MUSC 208 Winter 2014
John Ellinger Carleton College

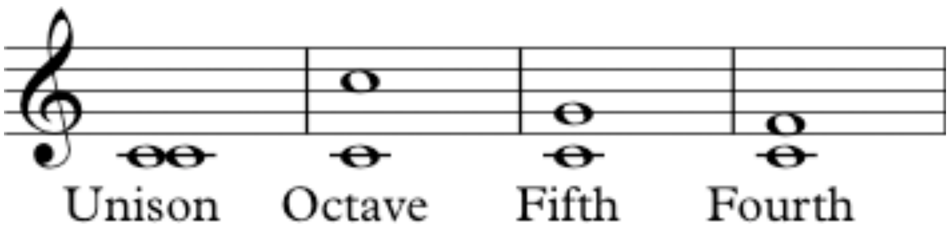
Lab 10 The Harmonic Series, Scales, Tuning, and Cents

Musical Intervals

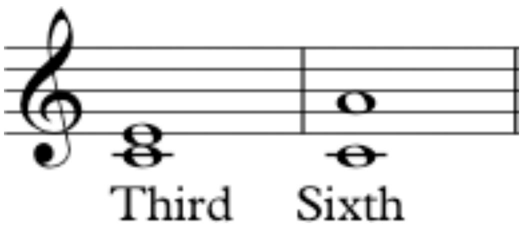
An interval in music is defined as the distance between two notes. In western European music intervals are named both by the number of inclusive half steps between the two notes and also by the inclusive number of lines and spaces separating the notes on the music staff.

Intervals are divided into classes based on consonance and dissonance. Consonant intervals are further divided into perfect and imperfect consonances. Using the notes of the C Major scale the basic intervals are shown below.

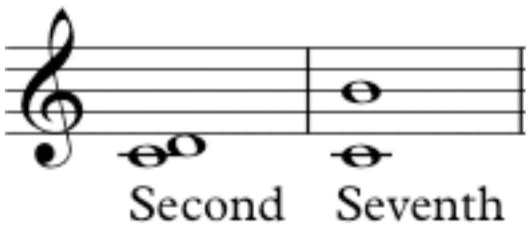
Perfect Consonant Intervals



Imperfect Consonant Intervals



Dissonant Intervals



One of the reasons consonant intervals sound pleasing to the ear is because the frequencies of the two notes are related by small integer ratios. The most consonant intervals appear early in the harmonic series.

Harmonic Series

When you press a key on a piano, blow into a wind or brass instrument, pluck a guitar string, or sing you generate a note at a certain frequency. That note is actually composed of several frequencies related by the harmonic series.

Harmonic Series Definition

1. *Mathematics* A series whose terms are in harmonic progression, especially the series $1 + {}^{1}/_{2} + {}^{1}/_{3} + {}^{1}/_{4} + ...$
2. *Music* A series of tones consisting of a fundamental tone and the consecutive harmonics produced by it.
http://www.thefreedictionary.com/harmonic+series

The frequency of each harmonic is an integer multiple of the fundamental frequency. The first harmonic is also called the fundamental frequency.

Name	Period	Frequency	100 Hz Fundamental
First Harmonic	1	f	100
2nd Harmonic	1/2	2f	200
3rd Harmonic	1/3	3f	300
4th Harmonic	1/4	4f	400
5th Harmonic	1/5	5f	500
6th Harmonic	1/6	6f	600
7th Harmonic	1/7	7f	700
8th Harmonic	1/8	8f	800
9th Harmonic	1/9	9f	900
10th Harmonic	1/10	10f	1000
11th Harmonic	1/11	11f	1100
12th Harmonic	1/12	12f	1200
13th Harmonic	1/13	13f	1300
14th Harmonic	1/14	14f	1400
15th Harmonic	1/15	15f	1500
16th Harmonic	1/16	16f	1600

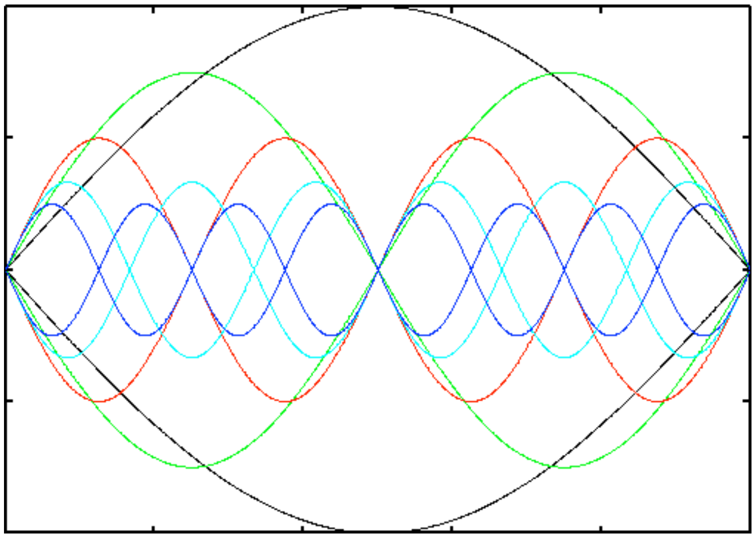
Harmonics, Overtones, And Partials

These terms are often used misused when referring to notes in the harmonic series. Harmonics and overtones refer to integer multiples of a fundamental frequency, the notes of the harmonic series. The fundamental frequency is the first harmonic. The first overtone is the second harmonic. A partial is a non integer multiple of the the fundamental frequency and does not occur in the harmonic series.

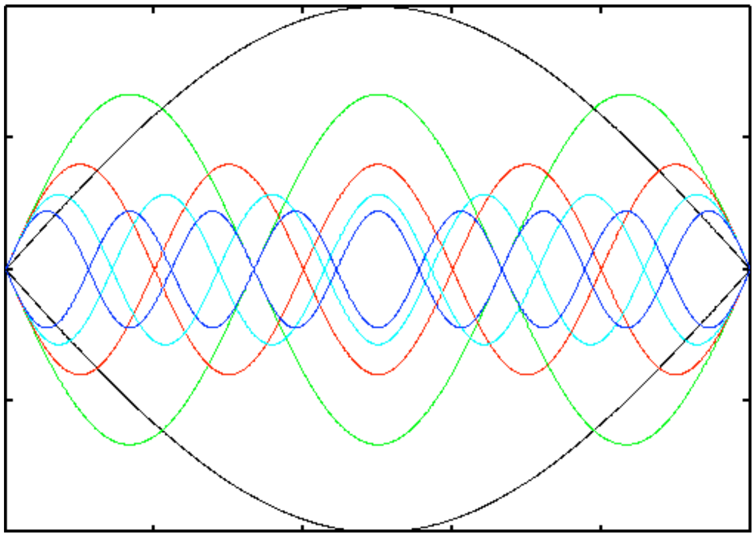
plotHarmonicSeries.m

Execute the Octave script plotHarmonicSeries.m in the m208Lab10 folder. The wavelengths (periods) of all notes found in the harmonic series fit an exact integer number of times into the period of the fundamental.

Even Harmonics 1 2 4 6 8

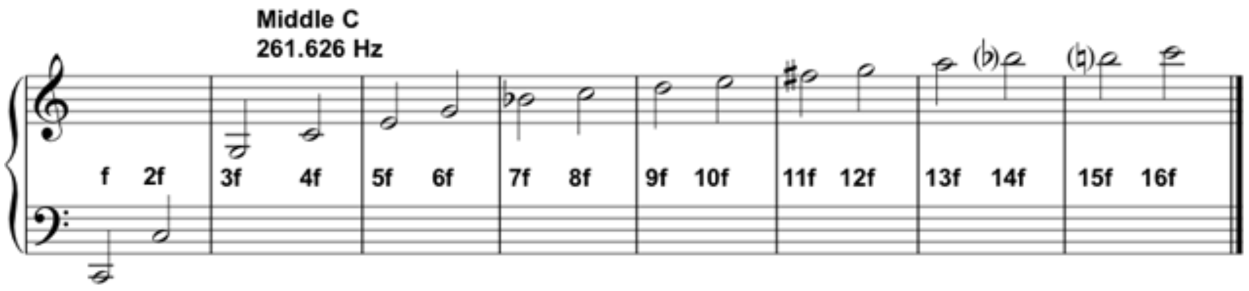


Odd Harmonics 1 3 5 7 9



Harmonic Series In Musical Notation

The first 16 harmonics of C2 (MIDI 36) are shown below. Harmonic series notes, 2f, 4f, 8f, and 16f (the octaves above C2) precisely match the notes on the piano. All other notes in the harmonic series will be out of tune with the piano by varying degrees. The reasons why will be explained throughout the remainder of this lab.



Play the Harmonic Series

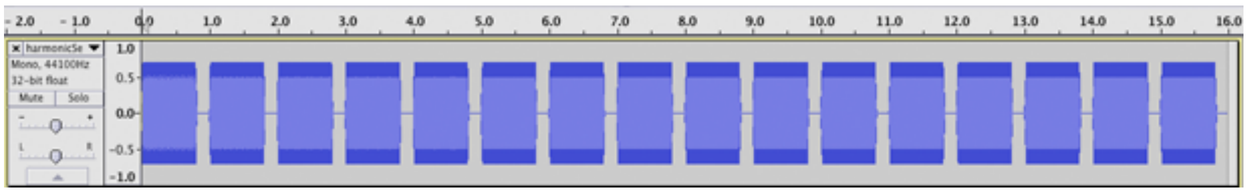
Enter and run this code in ChucK.

```
// playHarmonicSeries
// John Ellinger, Music 208, Spring2013
SinOsc s => Envelope e => dac;
16 => int numH;
function void playHarmonicSeries( float f0, dur d )
{
    for (1 => int ix; ix <= numH; ix++ )
    {
        <<< "Harmonic ", ix >>>;
        f0 * (ix) => s.freq;
        // reduce the gain to prevent clipping
        0.5 => s.gain;
        e.keyOn();
        d * .8 => now;
        e.keyOff();
        d * .2  => now;
    }
}
```

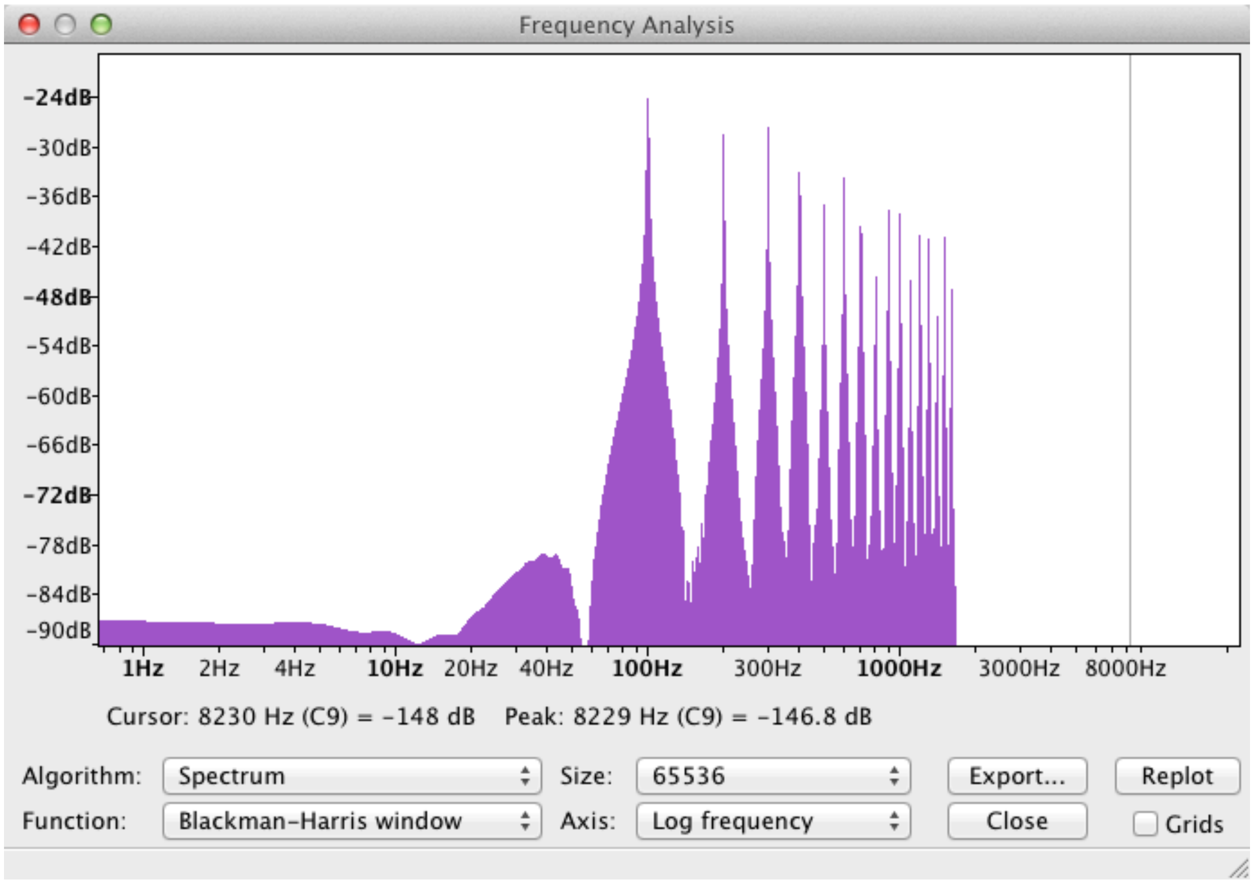
```
dac => WvOut w => blackhole;
"harmonicSeries.wav" => w.wavFilename;
playHarmonicSeries( 100, 1000::ms);
```

Examine the wav file in Audacity

Open the harmonicSeries.wav file in Audacity



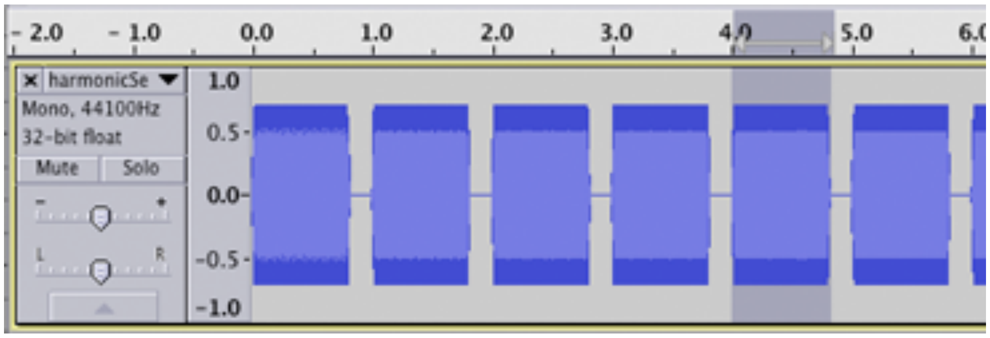
Select the entire waveform and choose Plot Spectrum from the Analyze menu. Set the controls as shown below. Slide the cursor across each peak and read the peak frequency. They should all be multiples of f0 = 100 Hz.



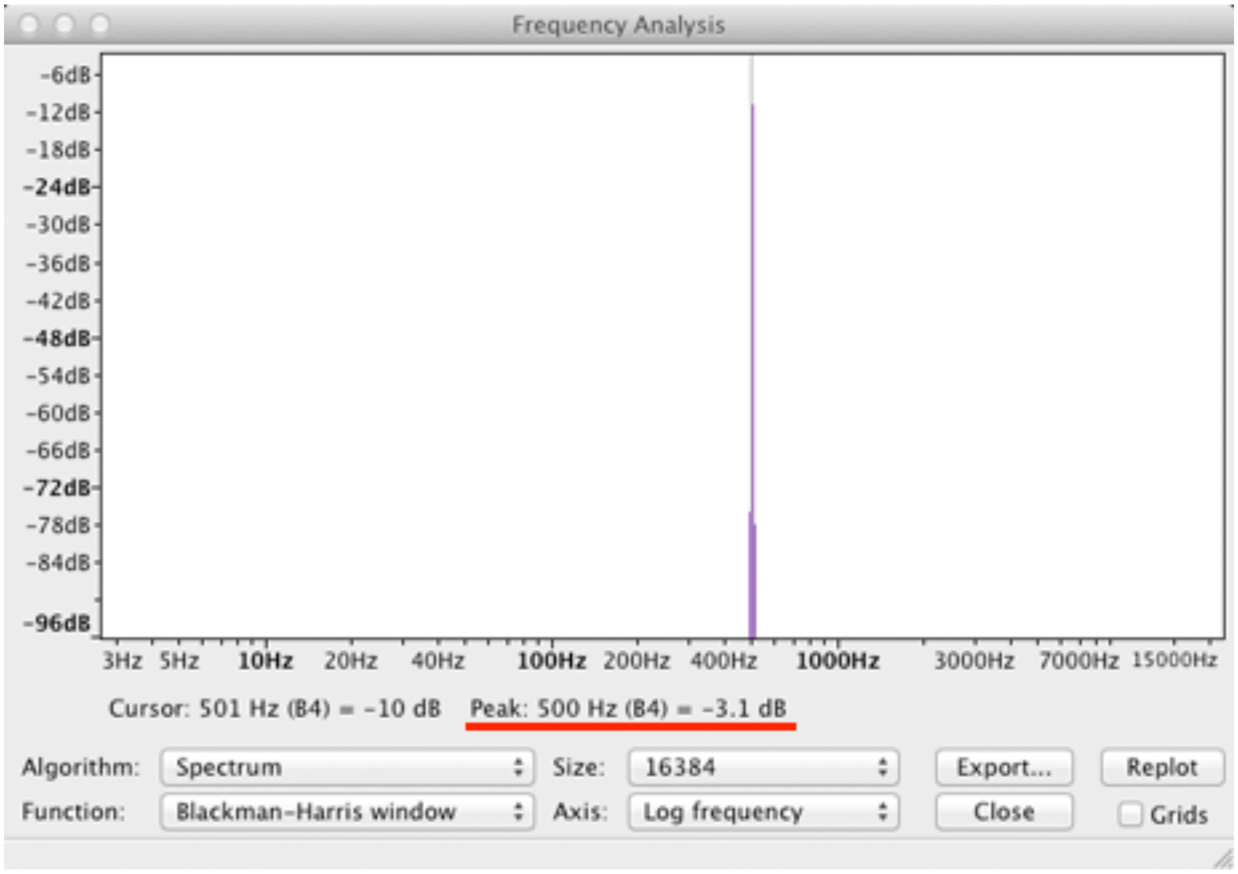
Cursor: 8230 Hz (C9) = -148 dB   Peak: 8229 Hz (C9) = -146.8 dB

Algorithm: Spectrum   Size: 65536
Function: Blackman-Harris window   Axis: Log frequency

Select harmonic 5.



Plot the spectrum again. Use the control settings shown below and you'll see a single frequency at 500 Hz.



Cursor: 501 Hz (B4) = -10 dB Peak: 500 Hz (B4) = -3.1 dB

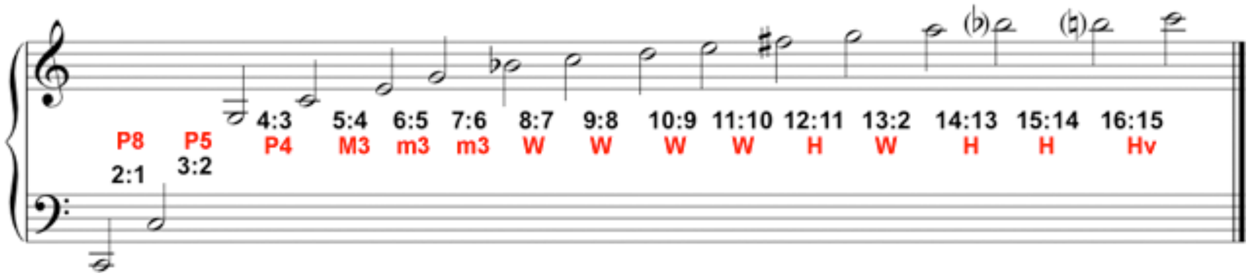
Algorithm: Spectrum
Size: 16384
Function: Blackman-Harris window
Axis: Log frequency

Frequency Ratios And Musical Intervals

"Pythagoras is credited by ancient Greek writers with having discovered the intervals of the octave, fifth, fourth, and double octave. Pythagoras and his followers attached great numerological significance to the fact that these most harmonious intervals were constructed strictly from ratios of the consecutive inters 1, 2, 3, and 4." Gareth Loy, 2006, Musimathics Vol 1, The MIT Press, page 48.

Pythagorean tuning was used for over 1000 years partly because of the mystical, almost religious, simplicity of the ratios 2:1, 3:2, and 4:3. These ratios where the numerator is one greater than the denominator held great significance to ancient Greek, Medieval and Renaissance music theorists. It was during the Renaissance with the rise of polyphonic music and a desire to transpose music into different keys that led to the development of alternate tuning systems.

Frequency Ratios Between Adjacent Harmonics, H / H-1



Here's a chart showing the harmonic series ratios and their closest music theory interval. An interesting feature is that harmonics 2f, 4f, 8f, and 16f are one, two, three, and four octaves above the fundamental frequency f. However, there is more than one ratio for minor thirds, whole steps, and half steps.

Harmonic	H / H-1 Ratio	Closest Music Theory Interval
f	1/1	unison
2f	2/1	P8 octave
3f	3/2	P5 perfect fifth
4f	4/3	P4 perfect fourth
5f	5/4	M3 major third
6f	6/5	m3 minor third
7f	7/6	m3 minor third
8f	8/7	W whole step M2
9f	9/8	W whole step M

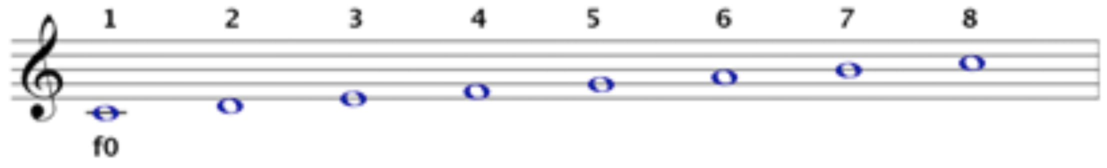
Pythagorean Scale

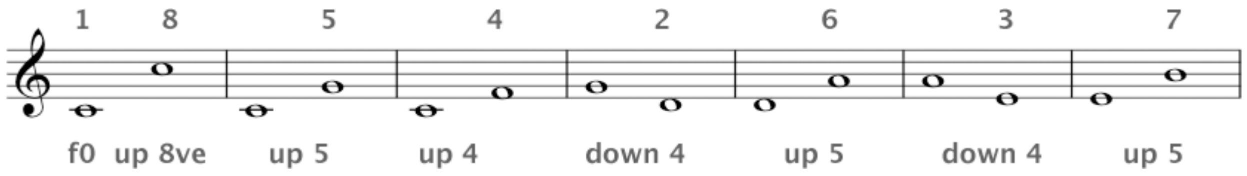
The Pythagorean scale that has been known for thousands of years. The first three ratios of the harmonic series, 2/1, 3/2, and 4/3, are used to build the scale.

Rules

1. Multiply by 2 to go up one octave.
2. Multiply by 3/2 to go up one fifth.
3. Multiply by 4/3 to go up one fourth.
4. Divide by 2 (multiply by 1/2) to go down one octave.
5. Divide by 3/2 (multiply by 2/3) to go down one fifth (or up 4, down 8).
6. Divide by 4/3 (multiply by 3/4) to go down one fourth (or up 5, down 8).

Using these rules we can construct the Pythagorean scale that is similar to the notes of our major scale.





Construct in this order	Directions	Formula
Note1	Fundamental Frequency	Note1 = f0
Note8	Note1 up one octave	Note1 * 2
Note5	Note1 up fifth	Note1 * 3/2
Note4	Note1 up fourth	Note1 * 4/3
Note2	Note5 down fourth	Note5 * 3/4
Note6	Note2 up fifth	Note2 * 3/2
Note3	Note6 down fourth	Note6 * 3/4
Note7	Note3 up fifth	Note3 * 3/2

Using these rules you can construct the Pythagorean scale for one octave from any starting frequency.

Create the Pythagorean Scale Class

Enter this code.

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

// IMPORTANT declare a public class so other files can use it
public class PythagoreanScaleClass
{
    float f0; // Harmonic Series fundamental frequency
    float scale[8]; // eight notes in the scale

    // initialize the fundamental frequency
    function void init( float fundamentalFrequency )
    {
        fundamentalFrequency => f0;
        buildScale();
    }

    function float note1()
    {
        return f0; // this is the only one that's correct
    }

    function float note8()
    {
        // note1 up octave
        return f0 * 2.0;
    }

    function float note5()
    {
        // note1 up fifth
        return f0 * 3.0 / 2.0;
    }

    function float note4()
    {
        // note1 up fourth
    return f0 * 4.0 / 3.0;
}

function float note2()
{
    // note5 down fourth
    return note5() * 3.0 / 4.0;
}

function float note6()
{
    // note2 up fifth
    return note2() * 3.0 / 2.0;
}

function float note3()
{
    // note5 down fourth
    return note6() * 3.0 / 4.0;
}

function float note7() // new
{
    // note3 up fifth
    return note3() * 3.0 / 2.0;
}

// construct each note of the scale
function void buildScale()
{
    note1() => scale[0];
    note2() => scale[1];
    note3() => scale[2];
    note4() => scale[3];
    note5() => scale[4];
    note6() => scale[5];
    note7() => scale[6];
    note8() => scale[7];
}
```

```
    function void printFreqs()
    {
        for ( 0 => int ix; ix < scale.cap(); ix++ )
        {
            <<< ix, "\t", scale[ix], "\t", Std.ftom( scale[ix] ) >>>;
        }
    }

    function void playOneNote(  int note, dur tmOn, dur tmOff )
    {
        SinOsc s => Envelope e => dac;
        // logic OR symbol is ||
        if ( ( note < 0 ) || (note > scale.cap() ) )
            <<< "BAD note index number:\t", note >>>;
        else
        {
            scale[ note ] => s.freq;
            e.keyOn();
            tmOn => now;
            e.keyOff();
            tmOff => now;
        }
        s =< e =< dac; // disconnect
    }

    function void playScale( dur noteDur )
    {
        for ( 0 => int ix; ix < scale.cap(); ix++ )
        {
            playOneNote(  ix, noteDur * 0.8, noteDur * 0.2 );
        }
    }
} // end class
```

Run and test the class. When it runs cleanly (no errors) stop the virtual machine. The class itself doesn't do anything yet. Because the PythagoreanScaleClass was declared public, it can be used by other source files. We need create a new source file to test it. Keep the PythagoreanScaleClass window open.

PythagoreanScaleTest.ck

Open a new ChucK file and enter this code. This new file will be used to test the PythagoreadScaleClass class.

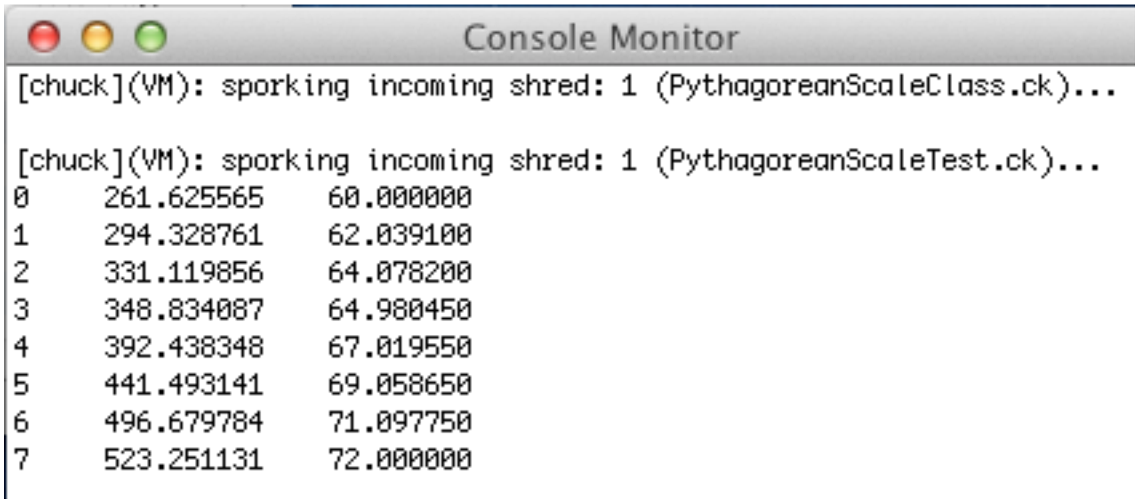
```
// PythagoreanScaleTest.ck
// John Ellinger Music 208 Winter 2014

// create an instance of the class
PythagoreanScaleClass psc;

// call the init (initialize) method to set f0
// this must be called before any other class methods
psc.init( Std.mtof( 60 ) );

// print the scale frequencies in Hz
psc.printFreqs();

// play the scale
psc.playScale();
```

Ready to Test

Stop the Virtual Machine.
Start the Virtual Machine.
Important: Run the PythagoreanScaleClass.ck file first. Anytime you change code in the public class, stop and restart the Virtual Machine.

Run the PythagoreanScaleTest.ck file second. If there are no errors in the PythagoreanTest.ck file you should hear scale played and see the following output.

Output



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

[chuck](VM): sporking incoming shred: 1 (PythagoreanScaleTest.ck)...
0    261.625565    60.000000
1    294.328761    62.039100
2    331.119856    64.078200
3    348.834087    64.980450
4    392.438348    67.019550
5    441.493141    69.058650
6    496.679784    71.097750
7    523.251131    72.000000
```

Notice that Pythagorean ratios do not result in integer valued MIDI note numbers in column three.

Equal Temperament

As appealing as the Pythagorean system of simple tuning ratios seemed, it did not work in practice, especially on keyboard instruments. Problems arose when you tried to play songs to a different keys. Some early keyboards divided the black keys into two parts, for example one half for F sharp and the other half for G flat.

The Equal Temperament tuning system we use today was established during the 18th and early 19th centuries. Equal Temperament divides the octave into twelve equal half steps. On the piano the only intervals that are perfectly in tune are octaves, everything else is equally out of tune.

The frequency ratio between half steps is :

$$2^{\frac{1}{12}}$$

In order to calculate the pitch of any note you need a reference frequency and you need to know how many half steps distant it is from that reference. The standard reference pitch is A 440 Hz (called A440) and is MIDI note number 69.

MIDI Note Number To Frequency

We've used this formula before. MIDI note number 69 (A440) as our reference pitch and "mnn" stands for MIDI Note Number.

$frequency = 440 * 2^{\frac{m-69}{12}}$, where m is the MIDI note number.

It's exactly what the ChucK method Std.mtof(60) does.

Equal Temperament Scale Class

Enter this code.

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

// IMPORTANT declare a public class so other files can use it
public class EqualTempScaleClass
{
    float midiNote1; // Harmonic Series fundamental frequency
    float scale[8]; // eight notes in the scale

    // initialize the fundamental frequency
    function void init( float midiNum )
    {
        midiNum => midiNote1;
        buildScale();
    }

    function float note1()
    {
        return midiNote1; // this is the only one that's correct
    }

    function float note2()
    {
        return midiNote1 + 2;
    }

    function float note3()
    {
        return midiNote1 + 4;
    }

    function float note4()
    {
        return midiNote1 + 5;
    }
```

```
function float note5()
{
    return midiNote1 + 7;
}

function float note6()
{
    return midiNote1 + 9;
}

function float note7() // new
{
    return midiNote1 + 11;
}

function float note8()
{
    return midiNote1 + 12;
}

// construct each note of the scale
function void buildScale()
{
    note1() => scale[0];
    note2() => scale[1];
    note3() => scale[2];
    note4() => scale[3];
    note5() => scale[4];
    note6() => scale[5];
    note7() => scale[6];
    note8() => scale[7];
}

function void printFreqs()
{
    for ( 0 => int ix; ix < scale.cap(); ix++ )
    {
        <<< ix, "\t", scale[ix], "\t", Std.mtof( scale[ix] ) >>>;
    }
}
```

```
    function void playOneNote(  int note, dur tmOn, dur tmOff )
    {
        SinOsc s => Envelope e => dac;
        // logic OR symbol is ||
        if ( ( note < 0 ) || (note > scale.cap() ) )
            <<< "BAD note index number:\t", note >>>;
        else
        {
            Std.mtof( scale[ note ] ) => s.freq;
            e.keyOn();
            tmOn => now;
            e.keyOff();
            tmOff => now;
        }
        s =< e =< dac; // disconnect
    }

    function void playScale( dur noteDur )
    {
        for ( 0 => int ix; ix < scale.cap(); ix++ )
        {
            playOneNote(  ix, noteDur * 0.8, noteDur * 0.2 );
        }
    }
} // end class
```

Run and test the class. When it runs cleanly (no errors) stop the virtual machine. The class itself doesn't do anything yet. Because the EqualTempScaleClass was declared public, it can be used by other source files. We need create a new source file to test it. Keep the EqualTempScaleClass window open.

EqualTempScaleTest.ck

Open a new ChucK file and enter this code. This new file will be used to test the PythagoreadScaleClass class.

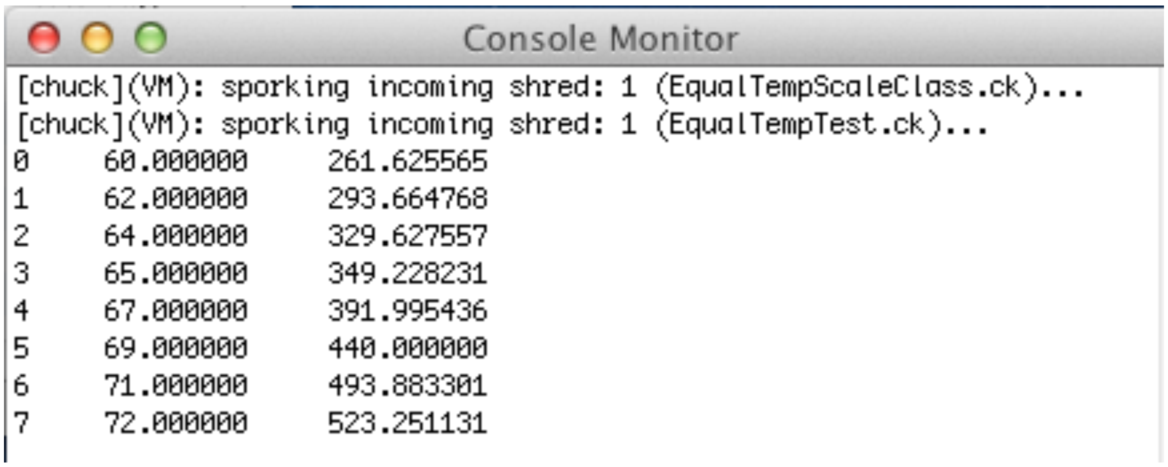
```
// EqualTempScaleClass.ck
// John Ellinger Music 208 Winter 2014

// create an instance of the class
EqualTempScaleClass psc;
// call the init (initialize) method to set f0
// this must be called before any other class methods
psc.init( Std.mtof( 60 ) );
// print the scale frequencies in Hz
psc.printFreqs();
// play the scale
psc.playScale();
```

Ready to Test

Stop the Virtual Machine.
Start the Virtual Machine.
Important: Run the EqualTempScaleClass.ck file first. Anytime you change code in the public class, stop and restart the Virtual Machine.

Run the EqualTempScaleTest.ck file second. If there are no errors in the EqualTempTest.ck file you should hear scale played and see the following output.

Output



```
[chuck](VM): sporking incoming shred: 1 (EqualTempScaleClass.ck)...
[chuck](VM): sporking incoming shred: 1 (EqualTempTest.ck)...
0     60.000000     261.625565
1     62.000000     293.664768
2     64.000000     329.627557
3     65.000000     349.228231
4     67.000000     391.995436
5     69.000000     440.000000
6     71.000000     493.883301
7     72.000000     523.251131
```

Notice that the MIDI note numbers are integers but the frequencies in column three differ from the Pythagorean ratio frequencies.

Pythagorean And Equal Tempered Scales Compared

If you play the two scales simultaneously pitch differences between the individual notes will be heard as beats. Beats per second is a rough approximation to frequency difference in Hz. Create this code.

PyETScalesCompare.ck

```
// PyEtScalesCompare.ck
// John Ellinger, Music 208, Spring2013

// create an instance of the PythagoreanScaleClass
PythagoreanScaleClass psc;
psc.init( Std.mtof( 60)  );

// create an instance of the EqualTempScaleClass
EqualTempScaleClass etsc;
etsc.init( 60 );

// play both scales simultaneously
for ( 0 => int ix; ix < psc.scale.cap(); ix++ )
{
    psc.scale[ix] => float fPY;
    Std.mtof( etsc.scale[ix] ) => float fET;

    spork ~ psc.playOneNote( ix, 5000::ms, 100::ms );
    spork ~ etsc.playOneNote( ix, 5000::ms, 100::ms );
    5100::ms => now;

    // Output Compare the MIDI (EqualTemp) notes to Pythagorean
    <<< ix+1, fET, fPY,  fET - fPY >>>;
}
```

Ready to Test

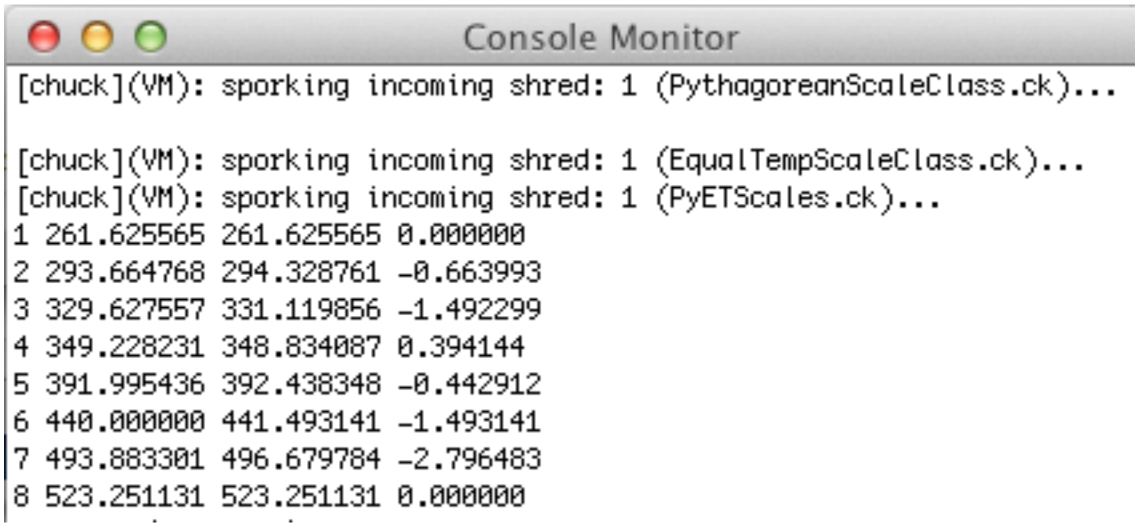
Stop the Virtual Machine.
Start the Virtual Machine.
Important: Run the PythagoreanScaleClass.ck file and the EqualTempScaleClass.ck file. Order does not matter here.

Run the PyEtScalesCompare.ck file next. If there are no errors in the you should hear the notes of each scale played together.

Output

The output compares the difference between Equal Temperament (MIDI notes) and Pythagorean ratios. As you can see the only interval that is in tune between the two scales are notes 1 and 8, the unison (fundamental frequency) and the octave.

Note 4 is the only note that is sharp. Notes 2, 3, 5, 6, 7 are flat.



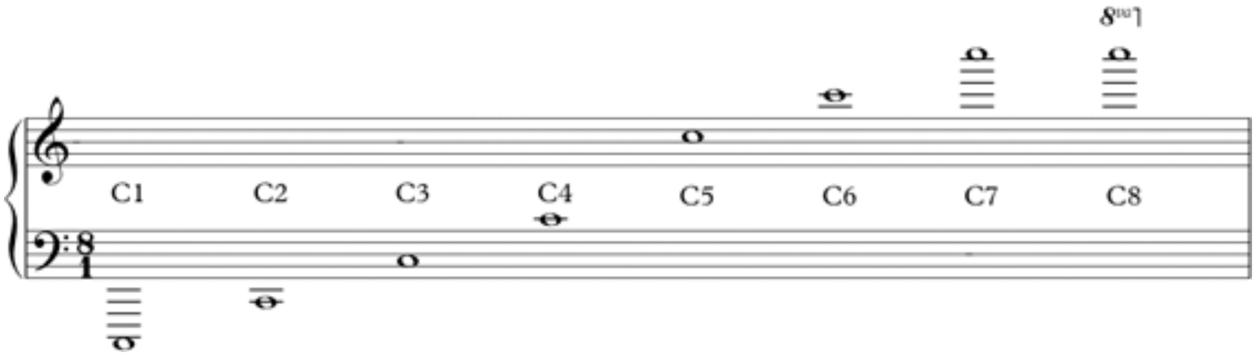
```
[chuck](VM): sporking incoming shred: 1 (PythagoreanScaleClass.ck)...

[chuck](VM): sporking incoming shred: 1 (EqualTempScaleClass.ck)...
[chuck](VM): sporking incoming shred: 1 (PyETScales.ck)...
1 261.625565 261.625565 0.000000
2 293.664768 294.328761 -0.663993
3 329.627557 331.119856 -1.492299
4 349.228231 348.834087 0.394144
5 391.995436 392.438348 -0.442912
6 440.000000 441.493141 -1.493141
7 493.883301 496.679784 -2.796483
8 523.251131 523.251131 0.000000
```

Why Equal Temperament?

The Equal Temperament system we use today is one of many tuning systems that have tried to reconcile the pure Pythagorean ratios of the Octave (2:1) and the Fifth (3:2). Here's the problem in a nutshell using the piano keyboard as an example.

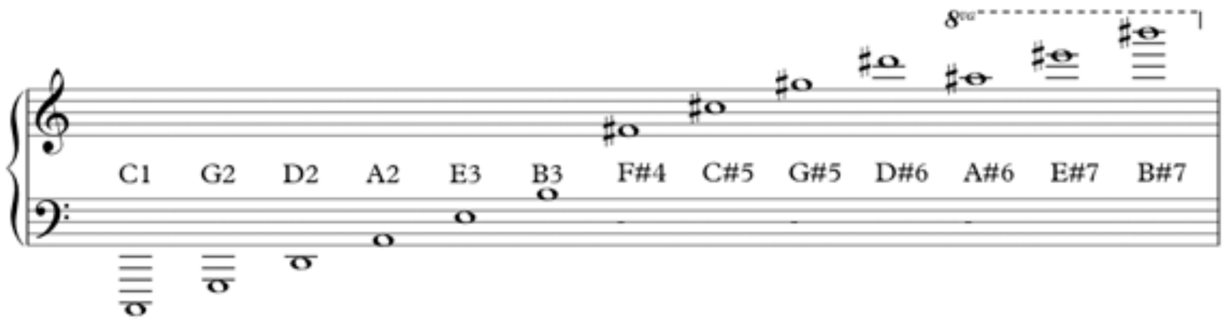
Start on the lowest C on the piano and continue up for 8 octaves until you reach the highest C on the piano. If you call the frequency of the lowest C ***f***, then the ending frequency is $f * 2^7$.

Seven Octaves on the Piano From C1 to C8



Now do the same thing tuning by fifths, a 3:2 ratio. After 12 fifths you'll reach the B#7 on the piano which should sound the same as C8, but it won't. If the starting frequency is ***f***, then the ending frequency is $f * \left(\frac{3}{2}\right)^{12}$.

Twelve Fifths on the Piano ending with B#7



The Comma Of Pythagoras

The difference by which (3/2)^12 exceeds 2^7 is known as the comma of Pythagoras.

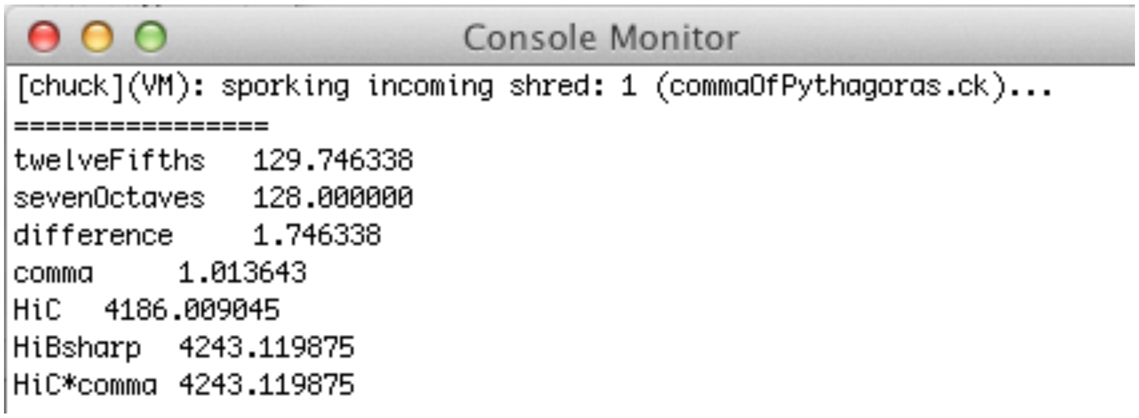
$$\frac{\left(\frac{3}{2}\right)^{12}}{2^7} = \text{comma of Pythagoras} = 1.01364$$

Enter and run this code to hear and see the difference.

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

Math.pow( 2, 7 ) => float sevenOctaves;
Math.pow( 3.0/2.0, 12 ) => float twelveFifths;

twelveFifths - sevenOctaves => float diff;
twelveFifths / sevenOctaves => float comma;

Std.mtof(24) * sevenOctaves => float hiC; // highest C8 on piano
Std.mtof(24) * twelveFifths => float hiBsharp; // Pythagorean
B# != C8

SinOsc sOctave => dac;  // 7 octave sine wave
SinOsc sFifth => dac; // 12 fifths sine wave
hiC => sOctave.freq;
hiBsharp => sFifth.freq;

0.4 => sOctave.gain;
0.0 => sFifth.gain;
1000::ms => now;

0.0 => sOctave.gain;
0.4 => sFifth.gain;
1000::ms => now;

0.4 => sOctave.gain;
0.4 => sFifth.gain;
1000::ms => now;
```

```
// Output
<<< "================", "" >>>;
<<< "twelveFifths\t", twelveFifths >>>;
<<< "sevenOctaves\t", sevenOctaves >>>;
<<< "difference\t", diff >>>;
<<< "comma\t", comma >>>;
<<< "HiC\t", hiC >>>;
<<< "HiBsharp\t", hiBsharp >>>;
<<< "HiC*comma\t", hiC * comma >>>;
```

Output



```
[chuck](VM): sporking incoming shred: 1 (commaOfPythagoras.ck)...
================
twelveFifths   129.746338
sevenOctaves   128.000000
difference     1.746338
comma     1.013643
HiC   4186.009045
HiBsharp  4243.119875
HiC*comma 4243.119875
```

One method of tuning the piano is to tune all octaves perfectly and then flatten each fifth in the cycle of fiths shown above by 1/11 of a comma. That way the cycle of fifths will end on the same frequency as the cycle of octaves. Piano tuners found that the when the interval of a fifth is flattened so that it beats 3 times every 5 seconds, that was the right amount.

On the guitar, every fret is positioned in Equal Temperament half steps with the 12th fret being the Octave and the 7th fret the fifth. Many guitar players use a method of tuning in harmonics where they play the harmonic on the fifth fret of a lower string and compare it to the harmonic on the seventh fret of the next higher string. If the harmonics match they think it's in tune. Mathematically it's not. The 5th fret harmonic is two octaves above the the open string and the 7th fret harmonic is one octave plus a fifth above the open string. The harmonics produce the pure Pythagorean ratios, the frets produce Equal Temperament ratios. When the harmonic method of tuning is used over all six strings the errors compound themselves.

String Ensembles and Choral groups often use pure ratios in their performances because they are not bound by Equal Temperament. A violinist trained to produce pure intervals sometimes has trouble adjusting their intonation when playing with a piano.

Cents

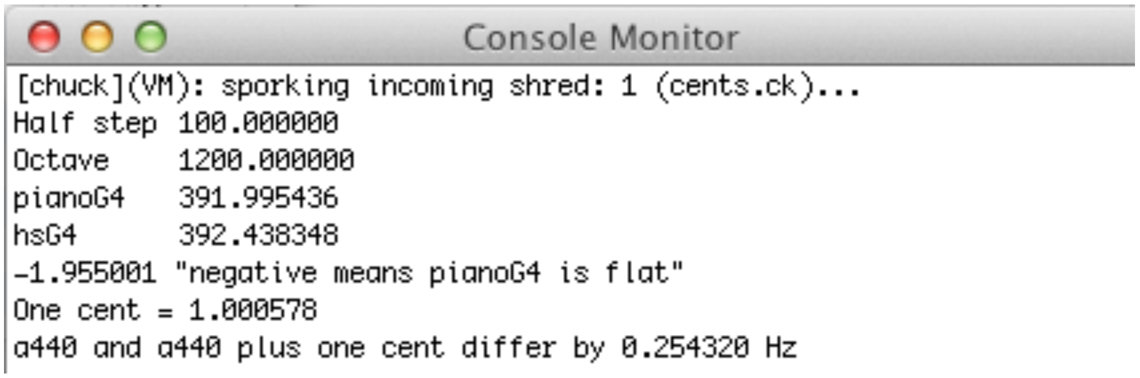
The audio unit used for measuring small differences in frequency is called a cent. By definition there are 1200 cents in one octave. A half step is to 100 cents. This is the general formula to find the cents difference between any two frequencies.

$$centDifference = 1200 * \log_2\left(\frac{f1}{f2}\right)$$

Enter and run this code.

```
// cents.ck
// John Ellinger Music 208 Winter 2014
function float calcCentDifference( float f1, float f2 )
{
    return 1200 * Math.log2( f1 / f2 );
}

// Run some tests
// Half step == 100 cents
<<< "Half step\t", calcCentDifference( Std.mtof(61),
Std.mtof(60) ) >>>;

// Octave == 1200 cents
<<< "Octave\t", calcCentDifference( Std.mtof(72), Std.mtof(60) ) >>>;

// Difference between Harmonic Series G4 and Piano G4
Std.mtof(67) => float pianoG4;
Std.mtof(36) * 6 => float hsG4; // starting from C2, G4 is 6th
harmonic
<<< "pianoG4\t", pianoG4 >>>;
<<< "hsG4\t\t", hsG4 >>>;
<<< calcCentDifference( pianoG4, hsG4 ), "\"negative means pianoG4 is flat\"" >>>;

// One cent  1.000578
<<< "One cent =", Math.pow( 2, 1.0/1200 ) >>>;
440 => float a440;
440 * 1.000578 => float a440plusOneCent;
<<< "a440 and a440 plus one cent differ by", a440plusOneCent - a440,  "Hz" >>>;
```

Output

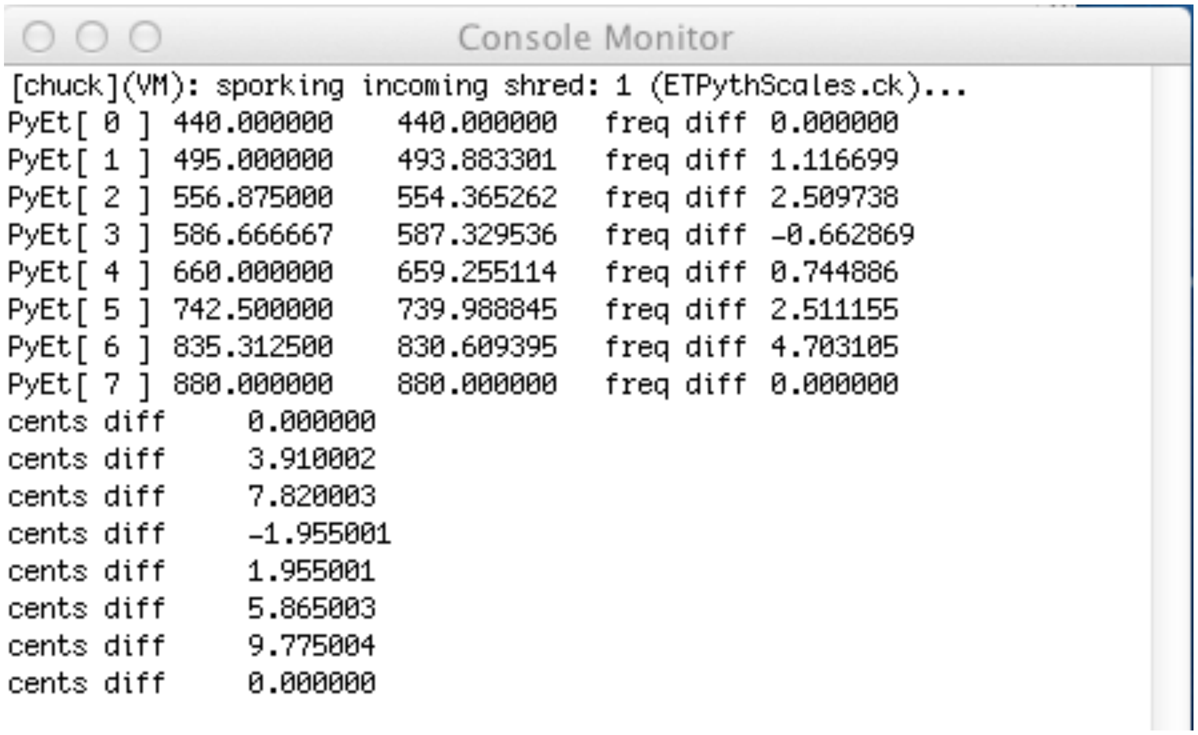


```
[chuck](VM): sporking incoming shred: 1 (cents.ck)...
Half step 100.000000
Octave    1200.000000
pianoG4   391.995436
hsG4      392.438348
-1.955001 "negative means pianoG4 is flat"
One cent = 1.000578
a440 and a440 plus one cent differ by 0.254320 Hz
```

Calculate the Cents Difference Between Notes of the Pythagorean and Equal Tempered Scales

Add a for loop to the end of the ETPythScales.ck program that calculates the cent difference between the Pythagorean and Equal Tempered scales.

```
// from PyEtScalesCompare.ck
for ( 0 => int ix; ix < psc.scale.cap(); ix++ )
{
    psc.scale[ix] => float fPY;
    Std.mtof( etsc.scale[ix] ) => float fET;

    spork ~ psc.playOneNote( ix, 5000::ms, 100::ms );
    spork ~ etsc.playOneNote( ix, 5000::ms, 100::ms );
    5100::ms => now;

    // Output Compare the MIDI (EqualTemp) notes to Pythagorean
    <<< ix+1, fET, fPY,  fET - fPY >>>;

    // Write compare cents code here
    <<< ix+1, fET, fPY,  fET - fPY >>>;
}
```



Console Monitor

```
[chuck](VM): sporking incoming shred: 1 (ETPythScales.ck)...
PyEt[ 0 ] 440.000000   440.000000   freq diff 0.000000
PyEt[ 1 ] 495.000000   493.883301   freq diff 1.116699
PyEt[ 2 ] 556.875000   554.365262   freq diff 2.509738
PyEt[ 3 ] 586.666667   587.329536   freq diff -0.662869
PyEt[ 4 ] 660.000000   659.255114   freq diff 0.744886
PyEt[ 5 ] 742.500000   739.988845   freq diff 2.511155
PyEt[ 6 ] 835.312500   830.609395   freq diff 4.703105
PyEt[ 7 ] 880.000000   880.000000   freq diff 0.000000
cents diff     0.000000
cents diff     3.910002
cents diff     7.820003
cents diff     -1.955001
cents diff     1.955001
cents diff     5.865003
cents diff     9.775004
cents diff     0.000000
```