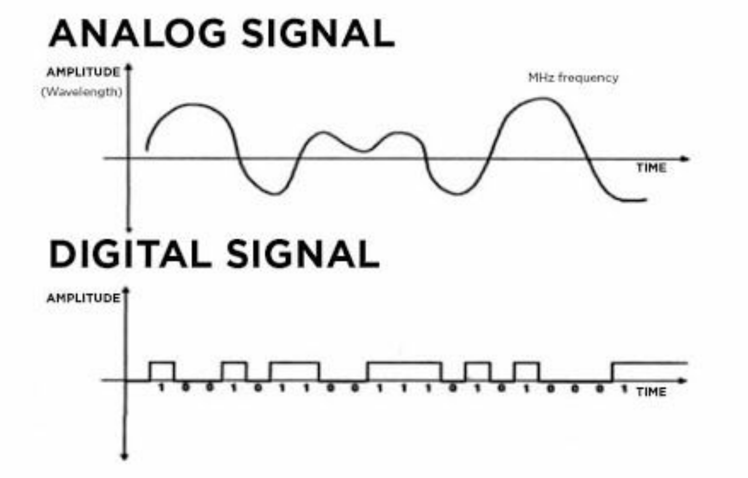
05 - Analog to Digital Converter (ADC)

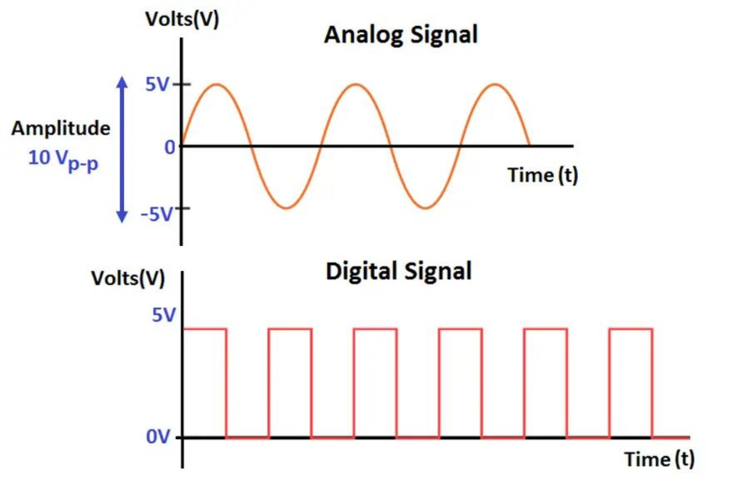
CEG 4330/6330 - Microprocessor-Based Embedded Systems

Max Gilson

Analog vs Digital

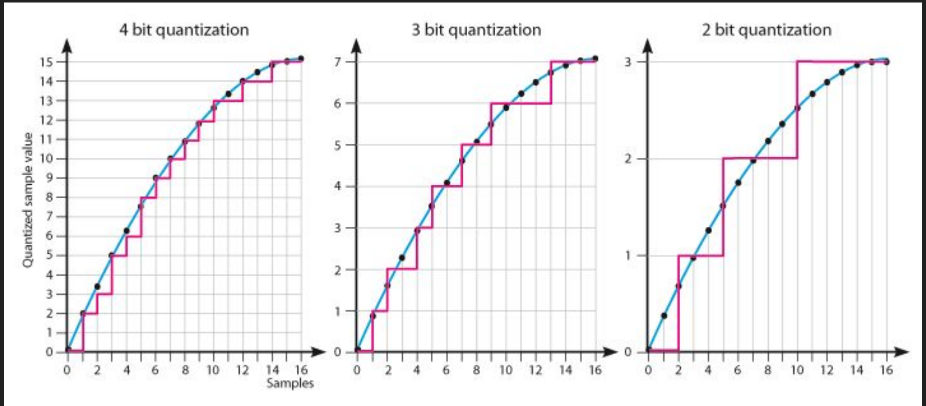
- Digital signals can be HIGH and LOW
  - Arduino HIGH and LOW is 5V and 0V (ground)
- Analog signals are not limited to HIGH and LOW
  - Can be any voltage between HIGH and LOW or even negative

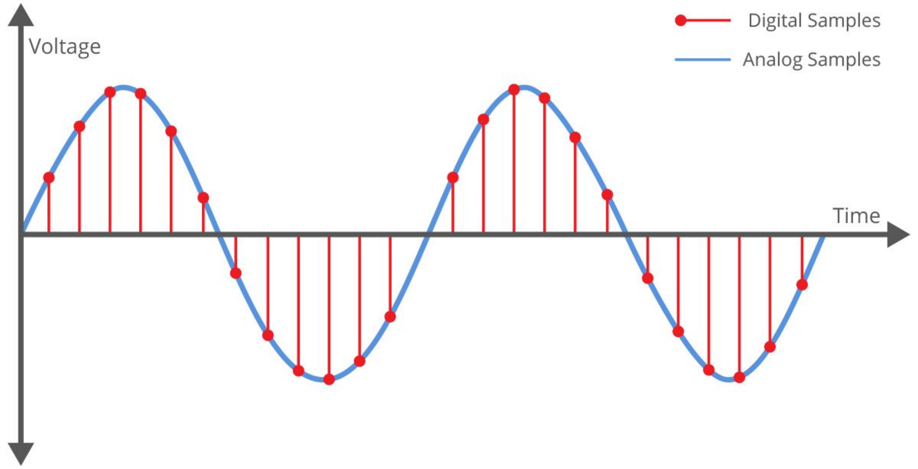




Sampling

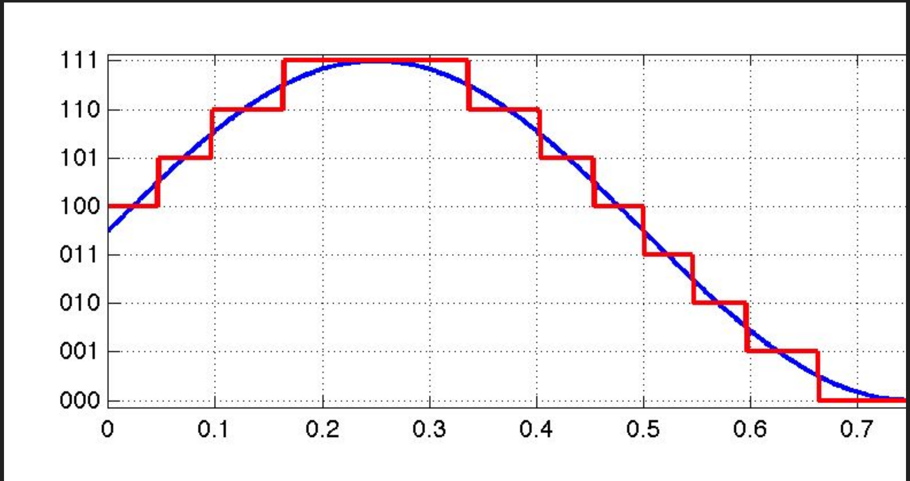
- To measure an analog signal with a digital computer, the computer must sample the analog signal periodically
- Each sample is recorded at a frequency and with n-bit quantization
  - Using low frequency or low quantization results in poor signal conversion





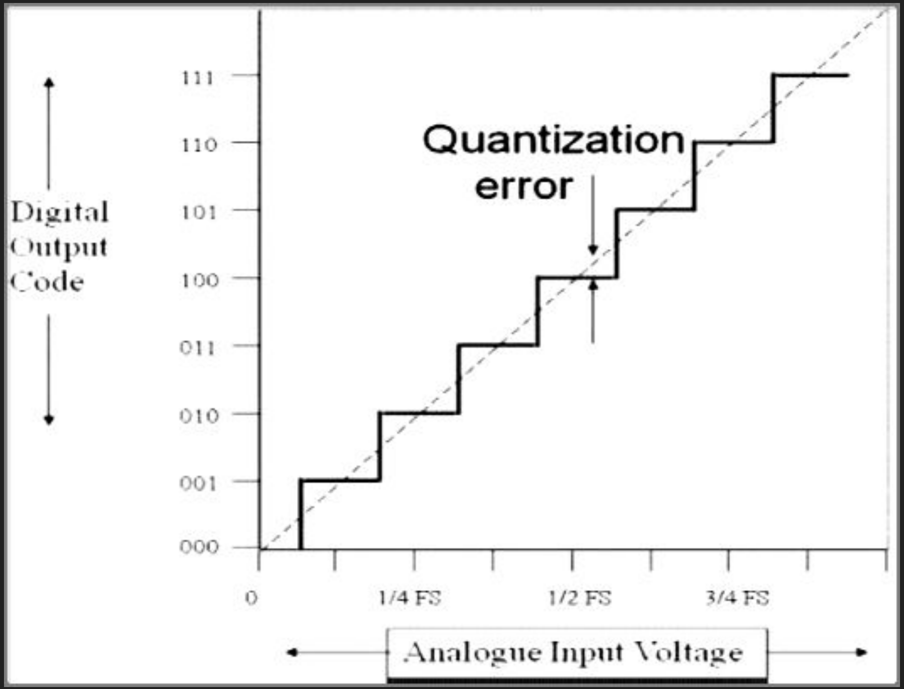
Quantization

- Higher quantization provides higher resolution of digital conversion
- Lower quantization is cheaper but creates a more "chunky" digital conversion
- Sometimes you must make a design trade off for accuracy vs cost



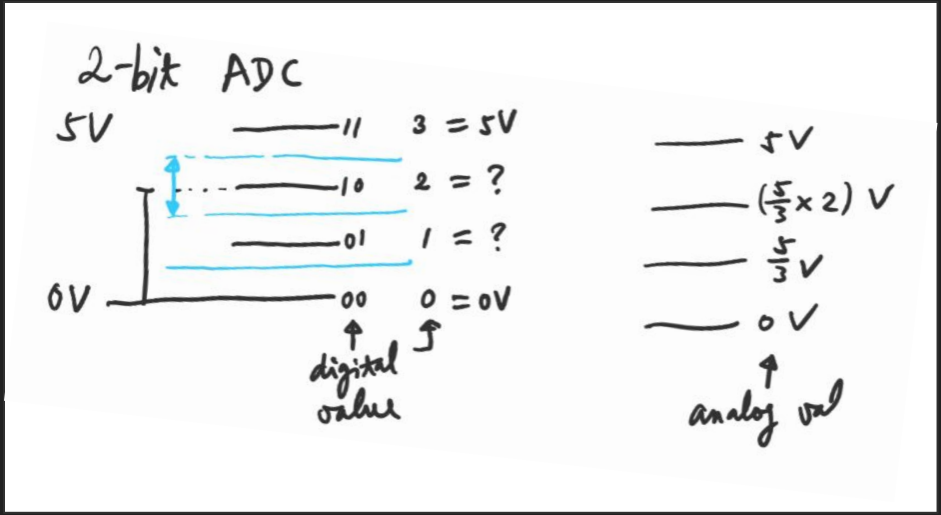
Quantization Error

- Quantization error, is the difference between the actual signal's value and the digital value
- Higher quantization provides less error
- Depends on thresholds between discrete digital values



Quantization Mapping

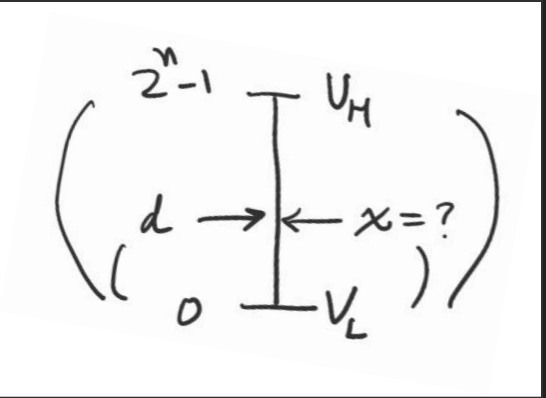
- How do we translate bits into a value?
- Assume 2-bit ADC with 5V maximum:
  - 11: (5/3)*3 = 5.00V
  - 10: (5/3)*2 = 3.33V
  - 01: (5/3)*1 = 1.67V
  - 00: (5/3)*0 = 0.00V



Quantization Mapping (cont.)

- For n-bit ADC:
  - 2^n-1 = max digital value (integer)
  - 0 = min digital value (integer)
  - Vmax = max analog value (voltage)
  - Vmin = min analog value (voltage)
- To calculate the analog value:

where d is the measured ADC integer
and x is the calculated analog value



$$\frac{d}{2^n - 1} = \frac{x - V_{\min}}{V_{max} - V_{\min}}$$

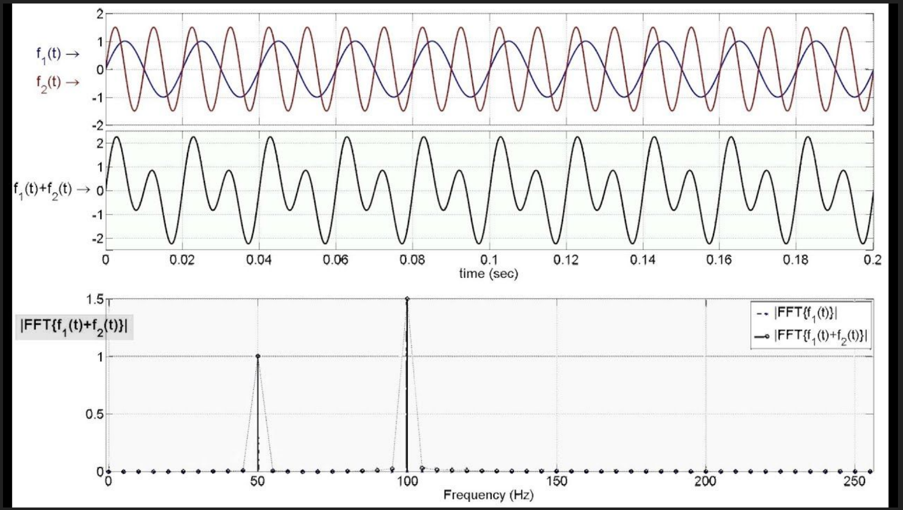
$$x = V_{\min} + \frac{V_{\max} - V_{\min}}{2^n - 1} d$$

Quantization Mapping (cont.)

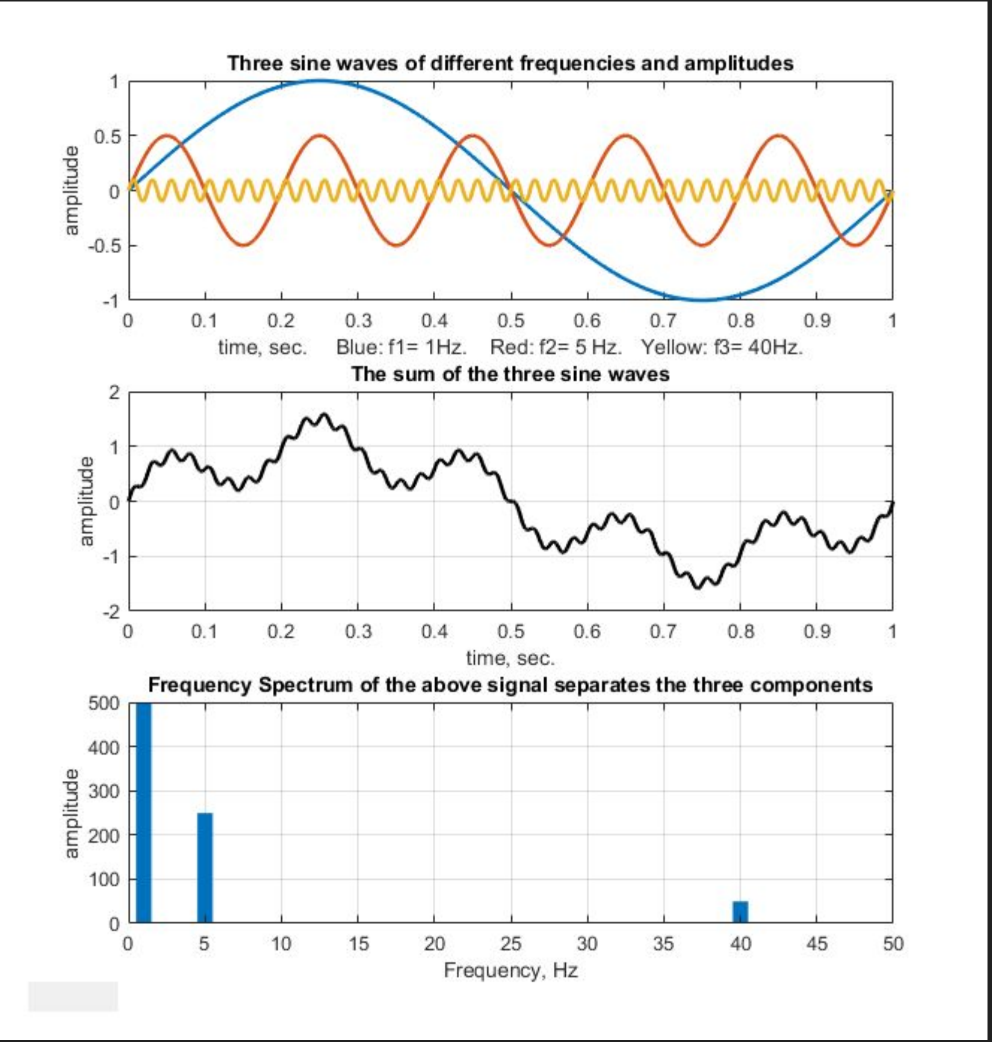
- Arduino map() function makes this very easy:
  - map(value, fromLow, fromHigh, toLow, toHigh)
  - value: the number to map
  - fromLow: the lower bound of the value’s current range
  - fromHigh: the upper bound of the value’s current range
  - toLow: the lower bound of the value’s target range
  - toHigh: the upper bound of the value’s target range
- Does not provide decimal

Understanding Fast Fourier Transform (FFT)

- Fast Fourier Transform (FFT) is a way of representing a signal in the frequency domain
- It shows magnitudes (or strengths) of frequencies for a given signal
- The x-axis represents frequency
- The y-axis represents magnitude



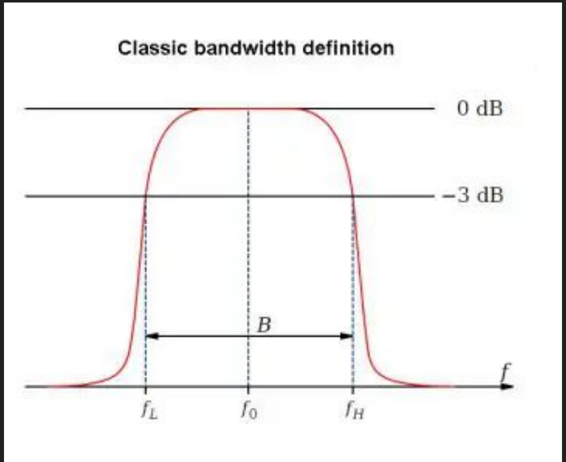
Understanding Fast Fourier Transform (FFT) (cont.)

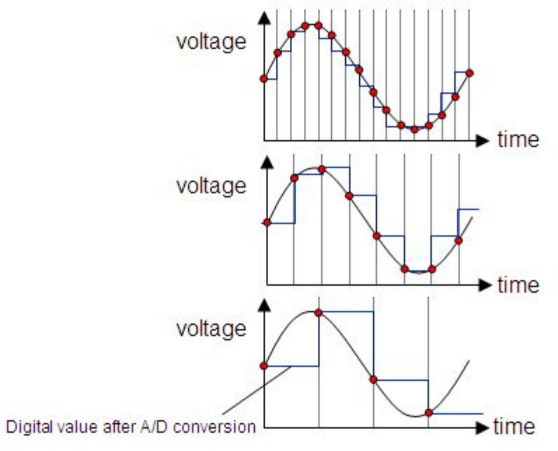


You’ve Probably Seen FFTs Before!

Nyquist Rate and Sampling Rate

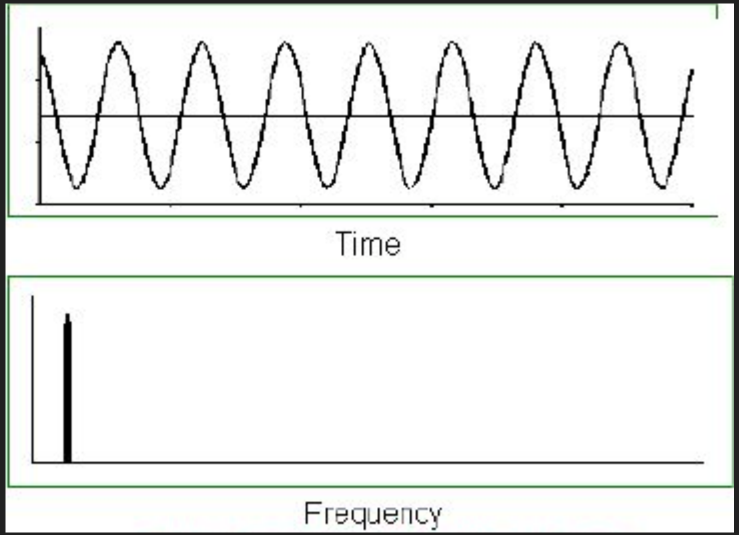
- Nyquist Rate = 2B
  - Where B is the bandwidth of the signal
- Sampling Rate must be greater than Nyquist Rate to fully recover signal
- Example:
  - Bandwidth = 10 Khz
  - Nyquist Rate = 20 Khz
  - Sampling Rate > 20 Khz





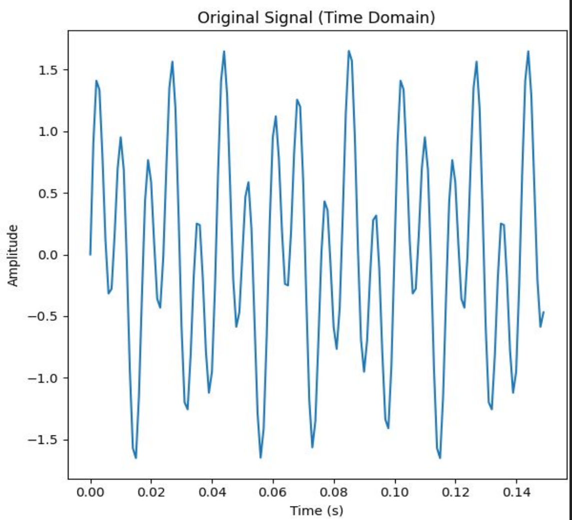
Choosing a Sampling Frequency

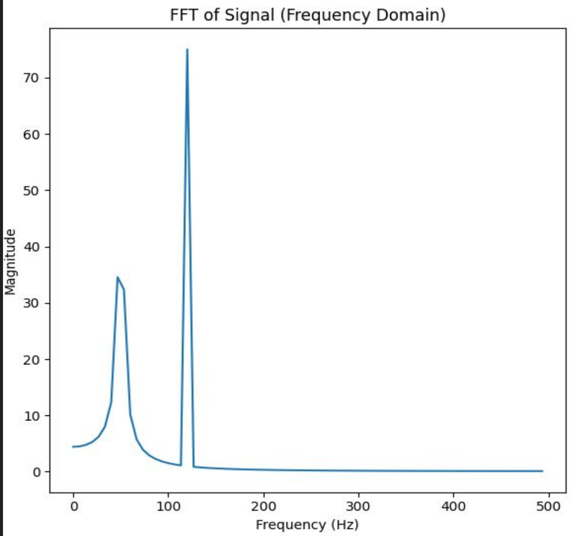
- Assume an ideal sine wave at 1 Hz
- Nyquist Rate = 2 Hz
- Sampling Frequency > 2 Hz
- Pick a sampling frequency that fully recovers the signal
  - > 2 Hz minimum
  - 3 Hz
  - 4 Hz
  - 1000 Hz



Finding the Bandwidth

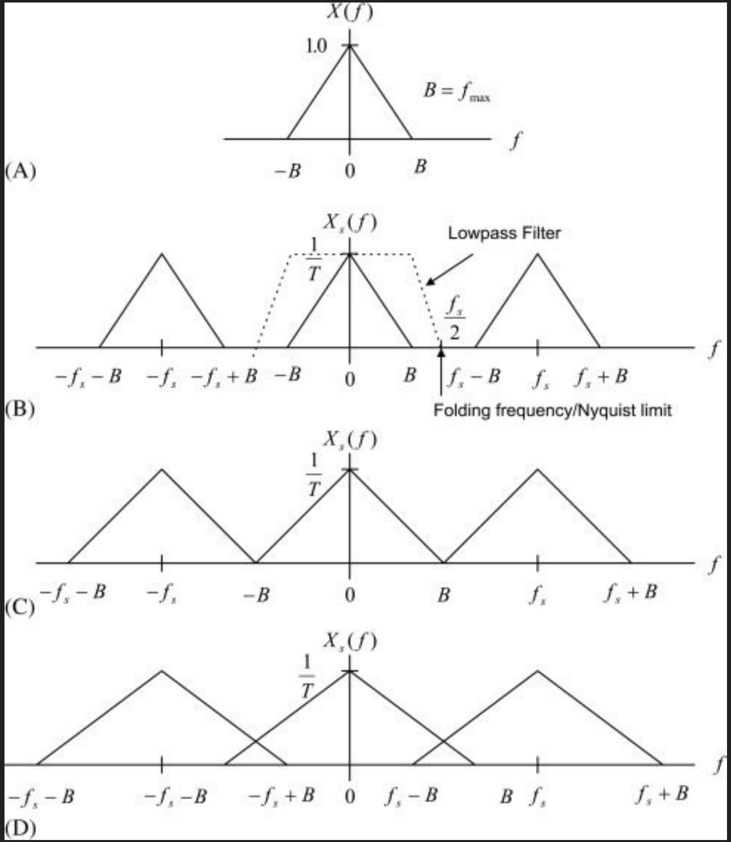
- With a signal that has multiple frequency components, we consider the highest significant frequency to be the bandwidth
- First, perform an FFT
- Second, find the highest frequency, select this as B
- Third, select a sampling frequency > 2B
- In this example, B = 120 Hz and we can select a sampling frequency of >240 Hz





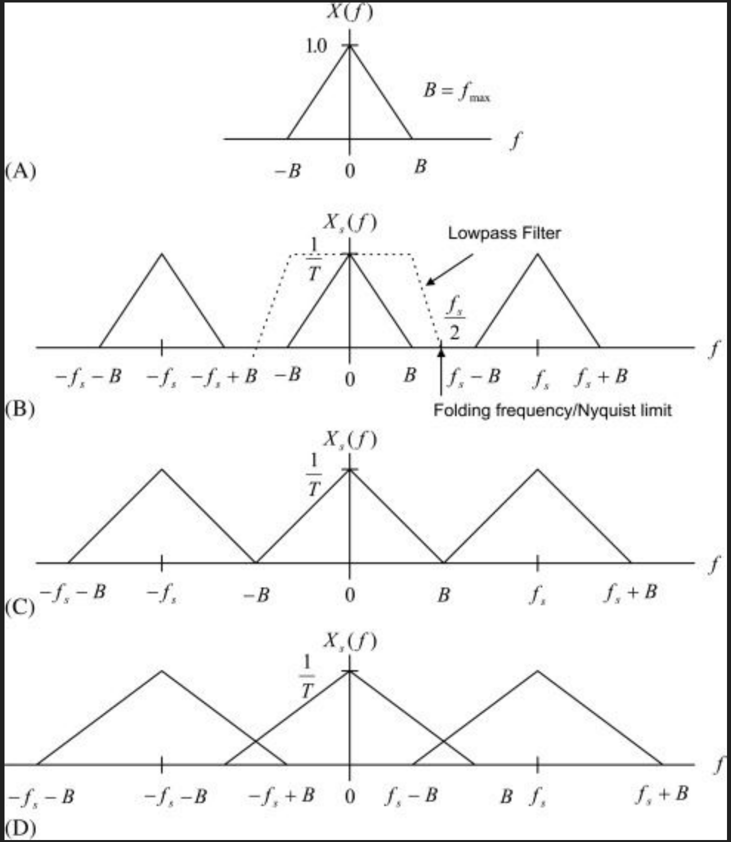
Oversampling vs Undersampling

- A: shows the signals bandwidth in the frequency domain where B is the bandwidth
- B: shows oversampling
  - $f_s > 2B$
- C: sampling at 2B Hz
  - $f_s = 2B$
- D: shows undersampling
  - $f_s < 2B$



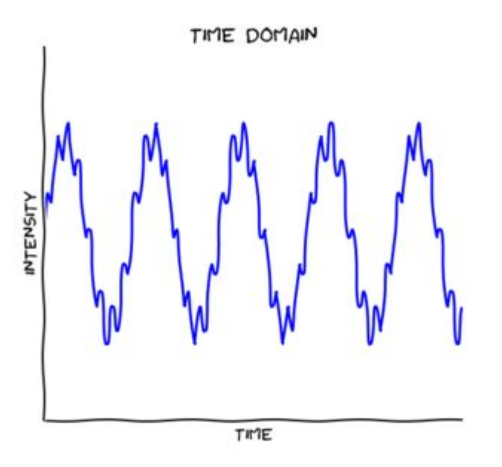
Oversampling vs Undersampling (cont.)

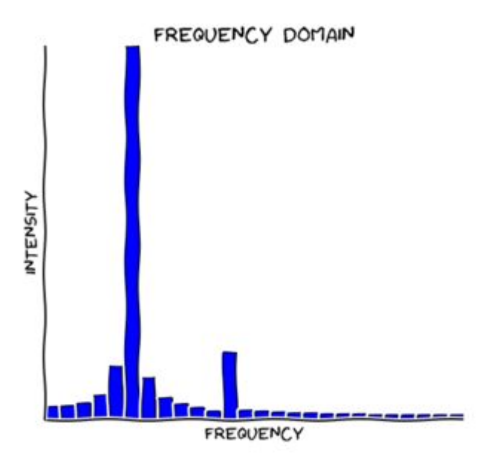
- Just by sampling a signal, frequencies are replicated at $n*f_s$ where n = 0, 1, 2, …
- Since we have these replicated frequencies, choosing a sampling rate > 2B is required otherwise we'll lose signal clarity



Bandwidth is Never Ideal

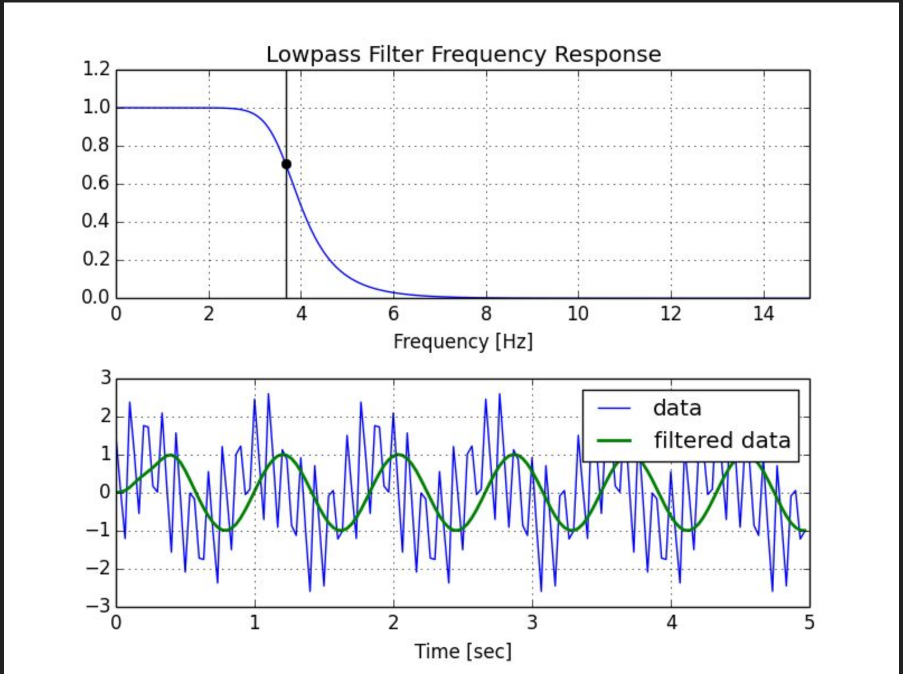
- There will always be some amount of noise in your signal
- This noise can be seen in the frequency domain
- To help fight noise, we can use a low pass filter tuned to B frequency to prevent noise during ADC





Low Pass Filters

- Low pass filters only “pass” the low frequencies to the output
- Allows us to “filter” or weaken the higher frequency components before sending to ADC
- This allows us to convert only the data or frequencies that are important (not noise)

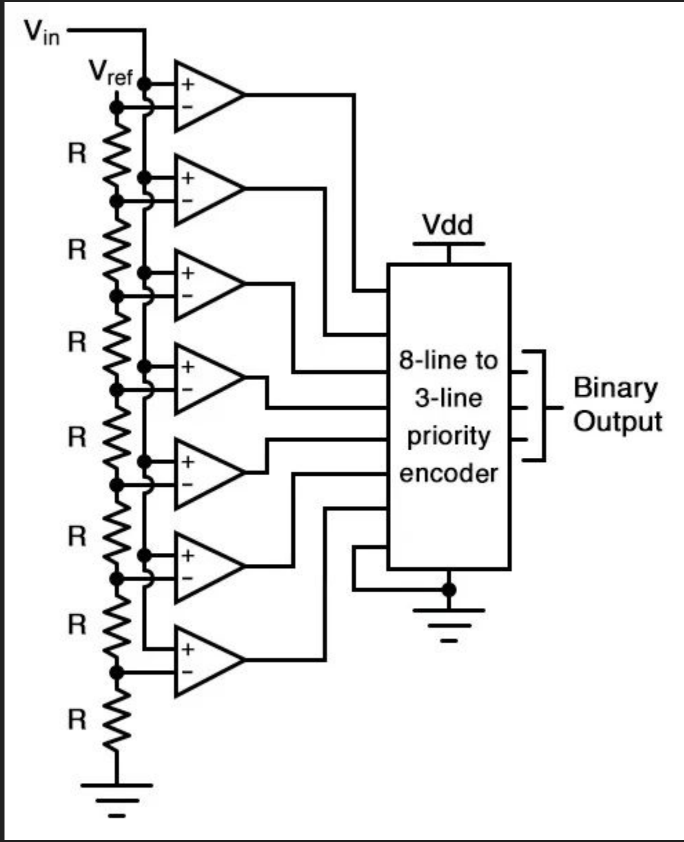


How to Design an ADC

- Comparators:
  - Can use comparators to check each boundary of quantization threshold
  - For 2^{10}-1 thresholds, 1023 comparators are required
  - Very fast but expensive
- Successive Approximation (used by Arduino):
  - Generate an analog signal iteratively, and compare each iteration to the signal value
  - Check if the value is above or below the current MSB and move to the next value
  - Very slow but cheap

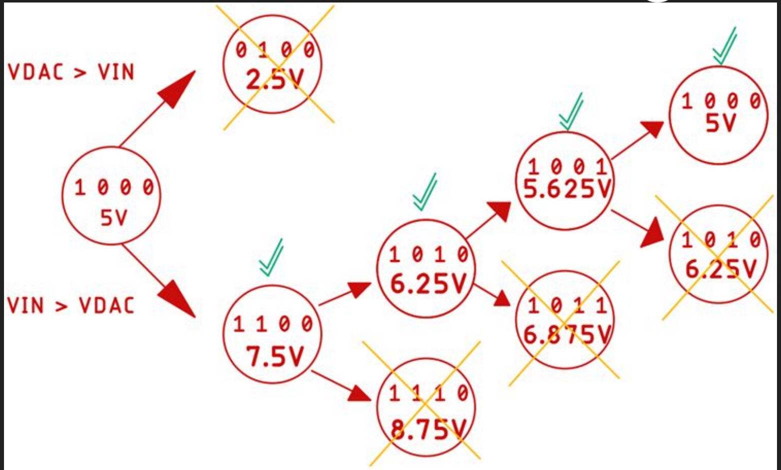
How Comparators Convert to Digital

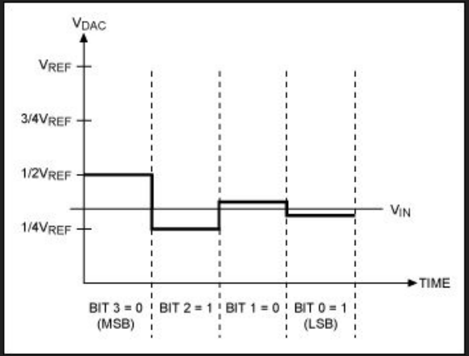
- Comparators are lined up, and fed different threshold voltages
- The outputs of the comparators are fed into an encoder to convert the threshold values to a digital value
- This requires a lot of hardware but offers great performance
- For 2^{10}-1 thresholds, 1023 comparators are required

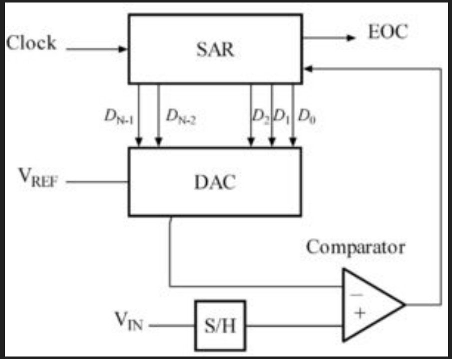


How Successive Approximation Converts to Digital

- Compare the input signal to the digital value if the MSB is set
- If input signal is greater, set the bit and continue with next MSB
- If input signal is less, clear the bit and continue to with next MSB

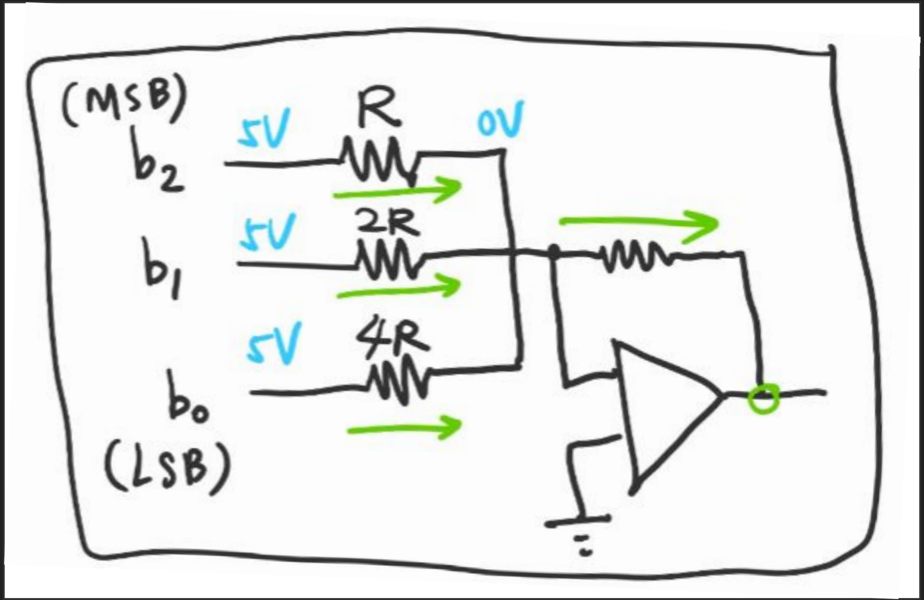






Designing a DAC for Successive Approximation

- Using an operational amplifier, you can "sum" the bits of a digital output to produce an analog signal that is the sum of weighted currents
- Weighting:
  - Bit 0: 1
  - Bit 1: 2
  - Bit 3: 4
- Very cheap to produce



Arduino ADC

- Arduino UNO:
  - 10-bit ADC
  - 0-5V input
  - 6 analog channels (each shares 1 ADC)
- Uses Successive Approximation

Arduino ADC Sample Code

```
int analogPin = A3; // potentiometer wiper (middle terminal) connected to analog pin 3
                    // outside leads to ground and +5V
int val = 0;  // variable to store the value read

void setup() {
  Serial.begin(9600);           //  setup serial
}

void loop() {
  val = analogRead(analogPin);  // read the input pin
  Serial.println(val);          // debug value
}
```

analogRead() Is Not Always Best

- Sometimes you need to record multiple samples in a row
- Sometimes you need to capture samples very fast
- analogRead() is a function, is slow, and only captures 1 value whenever it is called
- Uses polling
  - Notice the while loop!

analogRead() Library Code

```
int analogRead(uint8_t pin)
{
    if (pin >= 14) pin -= 14; // allow for channel or pin numbers

    // set the analog reference (high two bits of ADMUX) and select the
    // channel (low 4 bits).  this also sets ADLAR (left-adjust result)
    // to 0 (the default).
    ADMUX = (analog_reference << 6) | (pin & 0x07);

    // without a delay, we seem to read from the wrong channel
    //delay(1);

    // start the conversion
    sbi(ADCSRA, ADSC);

    // ADSC is cleared when the conversion finishes
    while (bit_is_set(ADCSRA, ADSC));

    // ADC macro takes care of reading ADC register.
    // avr-gcc implements the proper reading order: ADCL is read first.
    return ADC;
}
```

ADC Register Control

- ADCSRA
  - Enable ADC
  - Start Conversion
  - Auto Trigger
    - Trigger ADC on a timer or pin
    - Allows you to choose your own sampling frequency
    - Free running is fastest possible frequency
  - ADC Prescaler Select (128 default in init())
    - Conversion takes multiple clock cycles
    - First 25 clock cycles, then 13 clock cycles
    - Default: 125 Khz / 13 avg. cycles = 9.6 Khz sampling frequency
  - ADC Interrupt
    - Interrupt enable and interrupt flags

ADC Register Control (cont.)

- ADMUX
  - ADC Reference Selection
    - Select between 5V reference voltage or other
    - Choosing 3.3V reference voltage, gives higher resolution readings from 0V to 3.3V
  - Left Justified
    - Shifts the ADCL bits left into the ADCH
    - Forces the 8 MSB to be in the ADCH register
    - Mostly used in memory constrained situations
  - ADMUX
    - Select analog pin to input into ADC

ADC Interrupt

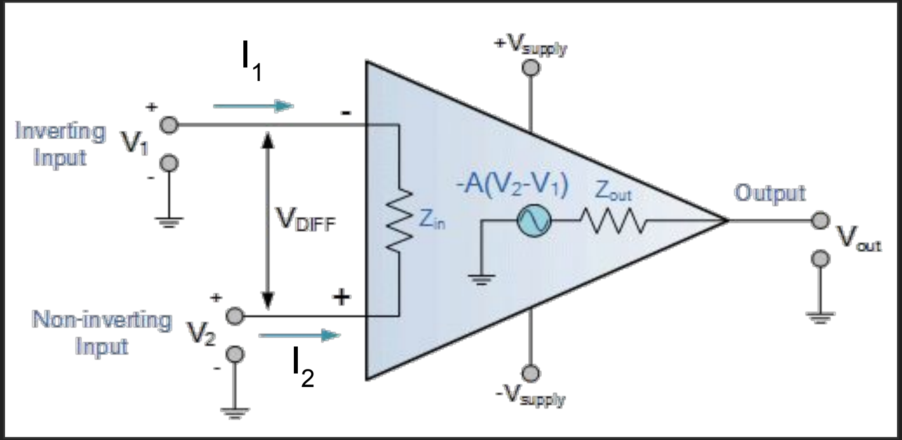
- ADC Conversion Complete Interrupt
  - Use: ADC_vect
- Triggers ISR when a conversion on ADC is complete
- Much faster than using analogRead()
  - Allows for immediate processing of ADC value after conversion is complete, without waiting for conversion

Signal Conditioning

- Often times, our analog signal is not in a perfect range of 0V to 5V
  - Example: 1S Lipo Battery (same battery in smartphones) has a range of 3.2V to 4.2V
    - 3.2V is 0% battery
    - 4.2V is 100% battery
- Sometimes, the voltage range is too small to be measured in 0V to 5V
  - Example: unprocessed microphone output may be 0.001V to 0.100V
    - This is too small for the Arduino to measure consistently
- Signal conditioning will allow us to convert the sensor’s output range to the ADC’s input range
  - Amplify and offset voltage

Signal Conditioning using Operational Amplifier

- Operational Amplifier (Op Amp)
  - We can make different circuits:
  - Non-inverting amplifier
  - Inverting amplifier
  - Inverting amplifier with DC offset
- $V_1 = V_2$
- $I_1 = I_2 = 0$
- Ohm's Law:
  - $V = I * R$



Non-Inverting Amplifier

$$V_1 = V_2 = V_{in}$$

$$V_{in} = I_{R2} \cdot R2$$

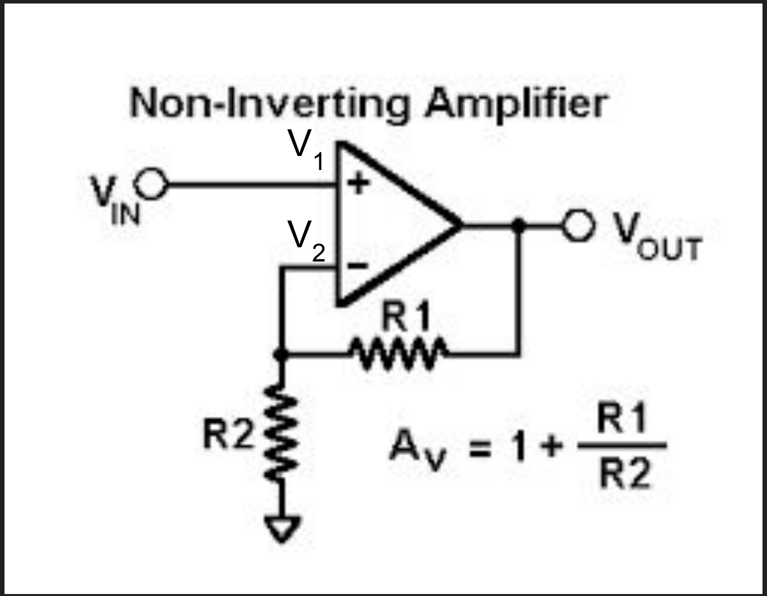
$$I_{R2} = I_{R1} = I$$

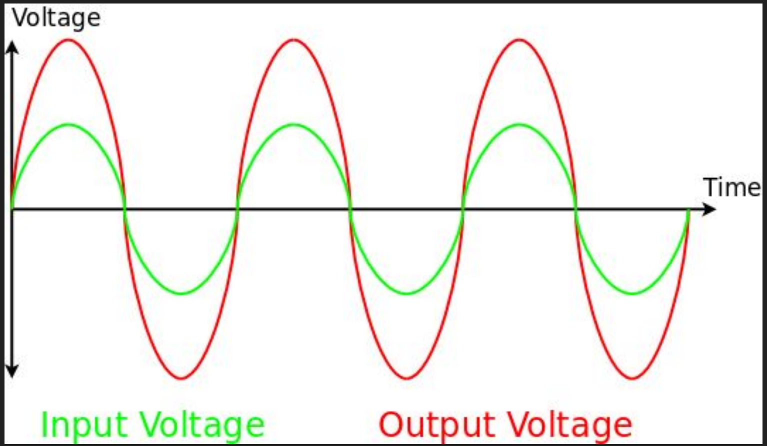
$$V_{out} = I \cdot (R1 + R2)$$

$$V_{out} = V_{in} \cdot \frac{R1 + R2}{R2}$$

$$V_{out} = V_{in} \cdot \left(1 + \frac{R1}{R2}\right)$$

$$A_v = 1 + \frac{R1}{R2}$$





Inverting Amplifier

$$V_1 = V_2 = 0$$

$$V_{in} = I_{Ri} \cdot R_i$$

$$I_{Rf} = I_{Ri} = I$$

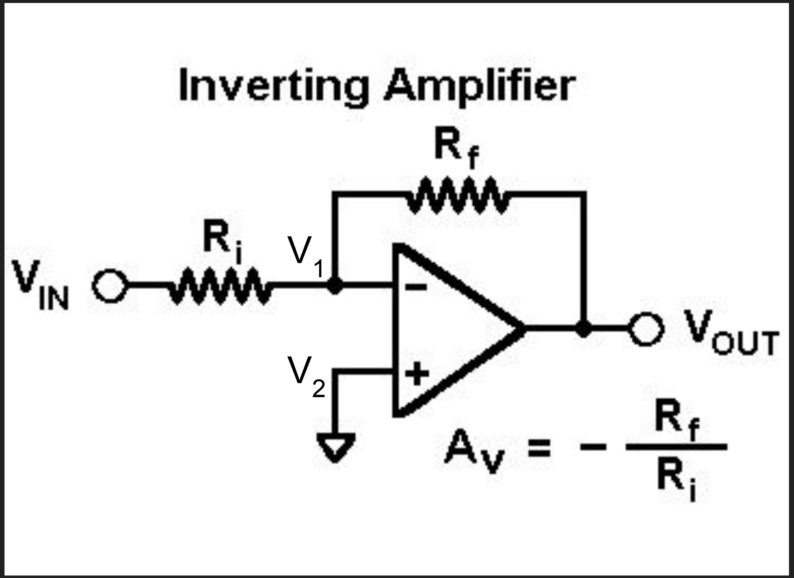
$$V_{Rf} = -V_{out}$$

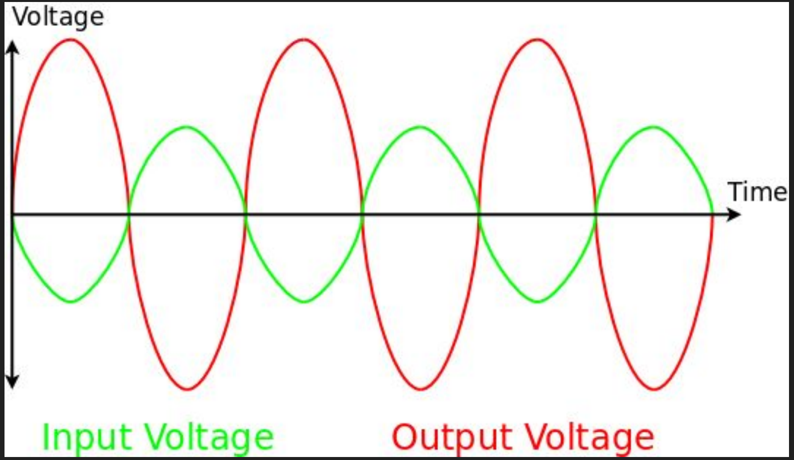
$$V_{out} = -V_{Rf}$$

$$V_{out} = -(R_f \cdot I)$$

$$V_{out} = -\left(\frac{R_f}{R_i}\right) \cdot V_{in}$$

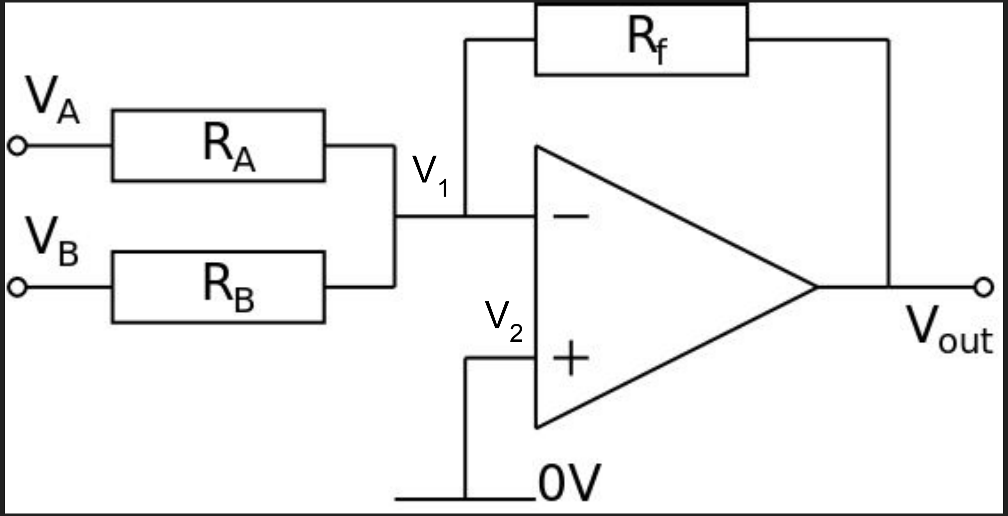
$$A_v = -\left(\frac{R_f}{R_i}\right)$$

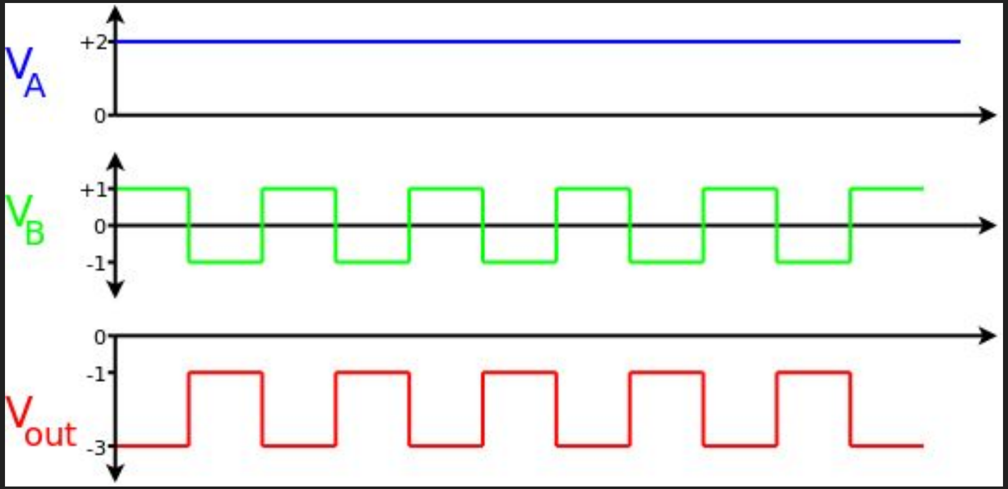




Inverting Amplifier With DC Offset

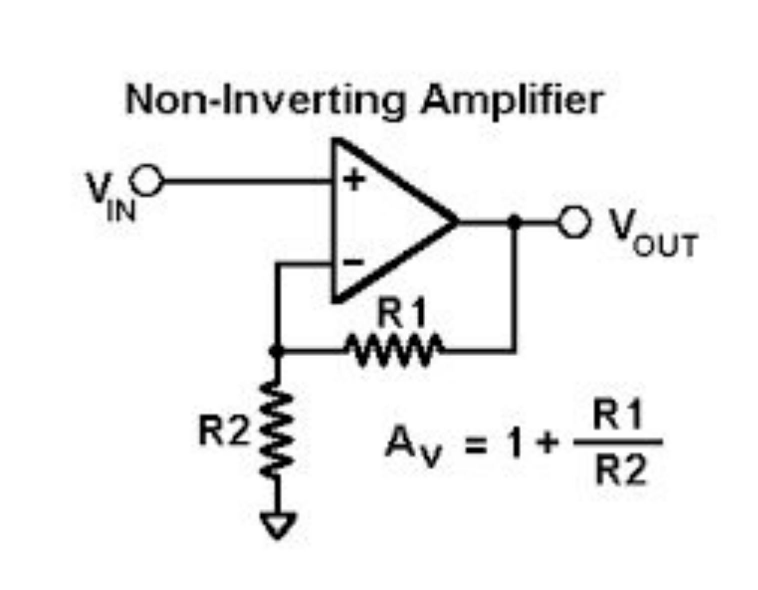
$$V_1 = V_2 = 0$$
$$V_A = I_{RA} \cdot R_A$$
$$V_B = I_{RB} \cdot R_B$$
$$I_{RA} + I_{RB} = I_{Rf} = I$$
$$V_{Rf} = -V_{out}$$
$$V_{out} = -V_{Rf}$$
$$V_{out} = -(R_f \cdot I)$$
$$V_{out} = -(R_f) \cdot (I_{RA} + I_{RB})$$
$$V_{out} = -(R_f) \cdot \left(\frac{V_A}{R_A} + \frac{V_B}{R_B}\right)$$
$$V_{out} = -\left(\frac{R_f}{R_A}\right) \cdot V_A + -\left(\frac{R_f}{R_B}\right) \cdot V_B$$





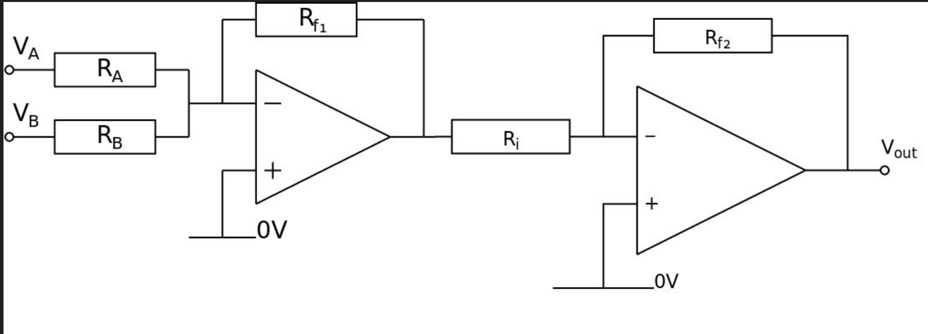
Design Challenge 1

- Given an input signal from 0 mV to 2 mV, create a signal conditioning circuit to measure this signal with the Arduino Uno's 0V to 5V ADC
- Non-inverting amplifier:
- 2500 amplifier gain
- Av = gain = 1 + R1/R2 = 2500
- R1/R2 = 2499
- R1 = 2,400,000 Ω = 2,400 KΩ = 2.4 MΩ
- R2 = 960 Ω = 0.960 KΩ
- Av = 1 + (2400000/960) = 2501



Design Challenge 2

- Given an input signal from -20 mV to 30 mV, create a signal conditioning circuit to measure this signal with the Arduino Uno’s 0V to 5V ADC
- Two-stage amplifier:
- 100 amplifier gain
- 2V shift UP after amplifying
- Rf1 = Rf2 = 10 KΩ
- RA = Ri = 1 KΩ
- Gain = 10 x 10 = 100
- Assume VB = 5V
  - RB = 250 KΩ



Design Challenge 3

- Given an input signal from 10 mV to 20 mV, create a signal conditioning circuit to measure this signal with the Arduino Uno's 0V to 5V ADC
- Two-stage amplifier:
- 500 amplifier gain
- 5V shift DOWN after amplifying
- Rf1 = 50 KΩ
- Rf2 = 10 KΩ
- RA = Ri = 1 KΩ
- Gain = 50 x 10 = 100
- Assume VB = 5V
    - RB = 10 KΩ