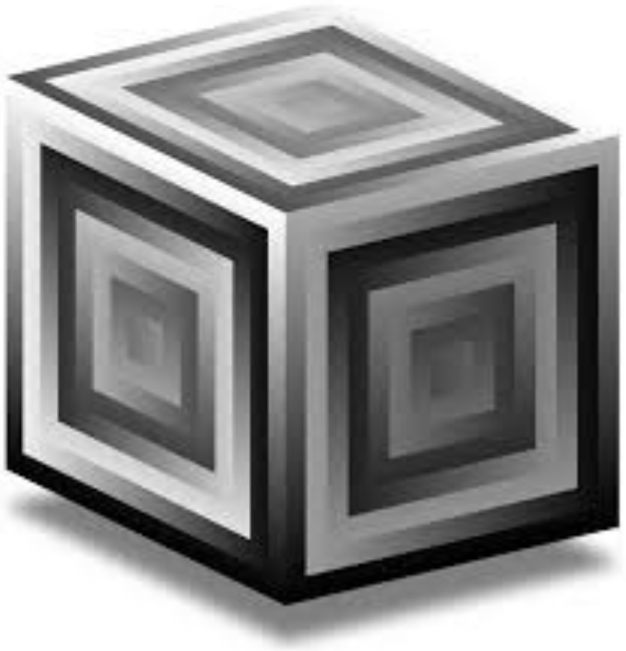
Digital Representation

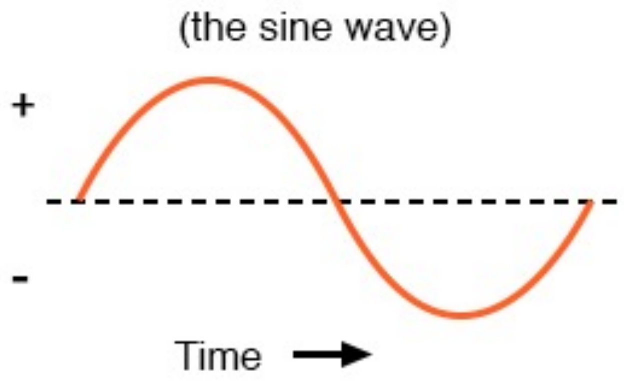
Topics Addressed

- Continuous Waveforms
- Sample Rate
- Aliasing
- Nyquist Frequency/Rate
- Wavetable Synthesis
- Band Limited Oscillators
- DAC/ADC



Continuous Waveforms

- All musical instruments produce a continuous waveform.
- Vinyl discs and cassette tapes store continuous waveforms.
- Speakers drive air to produce continuous waveforms by voltage.
- Microphones capture changes in air pressure and convert to a continuous voltage.

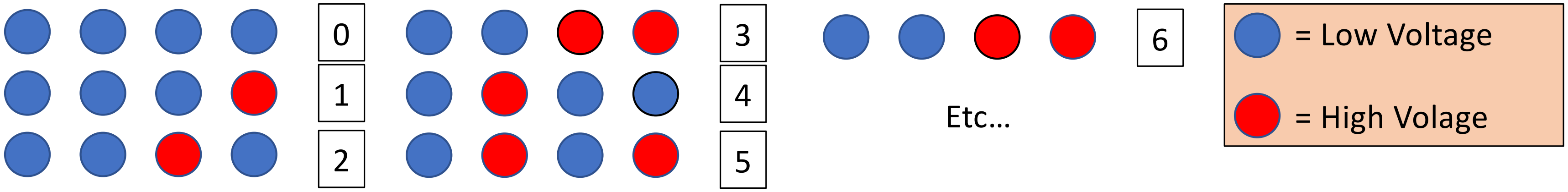


COMPUTERS ARE DIGITAL!!!!!!!!!

Computer Storage

Take CS240 to learn more!

- Computers do two primary things: store data and process data
- Data is stored by using collections of binary bits that are abstractions for high voltage or low voltage in an electrical circuit.
  - Computers have a finite number of these bits. After all, they are a physical device.
- Suppose we have a very tiny computer that only had 4 bits worth of storage for a single number. How many different numbers could we represent?

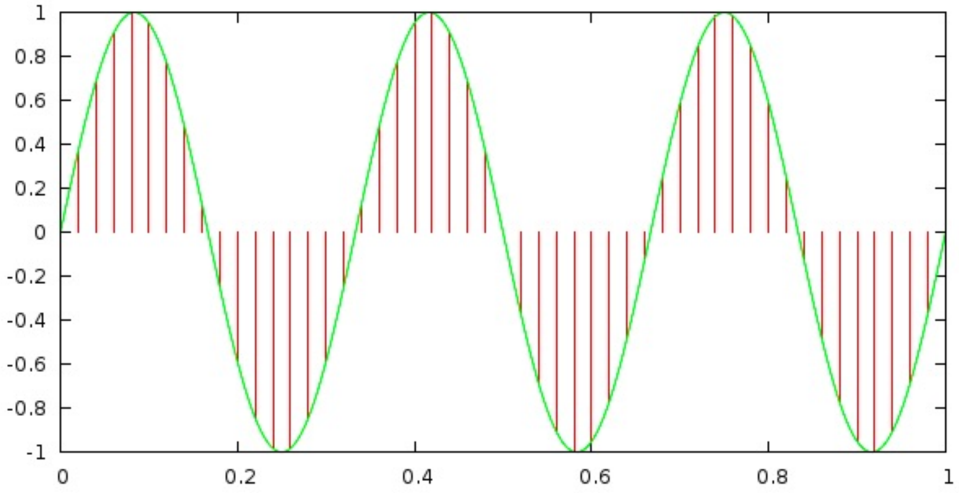


Implications

- If we only have a finite number of bits no matter what that finite number is, can we represent all the integers? No!
- Yikes! This means we have to make concessions. Moreover, we don't want to use all the bits in our machine to represent a single number.
  - In truth, our modern computers use up to 64 bits to represent a number, allowing 2^{64} different numbers.
  - We also only have 64 bits for floating point numbers as well.
  - If you look at the documentation for SuperCollider, you'll see that the `Float` class stores a 64-bit number while the `Integer` class stores a 32-bit number.
- Any computer will be unable to perfectly represent infinite forms like numbers.

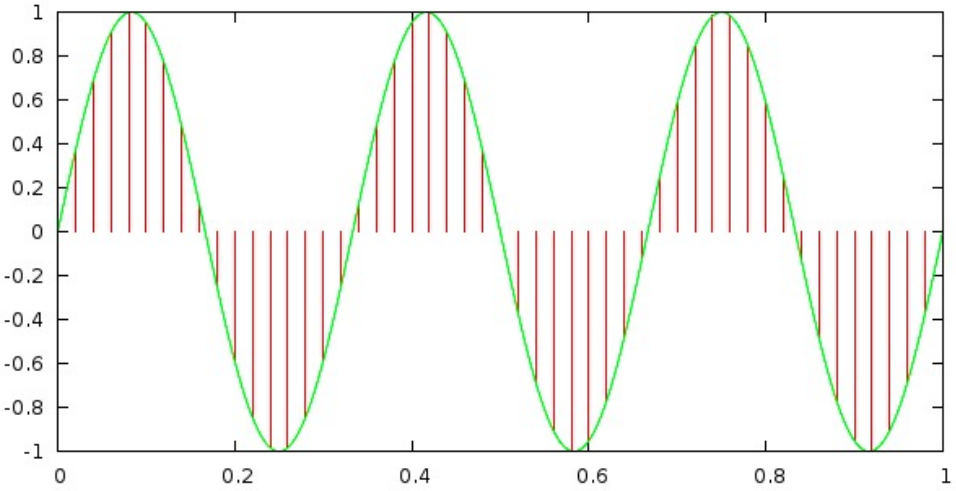
Continuous Waveforms

- Continuous waveforms require an infinite number of data points to represent. Therefore, we will need to make some concessions.
- Our strategy will be to sample the signal we are trying to produce.



A signal is simply fluctuating data points between -1 and 1 over a period of time. The red lines here indicate both when the sample occurs and what the value is between -1 and 1 (represented by the length of the line).

Continuous Waveforms



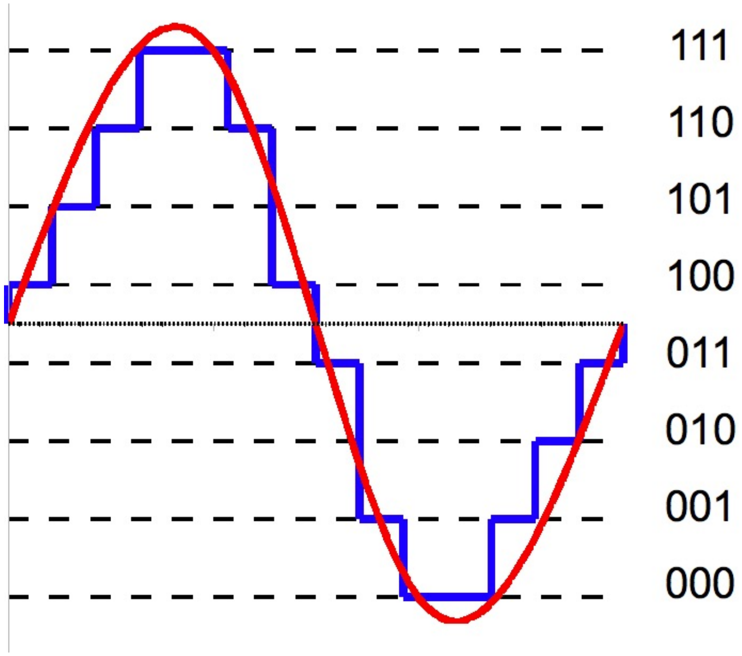
Time	Amplitude
0	0
0.1	0.35
0.2	0.65
0.3	0.9
0.4	1
0.5	0.95
0.6	0.75

Etc…

- Samples are generated at regular, evenly-spaced time periods.
- The number of the sample is the amplitude of the signal
- Sampling gives a finite approximation of the continuous signal

Quantization

- Concessions need to be made in terms of the number of samples generated for a continuous waveform as we just saw.
- Concessions **also** need to be made for exactness of the amplitude measured. There are infinite number of possible amplitudes generated from a signal between -1 and 1 amplitude.
    - Solution: quantize the amplitude.
- Bit depth specifies the number of bits allocated to each sample of audio.
    - More bits = more resolution = more accurate amplitude sample
    - Common bit depths: 24-bit and 16-bit



Sample Rate

- In order to achieve a good approximation of the continuous signal, many samples need to be produced.
- The sample rate for a signal specifies the number of samples per second.
- The sample rate applies to both incoming and outgoing signals. With incoming signals we sample at the sample rate from a continuous physical voltage. With outgoing signals, we output samples at the sample rate to generate a continuous voltage (which is usually passed on to our speakers).
- The standard rate is 44,100Hz, meaning that 44,100 samples are generated for every second of sound.
- Other standard rates include 48kHz (i.e., 48,000Hz) or even 192kHz.
- Most of the time choosing a higher sampling rate won't make any audible difference to us but it could potentially.

Exercise

- Consider a simple continuous sine wave with frequency of 100. What would the first five samples be if the sample rate were 44.1kHz?

$$\text{Time between samples} = \frac{1}{sample\ rate}$$
$$= \frac{1}{44100} = 2.268 \ * 10^{-5}$$

Mathematically, we can model a sine wave using sin. A sine wave has a period of 0 to 2pi. Therefore, $\sin(2\pi t)$ where t is time in seconds models a wave with frequency 1Hz. $\sin(2\pi t \ * f)$ then models a sine wave of any frequency. Plug in our times to get the requisite amplitude.

Sample Number	Time elapsed (in seconds)	Amplitude
0	0	0
1	0.00002268	.01425
2	0.00004535	.02849
3	0.00006803	.04273
4	0.00009070	.05699

Relevant Equations

- Time elapsed between samples (i.e, the sampling period)

$$\frac{1}{sample\ rate}$$

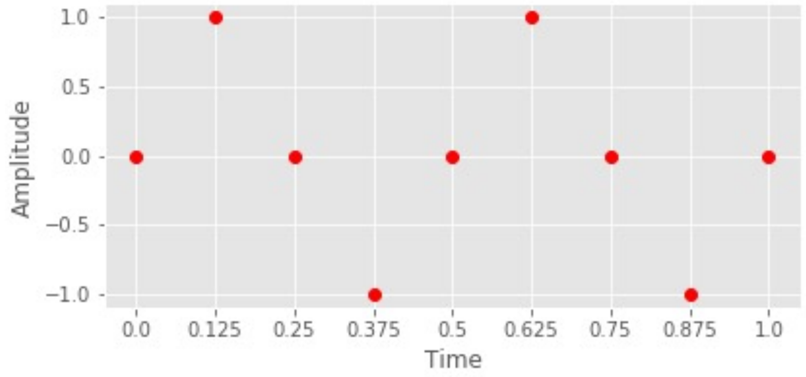
- A sine wave with frequency f in Hz and time elapsed t in seconds

$$\sin(2\pi f\ * t)$$

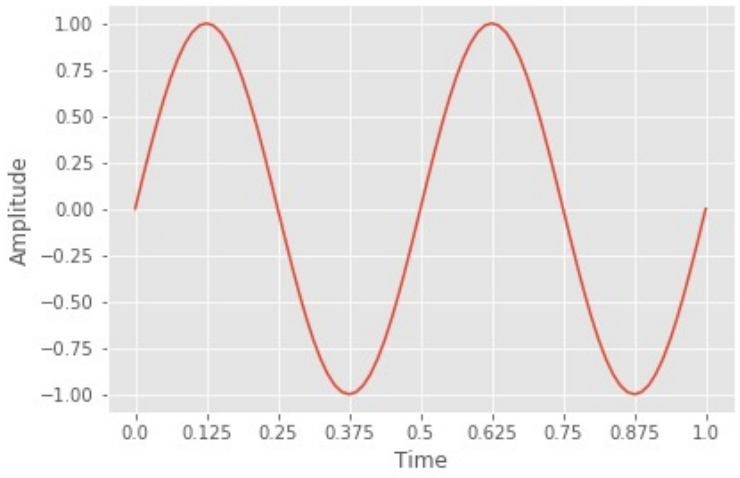
Plotting Samples

- Imagine we had an array of samples from a sine wave. What could we infer about such a sine wave? Frequency?
- Say we had the following samples as an array [0, 1, 0, -1, 0, 1, 0, -1, 0] and our sample rate was 8Hz (yes, absurdly low).
- What can we say about this sine wave?

Let’s plot the samples first



?????

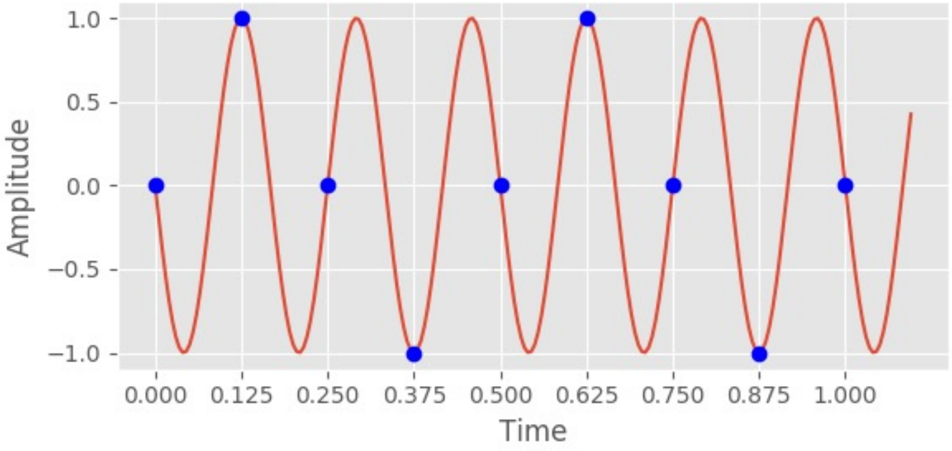


- Suggestive of a sine wave with frequency 2Hz
- Our ears will definitively hears this frequency as 2Hz
- However, is that the only possible wave given the number of samples that we have?

Other Possibilities

Let's say that we increased the sampling rate to 20Hz and we find that our original samples of [0, 1, 0, -1, 0 … etc.] actually come from this waveform!?

Maybe its actually 6Hz and not 2Hz? How do we know???

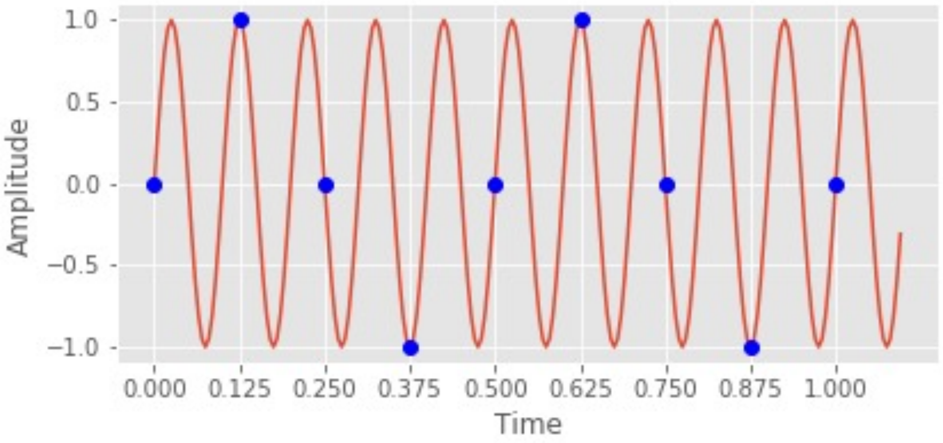


Yikes!

Other Possibilities

Let’s say that we increased the sampling rate to an even higher sampling rate and we find our original samples of [0, 1, 0, -1, 0 … etc.] come from this waveform!?

Maybe its actually 10Hz and not 2Hz? How do we know???

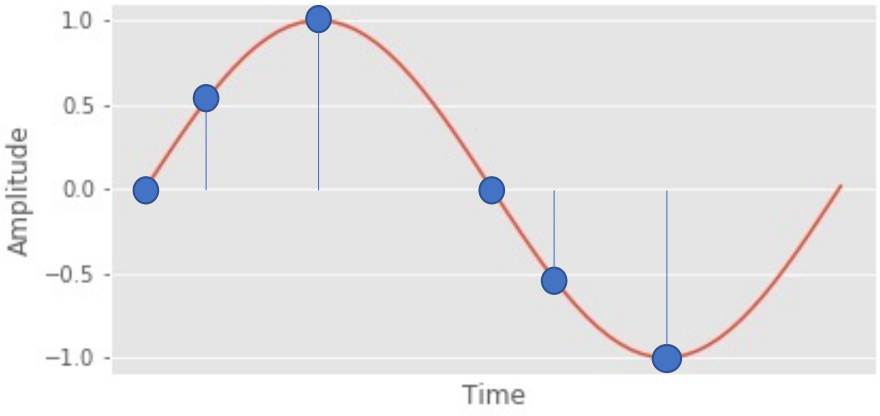


Yikes!

Aliasing

- The previous few slides depict a fundamental issue when we try to discretize continuous signals.
- If our sampling rate is too low for the signals we are trying to represent then we could generate samples that will actually produce a **different** frequency.
  - Suppose our sine wave in the previous examples was actually 10Hz. With a sample rate of just 8Hz, we generated samples that would be perceived by our ears as 2Hz. Problem!
- We call such falsely generated lower frequencies aliases.
- Combatting aliasing in signals is a real problem in Digital Signal Processing.
- Problem: finite sampling cannot capture an infinite number of frequencies.
  - Sampling rate restricts and determines the range of frequencies that can be represented.

Minimum Samples



How many samples does it take to accurately represent one period of a given frequency? We need to have a sample from the “high” amplitude and a sample from the “low” amplitude to have an accurate frequency. That recreates the physical phenomenon of compression and rarefaction.

Four samples will work. We get a basic outline of the sinewaves peaks and troughs. Can we use fewer?

Two samples seem to work. Can get a ”high” and “low” amplitude. Does it work in all cases?

No. So to guarantee we must sample at a rate greater than twice this frequency.

Nyquist Frequency/Rate

- We can generalize our results from the previous slide. Given an analog signal $x(t)$ whose constituent sine waves includes a maximum frequency B (i.e., the bandwidth), then the sample rate f_s must be $f_s > 2B$ to prevent aliasing.
- The Nyquist Rate is $2B$. Intuitively, this is the threshold for the minimum sampling rate to accurately capture $x(t)$.
- The Nyquist Frequency is $f_s/2$. Intuitively, this is the maximum frequency that can be sampled without aliasing.

Important: sometimes these terms are used interchangeably and sometimes they mean different things. Be careful when reading other materials. In essence, they both express the limit of sampling.

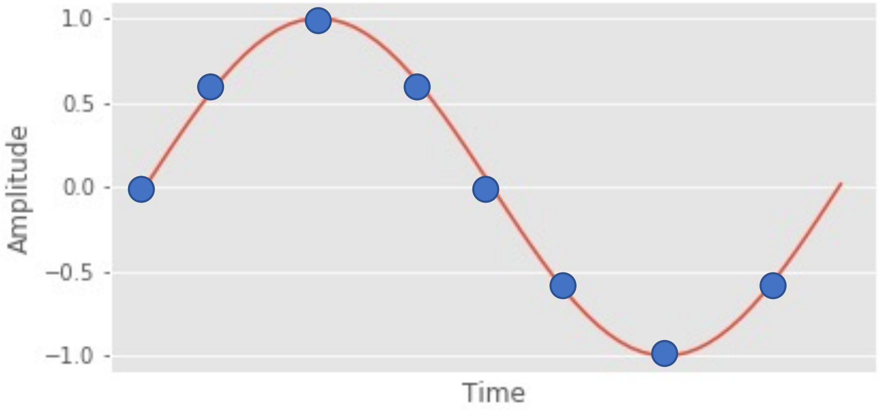
Preventing Aliasing

- Continuous signals such as the human voice, an instrument, nature, … etc. nearly always contain frequencies that exceed the Nyquist Frequency for our computers.
- Every sound card or audio interface contains an analog to digital converter (ADC for short) which handles the process of converting continuous signals that we record into discrete samples.
- To prevent aliasing, continuous signals are passed through an anti-aliasing filter which removes these upper frequencies before sampling occurs.
    - This is a very simplified version of an incredibly complex topic.
- We also have to be mindful of not generating our own aliases by attempting to produce frequencies above the Nyquist Frequency.

Wavetable Synthesis

- How do we actually generate a sinusoidal wave from the computer?
  - We want to generate the samples of a sine wave
  - We could sample from an analog sine wave but it is much easier to generate the samples ourselves mathematically.
- Solution: wavetable
  - Let’s create the samples ourselves by hand
  - We will generate one single period of a sine wave and loop through those samples when we will want to playback the wave.

Generating Sine Wavetable



Let’s assume we have a sample rate of 44.1kHz. Assume we cycle through the samples of this table at the sample rate. What frequency wave have we generated?

Answer: 44100/8 = 5512.5Hz

Amplitude
0
.707
1
.707
0
-.707
-1
-.707

Creating Other Frequencies

- How do we create other frequencies?
- Option 1: generate a wave table for every frequency we want to create.
  - Pro: accurate wave table
  - Con: terrible usage of memory. We could have many many wavetables!
- Option 2: “playback” the wave table at different rates
  - Pro: efficient usage of memory
  - Con: inaccuracies in samples (more soon)
- It turns out that Option 2 is the most common, though some oscillators/synthesizers will use multiple wavetables for different parts of the frequency spectrum.

Amplitude
0
.707
1
.707
0
-.707
-1
-.707

Creating Other Frequencies

- Let’s say we want to generate a sine wave of frequency 11025Hz.
- We know our table when looped through and played back at the sample rate (44.1kHz) generates a sine wave of 5512.5Hz.
- What samples from our table do we need to create 11025Hz?

Amplitude
0
.707
1
.707
0
-.707
-1
-.707

Generated Samples

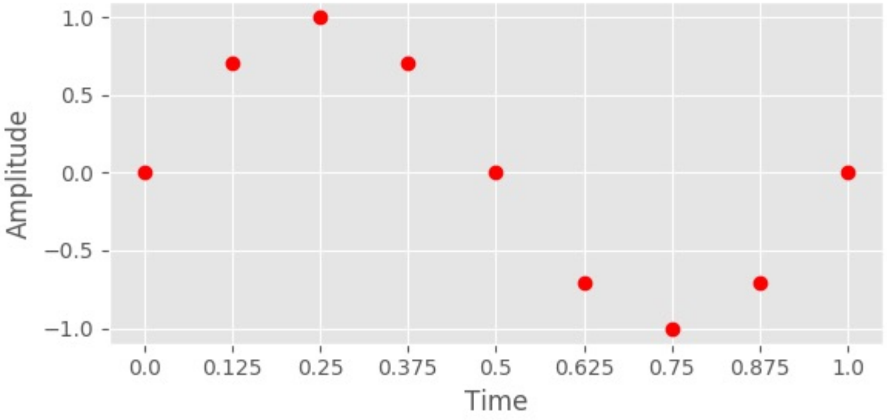
0	1	0	-1	etc…

Step size of 2

Need to generate 44100 samples per second. This means we will go through this single period 11025 times to generate a frequency of 11025Hz.

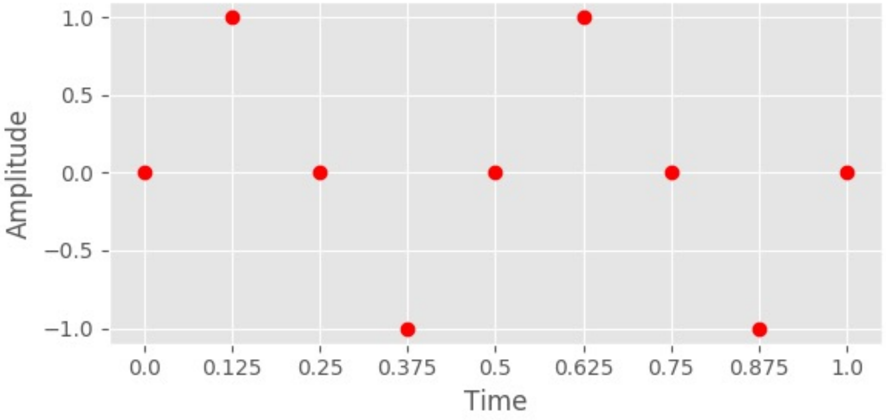
Creating Other Frequencies

Step Size = 1



Our first 9 samples of $N = 8$ samples of a **single** period of a sine wave. At a sample rate of 44.1kHz, we play $44100/8 = 5512.5$ cycles of this sine wave second. Therefore, our frequency is 5512.5Hz

Step Size = 2



Our first 9 samples of $N = 8$ samples of **two** periods of a sine wave. At a sample rate of 44.1kHz, we play $44100/8 = 5512.5$ cycles of this sine wave second. Therefore, our frequency is 5512.5Hz * 2 = 11,025Hz.

Creating Other Frequencies

- We can generalize our discovery from the previous slide to discover the playback rate for any frequency we want to produce where *N* is the number of samples in our table, f_x is the frequency we want to produce, f_s is the sample rate, and s is the playback rate or step size:

$$s = \frac{N f_x}{f_s}$$

- When the playback rate s (we can also think about this as the step size of moving through our table) is an integer, it’s easy to select which samples should be used.
- What should we do for non-integer s?

Creating Other Frequencies

Let’s say I want to generate samples for a wave of 8268.75Hz using our table of 8 samples. Using the formula on the previous slide, this results in a playback rate of 1.5, meaning we step through the table by every 1.5 sample.

Sample #	Amplitude
0	0
1	.707
2	1
3	.707
4	0
5	-.707
6	-1
7	-.707

Generated Samples

0	1.5	3	4.5	6	7.5	1	2.5	4	
0	.707 or 1	.707	0 or -.707	-1	-.707 or 0	.707	1 or .707	0	etc…

What sample should I choose when my value lies **between** two samples in my table?

Truncation/Rounding

Choice based on wavetable	0	.707 or 1	.707	0 or -.707	-1	-.707 or 0	.707	1 or .707	0	etc…
Truncation – choose lower sample's value	0	.707	.707	0	-1	-.707	.707	1	0	etc…
		1.5		4.5		7.5		10.5		
Rounding (based on step size) – choose closer one	0	1	.707	-.707	-1	0	.707	.707	0	etc…
True value given its own wavetable	0	.924	.707	-.383	-1	-.383	.707	.924	0	etc…

Roundup to higher sample's value because step size would roundup

Linear Interpolation

- Both truncation and rounding produce poor approximations when we want a sample that exists between two other samples. It can cause distortion to the wave.
  - Advantage of truncation and rounding: quick to produce a value for all samples
- Another option is to estimate what that value would be.
- Many different approaches we could take.
- Simplest is linear interpolation.
- Others include cubic hermite, polynomial, Lagrangian, sinc interpolation… etc.
  - Efficiency of producing samples varies! Remember we need to generate these samples quickly!

Linear Interpolation

Sample/ Index *n*	Amplitude *S*[*n*]
0	0
1	.707
2	1
3	.707
4	0
5	-.707
6	-1
7	-.707

- Linear interpolation draws a line between the values of two adjacent samples and estimates what the sample value would be based on the step size.
- Assuming we have a step size s = 1.5, then the samples we need from the our table will be samples 0, 1.5, 3, 4.5, 6 … etc. We will need to use linear interpolation to calculate 1.5, 4.5 ... etc.
- Let’s take 1.5 for example. 1.5 is not any of the *n* indices in our wavetable. In other words, $S[1.5]$ is not valid.

Linear Interpolation

Sample/ Index *n*	**Amplitude *S*[*n*]**
0	0
1	.707
2	1
3	.707
4	0
5	-.707
6	-1
7	-.707

$$(n_0, a_0) = (1, .707)$$

$$(n_{0+1}, a_{0+1}) = (2, 1)$$

- Consider the case of sample 1.5 which does not have a valid index n in our wavetable. We will use linear interpolation to calculate that value.
- Using our good friend point-slope form (i.e., $y - y_1 = m(x - x_1)$) which is one form of the equation for line, let's calculate the equation of the line between the lower and higher sample for 1.5 (i.e., 1 and 2). *x* is really our sample index *n* and *y* is our amplitude. Therefore, the lower sample of 1.5 which we call the point (n_0, a_0) is $(1, .707)$, and the upper sample of 1.5 is $(2, 1)$.
- First, calculate *m*. The slope between these two points is 0.293
- Second, use point-slope form to calculate the estimated amplitude. $a = m(n - n_o) + a_o = .293(1.5 - 1) + .707 = .853$
- Rounding produced 1. Truncation produced .707. The true value should be .924.

Linear Interpolation

											Total Error:
Linear interpolation	0	.853	.707	-.354	-1	-.354	.707	.853	0	etc…	.2
Truncation – choose lower value always	0	.707	.707	0	-1	-.707	.707	1	0	etc…	.6
Rounding (based on step size) – choose closer one	0	1	.707	-.707	-1	0	.707	.707	0	etc…	.4
True value given its own wavetable	0	.924	.707	-.383	-1	-.383	.707	.924	0	etc…	0

Interpolation

- Interpolation provides the benefit of rendering more accurate samples. Linear interpolation is just one of many kinds of interpolation used.
- Disadvantage: it takes processor time to generate new samples when they fall outside the constraints of wavetables.
- Linear interpolation does not provide the most accurate results as other forms of interpolation but straddles the duality of accuracy vs. computational expense decently.
- You will explore more in the upcoming assignment!

SinOsc and PlayBuf

From the documentation of SinOsc:

Generates a sine wave. Uses a wavetable lookup oscillator with linear interpolation. Frequency and phase modulation are provided for audio-rate modulation. Technically, SinOsc uses the same implementation as Osc except that its table is fixed to be a sine wave made of 8192 samples.

From the documentation of PlayBuf:

Plays back a sample resident in memory. [For rate]: *interpolation is cubic.*

Experiment with SinOsc

Fiddle around with various frequencies for SinOsc above the Nyquist frequency (i.e., above 22050 Hz for a f_s of 44100Hz). Do we produce audible frequencies? Why? Can you determine a relationship?

```
{SinOsc.ar(<your test frequency, 0, 0.1) ! 2}.play;
```

Sampling Rate

$$f_s = 44100Hz$$

Stepsize

$$s = \frac{Nf_x}{f_s}$$

What about complex sounds?

- So far we have only looked at wavetable synthesis for sine waves but we can use wavetable synthesis for other complex waves like sawtooth or triangle
- Complex sounds require extra precautions because they are a sum of multiple sine waves.
- We need to be sure that our wavetable does not include frequencies above the Nyquist Frequency.
- We need to be cautious that playing back our wavetable at higher rates does not produce aliasing either!

Band Limited Oscillators

- Consider the sawtooth wave which for each harmonic *n* has amplitude $\frac{1}{n}$. A true sawtooth wave is equivalent to the sum of all harmonics from the fundamental to infinity (i.e., $\sum_{n=1}^{\infty} \frac{\sin(2\pi f n t)}{n}$)
- This is of course will generate frequencies **above** the Nyquist frequency.
- Band Limiting an oscillator excludes those frequencies that rise above the Nyquist frequency and usually several more. In fact, that is what we did when we used additive synthesis to build our own sawtooth waves.

👀

Remember

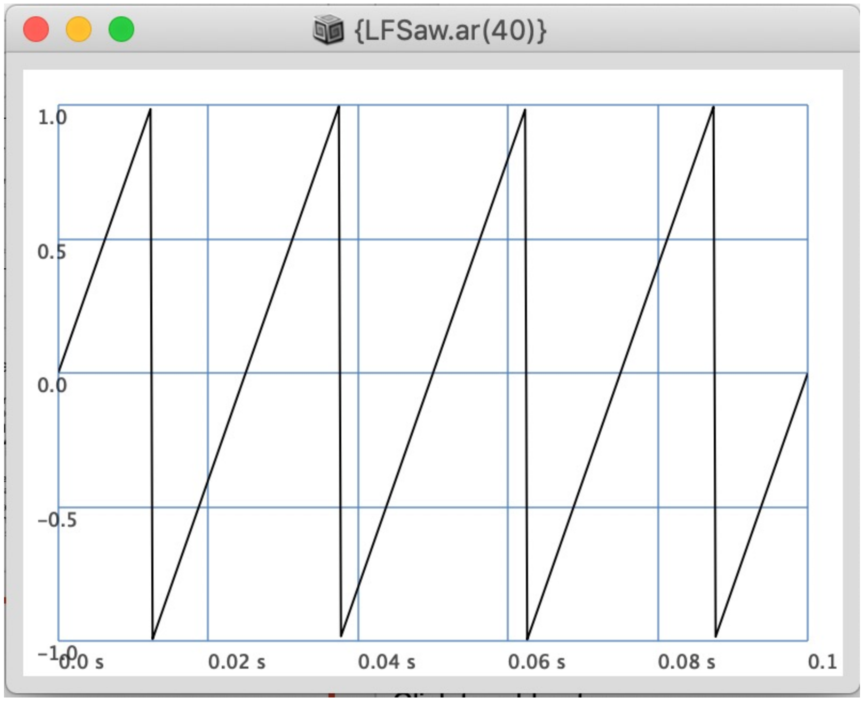
```
(
~saw = {
    arg freq = 300, funAmp = 0.6;
    var sig = {
        |i| // One less than the harmonic num which are one indexed (not zero)
        SinOsc.ar(freq * (i + 1), 0, funAmp/(i + 1)) // Freq and amp come from harmonic number
    }.dup(30) sum; // 30 represents the number of harmonics.  Sum sums the contents of an array
    sig ! 2; // Return the stereo signal.  ! equivalent to dup.
};
)


~saw.plot; // Functions that return a UGen or an array of UGens can be plotted!
~saw.play;
```

Bandlimited vs. Non-bandlimited

- Non-bandlimited waveforms still serve useful purposes especially at lower frequencies.
- A non-bandlimited oscillator like a sawtooth wave with rich harmonics will contain some aliasing but the frequencies will be attenuated in amplitude by having a high harmonic number. So its tolerated.
  - Advantage is a rich harmonic sound because all harmonics are present.
  - Disadvantage is that the aliasing is untenable at higher frequencies
- A bandlimited oscillator will contain fewer harmonics and a less “pure” sound but no aliasing at higher frequencies.
  - Note that sometimes multiple wavetables can be used

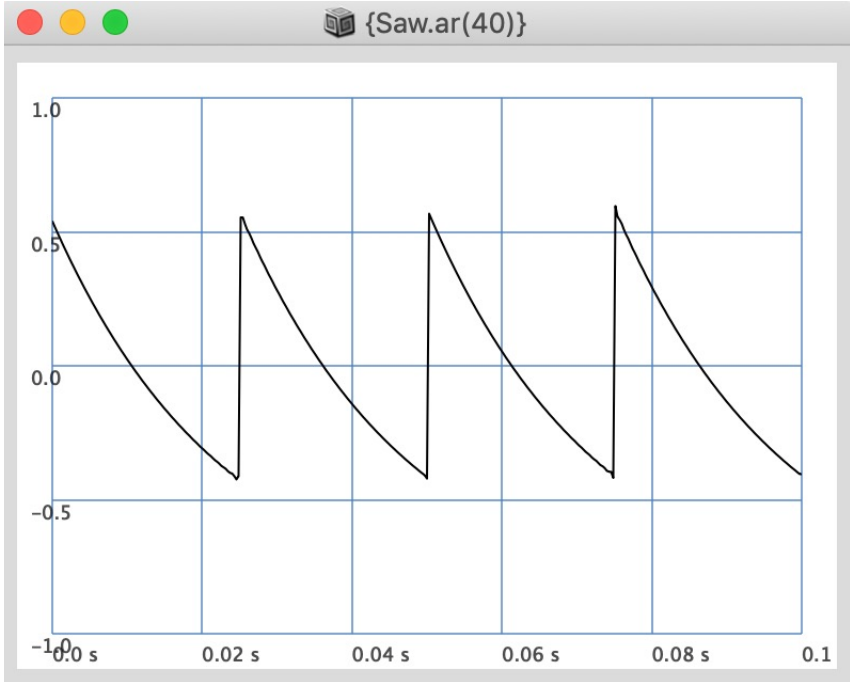
Comparison – Low Frequencies

Non-band limited sawtooth wave



{LFSaw.ar(40)}

Band limited sawtooth wave

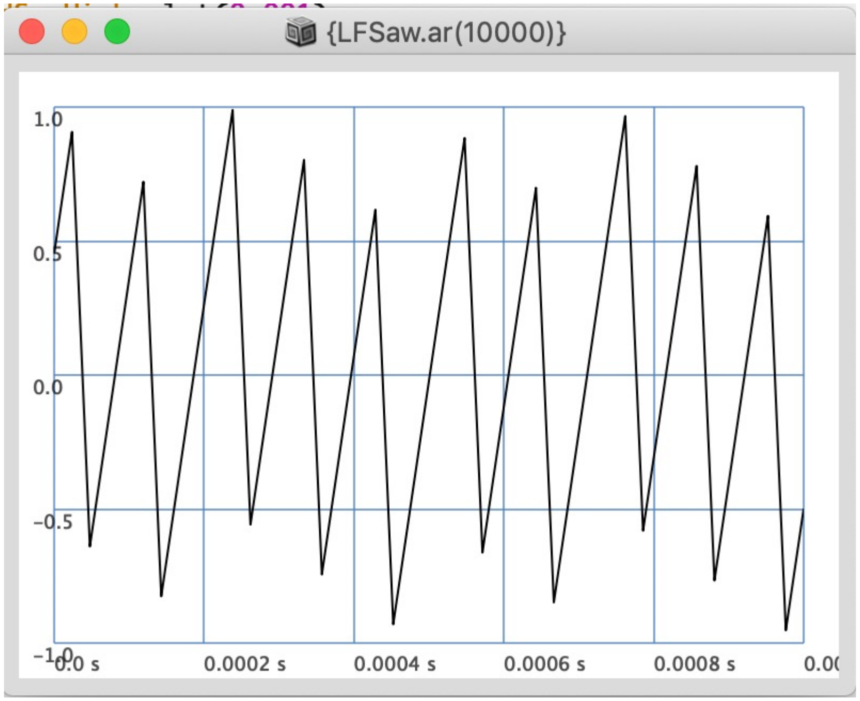


{Saw.ar(40)}

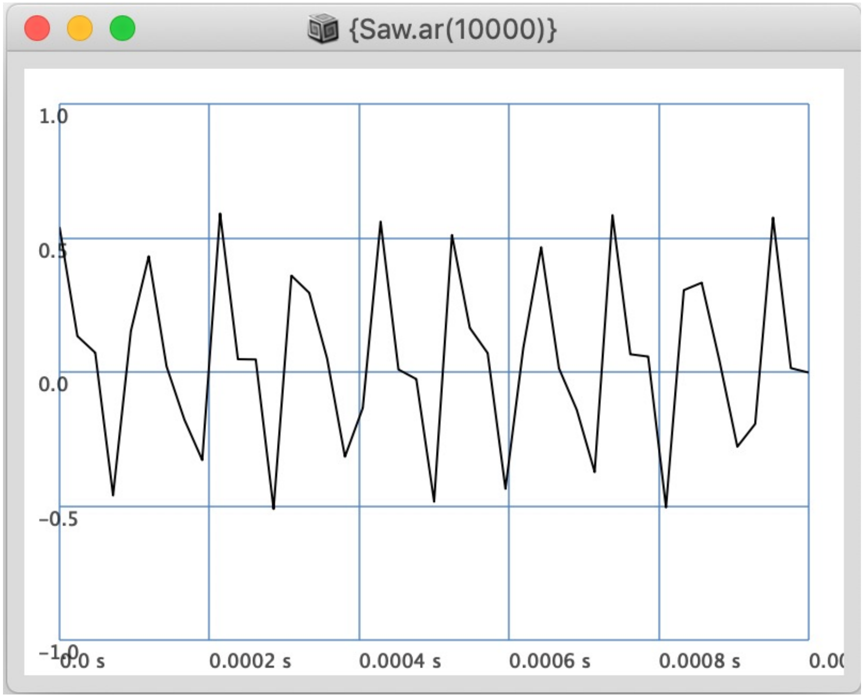
Note the slight curvature in the sawtooth ramp

Comparison – High Frequencies

Non-band limited sawtooth wave



Band limited sawtooth wave



Still distortion because not all harmonics are present

Hearing Aliasing

```
{LFSaw.ar(XLine.kr(20, 15000, 3))!2}.play;
```

Listen to the additional noise and distortion particular
as the frequency hits higher levels.

Oscillators in Supercollider

Most oscillators in SuperCollider use wavetable synthesis and are either bandlimited or non-bandlimited. Non-band limited oscillators are usually named with the prefix “LF”, standing for low frequency and are intended to be used as such.

LFPulse
LFSaw
LFTri
LFPar – parabolic
LFCub – cubic
…etc.

Pulse
Saw
Sin
Blip
…etc.

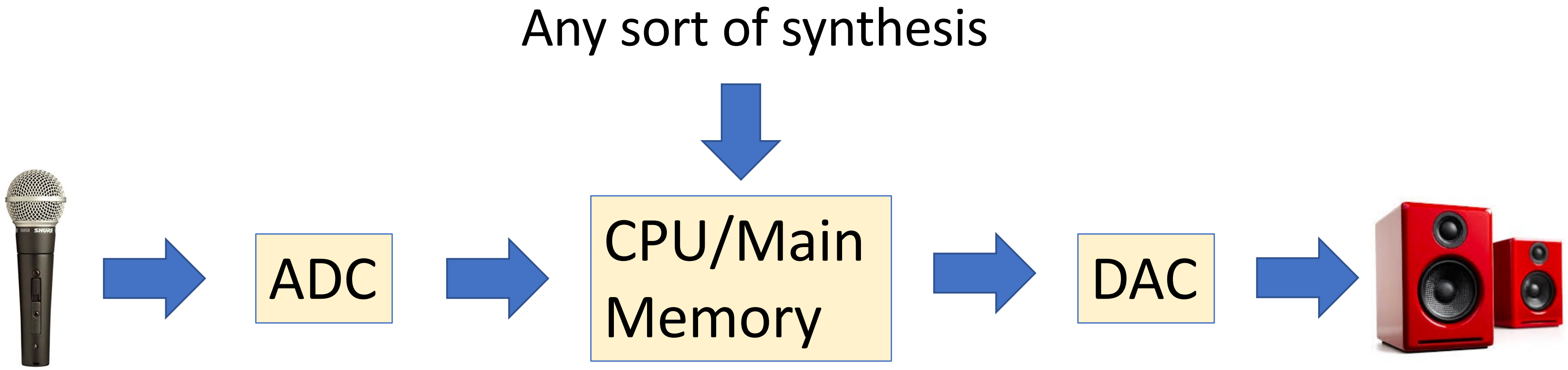
.ar vs .kr for SuperCollider UGens

- SuperCollider calculates audio in groups of samples called *blocks*.
  - Why? Audio must be relayed from your program to your speakers using OS (operating system) calls. I/O system calls are slow. Better to send a chunk instead of individual samples.
- An audio UGen (.ar) will calculate all the samples in the block.
  - Pros: accurate. Necessary for generating audio signals.
  - Cons: less efficient. Matters when dealing with lots of processing.
- A control UGen (.kr) will calculate **one** value for the entire block.
  - Pros: more efficient
  - Cons: less accurate. Okay generally when we want to modulate parameters.
- The default block size in SuperCollider is 64 but can be adjusted. 64 sample block size at $f_s = 44{,}100$ is approximately 689 blocks a second for a period of 0.00145 seconds.

ADC/DAC

- Analog to Digital Converter (ADC) – the process of converting a continuous voltage signal into discrete samples
  - Will often include an anti-aliasing filter to reduce frequencies above the Nyquist frequency
- Digital to Analog Converter (DAC) – the process of converting a digital signal consisting of samples back into a continuous voltage signal
- All computers with sound capabilities must have these two components, either on a separate sound card or on the motherboard

Signal Chain



Any sort of synthesis

ADC

CPU/Main Memory

DAC

Further Topics

- Bit depth
- Dithering
- Quantization
- DC offset
- Frequency Domain
- Much more on hardware implementations of ADC's and DAC's
    - Summing Amplifier
    - Successive-Approximation ADC
    - Etc…

In Summary

- Computers store digital information which is problematic for continuous data like audio signals
- Aliasing occurs when the sample rate is not high enough to accommodate the maximum frequency of an audio signal.
  - We can create aliasing when we sample a continuous signal at too low of a rate.
  - We can create aliasing when we digitally create samples of a signal whose maximum frequency exceeds the Nyquist rate (i.e., `LFPulse`, `LFSaw`, … etc.)
- A solution to classic waveforms is to use bandlimited versions that cap the maximum frequency.