MUSC 208 Winter 2014
John Ellinger Carleton College

The Five Common Synthesizer Waveforms

Many hardware and software synthesizers allow you to select different waveform shapes that are used to construct and mix sounds together. The same waveforms are often available in the LFO's (Low Frequency Oscillators) that add color and motion to the sounds.

The five most common waveforms are:

1. Sine
2. Saw
3. Square
4. Triangle
5. Pulse

They are all included in ChucK. http://chuck.cs.princeton.edu/doc/program/ugen.html

[ugen]: SinOsc

- *sine oscillator*
- *see examples: whirl.ck*

(control parameters)

- **.freq** - (float , READ/WRITE) - *oscillator frequency (Hz), phase-matched*
- **.sfreq** - (float , READ/WRITE) - *oscillator frequency (Hz)*
- **.phase** - (float , READ/WRITE) - *current phase*
- **.sync** - (int , READ/WRITE) - *(0) sync frequency to input, (1) sync phase to input, (2) fm synth*

[ugen]: PulseOsc

- *pulse oscillators*
- *a pulse wave oscillator with variable width.*

(control parameters)

- **.freq** - (float , READ/WRITE) - *oscillator frequency (Hz), phase-matched*
- **.sfreq** - (float , READ/WRITE) - *oscillator frequency (Hz)*
- **.phase** - (float , READ/WRITE) - *current phase*
- **.sync** - (int , READ/WRITE) - *(0) sync frequency to input, (1) sync phase to input, (2) fm synth*
- **.width** - (float , READ/WRITE) - *length of duty cycle (0-1)*

[ugen]: SqrOsc

- *square wave oscillator (pulse with fixed width of 0.5)*

(control parameters)

- **.freq** - (float , READ/WRITE) - *oscillator frequency (Hz), phase-matched*
- **.sfreq** - (float , READ/WRITE) - *oscillator frequency (Hz)*
- **.phase** - (float , READ/WRITE) - *current phase*

- **.sync** - (int , READ/WRITE) - *(0) sync frequency to input, (1) sync phase to input, (2) fm synth*
- **.width** - (int , READ/WRITE) - *length of duty cycle (0 to 1)*

[ugen]: TriOsc

- *triangle wave oscillator*

(control parameters)

- **.freq** - (float , READ/WRITE) - *oscillator frequency (Hz), phase-matched*
- **.sfreq** - (float , READ/WRITE) - *oscillator frequency (Hz)*
- **.phase** - (float , READ/WRITE) - *current phase*
- **.sync** - (int , READ/WRITE) - *(0) sync frequency to input, (1) sync phase to input, (2) fm synth*
- **.width** - (float , READ/WRITE) - *control midpoint of triangle (0 to 1)*

[ugen]: SawOsc

- *sawtooth wave oscillator (triangle, width forced to 0.0 or 1.0)*

(control parameters)

- **.freq** - (float , READ/WRITE) - *oscillator frequency (Hz), phase-matched*
- **.sfreq** - (float , READ/WRITE) - *oscillator frequency (Hz)*
- **.phase** - (float , READ/WRITE) - *current phase*
- **.sync** - (int , READ/WRITE) - *(0) sync frequency to input, (1) sync phase to input, (2) fm synth*
- **.width** - (float , READ/WRITE) - *increasing (w > 0.5) or decreasing (w < 0.5)*

We've used the (SinOsc). Let's listen to the others.

```
// fiveWaveforms
// John Ellinger Music 208 Winter 2014
SinOsc sinwav;
SawOsc sawwav;
SqrOsc sqrwav;
TriOsc triwav;
PulseOsc pulsewav;
// default frequency is 220 Hz
0.7 => sinwav.gain => sawwav.gain => sqrwav.gain
    => triwav.gain => pulsewav.gain;

dac => WvOut w => blackhole;
"fiveWavforms.wav" => w.wavFilename;
```

```
750::ms => dur dura;

<<< "Sine wave" >>>;
sinwav => dac;
dura => now;
sinwav =< dac; // disconnect;
dura => now;

<<< "Sawtooth wave" >>>;
sawwav => dac;
dura => now;
sawwav =< dac; // disconnect;
dura => now;

<<< "Square wave" >>>;
sqrwav => dac;
dura => now;
sqrwav =< dac; // disconnect;
dura => now;

<<< "Triangle wave" >>>;
triwav => dac;
dura => now;
triwav =< dac; // disconnect;
dura => now;

<<< "Pulse wave, width = .05" >>>;
.05 => pulsewav.width;
pulsewav => dac;
dura => now;
pulsewav =< dac; // disconnect;
dura => now;

<<< "Pulse wave, width = .25" >>>;
.25 => pulsewav.width;
pulsewav => dac;
dura => now;
pulsewav =< dac; // disconnect;
dura => now;
<<< "Pulse wave, width = .50, a Square wave" >>>;
```

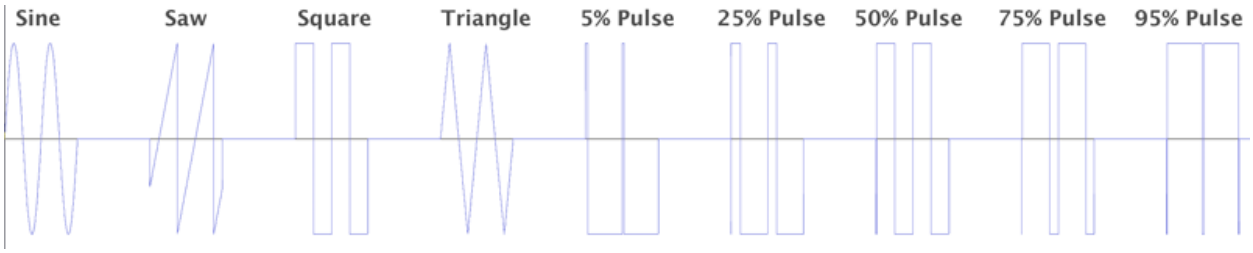
```
.50 => pulsewav.width; // square wave
pulsewav => dac;
dura => now;
pulsewav =< dac; // disconnect;
dura => now;

<<< "Pulse wave, width = .75" >>>;
.75 => pulsewav.width;
pulsewav => dac;
dura => now;
pulsewav =< dac; // disconnect;
dura => now;

<<< "Pulse wave, width = .95" >>>;
.95 => pulsewav.width;
pulsewav => dac;
dura => now;
pulsewav =< dac; // disconnect;
dura => now;
null @=> w;
```

Examine The Five Waveforms In Audacity

Change the duration to 400::samp and run the program again

```
400::samp => dur dura;
```

Look at the waveforms in Audacity.

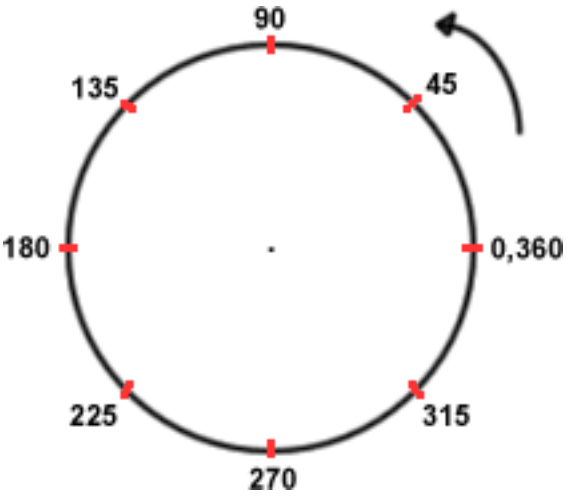


Sine And Cosine Phase

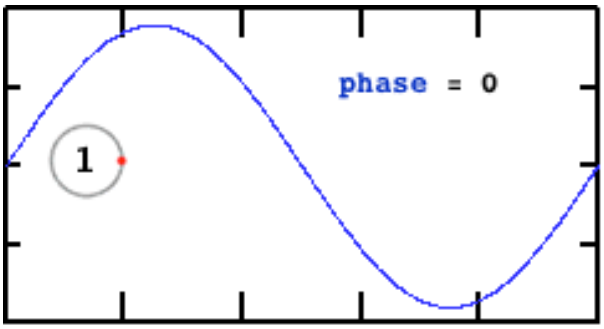
Phase refers to the starting point of the sine wave. Sine waves and cosine waves are π/2 radians or 90° out of phase with each other.

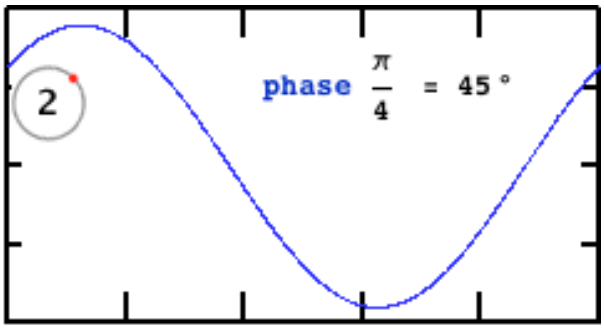
$$\cos(\theta) = \sin\left(\theta + \frac{\pi}{2}\right)$$

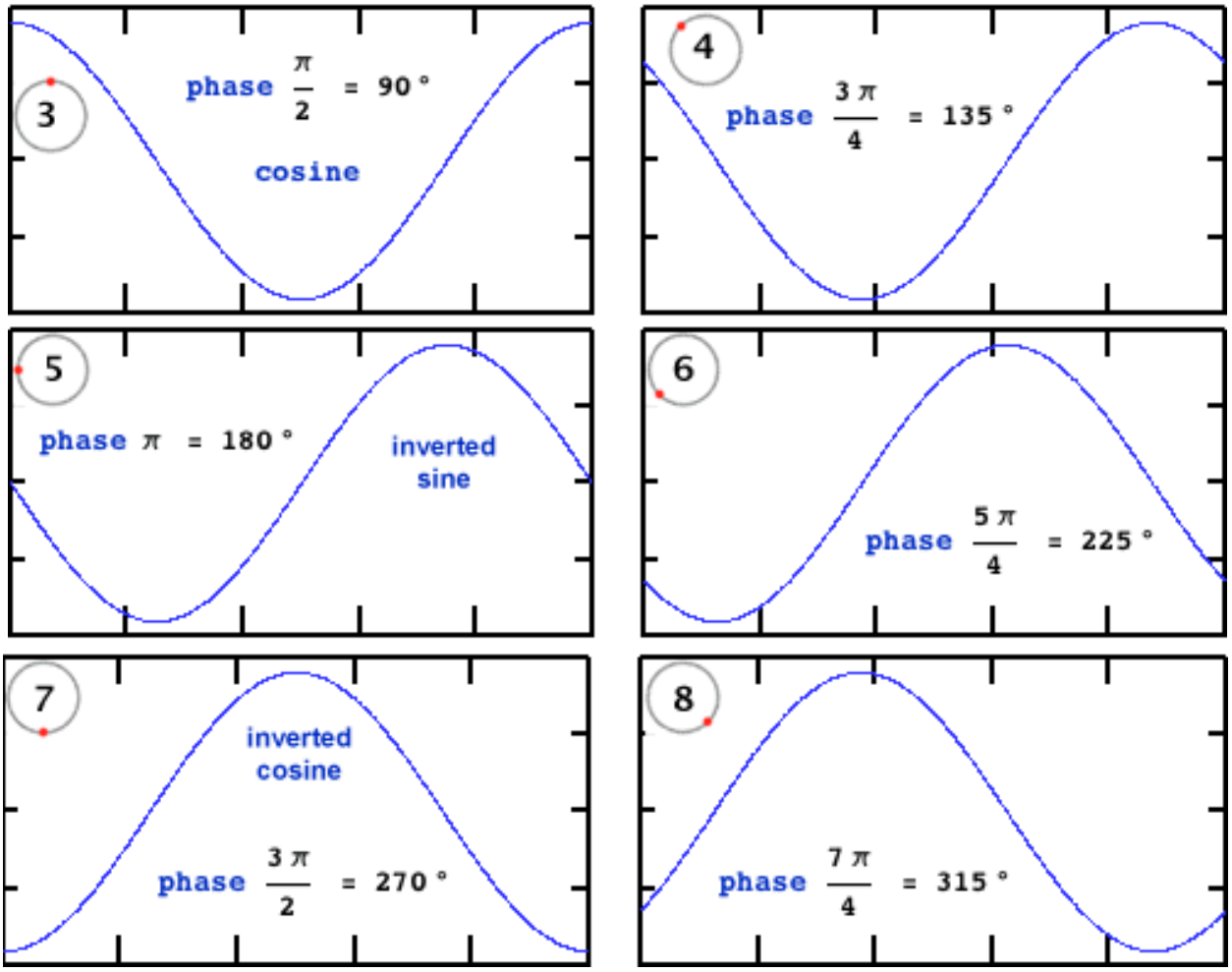
This table shows phase relationships between fractions of one revolution, degrees, and radians.



These Octave plots show eight sine waves incrementally phase shifted by π/4 radians.



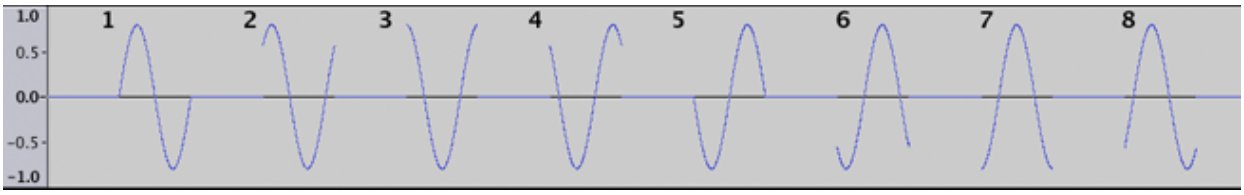





```
// phaseTest1.ck
// John Ellinger Music 208 Winter2014
100 => float freq;
second/samp => float SR;
(SR / freq) + 1 => float sampPeriodPlus1;
<<< "sampPeriodPlus1", sampPeriodPlus1 >>>;

SinOsc s1; // don't connect yet
freq => s1.freq;
0.8 => s1.gain;
// Setup WvOut
dac => WvOut w => blackhole;
"phaseTest1.wav" => w.wavFilename; // or
w.wavFilename("phaseTest1.wav");
```

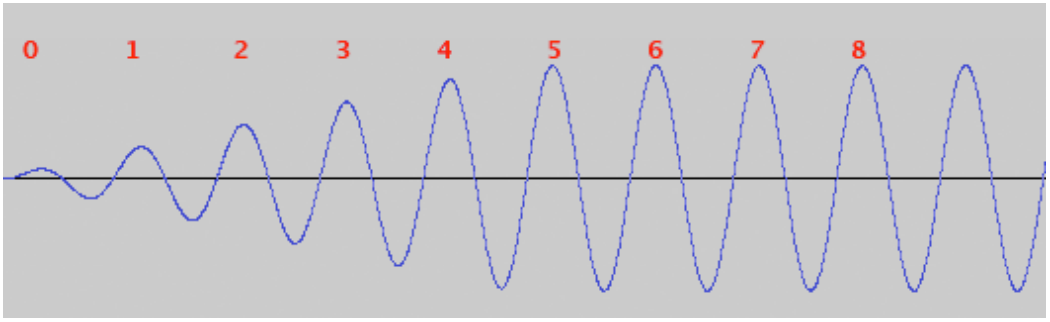
```
// ChucK phase increment = ?/4
8.0 => float phaseIncrement;
// delay
sampPeriodPlus1::samp => now;
for ( 0 => int ix; ix < 8; ix++ )
{
    s1 => dac; // connect sound
    ix / phaseIncrement => s1.phase;
    sampPeriodPlus1::samp => now;
    s1 =< dac; // disconnect sound and delay
    sampPeriodPlus1::samp => now;
}
null @=> w; // close WvOut file
```

The vertical lines at beginning and ending of the sound have been removed to better illustrate the starting phase.




```
// phaseTest2.ck
// John Ellinger Music 208 Winter2014

SinOsc s1 => Envelope e => dac;
220 => s1.freq;
0.8 => s1.gain;
8.0 => float phaseIncrement;
500::ms => dur dura;

function void playPhases( float phase, dur dura )
{
    1 => e.keyOn;
    phase => s1.phase;
    dura => now;
    1 => e.keyOff;
    dura => now;
}
```

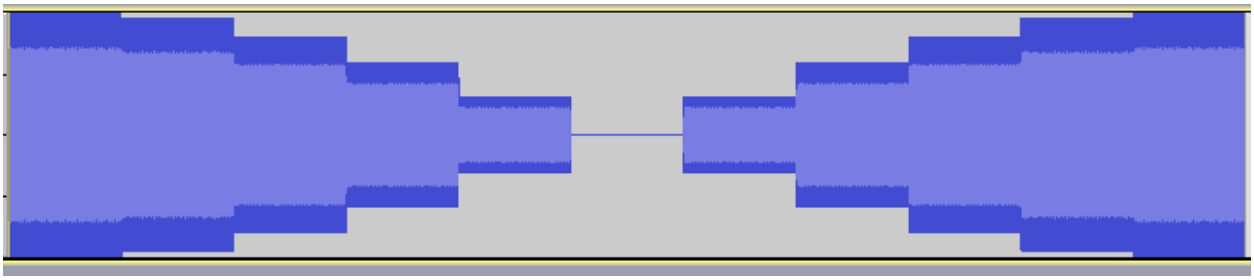
```
// Play one at a time
for ( 0 => int ix; ix < 8; ix++ )
{
    playPhases( ix / phaseIncrement, 500::ms );
}

// delay
1:: second => now;

// Play all at once
dac => WvOut w => blackhole;
"phaseTest2.wav" => w.wavFilename;
spork ~ playPhases( 0, 1::second );
spork ~ playPhases( 1.0/phaseIncrement, 1::second );
spork ~ playPhases( 2.0/phaseIncrement, 1::second);
spork ~ playPhases( 3.0/phaseIncrement, 1::second );
spork ~ playPhases( 4.0/phaseIncrement, 1::second );
spork ~ playPhases( 5.0/phaseIncrement, 1::second );
spork ~ playPhases( 6.0/phaseIncrement, 1::second );
spork ~ playPhases( 7.0/phaseIncrement, 1::second );
// play
1:: second => now;
null @=> w; // close WvOut file
```



The sum of several sine waves of the same frequency but with different phases results in a sine wave of the same frequency but possibly different amplitude. Phase has no effect on the frequency we hear.

When a waveform is combined with a copy of itself phase shifted by 180°, the two waves cancel each other and nothing will be heard.

```
// phaseTest3.ck
// John Ellinger Music 208 Winter2014

SinOsc s1 => dac;
SinOsc s2 => dac;
220 => s1.freq => s2.freq;
0.5 => s1.gain => s2.gain;
0 => s1.phase;

dac => WvOut w => blackhole;
w.wavFilename( "phaseTest3.wav" );

for ( 0 => int ix; ix <= 10; ix++ )
{
    ix / 10.0 => s2.phase; // 180 degrees out of phase
    <<< "Phase", s2.phase() >>>;
    1::second => dur dura;
    dura => now;

}
null @=> w; // close WvOut file
```



ChucK Phase

ChucK phase is measured as percentage of rotation around the unit circle. This table shows common measurement units of rotation.

Units				Values					
Revolutions	0	$1/8$	$1/4$	$3/8$	$1/2$	$5/8$	$3/4$	$7/8$	1
Degrees	0	45	90	135	180	225	270	315	360
Radians	0	$\frac{\pi}{4}$	$\frac{\pi}{2}$	$\frac{3\pi}{4}$	π	$\frac{5\pi}{4}$	$\frac{3\pi}{2}$	$\frac{7\pi}{4}$	2π
Chuck Phase	0	0.125	0.25	0.375	0.5	0.625	0.75	0.875	1.0

Angular Frequency Of The Sampling Rate

Divide the unit circle into 44100 pie slices. One revolution around the unit circle in one second is 1 Hz or 44100 ssamples per second or 2π radians per second. The angle of each slice in radians is the sampling rate phase increment. In the world of digital sound phase is measured in radians. Radians per second is called the angular frequency and is denoted by $\omega = 2\pi f$.

```
// phaseTest4.ck
// John Ellinger Music 208, Winter 2015
2*pi => float TWO_PI;
<<< "TWO_PI", TWO_PI >>>;

2*Math.PI => TWO_PI;
<<< "TWO_PI", TWO_PI >>>;

Math.TWO_PI => TWO_PI;
<<< "TWO_PI", TWO_PI >>>;

second/samp => float SR;
SR * 5.001 => float totalSamples;
SR / TWO_PI => float sr_radianFrequency;
1/sr_radianFrequency => float sr_phaseIncrement;
```

```
// Print variables
<<< "SR in samples per second", SR>>>;
<<< "sr_radianFrequency in radians per second",
sr_radianFrequency >>>;
<<< "sr_phaseIncrement in seconds per radian", sr_phaseIncrement
>>>;
<<< "totalSamples", totalSamples >>>;

0 => float sr_phase;
0 => int wrapAround;
for ( 1 => int n; n <= totalSamples; n++ )
{
    sr_phase + sr_phaseIncrement => sr_phase;

    if ( sr_phase >= TWO_PI )
    {
        wrapAround++;
        <<< "wrapAround", wrapAround, "at sample", n, "sr_phase",
sr_phase >>>;
        sr_phase - TWO_PI => sr_phase;
        <<< "====> sr_phase reset to", sr_phase >>>;
    }
}
```

Note that the phase wraps around 2π radioans (44100 samples) once per second.

```
[chuck](VM): sporking incoming shred: 1 (phaseTest4.ck)...
TWO_PI 6.283185
TWO_PI 6.283185
TWO_PI 6.283185
SR in samples per second 44100.000000
sr_radianFrequency in radians per second 7018.732990
sr_phaseIncrement in seconds per radian 0.000142
totalSamples 220544.100000
wrapAround 1 at sample 44100 sr_phase 6.283185
====> sr_phase reset to 0.000000
wrapAround 2 at sample 88200 sr_phase 6.283185
====> sr_phase reset to 0.000000
wrapAround 3 at sample 132300 sr_phase 6.283185
====> sr_phase reset to 0.000000
wrapAround 4 at sample 176400 sr_phase 6.283185
====> sr_phase reset to 0.000000
wrapAround 5 at sample 220500 sr_phase 6.283185
====> sr_phase reset to 0.000000
```

Phase Increment Of Any Frequency

The phase increment for any frequency is that frequency multiplied by the sampling rate phase increment. In this case 2π is one period of that frequency.

```
// phaseTest5.ck
// John Ellinger Music 208 Winter 2014

Math.TWO_PI => float TWO_PI;
second/samp => float SR;
441.0 => float freq;
// period = 100 samples
SR / freq => float period;
TWO_PI / SR => float sr_phaseIncrement;
sr_phaseIncrement * freq => float freq_phaseIncrement;
501 => float totalSamples;

// Print variables
<<< "TWO_PI", TWO_PI >>>;
<<< "SR", SR >>>;
<<< "sr_phaseIncrement", sr_phaseIncrement >>>;
<<< "freq", freq >>>;
<<< "period", period >>>;
<<< "sr_phaseIncrement", sr_phaseIncrement >>>;
<<< "freq_phaseIncrement", freq_phaseIncrement >>>;
<<< "totalSamples", totalSamples >>>;

0 => float sr_phase;
0 => float freq_phase;
0 => int numPeriods;
0 => int wrapAround;

for ( 0 => int n; n < totalSamples; n++ )
{
    freq_phase + freq_phaseIncrement => freq_phase;

    if ( freq_phase >= TWO_PI )
    {
        freq_phase - TWO_PI => freq_phase;
        wrapAround++;
```

```
        <<< "wrapAround", wrapAround, "at sample", n,
            "sr_phase", sr_phase, "freq_phase", freq_phase >>>;
        freq_phase / TWO_PI => freq_phase;
        <<< "====> freq_phase reset to", freq_phase >>>;
    }
}
```

Notice that the sample rate phase wraps at multiples of the number of samples in one period of the frequency 441 Hz and that the freq_phase begins at different phases at each wrap around.

```
[chuck](VM): sporking incoming shred: 1 (phaseTest5.ck)...
TWO_PI 6.283185
SR 44100.000000
sr_phaseIncrement 0.000142
freq 441.000000
period 100.000000
sr_phaseIncrement 0.000142
freq_phaseIncrement 0.062832
totalSamples 501.000000
wrapAround 1 at sample 100 sr_phase 0.000000 freq_phase 0.062832
====> freq_phase reset to 0.010000
wrapAround 2 at sample 200 sr_phase 0.000000 freq_phase 0.010000
====> freq_phase reset to 0.001592
wrapAround 3 at sample 300 sr_phase 0.000000 freq_phase 0.001592
====> freq_phase reset to 0.000253
wrapAround 4 at sample 400 sr_phase 0.000000 freq_phase 0.000253
====> freq_phase reset to 0.000040
wrapAround 5 at sample 500 sr_phase 0.000000 freq_phase 0.000040
====> freq_phase reset to 0.000006
```

Let's see what a frequency of 300 Hz would sound like.

PhaseTest6.ck

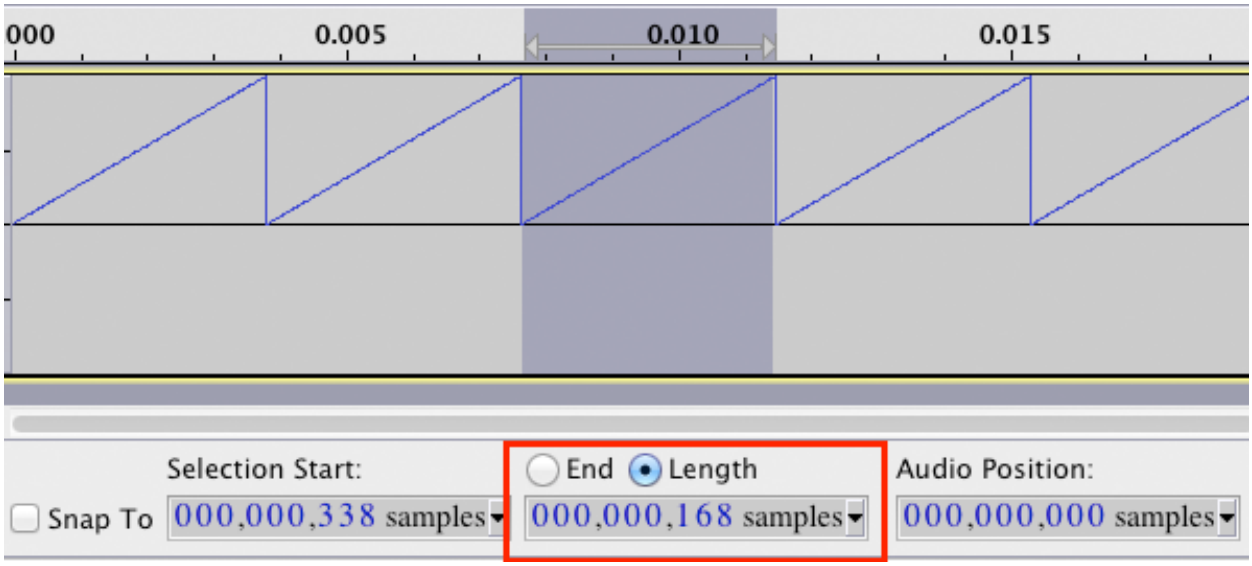
```
// phaseTest6.ck
// John Ellinger Music 208 Winter 2014
Impulse imp => Gain g => dac => WvOut w => blackhole;
w.wavFilename( "phase6Test.wav" );

Math.TWO_PI => float TWO_PI;
second/samp => float SR;
```

```
Std.mtof( 60 ) => float freq;
// period = 100 samples
SR / freq => float period;
TWO_PI / SR => float sr_phaseIncrement;
sr_phaseIncrement * freq => float freq_phaseIncrement;
1000 => float totalSamples;

// Print variables
<<< "TWO_PI", TWO_PI >>>;
<<< "SR", SR >>>;
<<< "freq in Hz", freq >>>;
<<< "period in samples", period >>>;
<<< "sr_phaseIncrement in radians", sr_phaseIncrement >>>;
<<< "freq_phaseIncrement in radians", freq_phaseIncrement >>>;
<<< "totalSamples", totalSamples >>>;

0 => float sr_phase;
0 => float freq_phase;
0 => int numPeriods;
0 => int wrapAround;

for ( 0 => int n; n < totalSamples; n++ )
{
    freq_phase + freq_phaseIncrement => freq_phase;

    if ( freq_phase >= TWO_PI )
    {
        freq_phase - TWO_PI => freq_phase;
        wrapAround++;
        <<< "wrapAround", wrapAround, "at sample", n,
           "sr_phase", sr_phase, "freq_phase", freq_phase >>>;
        freq_phase / TWO_PI => freq_phase;
         <<< "====> freq_phase reset to", freq_phase >>>;
    }

    freq_phase / TWO_PI  => imp.next;
    1::samp => now;
}

null @=> w; // close file
```

The outpt reports that there are 168.5 samples in one period of a 261.62 Hz frequency (Middle C).

```
[chuck](VM): sporking incoming shred: 1 (phase6Test.ck)...
TWO_PI 6.283185
SR 44100.000000
freq in Hz 261.625565
period in samples 168.561509
sr_phaseIncrement in radians 0.000142
freq_phaseIncrement in radians 0.037275
totalSamples 1000.000000
wrapAround 1 at sample 168 sr_phase 0.000000 freq_phase 0.016345
====> freq_phase reset to 0.002601
wrapAround 2 at sample 337 sr_phase 0.000000 freq_phase 0.018946
====> freq_phase reset to 0.003015
wrapAround 3 at sample 506 sr_phase 0.000000 freq_phase 0.019360
====> freq_phase reset to 0.003081
wrapAround 4 at sample 675 sr_phase 0.000000 freq_phase 0.019426
====> freq_phase reset to 0.003092
wrapAround 5 at sample 844 sr_phase 0.000000 freq_phase 0.019437
====> freq_phase reset to 0.003093
```

Open phase6Test.wav in Audacity and display the number of samples in one period. We just generated a unipolar ramp or sawtooth wave with amplitudes between 0 and 1.0.



Change a Normalized Unipolar Wave to a Normalized Bipolar Wave

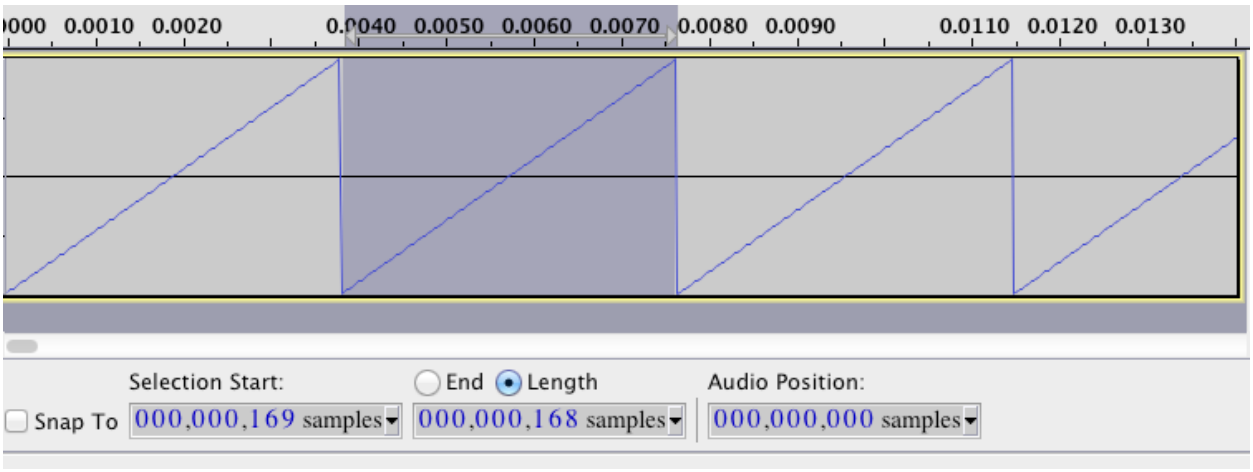
Normalized means the amplitude range is 0 – 1.0 for a unipolar wave and ±1.0 for a bipolar wave. We can turn any normalized unipolar wave into a normalized bipolar wave in two simple steps.

1. Multiply all amplitudes by 2.
2. and subtract 1.0.

Save phase6Test.ck as phase7Test.ck and make these changes.

```
// phase7Test.ck
// John Ellinger Music 208 Winter 2014
Impulse imp => Gain g => dac => WvOut w => blackhole;
w.wavFilename( "phase7Test.wav" );

( more code )

for ( 0 => int n; n < totalSamples; n++ )
{
    freq_phase + freq_phaseIncrement => freq_phase;

    if ( freq_phase >= TWO_PI )
    {
        freq_phase - TWO_PI => freq_phase;
        wrapAround++;
        // <<< "wrapAround", wrapAround, "at sample", n,
        //   "sr_phase", sr_phase, "freq_phase", freq_phase >>>;
        (freq_phase / TWO_PI) => freq_phase;
        // <<< "====> freq_phase reset to", freq_phase >>>;
    }

    ( 2 * freq_phase / TWO_PI ) - 1  => imp.next;
    1::samp => now;
}
```

Run the code and look at the phase7Test.wav in Audacity.



Build A Sawtooth Wave Using The Phase Increment Formula

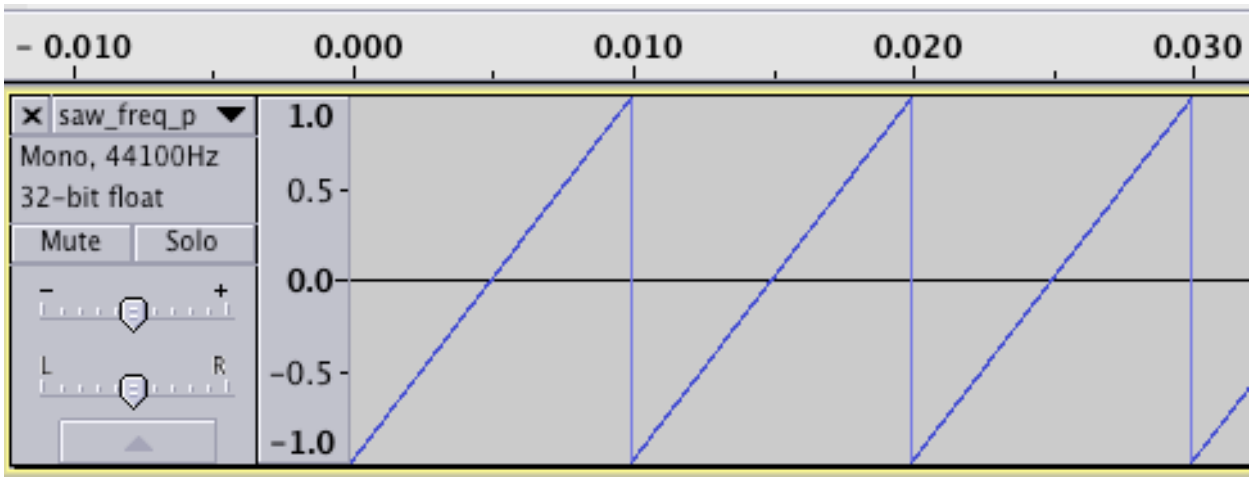
Save phase7Test.ck as genSaw.ck. Modify phase7Test.ck to create a genSaw() function that will generate sawtooth wave for any amplitude, frequency, and duration.

```
// genSaw.ck
// John Ellinger Music 208 Winter 2014
Impulse imp => Gain g => dac => WvOut w => blackhole;
w.wavFilename( "genSaw.wav" );

function void genSaw( float amp, float freq, dur dura )
{
    second / samp => float SR;
    dura / samp => float totalSamples;
    Math.TWO_PI => float TWO_PI;
    TWO_PI / SR => float sr_phaseIncrement;
    sr_phaseIncrement * freq => float freq_phaseIncrement;

    0 => float sr_phase;
    0 => float freq_phase;
    for ( 0 => int n; n < totalSamples; n++ )
    {
        freq_phase + freq_phaseIncrement => freq_phase;
```

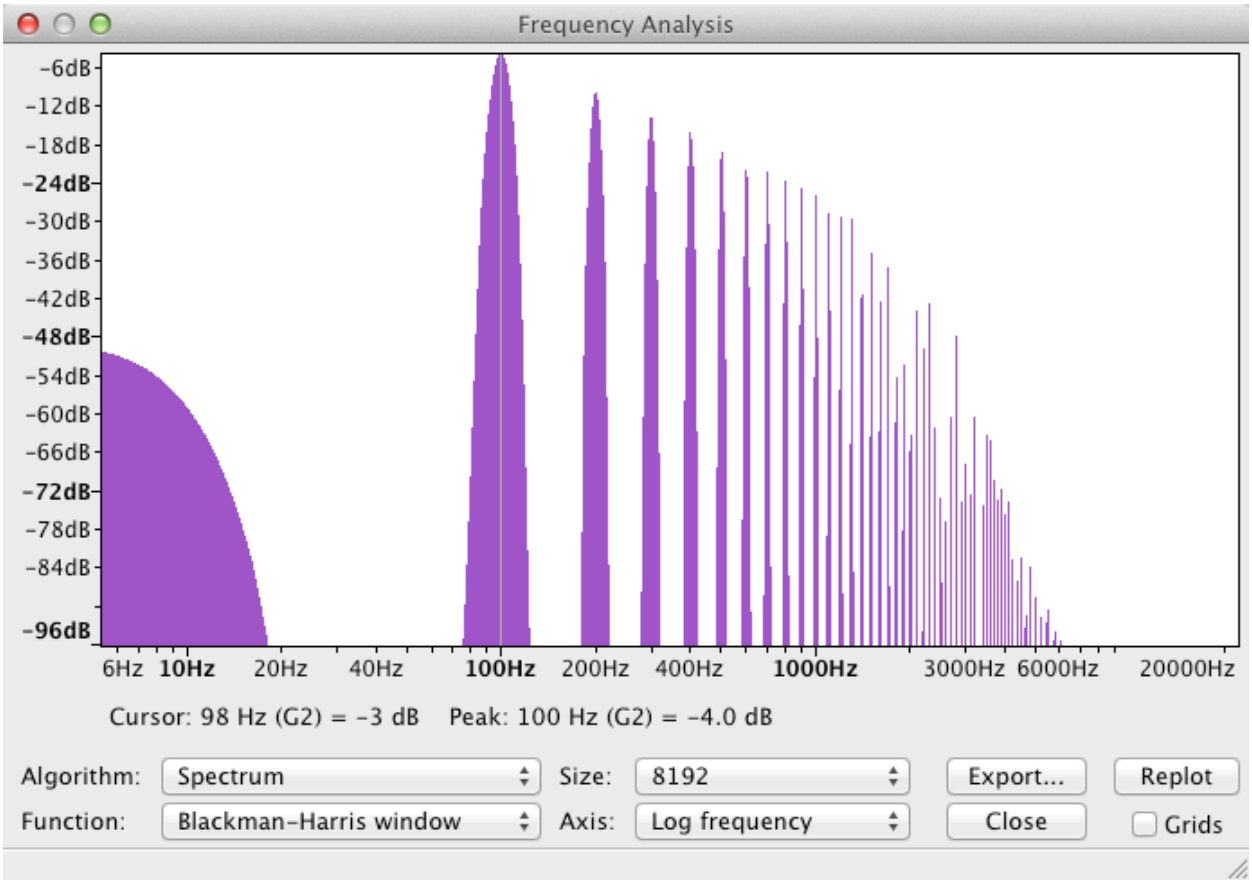
```
        if ( freq_phase >= TWO_PI )
        {
            freq_phase - TWO_PI => freq_phase;
            (freq_phase / TWO_PI) => freq_phase;
        }
        ( 2 * freq_phase / TWO_PI ) - 1  => imp.next;
        1::samp => now;
    }
}
```

```
genSaw( 1.0, 100, 1000::ms );
null @=> w; // close file
```

Open saw_phaseIncrement in Audacity and look at the waveform.

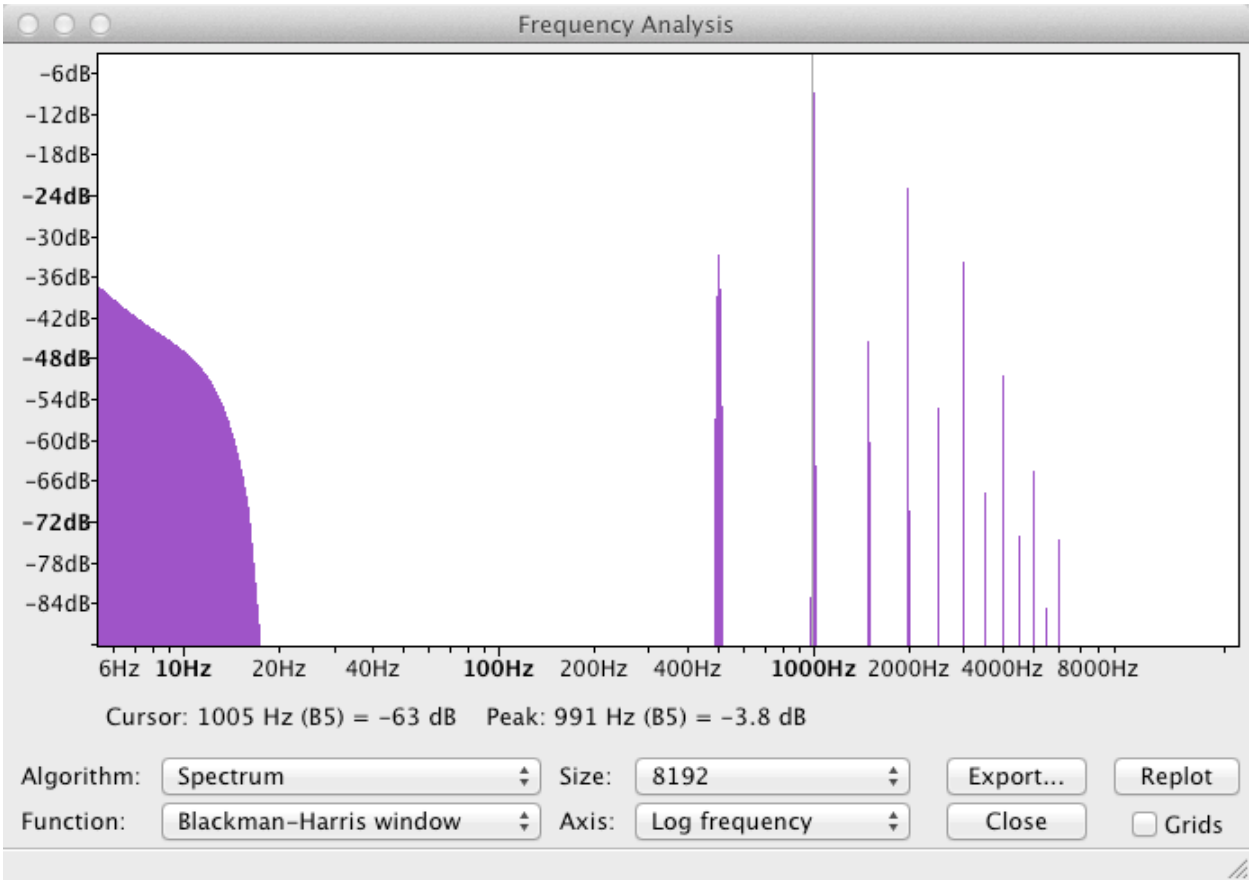


Plot the Spectrum.



As you move the cursor over each peak you'll observe the Peak frequencies occur in multiples of 100 Hz, the fundamental frequency.

Try it again with a frequency of 1000 Hz. Open the wav file and plot the spectrum in Audacity. You can see the largest peak near 1000 Hz but there is also a large peak around 500 Hz. Evidently this version of the genSaw does not produce bandlimited waveforms.

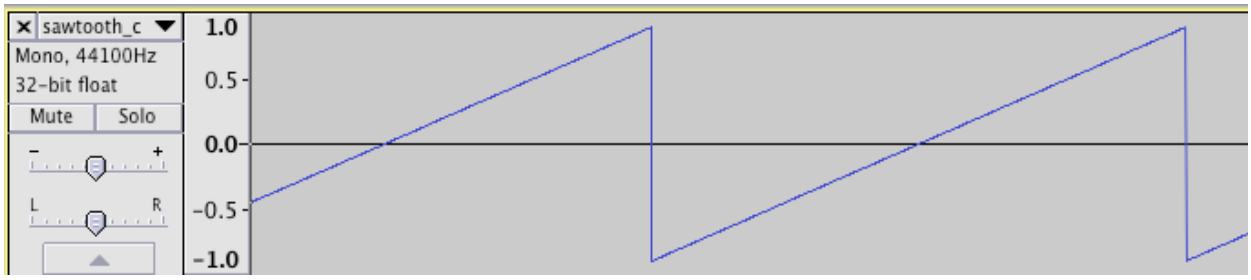


The Chuck SawOsc Object

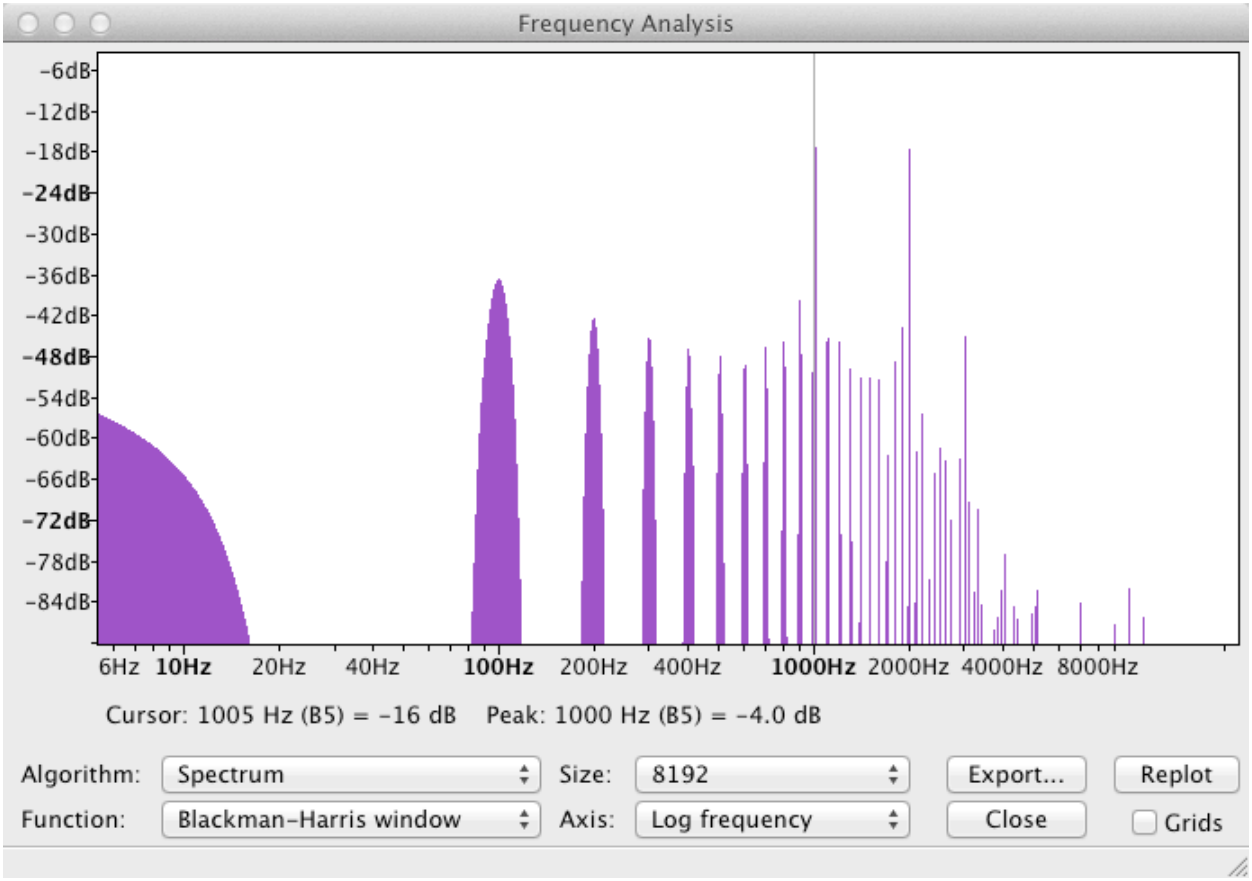
Let's see if the ChucK sawtoogh can do any better.

```
// chuckSaw.ck
// John Ellinger Music 208 Winter 2014
SawOsc saw => dac => WvOut w => blackhole;
"chuckSaw.wav" => w.wavFilename;
1000 => saw.freq;
1.0 => saw.gain;
1000::ms => now;
null @=> w; // close WvOut file
```

Open chuckSaw.wav in Audacity. The ChucK wave is looks similar but does not start at –1, as the genSaw did.



The spectrum has even more aliased frequencies.



Bandlimited Waveforms

In order to create a bandlimited sawtooth wave, we need to keep all frequencies below SR/2 = 22050 Hz. The solution for doing that comes from the Fourier Series.

The Fourier Series

Jean Baptiste Joseph Fourier (1768-1830) developed the Fourier during his study of heat conduction. Hundreds of books, treatises and college and graduate school mathematics and engineering courses are devoted to the study of Fourier analysis.

The basic premise of Fourier's theorem states that: ***any*** continuous periodic waveform can be transformed into the sum of simple sine and cosine waves of varying amplitudes and phases at ***integer*** multiples of a fundamental frequency. For audio signals it can be expressed like this:

$$x(t) = \sum_{n=1}^{\infty} A_n \sin(2\pi n f_0 t + \theta_n)$$

This formula says that a sawtooth wave, a square wave, a triangle wave, a piano sound, a violin sound, or any periodic sound can be computed by carefully choosing the amplitude and phase values and summing the Fourier series with a fundamental frequency of f_0.

The Five Common Synthesis Waveforms

1. Sine
2. Sawtooth
3. Square
4. Triangle
5. Pulse

Each of the five common waveforms can be constructed mathematically as a sum of harmonically related sine waves.

Sawtooth Wave Synthesis

The formula for a sawtooth wave consructed from a sum of sine waves at integer multiples of a fundamental frequency is shown below.

$$Sawtooth = \frac{2}{\pi}\sum_{n=1}^{\infty}\frac{1}{n}\sin(n\omega)$$

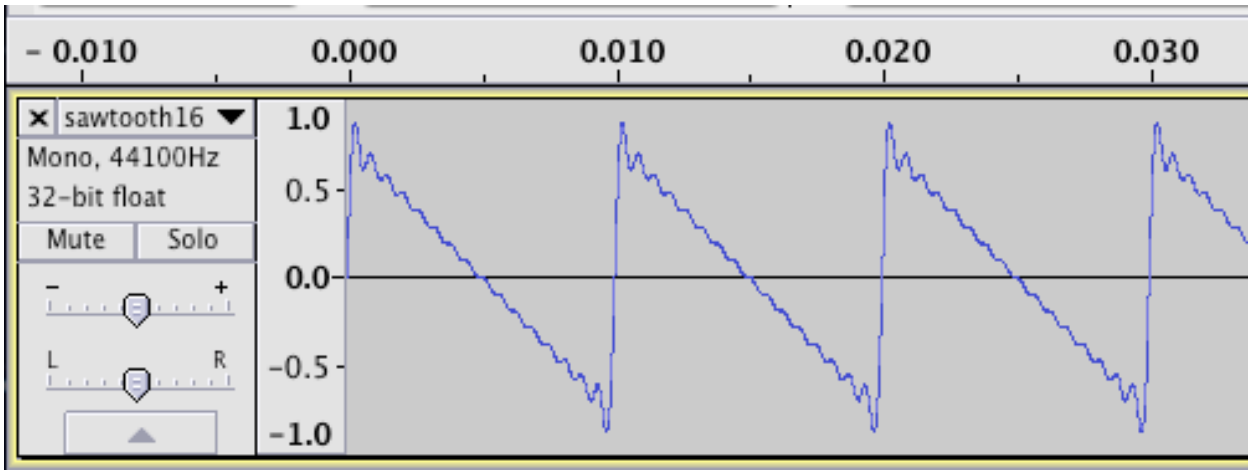
$$= \frac{2}{\pi}\left(\sin(\omega)+\frac{1}{2}\sin(2\omega)+\frac{1}{3}\sin(3\omega)+\frac{1}{4}\sin(4\omega)+\frac{1}{5}\sin(5\omega)...\right)$$

$$\omega = 2\pi f_0$$

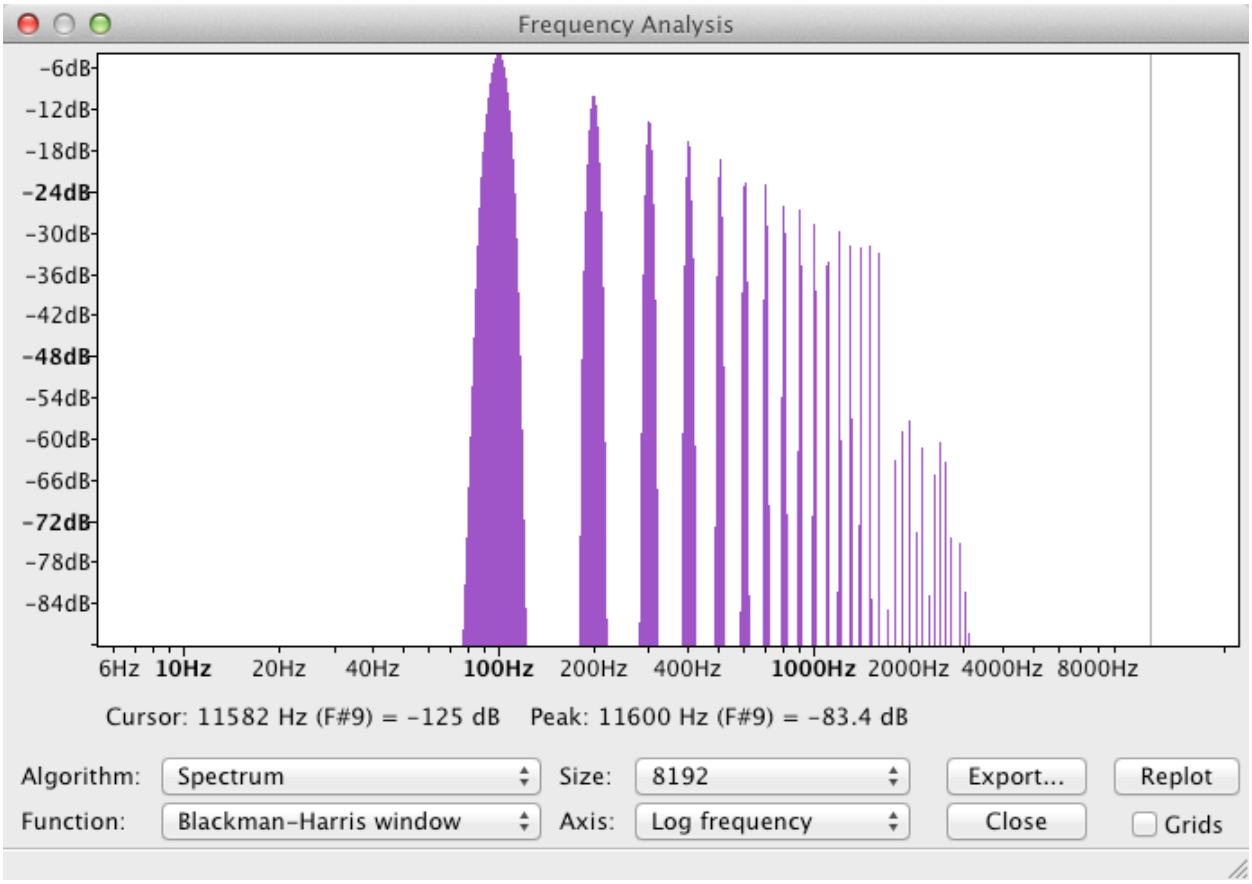
```
// sawtooth_fourierSeries
// John Ellinger Music 208 Winter2014

16 => int numHarmonics;
// create an arry with one extra SinOsc
// arrays begin at 0, harmonics begin at 1
// we'll ignore sinRA[0]
SinOsc sinRA[ numHarmonics + 1 ];

dac => WvOut w => blackhole;
"sawtooth_fourier.wav" => w.wavFilename;

second/samp => float SR; // Sample Rate
2 / pi => float TWO_OVER_PI;

1.0 => float amp;
100 => float f0;
float dB;

for (1 => int n; n < sinRA.cap(); n++ )
{
    f0 * n => sinRA[n].freq;
    amp * TWO_OVER_PI * (1.0 / n ) => sinRA[n].gain;
```

```
    sinRA[n] => dac;
    // print out the amplitude to decibel conversion
    Math.rmstodb( sinRA[n].gain()  ) - 100 => dB;
    <<< n, dB >>>;

}
500::ms => now;
null @=> w; // close WvOut file
```

Open sawtooth_fourider.wav in Audacity and look at the waveform.



The squiggles you see cause the high frequency components above 1600 Hz in the spectrum.



Compare the dB readings for each peak in Audacity with the printout in the Console Monitor window. They should agree.

```
[chuck](VM): sporking incoming shred: 1 (sawtooth_fourierSeries.ck)...

1 -3.922398
2 -9.942998
3 -13.464823
4 -15.963598
5 -17.901798
6 -19.485423
7 -20.824358
8 -21.984198
9 -23.007248
10 -23.922397
11 -24.750251
12 -25.506023
13 -26.201265
14 -26.844958
15 -27.444222
16 -28.004798
```

Ascending Sawtooth

If you alternate plus and minus signs in the sawtooth formula you reverse the direction of the ramp.

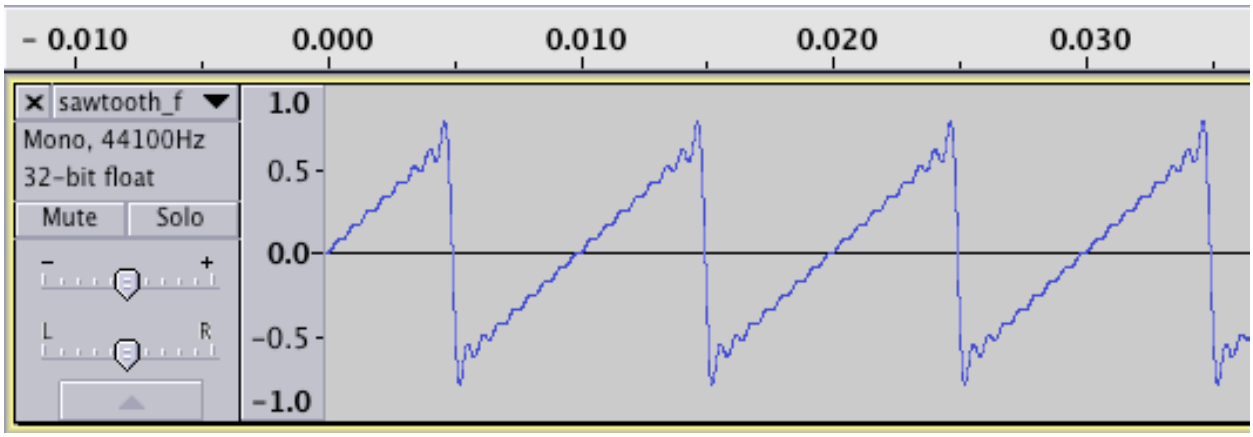
$$Sawtooth = \frac{2}{\pi}\sum_{n=1}^{\infty}\frac{-1^{(n+1)}}{n}\sin(n\omega)$$

$$= \frac{2}{\pi}\left(\sin(\omega) - \frac{1}{2}\sin(2\omega) + \frac{1}{3}\sin(3\omega) - \frac{1}{4}\sin(4\omega) + \frac{1}{5}\sin(5\omega)...\right)$$

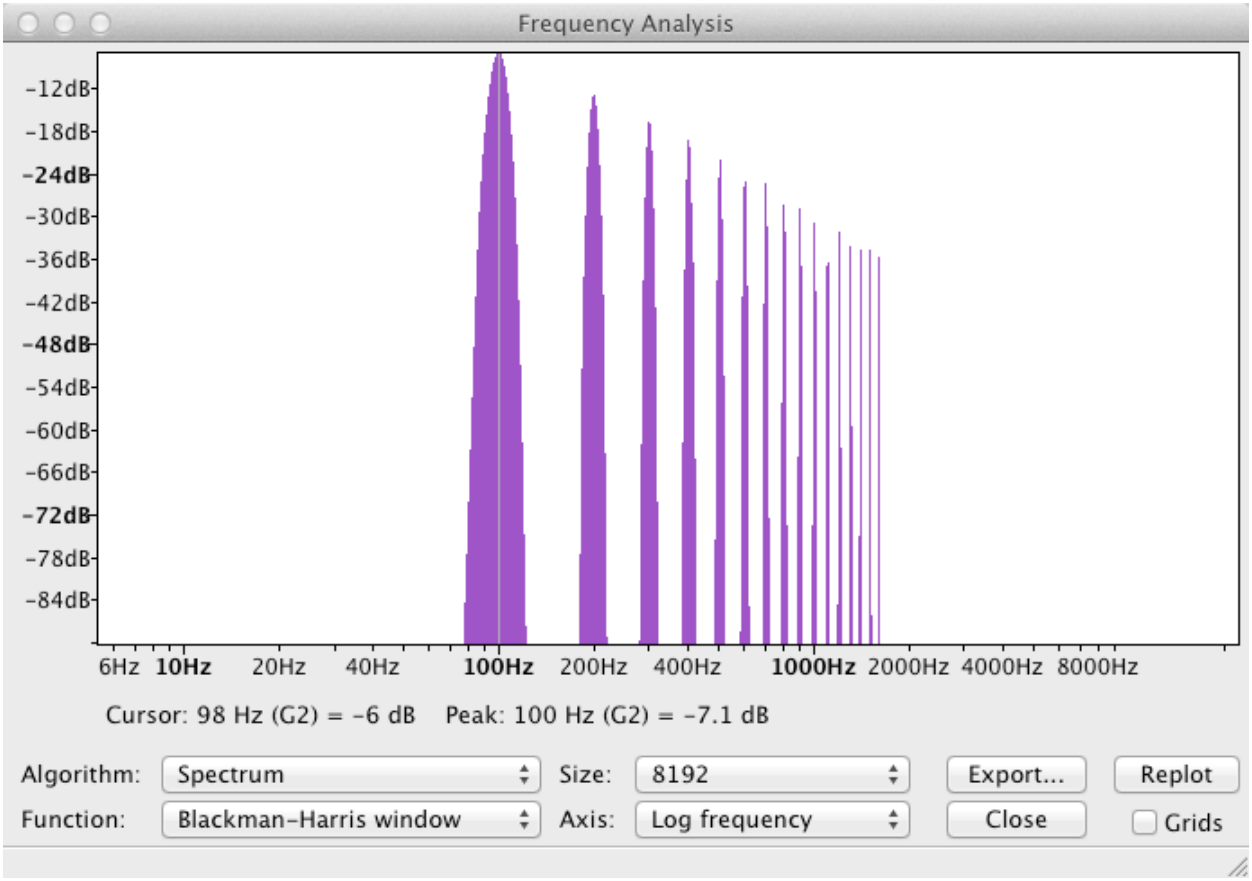
$$\omega = 2\pi f_0$$

```
// sawtooth_ascending_fourier.ck
// John Ellinger Music 208 Winter2014

16 => int numHarmonics;
// create an arry with one extra SinOsc
// arrays begin at 0, harmonics begin at 1
// we'll ignore sinRA[0]
SinOsc sinRA[ numHarmonics + 1 ];
dac => WvOut w => blackhole;
"sawtooth_fourier_ascending.wav" => w.wavFilename;

second/samp => float SR; // Sample Rate
2 / pi => float TWO_OVER_PI;

0.7 => float amp;
100 => float f0;
float maxFreq;
float sign;
float dB;

for (1 => int n; n < sinRA.cap(); n++ )
{
    f0 * n => maxFreq;
    maxFreq => sinRA[n].freq;
    Math.pow( -1.0, n+1 ) => sign;
    amp * sign * TWO_OVER_PI * (1.0 / n ) => sinRA[n].gain;
    sinRA[n] => dac;
    // check for clipping
    if ( maxFreq > SR/2  )
    {
        <<< "**** CLIPPING:", maxFreq $ int, "Hz at index", n
>>>;
        <<< "aliased frequency is",  SR - maxFreq >>>;
    }
}
<<< "Max Frequency is:", maxFreq $ int, "Hz" >>>;

500::ms => now;
null @=> w; // close WvOut file
```

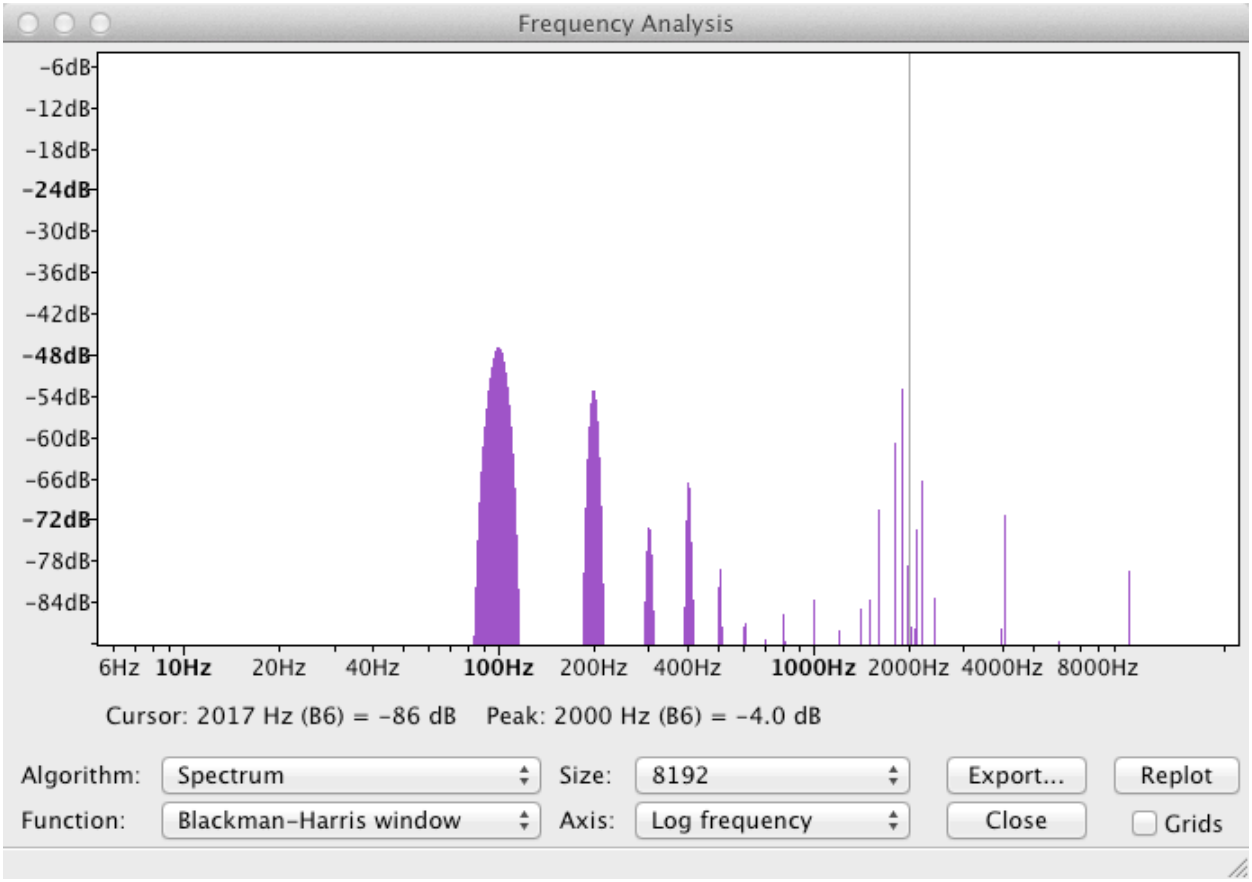
The Chuck sawtooth wave ramps in the other direction.



The spectrum should be identical.



Try it again with fo = 2000 Hz. This time alias frequencies appear.




```
[chuck](VM): sporking incoming shred: 1
(sawtooth_ascending_fourier.ck)...
Harmonic 12 ALIAS 24000 Hz aliased frequency 20100.000000
Harmonic 13 ALIAS 26000 Hz aliased frequency 18100.000000
Harmonic 14 ALIAS 28000 Hz aliased frequency 16100.000000
Harmonic 15 ALIAS 30000 Hz aliased frequency 14100.000000
Harmonic 16 ALIAS 32000 Hz aliased frequency 12100.000000
Max Frequency is: 32000 Hz
```

Square Wave Synthesis

The formula for a sawtooth wave consructed from a sum of sine waves at integer multiples of a fundamental frequency is shown below.

$$Square = \frac{4}{\pi}\left(\sum_{n=1}^{\infty}\frac{1}{n}\sin(n\omega)\right);\quad \text{n is odd}$$

$$= \frac{4}{\pi}\left(\sin(\omega)+\frac{1}{3}\sin(3\omega)+\frac{1}{5}\sin(5\omega)+\frac{1}{7}\sin(7\omega)+\frac{1}{9}\sin(9\omega)...\right)$$

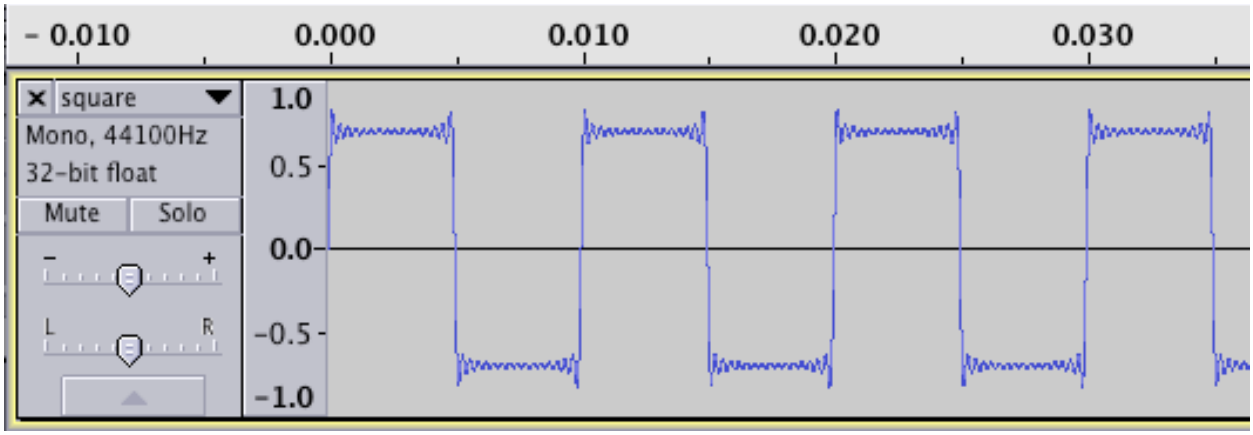
$$\omega = 2\pi f_0$$

Create this ChucK code

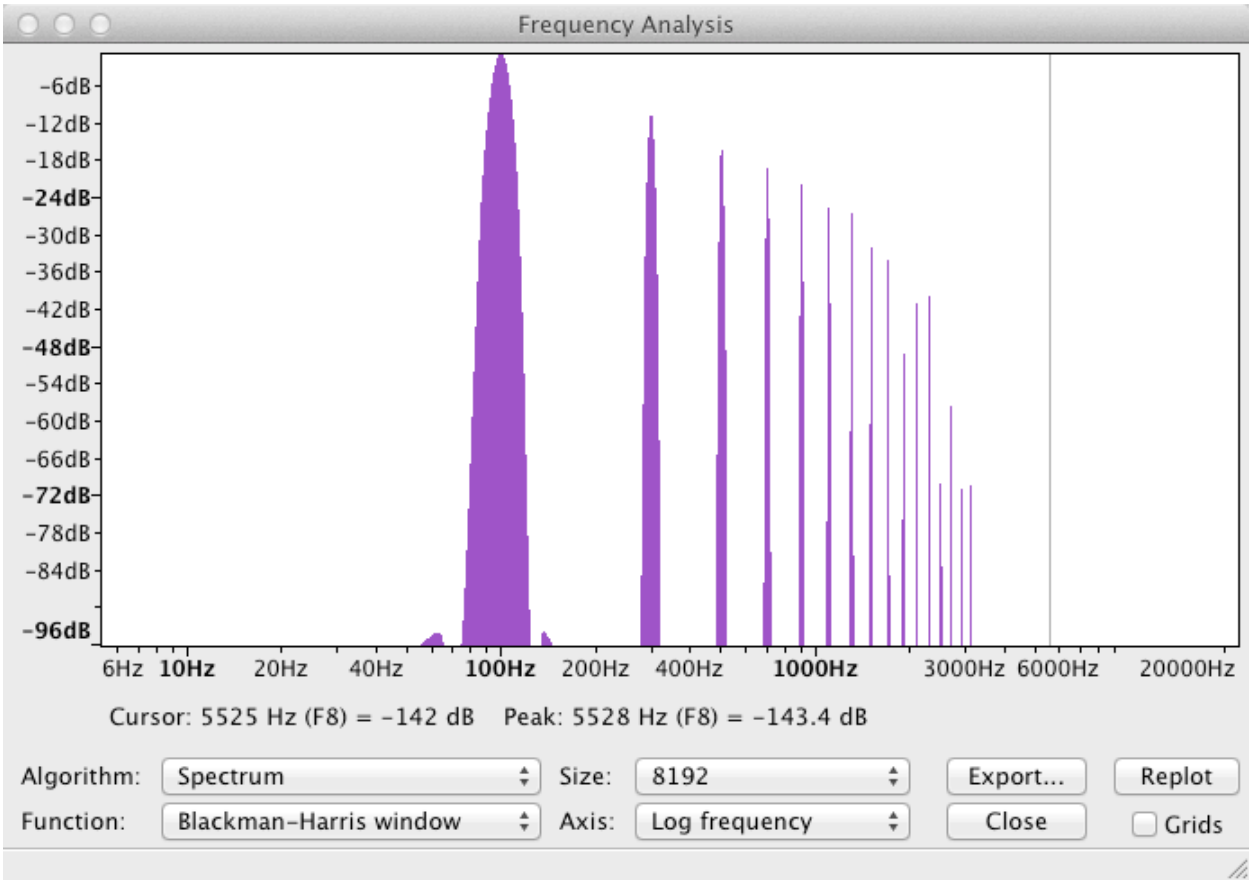
```
// squareWave
// John Ellinger, Music 208, Spring2013

16 => int numHarmonics;
// create an arry with one extra SinOsc
// arrays begin at 0, harmonics begin at 1
// we'll ignore sinRA[0]
// we're only using odd harmonics so double the size of the
array
SinOsc sinRA[ numHarmonics*2 + 1  ]; // we won't use sinRA[0];
dac => WvOut w => blackhole;
"square.wav" => w.wavFilename;

100 => float f0;
0.7 => float amp;
4.0 / pi => float FOUR_OVER_PI;
float dB;
float freq;
```

```
for (1 => int ix; ix < sinRA.cap(); ix++ )
{
    if ( ix % 2 == 0 ) // skip the rest if ix is even number
        continue;
    // test for exceeding Nyquist limit
    f0 * (ix) => freq;
    if ( freq > 22050 )
        break; // do not continue

    f0 * (ix) => sinRA[ix].freq;
// reduce the gain to prevent clipping
amp * FOUR_OVER_PI * (1.0 / ix ) => sinRA[ix].gain;
Math.rmstodb( sinRA[ix].gain() ) - 100 => dB;
<<< ix, "\tamplitude\t", sinRA[ix].gain(), "\tdB\t", dB >>>;
sinRA[ix] => dac;
}
500::ms => now;
null @=> w; // close WvOut file
```

Open sawtooth.wav in Audacity and look at the waveform.



Select the entire waveform and choose Plot Spectrum from the Analyze menu.



Position the cursor over each peak and examine the frequency componenets. They should all be odd numbered multiples of 100 Hz. Compare the dB readings for each peak in Audacity with the printout in the Console Monitor window. They should agree.

```
[chuck](VM): sporking incoming shred: 1 (squareWave.ck)...
1 freq    100.000000   amplitude 0.891268     dB    -0.999837
3 freq    300.000000   amplitude 0.297089     dB    -10.542262
5 freq    500.000000   amplitude 0.178254     dB    -14.979237
7 freq    700.000000   amplitude 0.127324     dB    -17.901798
9 freq    900.000000   amplitude 0.099030     dB    -20.084687
11 freq   1100.000000  amplitude 0.081024     dB    -21.827691
13 freq   1300.000000  amplitude 0.068559     dB    -23.278704
15 freq   1500.000000  amplitude 0.059418     dB    -24.521662
17 freq   1700.000000  amplitude 0.052428     dB    -25.608815
19 freq   1900.000000  amplitude 0.046909     dB    -26.574909
21 freq   2100.000000  amplitude 0.042441     dB    -27.444222
23 freq   2300.000000  amplitude 0.038751     dB    -28.234394
25 freq   2500.000000  amplitude 0.035651     dB    -28.958637
27 freq   2700.000000  amplitude 0.033010     dB    -29.627112
29 freq   2900.000000  amplitude 0.030733     dB    -30.247797
31 freq   3100.000000  amplitude 0.028751     dB    -30.827071
```

Turn squareWave Into A Function

Create genSquareWave(float amp, float freq, dur dura) that will create a square wave at any amplitude, frequency, and duration. It does not have to be bandlimited.

Triangle Wave Synthesis

The formula for a sawtooth wave consructed from a sum of sine waves at integer multiples of a fundamental frequency is shown below. Note the alternating plus - minus signs

$$Triangle = \frac{8}{\pi^2}\left(\sum_{n=1}^{\infty}\frac{-1^{(n+1)}}{n^2}\sin(n\omega)\right);\ \text{n is odd}$$

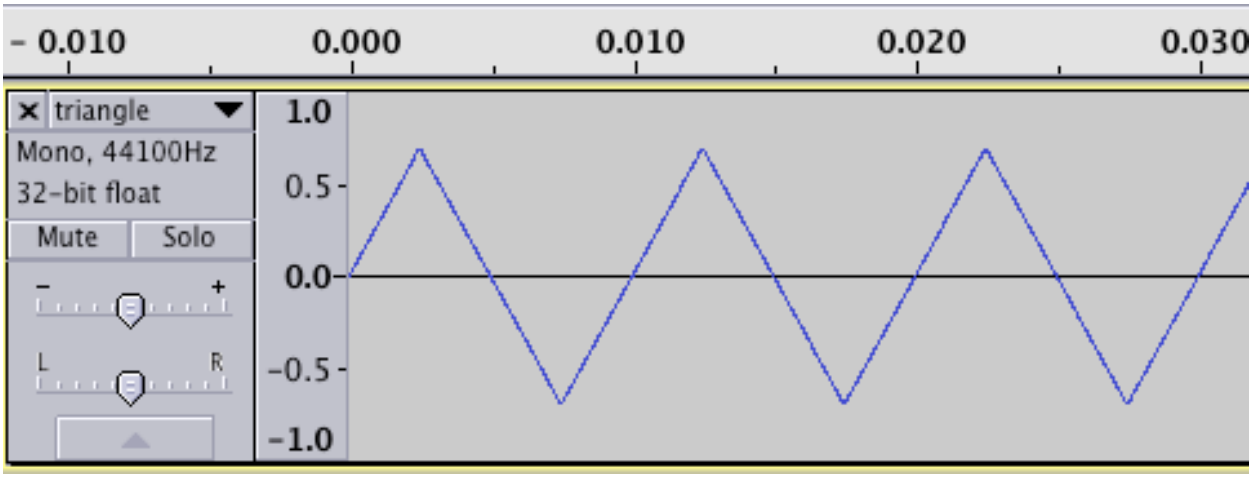
$$= \frac{8}{\pi^2}\left(\sin(\omega)-\frac{1}{9}\sin(3\omega)+\frac{1}{25}\sin(5\omega)-\frac{1}{49}\sin(7\omega)+\frac{1}{81}\sin(9\omega)...\right)$$

$$\omega = 2\pi f_0$$

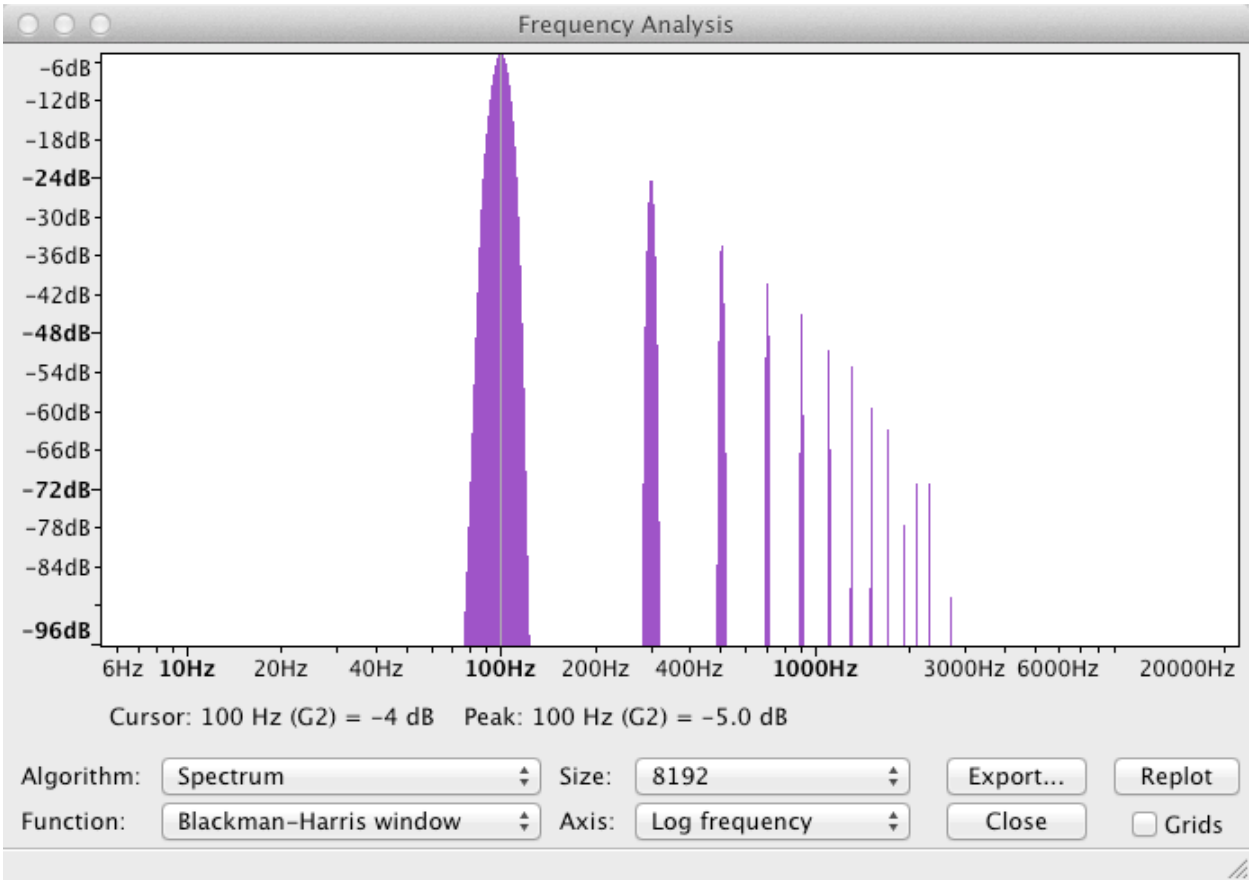
```
// triangleWave
// John Ellinger Music 208 Winter 2014

16 => int numHarmonics;
// create an arry with one extra SinOsc
// arrays begin at 0, harmonics begin at 1
// we'll ignore sinRA[0]
// we're only using odd harmonics so double the size of the
array
SinOsc sinRA[ numHarmonics*2 + 1  ]; // we won't use sinRA[0];
dac => WvOut w => blackhole;
"triangle.wav" => w.wavFilename;

100 => float f0;
0.7 => float amp;
8 / (pi*pi) => float EIGHT_OVER_PI_SQUARED;
float dB;
float freq;
-1 => int sign;

for (1 => int ix; ix < sinRA.cap(); ix++ )
{
    if ( ix % 2 == 0 ) // skip the rest if ix is even number
        continue;
```

```
    // test for exceeding Nyquist limit
    f0 * (ix) => freq;
    if ( freq > 22050 )
        break; // do not continue

    f0 * (ix) => sinRA[ix].freq;
    // reduce the gain to prevent clipping
    sign * -1 => sign; // flip sign every other time
    sign * amp * EIGHT_OVER_PI_SQUARED * ( 1.0/(ix*ix) ) =>
sinRA[ix].gain;
    Math.rmstodb( Math.fabs(sinRA[ix].gain() ) ) - 100 => dB;
    <<< ix, "freq\t", freq, "\tamp\t", sinRA[ix].gain(), "\tdB
\t", dB >>>;
    sinRA[ix] => dac;
}
500::ms => now;
null @=> w; // close WvOut file
```

Open triangle.wav in Audacity and look at the waveform.



Select the entire waveform and choose Plot Spectrum from the Analyze menu.



Position the cursor over each peak and examine the frequency componenets. They should all be multiples of 100 Hz. Compare the dB readings for each peak in Audacity with the printout in the Console Monitor window. They should agree.

```
[chuck](VM): sporking incoming shred: 1 (triangleWave.ck)...
1 freq    100.000000   amp   0.567399     dB    -4.922235
3 freq    300.000000   amp   -0.063044    dB    -24.007084
5 freq    500.000000   amp   0.022696     dB    -32.881034
7 freq    700.000000   amp   -0.011580    dB    -38.726156
9 freq    900.000000   amp   0.007005     dB    -43.091935
11 freq   1100.000000  amp   -0.004689    dB    -46.577941
13 freq   1300.000000  amp   0.003357     dB    -49.479969
15 freq   1500.000000  amp   -0.002522    dB    -51.965885
17 freq   1700.000000  amp   0.001963     dB    -54.140191
19 freq   1900.000000  amp   -0.001572    dB    -56.072379
21 freq   2100.000000  amp   0.001287     dB    -57.811006
23 freq   2300.000000  amp   -0.001073    dB    -59.391348
25 freq   2500.000000  amp   0.000908     dB    -60.839835
27 freq   2700.000000  amp   -0.000778    dB    -62.176785
29 freq   2900.000000  amp   0.000675     dB    -63.418154
31 freq   3100.000000  amp   -0.000590    dB    -64.576702
```

Turn triangleWave Into A Function.

Create genTriangleWave(float amp, float freq, dur dura) that will create a square wave at any amplitude, frequency, and duration. It does not have to be bandlimited.

Pulse Wave Synthesis

The formula for a sawtooth wave consructed from a sum of sine waves at integer multiples of a fundamental frequency is shown below.

$$Pulse = \sum_{n=1}^{\infty} \frac{1}{n} \cos(n\omega)$$

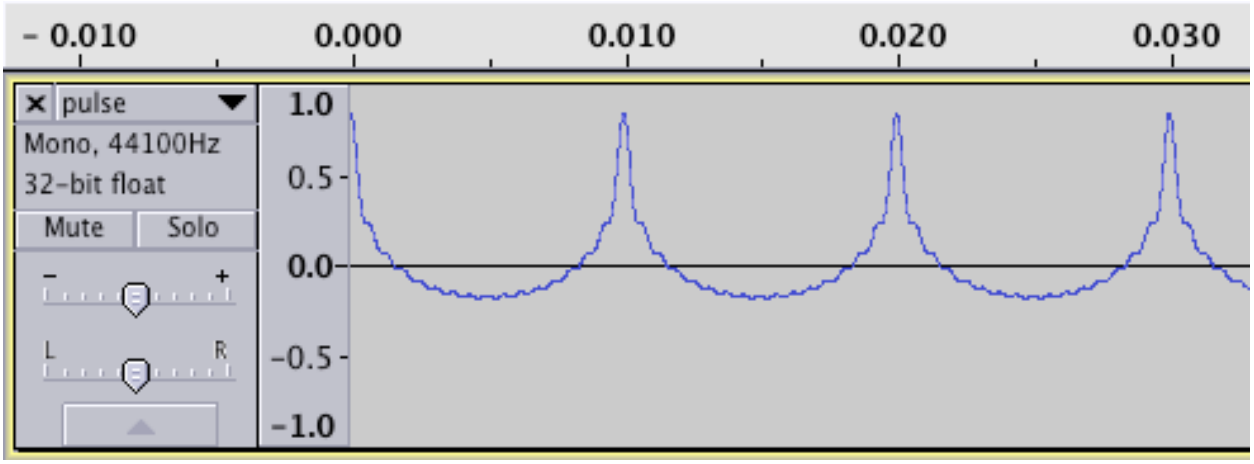
$$\omega = 2\pi f_0$$

$$= \sin(\omega) + \frac{1}{2}\cos(2\omega) + \frac{1}{3}\cos(3\omega) + \frac{1}{4}\cos(4\omega) + \frac{1}{5}\cos(5\omega)...$$

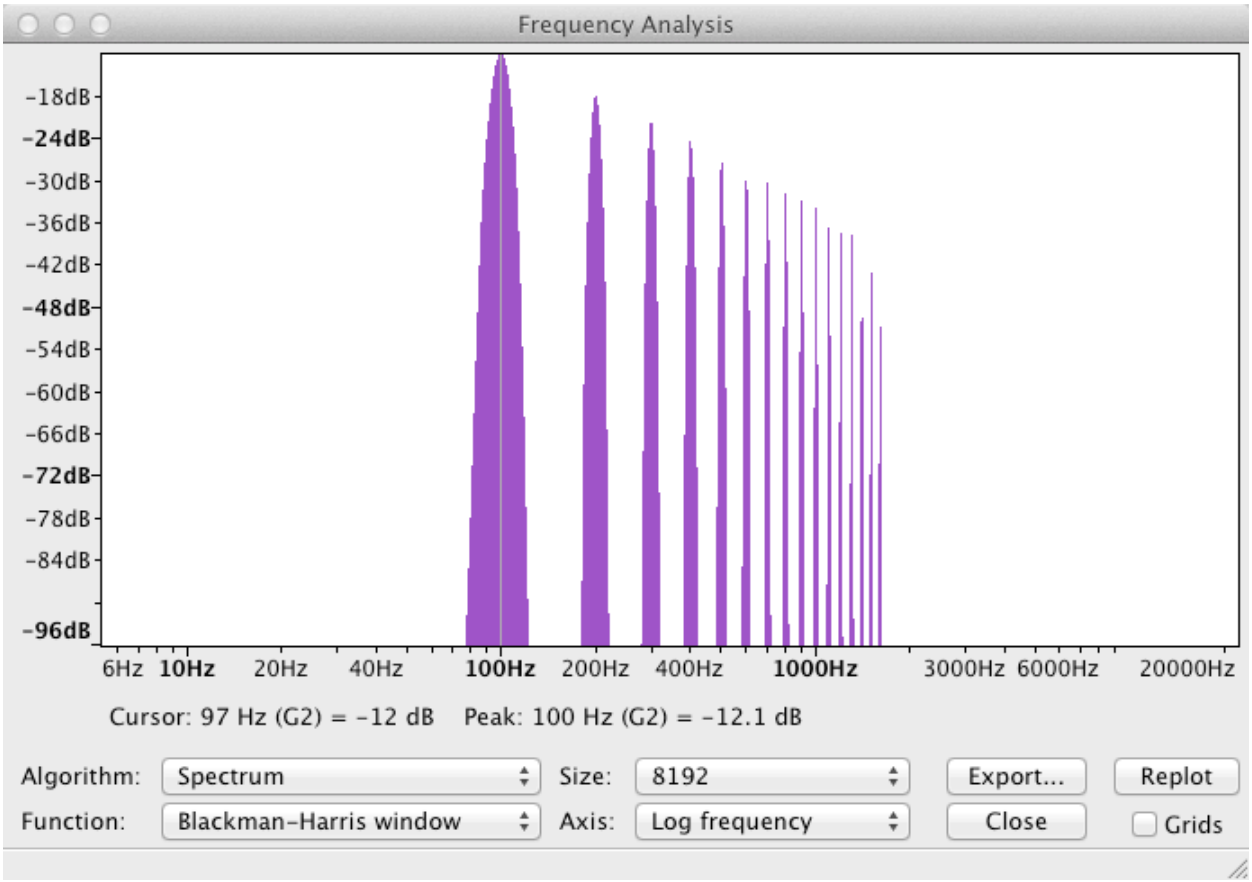
```
// pulse1.ck
// John Ellinger Music 208 Winter2014

16 => int numHarmonics;
// create an arry with one extra SinOsc
// arrays begin at 0, harmonics begin at 1
// we'll ignore sinRA[0]
SinOsc sinRA[ numHarmonics + 1 ];
dac => WvOut w => blackhole;
"pulse1.wav" => w.wavFilename;

second/samp => float SR; // Sample Rate
2 / pi => float TWO_OVER_PI;

0.7 => float amp;
100 => float f0;
float dB;

for (1 => int ix; ix < sinRA.cap(); ix++ )
{
    // shift phase by ?/2
    .25 => sinRA[ix].phase;
    f0 * ix => sinRA[ix].freq;
    amp * TWO_OVER_PI * (1.0 / ix ) => sinRA[ix].gain;
    sinRA[ix] => dac;
```

```
    // print out the amplitude to decibel conversion
    Math.rmstodb( sinRA[ix].gain()  ) - 100 => dB;
    <<< ix, "freq\t", sinRA[ix].freq(), "\tamp\t",
sinRA[ix].gain(), "\tdB\t", dB >>>;

}
500::ms => now;
null @=> w; // close WvOut file
```

Open pulse.wav in Audacity and look at the waveform.



Select the entire waveform and choose Plot Spectrum from the Analyze menu.



Position the cursor over each peak and examine the frequency componenets. They should all be multiples of 100 Hz. Compare the dB readings for each peak in Audacity with the printout in the Console Monitor window. They should agree.

```
[chuck](VM): sporking incoming shred: 1 (pulse1.ck)...
1 freq     100.000000   amp   0.445634     dB    -7.020437
2 freq     200.000000   amp   0.222817     dB    -13.041037
3 freq     300.000000   amp   0.148545     dB    -16.562862
4 freq     400.000000   amp   0.111408     dB    -19.061637
5 freq     500.000000   amp   0.089127     dB    -20.999837
6 freq     600.000000   amp   0.074272     dB    -22.583462
7 freq     700.000000   amp   0.063662     dB    -23.922397
8 freq     800.000000   amp   0.055704     dB    -25.082237
9 freq     900.000000   amp   0.049515     dB    -26.105287
10 freq    1000.000000  amp   0.044563     dB    -27.020437
11 freq    1100.000000  amp   0.040512     dB    -27.848291
12 freq    1200.000000  amp   0.037136     dB    -28.604062
13 freq    1300.000000  amp   0.034280     dB    -29.299304
14 freq    1400.000000  amp   0.031831     dB    -29.942997
15 freq    1500.000000  amp   0.029709     dB    -30.542262
16 freq    1600.000000  amp   0.027852     dB    -31.102837
```

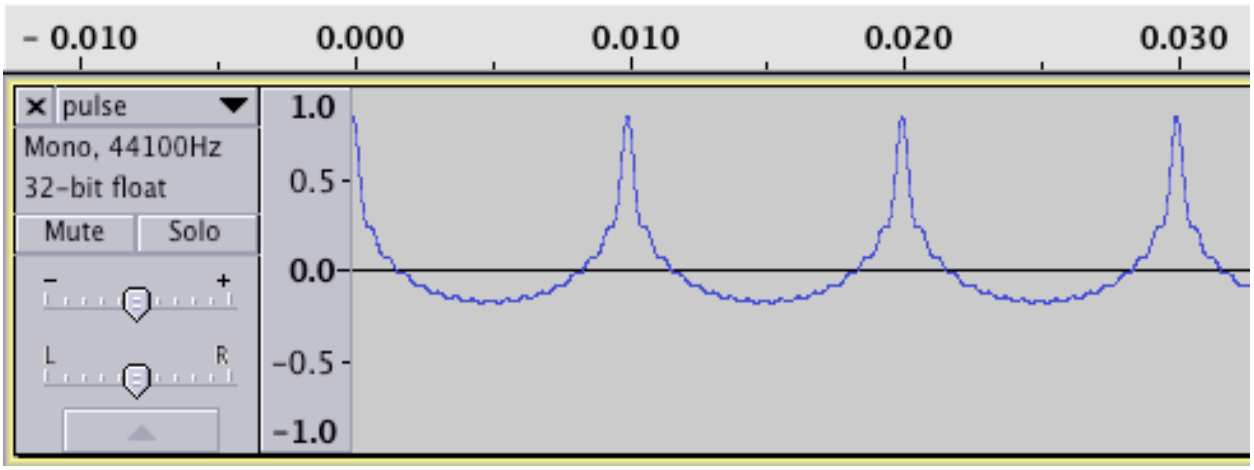
Pulse 2

A pulse wave can also be created by combining a sawtooth wave with a phase shifted version of itself.

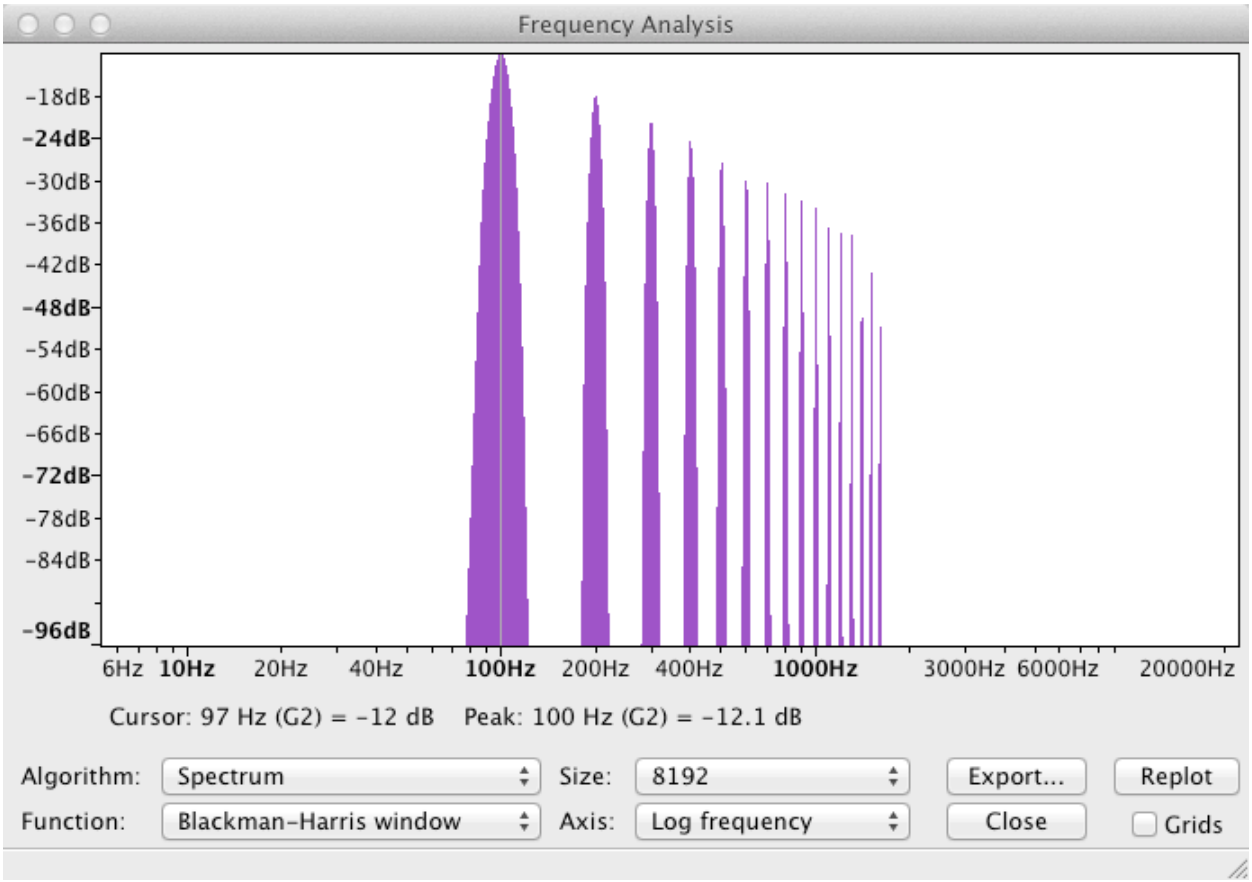
```
// pulse2
// John Ellinger, Music 208, Spring2013

16 => int numHarmonics;
// create an arry with one extra SinOsc
// arrays begin at 0, harmonics begin at 1
// we'll ignore sinRA[0]
SinOsc sinRA[ numHarmonics + 1 ];
SinOsc sinRA2[ numHarmonics + 1 ];
dac => WvOut w => blackhole;
"pulse2.wav" => w.wavFilename;

0.2 => float amp;
100 => float f0;
float dB;
float freq;
for (1 => int ix; ix < sinRA.cap(); ix++ )
{
    // test for exceeding Nyquist limit
    f0 * (ix) => freq;
    if ( freq > 22050 )
        break; // do not continue

    // cosine phase shifted version
    .25 => sinRA2[ix].phase;

    f0 * (ix) => sinRA[ix].freq => sinRA2[ix].freq;
    // reduce the gain to prevent clipping

    amp*(1.0/ix) => sinRA[ix].gain => sinRA2[ix].gain;
    Math.rmstodb( sinRA[ix].gain() ) - 100 => dB;
    <<< ix, "freq\t", sinRA[ix].freq(), "\tamp\t",
sinRA[ix].gain(), "\tdB\t", dB >>>;
    sinRA[ix] => dac;
```

```
    sinRA2[ix] => dac;
}

500::ms => now;
null @=> w;
```

Open pulse.wav in Audacity and look at the waveform.



Select the entire waveform and choose Plot Spectrum from the Analyze menu.



Position the cursor over each peak and examine the frequency componenets. They should all be multiples of 100 Hz. Compare the dB readings for each peak in Audacity with the printout in the Console Monitor window. They should agree.

```
[chuck](VM): sporking incoming shred: 1 (pulse2.ck)...
1 freq     100.000000   amp   0.200000     dB    -13.979400
2 freq     200.000000   amp   0.100000     dB    -20.000000
3 freq     300.000000   amp   0.066667     dB    -23.521825
4 freq     400.000000   amp   0.050000     dB    -26.020600
5 freq     500.000000   amp   0.040000     dB    -27.958800
6 freq     600.000000   amp   0.033333     dB    -29.542425
7 freq     700.000000   amp   0.028571     dB    -30.881361
8 freq     800.000000   amp   0.025000     dB    -32.041200
9 freq     900.000000   amp   0.022222     dB    -33.064250
10 freq    1000.000000  amp   0.020000     dB    -33.979400
11 freq    1100.000000  amp   0.018182     dB    -34.807254
12 freq    1200.000000  amp   0.016667     dB    -35.563025
13 freq    1300.000000  amp   0.015385     dB    -36.258267
14 freq    1400.000000  amp   0.014286     dB    -36.901961
15 freq    1500.000000  amp   0.013333     dB    -37.501225
16 freq    1600.000000  amp   0.012500     dB    -38.061800
```