depend on both the past (left of y-axis) and future (right of y-axis)

```
x = -2*pi:.1:2*pi;
y = sinc(x);
plot(x,y)
grid on
title('Time Domain of Ideal Low Pass Filter')
ylabel('Amplitude')
xlabel('Time')
saveas(gcf, 'Notes/images/ideal_low_pass_time_domain.png')
```

```
org_babel_eoe
```



6.4.2 A Window to Filter Through

- By multiplying the ideal filter response(which is a sinc wave) by a proper window function we can truncate it
  - Thus our filter's impulse response will become finite
  - Once finite we can also shift the filter's response to the left t make is causal
  - *Which window do we use?*

- More samples in your window the worse the phase performance from the order of the filter being higher but more steep cutoff

1. **TODO** Get images

6.4.3 Comparison of Windows

- Different windows have different frequecy responses
  - The ultimate share of the FIR filter's response will be decided by the shape of the window's response
  - Tow Major Factors
    * Width of main lobe determines the sharpness of the transition band
    * Amplitude of the side lobes determines the amount of ripple in the final filter frequency response.

Window	Side Lobe Amplitude (dB)	Transition Width (Δ f)	Stop Band Attenuation
Rectangular	-13	0.9/N	-21
Hanning	-31	3.1/N	-44
Hamming	-41	3.3/N	-53
Blackman	-57	5.5/N	-74

- We can trade the length of the window (N) for eitehr computation speed or sharpness of transition
- We can trade the shapre of the window (Blackman vs. Hamming, etc...) for total attenuation and passband ripple
- Side lobe amplitude affects ripple, higher side lobe is more ripple
- Rectangular Window

$$w(n) = \begin{cases} 1 & 0 \leq n \leq N \\ 0 & else \end{cases} \quad (29)$$

- Hanning Window

$$w(n) = \begin{cases} 0.5 - 0.5cos(\frac{2\pi n}{N}) & 0 \leq n \leq N \\ 0 & else \end{cases} \quad (30)$$

- Hamming Window

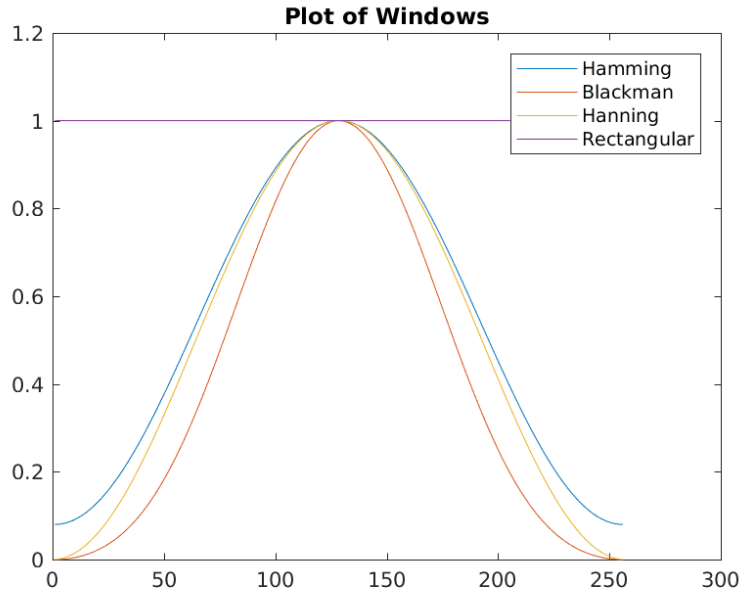
$$w(n) = \begin{cases} 0.54 - 0.46cos(\frac{2\pi n}{N}) & 0 \leq n \leq N \\ 0 & else \end{cases} \quad (31)$$

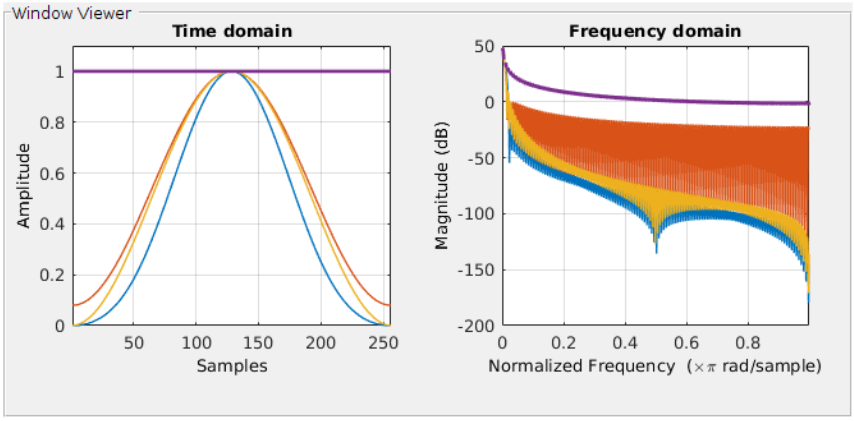
- Blackman Window

$$w(n) = \begin{cases} 0.42 - 0.5cos(\frac{2\pi n}{N}) - 0.08cos(\frac{4\pi n}{N}) & 0 \leq n \leq N \\ 0 & else \end{cases} \quad (32)$$

```
ham_window = hamming(256);
black_window = blackman(256);
han_window = hanning(256);
rect_window = rectwin(256)
plot(ham_window)
hold on
plot(black_window)
plot(han_window)
plot(rect_window)
axis([0 300 0 1.2])
title('Plot of Windows')
legend('Hamming', 'Blackman', 'Hanning', 'Rectangular');
saveas(gcf, '../Notes/images/windowing_plots.png')
```

```
org_babel_eoe
```





1. **TODO** Get image

6.4.4 Final Notes

- Specify filter requirements
  - Passband end frequency
  - Stopband start frequency
  - Stopband attenuation
  - Ripple requirements
- Select a window which will meet requirements
- Use requirements to construct ideal filter response
  - Set transition frequency to midpoint of passband end and stopband start
- Translate ideal filter response to infinite impulse response via sinc function

$$h(n) = \frac{\omega_c}{\pi} \frac{sin(\omega_c n)}{\omega_c n} \tag{33}$$

- Multiply the impulse response by a window
- Shift to make it causal (if necessary for real time processing)

6.5 FIR Design Example

- Let's consider a filter which
  - Is sampled at 100Khz
  - Passband up to 20KHz
  - Stopband at 40Khz down to 40dB
  - Minimize Ripple in Passband
- Given these specs, a rectangular window will not work for stop band attenuation
  - Try the other 3
- Find ideal response
  - Corner Frequency 30 Khz since it is midpoint between passband and stop band!
  - 30 KHz is 0.6π
  - $h(n) = 0.6\frac{sin(0.6\pi n}{0.6\pi n}$
  - Truncate via window functions and test frequency response

6.5.1 TODO Need code to re-create the images

6.5.2 FIR Discussion

- All window methods result in linear phase filters
  - The linear phase is due to the *shift* we employ to make the filter causal.
  - Without such a shift (filter after signal is captured) we can effectively have very little degradation to the phase of the signal.
    - This is typical of image processing where the entire signal is captured at once and then processed
  - Very good transition response was attained with relatively small filters
    - In our example the length N was set to 100 samples which is computationally efficient
  - There are more advanced methods which offer increased performance

    * In general the least squares approximation to an ideal filter will result in a filter of the lowest order for a given specification
    * Frequency domain sampling can also offer better frequency domain performance over windowing.
    * Both are computationally more difficult

6.6 Filter Realization

- Discuss how to implement in software.
  - Hardare (ASIC and FPGA) outside scope of course.
  - Should be easy to move from SW to HW

6.6.1 Transfer Functions and Difference Equations

- Instructor finds it easiest to implement filter from a difference equation
  - Convert transfer function and impulse response to difference equations and then writing the algorithm from the difference equation
- Transfer Function
  - What does 'z' mean?
    * The complex variable 'z' can be treated as a **shift** operator. Positive powers shift forward in time and negative backwards in time.
    * z^{-1} is a time delay of a single clock cycle.
  - We can rearrange any transfer function so that is shows the outputs as a function of current input, past inputs and past outputs

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_M b_m z^{-m}}{1 + \sum_N a_n z^{-n}} \tag{34}$$

$$Y(z) = X(z) \sum_M b_m z^{-m} - Y(z) \sum_N a_n z^{-n} \tag{35}$$

- Take the inverse Z Transform yields

$$y(n) = b_m x(n) + b_{m-1} x(n-1) + ... + a_n y(n-1) + ... \tag{36}$$

- x(n) is current input
- x(n-1) is previous input
- y(n-1) is previous output

6.6.2 Impulse Response

- For FIR we can proceed directly from the impulse response to difference equation
  - Each simple in impulse response represents a coefficient b_m
  - We multiply that coefficient by a shift equal to its distance from the origin

1. **TODO** make image

6.6.3 From Difference Equation to Software

- General outline here to be language agnostic
  - Matlab example will follow
- The basic algorithm for IIR Filters
  - Set initial values
    - These can be predetermined or simply read from the system for agiven number of clock cycles.
    - Need as many initial conditions as the order of the filter or number of delays in filter
  - Filter Loop
    - Read current input
    - Calculate output based on difference equation for filter
    - Return output
    - Store output and current input to be used for next cycle
- FIR Filter follow same process but no need to store output values.
  - Only rely on previos and current inputs

7 Filter Implementation

7.1 Filter Implementation

```
%% Startup and Globals

% ------------------------------------------------------------------------
% FilterImplementation.m
%
% Uses standard looping methods to implement an IIR filter in
% software.
%
% Copyright Jake Bailey, August 2015. All Rights Reserved.
%
% ------------------------------------------------------------------------

clear all;
close all;
clc;

%% Filter Parameters
% Using low-pass from course example
%        0.02554 z^2 + 0.07528 z + 0.01412
% H(z) = ---------------------------------
%        z^3 - 1.849 z^2 + 1.27 z - 0.3058
%
% The difference equation will look like:
% Y(n) = 0.02554*x(n-1) + 0.07528*x(n-2) + 0.01412*x(n-3)
% +1.849*y(n-1) - 1.27*y(n-2) + 0.3058*y(n-3)
% Coefficient arrays

b = [0.01412 0.07528 0.02554];
a = [-0.3058 1.27 -1.849 1];

%% Create some input

fs = 40000;
length = 100000;
n = 1:length;

% This input will give us two frequency peaks, one inside the passband and
% one in the stopband.

input = sin(94247.*(n./fs)) + sin(12566.*(n./fs));

% Show the original input, and that its frequency response is what we
% expect.

plotlength = 500;

% Plot original input
plot(n(1:plotlength),input(1:plotlength));
title('Original Plot')
saveas(gcf, '../Notes/images/filter_implementation_original.png')

```

```
H = fft(input);
mag = sqrt(real(H).^2 + imag(H).^2);
phase = atan2(imag(H),real(H));

freq_axis = pi.*n./length;
freq_length = length/2;

figure;

% Frequency Response magnitude and phase
subplot(2,1,1);
title('Original Magnitude and Phase')
plot(freq_axis(1:freq_length),mag(1:freq_length));

subplot(2,1,2);
plot(freq_axis(1:freq_length),phase(1:freq_length));
saveas(gcf, '../Notes/images/filter_implementation_original_mag_phase.png')

%% Main Filter Loop

% Initialize the sample arrays

x = zeros(1,length);
y = zeros(1,length);

% Note we're only running for a set period of time
% In real time applications you will often set this
% to an infinite while loop [i.e. while(true)]

for i = 1:length

 % Read input
  x(i) = input(i);

  % Skip until we have enough samples to process
  if i < 4

    continue %Keyword to skip this iteration of the loop
  end

  % Calculate output via our difference equation
  y(i) = (1/a(4))*(b(3)*x(i-1) + b(2)*x(i-2) + b(1)*x(i-3) - a(3)*y(i-1) - a(2)*y(i-2) - a(1)*y(i-3));

end


figure;
% Output
plot(n(1:plotlength),y(1:plotlength));
saveas(gcf, '../Notes/images/filter_implementation_filtered.png')
H = fft(y);
mag = sqrt(real(H).^2 + imag(H).^2);
phase = atan2(imag(H),real(H));

freq_axis = pi.*n./length;
freq_length = length/2;
```

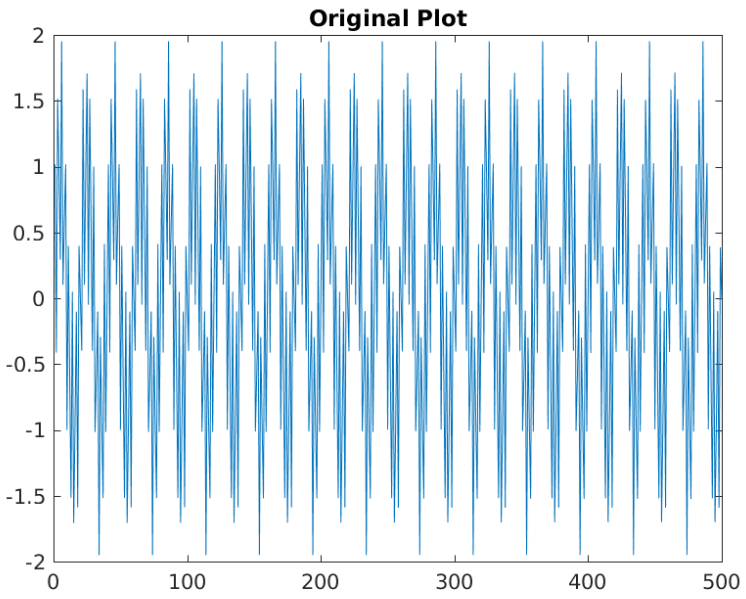
```
108
109 figure;
110
111 % mag and phase response of filtered output
112 subplot(2,1,1);
113 plot(freq_axis(1:freq_length),mag(1:freq_length));
114
115 subplot(2,1,2);
116 plot(freq_axis(1:freq_length),phase(1:freq_length));
117
118 saveas(gcf, '../Notes/images/filter_implementation_filtered_mag_phase.png')
```

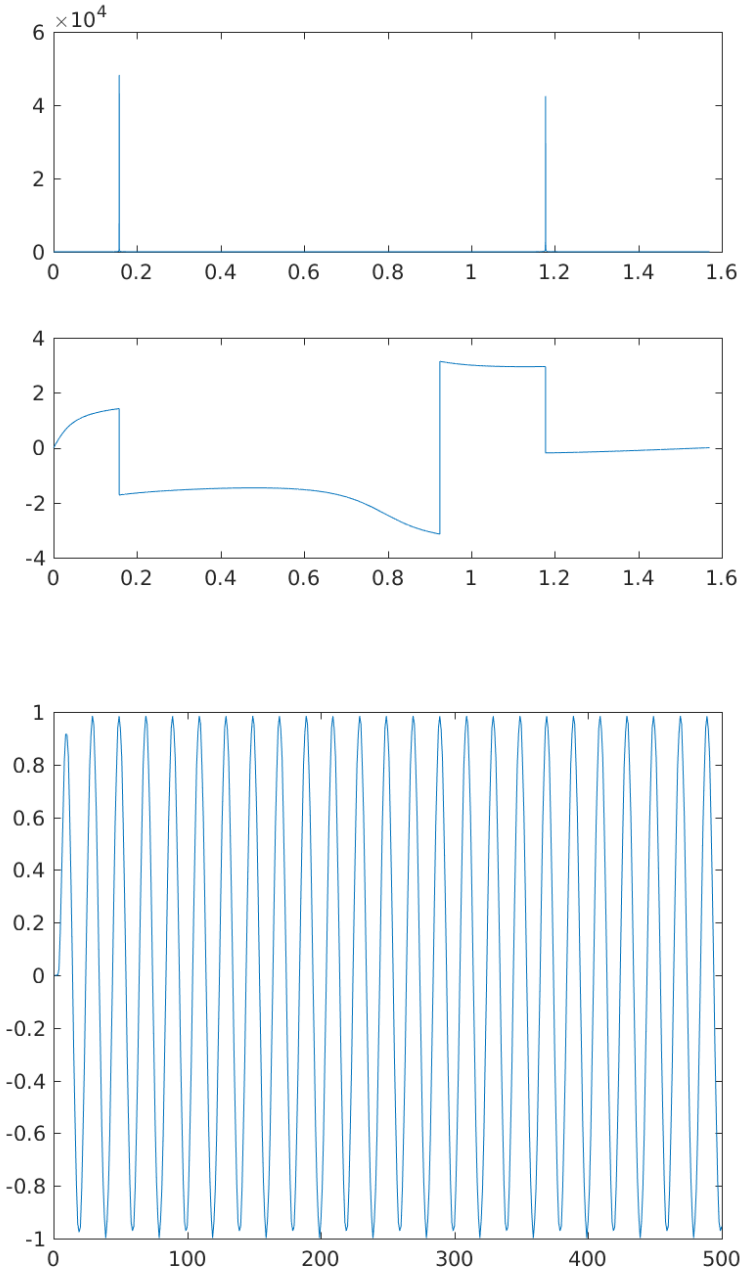
```
%% Startup and Globals

% ------------------------------------------------------------------------
% FilterImplementation.m
%
% Uses standard looping methods to implement an IIR filter in
% software.
%
% Copyright Jake Bailey, August 2015. All Rights Reserved.
%
% ------------------------------------------------------------------------

clear all;
close all;
clc;

%% Filter Parameters
% Using low-pass from course example
%        0.02554 z^2 + 0.07528 z + 0.01412
% H(z) = ---------------------------------
%        z^3 - 1.849 z^2 + 1.27 z - 0.3058
%
% The difference equation will look like:
% Y(n) = 0.02554*x(n-1) + 0.07528*x(n-2) + 0.01412*x(n-3)
% +1.849*y(n-1) - 1.27*y(n-2) + 0.3058*y(n-3)
% Coefficient arrays

b = [0.01412 0.07528 0.02554];
a = [-0.3058 1.27 -1.849 1];

%% Create some input
```

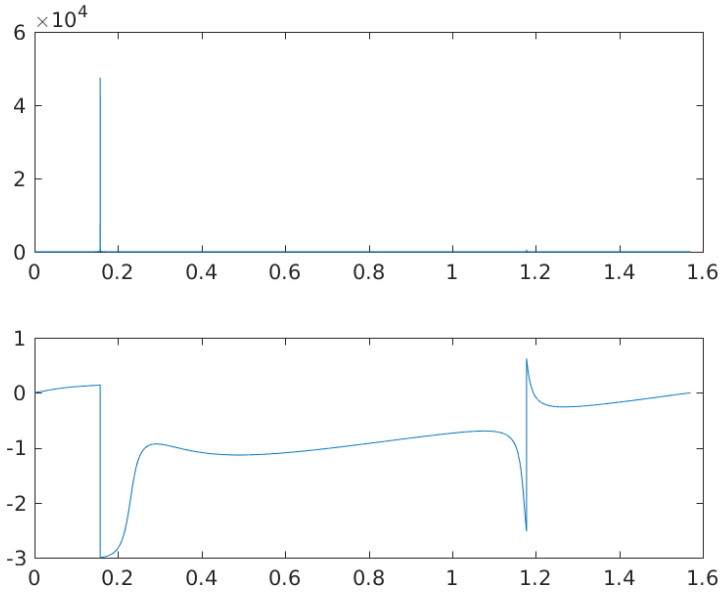
```
fs = 40000;
length = 100000;
n = 1:length;

% This input will give us two frequency peaks, one inside the passband and
% one in the stopband.

input = sin(94247.*(n./fs)) + sin(12566.*(n./fs));

% Show the original input, and that its frequency response is what we
% expect.

plotlength = 500;

% Plot original input
plot(n(1:plotlength),input(1:plotlength));
title('Original Plot')
saveas(gcf, '../Notes/images/filter_implementation_original.png')

H = fft(input);
mag = sqrt(real(H).^2 + imag(H).^2);
phase = atan2(imag(H),real(H));

freq_axis = pi.*n./length;
freq_length = length/2;

figure;

% Frequency Response magnitude and phase
subplot(2,1,1);
title('Original Magnitude and Phase')
plot(freq_axis(1:freq_length),mag(1:freq_length));

subplot(2,1,2);
plot(freq_axis(1:freq_length),phase(1:freq_length));
saveas(gcf, '../Notes/images/filter_implementation_original_mag_phase.png')

%% Main Filter Loop
```

```
% Initialize the sample arrays

x = zeros(1,length);
y = zeros(1,length);

% Note we're only running for a set period of time
% In real time applications you will often set this
% to an infinite while loop [i.e. while(true)]

for i = 1:length

 % Read input
  x(i) = input(i);

  % Skip until we have enough samples to process
  if i < 4

    continue %Keyword to skip this iteration of the loop
  end

  % Calculate output via our difference equation
  y(i) = (1/a(4))*(b(3)*x(i-1) + b(2)*x(i-2) + b(1)*x(i-3) - a(3)*y(i-1) - a(2)*y(i-2

end


figure;
% Output
plot(n(1:plotlength),y(1:plotlength));
saveas(gcf, '../Notes/images/filter_implementation_filtered.png')
H = fft(y);
mag = sqrt(real(H).^2 + imag(H).^2);
phase = atan2(imag(H),real(H));

freq_axis = pi.*n./length;
freq_length = length/2;

figure;

% mag and phase response of filtered output
```

```
subplot(2,1,1);
plot(freq_axis(1:freq_length),mag(1:freq_length));

subplot(2,1,2);
plot(freq_axis(1:freq_length),phase(1:freq_length));

saveas(gcf, '../Notes/images/filter_implementation_filtered_mag_phase.png')
'org_babel_eoe'

ans =

    'org_babel_eoe'
```




×10^4
6
4
2
0
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
4
2
0
-2
-4
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
100
200
300
400
500


×10^4
6
4
2
0
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1
0
-1
-2
-3
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6

8 IIR Design Example and Code

- How to get transfer function coefficients

8.1 Specific Matlab Functions

- butter
- bode
- pzmap
- tf

```
%% Startup and Globals
clear all;
clc;

%% Filter Parameters
fs = 40000;  % 40 KHz

wc = 2000; % 2KHz
ws = 12000; % 12 Khz
ripple = 0.1; % in dB
atten = 40;  % in dB

%% Intermediate Calculations for Butterworth Filter Design
del = 10^(ripple/20) - 1;
A = 10^(atten/20);
eps = sqrt((1/(1-del))^2 - 1);

wcd = wc/fs*2*pi;
wsd = ws/fs*2*pi;

wc_warp = tan(wcd/2);
ws_warp = tan(wsd/2);

d = eps/sqrt(A^2 - 1);
k = wc_warp/ws_warp;
del_s = sqrt(d^2/((1-del)^(-2) - 1));

N = ceil(log(d)/log(k));

% Pre-warping of frequencies
wc_min = wc_warp*((1-del)^(-2) - 1)^(-1/(2*N));
wc_max = ws_warp*((del_s)^(-2) - 1)^(-1/(2*N));

wn = (wc_max + wc_min)/2;

%% Actual Filter Design

% Hey! Look at that, MATLAB has a function to design Butterworth Filters!
[b,a] = butter(N,wn,'s')
```

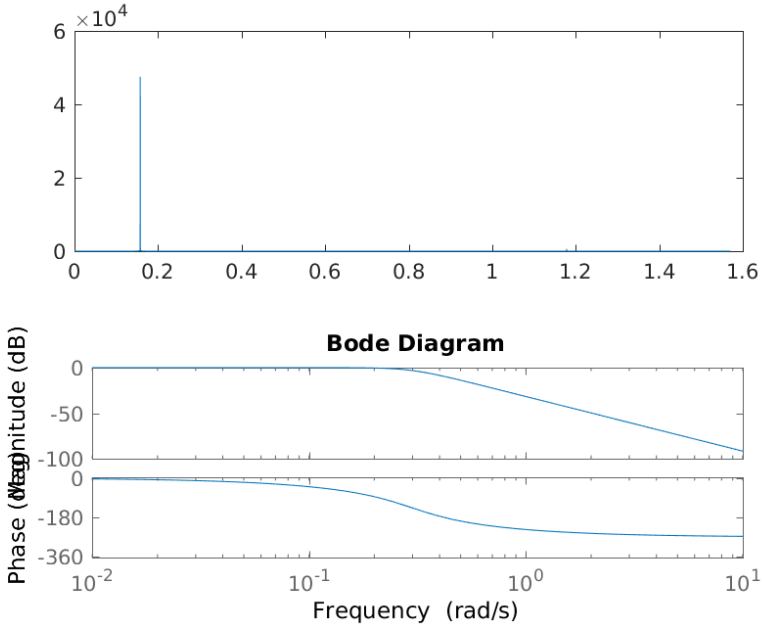
```
40
41  % This uses the coefficients to create a continuous time system object
42  sys = tf(b,a)
43  bode(sys);
44
45  % And this handy function converts that continuous time systems to
46  % a digital one for us, using the bilinear transformation.
47  d_sys = c2d(sys,2,'zoh')
48
49  % Find the poles and zeros
50  pz = pzmap(sys)
51
52  saveas(gcf, '../Notes/images/bode.png')
```

```
%% Startup and Globals
clear all;
clc;

%% Filter Parameters
fs = 40000;  % 40 KHz

wc = 2000; % 2KHz
ws = 12000; % 12 Khz
ripple = 0.1; % in dB
atten = 40;  % in dB

%% Intermediate Calculations for Butterworth Filter Design
del = 10^(ripple/20) - 1;
A = 10^(atten/20);
eps = sqrt((1/(1-del))^2 - 1);

wcd = wc/fs*2*pi;
wsd = ws/fs*2*pi;

wc_warp = tan(wcd/2);
ws_warp = tan(wsd/2);

d = eps/sqrt(A^2 - 1);
k = wc_warp/ws_warp;
del_s = sqrt(d^2/((1-del)^(-2) - 1));

N = ceil(log(d)/log(k));
```

```
% Pre-warping of frequencies
wc_min = wc_warp*((1-del)^(-2) - 1)^(-1/(2*N));
wc_max = ws_warp*((del_s)^(-2) - 1)^(-1/(2*N));

wn = (wc_max + wc_min)/2;

%% Actual Filter Design

% Hey! Look at that, MATLAB has a function to design Butterworth Filters!
[b,a] = butter(N,wn,'s')

b =

         0         0         0    0.0260


a =

    1.0000    0.5923    0.1754    0.0260

% This uses the coefficients to create a continuous time system object
sys = tf(b,a)

sys =

                0.02598
  -------------------------------------
  s^3 + 0.5923 s^2 + 0.1754 s + 0.02598

Continuous-time transfer function.
bode(sys);

% And this handy function converts that continuous time systems to
% a digital one for us, using the bilinear transformation.
d_sys = c2d(sys,2,'zoh')

d_sys =

  0.02554 z^2 + 0.07528 z + 0.01412
  ---------------------------------
```

```
  z^3 - 1.849 z^2 + 1.27 z - 0.3058

Sample time: 2 seconds
Discrete-time transfer function.

% Find the poles and zeros
pz = pzmap(sys)

pz =

  -0.2962 + 0.0000i
  -0.1481 + 0.2565i
  -0.1481 - 0.2565i

saveas(gcf, '../Notes/images/bode.png')
'org_babel_eoe'

ans =

    'org_babel_eoe'
```

- Bode Plot



9 FIR Design and Example Code

9.1 Matlab Functions

- freqz
- blackman
- hamming
- rectwin
- hann

```
%% Startup and Globals

clear all
clc
close all
n = -100000:100000;

h = zeros(1,length(n));

for i=1:length(n)
    if n(i) == 0
        h(i) = 1;
    else
        h(i) = sin(0.6*pi*n(i))/(0.6*pi*n(i));
    end
end

% This step ensures the DC gain of the filter is 0 (no change to DC
% signals).
gain = sum(h);
h = h./gain;

%% Hanning Window

hann_N = 100;
w_hann = zeros(1,hann_N);
h_hann = zeros(1,hann_N);

midpoint = ceil(length(n)/2);
window_start = midpoint - ceil(hann_N/2) - 1;
for i=1:hann_N
    w_hann(i) = 0.5 - 0.5*cos(2*pi*n(i)/hann_N);
    h_hann(i) = w_hann(i)*h(i+ window_start);
end

% This step ensures the DC gain of the filter is 0 (no change to DC
% signals).
gain = sum(h_hann);
h_hann = h_hann./gain;
```

```
40
41  %% Hamming Window
42
43  hamm_N = 100;
44  w_hamm = zeros(1,hamm_N);
45  h_hamm = zeros(1,hamm_N);
46
47  window_start = midpoint - ceil(hamm_N/2) - 1;
48
49  for i=1:hamm_N
50      w_hamm(i) = 0.54 - 0.46*cos(2*pi*n(i)/hamm_N);
51      h_hamm(i) = w_hamm(i)*h(i + window_start);
52  end
53
54  % This step ensures the DC gain of the filter is 0 (no change to DC
55  % signals).
56  gain = sum(h_hamm);
57  h_hamm = h_hamm./gain;
58
59  %% Blackman Window
60
61  black_N = 100;
62  w_black = zeros(1,black_N);
63  h_black = zeros(1,black_N);
64
65  window_start = midpoint - ceil(black_N/2) - 1;
66
67  for i=1:black_N
68      w_black(i) = 0.42 - 0.5*cos(2*pi*n(i)/black_N) + 0.08*cos(4*pi*n(i)/black_N);
69      h_black(i) = w_black(i)*h(i + window_start);
70  end
71
72  % This step ensures the DC gain of the filter is 0 (no change to DC
73  % signals).
74  gain = sum(h_black);
75  h_black = h_black./gain;
76
77  %% Comparing Frequency Responses
78  % Freqz will easily calculate the frequency response of an FIR filter.
79  [a, b] = freqz(h);
80  [a_hann, b_hann] = freqz(h_hann);
81  [a_hamm, b_hamm] = freqz(h_hamm);
82  [a_black, b_black] = freqz(h_black);
83  close all;
84
85  %% Plot results and save
86  figure;
87  hold on;
88  subplot(2,1,1);
89  plot(b,10*log(abs(a)));
90  title('Magnitude - Ideal');
91  ylabel('dB');
92  subplot(2,1,2);
93  phase = atan2(imag(a),real(a));
94  plot(b,phase);
95  title('Phase - Ideal');
```

```
96  ylabel('Radians');
97  saveas(gcf,'../Notes/images/Filter_Comparison_1.png');
98
99  figure;
100 subplot(2,1,1);
101 plot(b,10*log(abs(abs(a_hann))));
102 title('Magnitude - Hanning');
103 ylabel('dB');
104 subplot(2,1,2);
105 phase = atan2(imag(a_hann),real(a_hann));
106 plot(b,phase);
107 title('Phase - Hanning');
108 ylabel('Radians');
109 saveas(gcf,'../Notes/images/Filter_Comparison_2.png');
110
111 figure;
112 subplot(2,1,1);
113 plot(b,10*log(abs(abs(a_hamm))));
114 title('Magnitude - Hamming');
115 ylabel('dB');
116 subplot(2,1,2);
117 phase = atan2(imag(a_hamm),real(a_hamm));
118 plot(b,phase);
119 title('Phase - Hamming');
120 ylabel('Radians');
121 saveas(gcf,'../Notes/images/Filter_Comparison_3.png');
122
123 figure;
124 subplot(2,1,1);
125 plot(b,10*log(abs(abs(a_black))));
126 title('Magnitude - Blackman');
127 ylabel('dB');
128 subplot(2,1,2);
129 phase = atan2(imag(a_black),real(a_black));
130 plot(b,phase);
131 title('Phase - Blackman');
132 ylabel('Radians');
133 saveas(gcf,'../Notes/images/Filter_Comparison_4.png');
134
135 %close all;
```

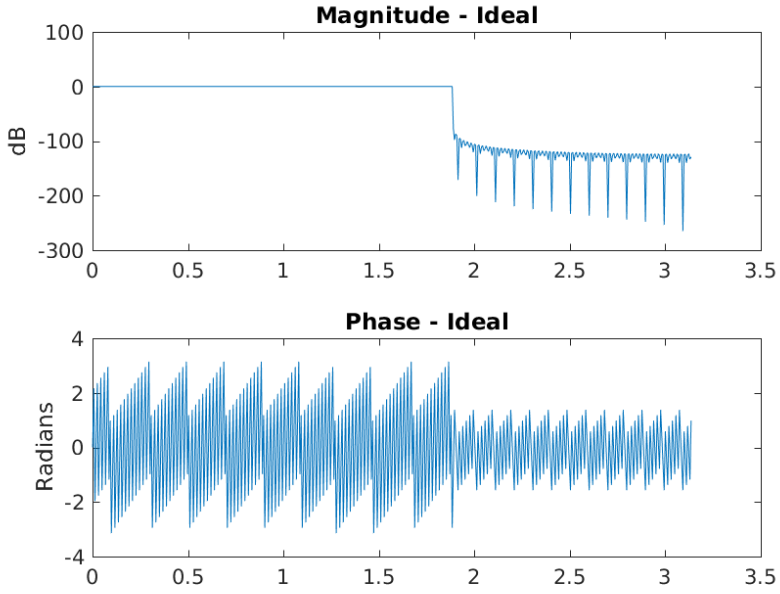
```
%% Startup and Globals

clear all
clc
close all
n = -100000:100000;

h = zeros(1,length(n));

for i=1:length(n)
```

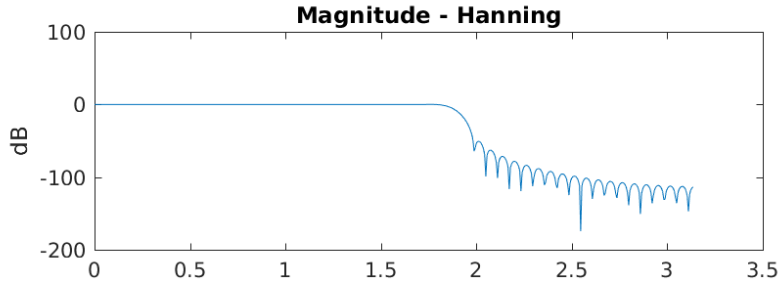
```
    if n(i) == 0
        h(i) = 1;
    else
        h(i) = sin(0.6*pi*n(i))/(0.6*pi*n(i));
    end
end

% This step ensures the DC gain of the filter is 0 (no change to DC
% signals).
gain = sum(h);
h = h./gain;

%% Hanning Window

hann_N = 100;
w_hann = zeros(1,hann_N);
h_hann = zeros(1,hann_N);

midpoint = ceil(length(n)/2);
window_start = midpoint - ceil(hann_N/2) - 1;
for i=1:hann_N
    w_hann(i) = 0.5 - 0.5*cos(2*pi*n(i)/hann_N);
    h_hann(i) = w_hann(i)*h(i+ window_start);
end

% This step ensures the DC gain of the filter is 0 (no change to DC
% signals).
gain = sum(h_hann);
h_hann = h_hann./gain;

%% Hamming Window

hamm_N = 100;
w_hamm = zeros(1,hamm_N);
h_hamm = zeros(1,hamm_N);

window_start = midpoint - ceil(hamm_N/2) - 1;

for i=1:hamm_N
    w_hamm(i) = 0.54 - 0.46*cos(2*pi*n(i)/hamm_N);
```

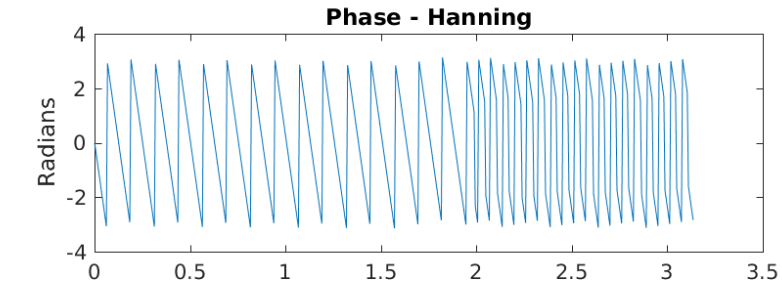
```
    h_hamm(i) = w_hamm(i)*h(i + window_start);
end

% This step ensures the DC gain of the filter is 0 (no change to DC
% signals).
gain = sum(h_hamm);
h_hamm = h_hamm./gain;

%% Blackman Window

black_N = 100;
w_black = zeros(1,black_N);
h_black = zeros(1,black_N);

window_start = midpoint - ceil(black_N/2) - 1;

for i=1:black_N
    w_black(i) = 0.42 - 0.5*cos(2*pi*n(i)/black_N) + 0.08*cos(4*pi*n(i)/black_N);
    h_black(i) = w_black(i)*h(i + window_start);
end

% This step ensures the DC gain of the filter is 0 (no change to DC
% signals).
gain = sum(h_black);
h_black = h_black./gain;

%% Comparing Frequency Responses
% Freqz will easily calculate the frequency response of an FIR filter.
[a, b] = freqz(h);
[a_hann, b_hann] = freqz(h_hann);
[a_hamm, b_hamm] = freqz(h_hamm);
[a_black, b_black] = freqz(h_black);
close all;

%% Plot results and save
figure;
hold on;
subplot(2,1,1);
plot(b,10*log(abs(a)));
title('Magnitude - Ideal');
```

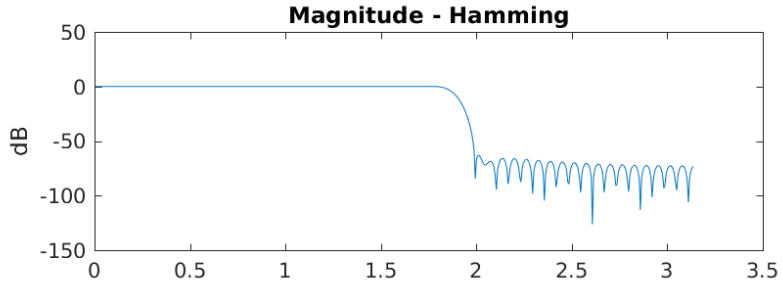
```
ylabel('dB');
subplot(2,1,2);
phase = atan2(imag(a),real(a));
plot(b,phase);
title('Phase - Ideal');
ylabel('Radians');
saveas(gcf,'../Notes/images/Filter_Comparison_1.png');

figure;
subplot(2,1,1);
plot(b,10*log(abs(abs(a_hann))));
title('Magnitude - Hanning');
ylabel('dB');
subplot(2,1,2);
phase = atan2(imag(a_hann),real(a_hann));
plot(b,phase);
title('Phase - Hanning');
ylabel('Radians');
saveas(gcf,'../Notes/images/Filter_Comparison_2.png');

figure;
subplot(2,1,1);
plot(b,10*log(abs(abs(a_hamm))));
title('Magnitude - Hamming');
ylabel('dB');
subplot(2,1,2);
phase = atan2(imag(a_hamm),real(a_hamm));
plot(b,phase);
title('Phase - Hamming');
ylabel('Radians');
saveas(gcf,'../Notes/images/Filter_Comparison_3.png');

figure;
subplot(2,1,1);
plot(b,10*log(abs(abs(a_black))));
title('Magnitude - Blackman');
ylabel('dB');
subplot(2,1,2);
phase = atan2(imag(a_black),real(a_black));
plot(b,phase);
```

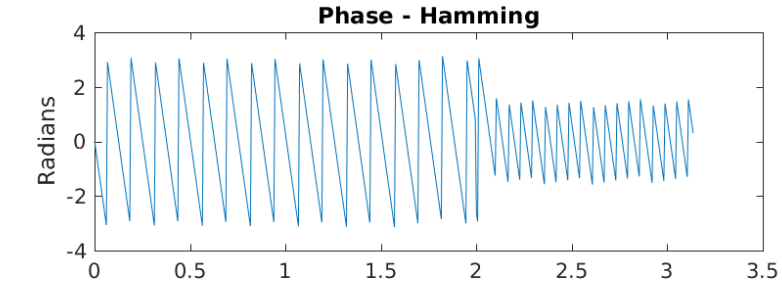
```
title('Phase - Blackman');
ylabel('Radians');
saveas(gcf,'../Notes/images/Filter_Comparison_4.png');

%close all;
'org_babel_eoe'

ans =

    'org_babel_eoe'
```

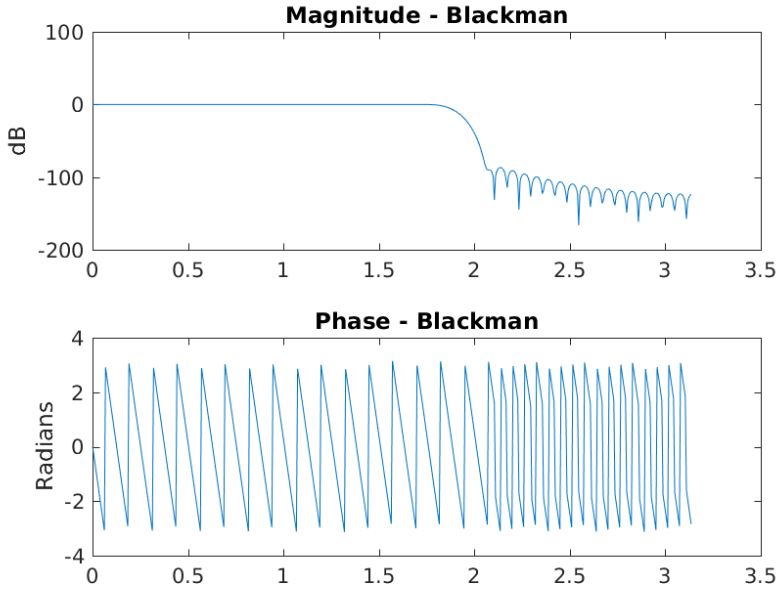



Magnitude - Hanning
dB
100
0
-100
-200
0
0.5
1
1.5
2
2.5
3
3.5


Phase - Hanning
Radians
4
2
0
-2
-4
0
0.5
1
1.5
2
2.5
3
3.5


Magnitude - Hamming
dB
50
0
-50
-100
-150
0
0.5
1
1.5
2
2.5
3
3.5


Phase - Hamming
Radians
4
2
0
-2
-4
0
0.5
1
1.5
2
2.5
3
3.5


Magnitude - Blackman
dB
100
0
-100
-200
0
0.5
1
1.5
2
2.5
3
3.5
Phase - Blackman
Radians
4
2
0
-2
-4
0
0.5
1
1.5
2
2.5
3
3.5

10 Image Filtering

10.1 Matlab Functions

- fspecial
- imshow
- imread
- rgb2gray
- conv2
- fft2
- ifft2
- tic
- toc

10.2 Filtering

- Working in greyscale is easier, in color you need to work on all 3 colors
- low pass filter is a blurring effect

```
%% Startup and Globals

clear all
clc
close all

%% Filter Parameters
width = 3840;
height = 2160;
n = 1000;
sig = 5;

% Read in the image
A = imread('../M Files/canyon.jpg');
% Convert it to grayscale to make things slightly simpler
A = rgb2gray(A);
% Display it.
imshow(A);

% Convert the image to floating point numbers
A = double(A);
% Create a gaussian kernel (similar to a sinc wave in 2D, gives us
```

```
% lowpass characteristics
H = fspecial('gaussian',n, sig);

% Do the filtering via direct convolution (and time it with tic/toc)
tic
A_conv = conv2(A,H,'same');
toc

% Display the result
figure;
imshow(A_conv, []);
%% FFT Convolution

% Dot he filterting via FFT convolution (and time it with tic/toc)
tic
A_freq = fft2(A,height+n-1,width+n-1);
H_freq = fft2(H,height+n-1,width+n-1);

Out_freq = A_freq.*H_freq;

Out = ifft2(Out_freq);
toc

% Display the result
figure;
imshow(Out,[]);
```

```
%% Startup and Globals

clear all
clc
close all

%% Filter Parameters
width = 3840;
height = 2160;
n = 1000;
sig = 5;

% Read in the image
A = imread('../M Files/canyon.jpg');
% Convert it to grayscale to make things slightly simpler
A = rgb2gray(A);
% Display it.
imshow(A);
[Warning: Image is too big to fit on screen; displaying at 33%]
[> In images.internal.initSize (line 71)
```

```
  In imshow (line 328)]

% Convert the image to floating point numbers
A = double(A);
% Create a gaussian kernel (similar to a sinc wave in 2D, gives us
% lowpass characteristics
H = fspecial('gaussian',n, sig);

% Do the filtering via direct convolution (and time it with tic/toc)
tic
A_conv = conv2(A,H,'same');
toc
Elapsed time is 5.282587 seconds.

% Display the result
figure;
imshow(A_conv, []);
[Warning: Image is too big to fit on screen; displaying at 33%]
[> In images.internal.initSize (line 71)
  In imshow (line 328)]
%% FFT Convolution

% Dot he filterting via FFT convolution (and time it with tic/toc)
tic
A_freq = fft2(A,height+n-1,width+n-1);
H_freq = fft2(H,height+n-1,width+n-1);

Out_freq = A_freq.*H_freq;

Out = ifft2(Out_freq);
toc
Elapsed time is 2.799704 seconds.

% Display the result
figure;
imshow(Out,[]);
[Warning: Image is too big to fit on screen; displaying at 25%]
[> In images.internal.initSize (line 71)
  In imshow (line 328)]
'org_babel_eoe'
```

```
ans =

    'org_babel_eoe'
```

11 Music Filtering

11.1 Matlab Functions

- audioread
- sound
- pause
- filter

```
%% Esatblish Our Parameters

% Number of poles for each filter
n_poles = 10;

% Where our corner will be
cutoff_frequency = 0.015;

% For use with bandpass/bandstop, we need two corners
freq_range = [0.10 0.4];

% The range of audio data we will use
start_sec = 10;
end_sec = 15;

%% Read Audio Data

% Read the actual file on disk
[data, Fs] = audioread('OVERWERK - After Hours - 01 Daybreak.flac');

% Credit to OVERWERK - http://www.overwerk.com/ for his great music
% Chop the data so we're only listening to 5 seconds instead of 6 minutes
chopped_data = data(Fs*start_sec:Fs*end_sec, :);

%% First Filter - Butterworth Lowpass

% Use MATLAB's built in butter function to compute the coefficients of the
% Butterworth filter

[b, a] = butter(n_poles,cutoff_frequency);

% Another MATLAB built-in, filter, will apply the filter we just designed
% to the audio data
filtered_song = filter(b,a,chopped_data);

% Play the music unedited first, then filtered
sound(chopped_data,Fs);
pause(end_sec + 1 - start_sec);
sound(filtered_song,Fs);
pause(end_sec + 1 - start_sec);

%% Second Filter - Butterworth Highpass
```

```
43
44  % Use MATLAB's built in butter function to compute the coefficients of the
45  % Butterworth filter
46  [b, a] = butter(n_poles,cutoff_frequency,'high');
47
48  % Another MATLAB built-in, filter, will apply the filter we just designed
49  % to the audio data
50  filtered_song = filter(b,a,chopped_data);
51
52  % Play the music unedited first, then filtered
53  %sound(chopped_data,Fs);
54  %pause(end_sec + 1 - start_sec);
55  sound(filtered_song,Fs);
56  pause(end_sec + 1 - start_sec);
57
58  %% Third Filter - Butterworth Bandpass
59
60  % Use MATLAB's built in butter function to compute the coefficients of the
61  % Butterworth filter
62  [b, a] = butter(n_poles,freq_range,'bandpass');
63
64  % Another MATLAB built-in, filter, will apply the filter we just designed
65  % to the audio data
66  filtered_song = filter(b,a,chopped_data);
67
68  % Play the music unedited first, then filtered
69  %sound(chopped_data,Fs);
70  %pause(end_sec + 1 - start_sec);
71  sound(filtered_song,Fs);
72  pause(end_sec + 1 - start_sec);
73
74  %% Final Filter - Butterworth Bandstop
75
76  % Use MATLAB's built in butter function to compute the coefficients of the
77  % Butterworth filter
78  [b, a] = butter(n_poles,freq_range,'stop');
79
80  % Another MATLAB built-in, filter, will apply the filter we just designed
81  % to the audio data
82  filtered_song = filter(b,a,chopped_data);
83
84  % Play the music unedited first, then filtered
85  %sound(chopped_data,Fs);
86  %pause(end_sec + 1 - start_sec);
87  sound(filtered_song,Fs);
88  pause(end_sec + 1 - start_sec);
```

```
%% Esatblish Our Parameters

% Number of poles for each filter
n_poles = 10;

% Where our corner will be
```

```
cutoff_frequency = 0.015;

% For use with bandpass/bandstop, we need two corners
freq_range = [0.10 0.4];

% The range of audio data we will use
start_sec = 10;
end_sec = 15;

%% Read Audio Data

% Read the actual file on disk
[data, Fs] = audioread('OVERWERK - After Hours - 01 Daybreak.flac');
{Error using audioread (line 74)
The filename specified was not found in the MATLAB path.
}

% Credit to OVERWERK - http://www.overwerk.com/ for his great music
% Chop the data so we're only listening to 5 seconds instead of 6 minutes
chopped_data = data(Fs*start_sec:Fs*end_sec, :);
{Undefined function or variable 'data'.
}

%% First Filter - Butterworth Lowpass

% Use MATLAB's built in butter function to compute the coefficients of the
% Butterworth filter

[b, a] = butter(n_poles,cutoff_frequency);

% Another MATLAB built-in, filter, will apply the filter we just designed
% to the audio data
filtered_song = filter(b,a,chopped_data);
{Undefined function or variable 'chopped_data'.
}

% Play the music unedited first, then filtered
sound(chopped_data,Fs);
{Undefined function or variable 'chopped_data'.
}
```

```
pause(end_sec + 1 - start_sec);
sound(filtered_song,Fs);
{Undefined function or variable 'filtered_song'.
}
pause(end_sec + 1 - start_sec);

%% Second Filter - Butterworth Highpass

% Use MATLAB's built in butter function to compute the coefficients of the
% Butterworth filter
[b, a] = butter(n_poles,cutoff_frequency,'high');

% Another MATLAB built-in, filter, will apply the filter we just designed
% to the audio data
filtered_song = filter(b,a,chopped_data);
{Undefined function or variable 'chopped_data'.
}

% Play the music unedited first, then filtered
%sound(chopped_data,Fs);
%pause(end_sec + 1 - start_sec);
sound(filtered_song,Fs);
{Undefined function or variable 'filtered_song'.
}
pause(end_sec + 1 - start_sec);

%% Third Filter - Butterworth Bandpass

% Use MATLAB's built in butter function to compute the coefficients of the
% Butterworth filter
[b, a] = butter(n_poles,freq_range,'bandpass');

% Another MATLAB built-in, filter, will apply the filter we just designed
% to the audio data
filtered_song = filter(b,a,chopped_data);
{Undefined function or variable 'chopped_data'.
}

% Play the music unedited first, then filtered
%sound(chopped_data,Fs);
```

```
%pause(end_sec + 1 - start_sec);
sound(filtered_song,Fs);
{Undefined function or variable 'filtered_song'.
}
pause(end_sec + 1 - start_sec);

%% Final Filter - Butterworth Bandstop

% Use MATLAB's built in butter function to compute the coefficients of the
% Butterworth filter
[b, a] = butter(n_poles,freq_range,'stop');

% Another MATLAB built-in, filter, will apply the filter we just designed
% to the audio data
filtered_song = filter(b,a,chopped_data);
{Undefined function or variable 'chopped_data'.
}

% Play the music unedited first, then filtered
%sound(chopped_data,Fs);
%pause(end_sec + 1 - start_sec);
sound(filtered_song,Fs);
{Undefined function or variable 'filtered_song'.
}
pause(end_sec + 1 - start_sec);
'org_babel_eoe'

ans =

    'org_babel_eoe'
```