MUSC 208 Winter 2014
John Ellinger, Carleton College

Lab 7 Stuck MIDI Notes, Impulse, Phasors, Random

Setup

Download and unzip m208Lab7.zip to your Desktop. Open Octave and cd to the m208Lab7 directory.

```
octave:9> cd /Users/je/Desktop/m208Lab7
octave:10> ls
kb.ck            music.wav       playCmajor.ck
```

Lab 7 will cover many of the same ideas introduced in Lab 6 and Homework 5-6. However, Lab 7 will implement them in ChucK instead of in Octave. We'll introduce the ChucK classes for Impulse and Phasor, and make use of different forms of the random function.

The ChucK Impulse class lets you work with a sound sample by sample. You can use it to create or play a waveform. The Phasor class has methods that work similar to the phase_increment and phase_index variables we used Octave. We'll use the Phasor class to generate a sine wave at a fixed frequency and slow down and speed up audio files. The lab will end using random functions to generate sound.

Homework 3-4 Problem 3 Optimized

playCmajor.ck in miniAudicle

Mac - open Au Lab first.

The playCmajor.ck scale in the lab folder illustrates several of the points mentioned in the lecture. Open the version of playCmajor.ck in the m208lab7 folder in miniAudicle.

Comment/Uncomment These Lines As Appropriate

```
23 // Shift key code is different on windows
24 // http://www.win.tue.nl/~aeb/linux/kbd/scancodes-1.html gives
25 // 42 => int which_ShiftLeft;  // win
26 // 54 => int which_ShiftRight; // win
27 // 225 => int which_Shift;     // mac
```

Type c d e f g a b C c and it seems to work. That's because the default piano sound has a rapid decay.

Stuck Notes

Mac - Quit and restart Au Lab to stop the stuck notes

Change the instrument to an organ that does not decay and you'll hear stuck notes. Change this

```
64 // change these as desired
65 0 => int patchNum;
66 60 => int scaleStart;
67
68 // infinite event loop
69 while( true )
70 {
```

to this

```
64 // change these as desired
65 19 => int patchNum;
66 48 => int scaleStart;
67
68 // infinite event loop
69 while( true )
70 {
```

Fix stuck notes

Every MIDI note on message (NON) must be matched with a MIDI note off message (NOF). The simplest NOF message is a NON with a velocity of zero.

Remove the /* and */ comments in the msg.isButtonUp() section of the while loop and run the program again. No more stuck notes.

Quit miniAudicle

Quit miniAudicle but leave Au Lab open.

Open Terminal

Open Terminal, change the working directory to m208lab7 and play the C major scale using chuck.

```
511,je@jemac:~/Desktop/m208lab7$ chuck playCmajor.ck
keyboard 'Keyboard' ready
```

Type c d e f g a b C c. It still works, sort of. Exit chuck with Ctrl-C.

Oh No! Stuck Notes Again

Fix the program so that Ctrl-C will exit cleanly.

Hint: Use the me.exit() function if Ctrl-c is captured on a key down message. The Ctrl key is a modifier key like the Shift key. Use the kb.ck program to find the key codes for Ctrl. Windows keyboards may have different results for left and right Ctrl key codes.

SndBuf Class

You've used the SndBuf class before to play wav files.

[ugen]: SndBuf

- *sound buffer (now interpolating)*
- *reads from a variety of file formats*
- *see examples: sndbuf.ck*

(control parameters)

- **.read** - (string , WRITE only) - *loads file for reading*
- **.chunks** - (int, READ/WRITE) - *size of chunk (# of frames) to read on-demand; 0 implies entire file, default; must be set before reading to take effect.*
- **.samples** - (int , READ only) - *get number of samples*
- **.length** - (dur, READ only) - *get length as duration*
- **.channels** - (int , READ only) - *get number of channels*
- **.pos** - (int , READ/WRITE) - *set position (0 < p < .samples)*
- **.rate** - (float , READ/WRITE) - *set/get playback rate (relative to file's natural speed)*
- **.interp** - (int , READ/WRITE) - *set/get interpolation (0=drop, 1=linear, 2=sinc)*
- **.loop** - (int , READ/WRITE) - *toggle looping*
- **.freq** - (float , READ/WRITE) - *set/get loop rate (file loops / second)*
- **.phase** - (float , READ/WRITE) - *set/get phase position (0-1)*
- **.channel** - (int , READ/WRITE) - *sel/get channel (0 < p < .channels)*
- **.phaseOffset** - (float , READ/WRITE) - *set/get a phase offset*
- **.write** - (string , WRITE only) - *loads a file for writing (or not)*

Enter and run this code. Save it as playSndBuf.ck.

You can use me.sourceDir() to set miniAudicles working directory without using the Preferences dialog.

```
// playSndBuf.ck

// read wav file into SndBuf
SndBuf buf => dac;
me.sourceDir() + "/music.wav" => string fname;
<<< "fname is" , fname >>>;

fname => buf.read;
0 => buf.pos;
buf.length() => now;

```

Impulse Class

http://chuck.cs.princeton.edu/doc/program/ugen.html

[ugen]: Impulse

- *pulse generator - can set the value of the current sample*
- *default for each sample is 0 if not set*

(control parameters)

- **.next** - (float , READ/WRITE) - *set value of next sample to be generated. (note: if you are using the **UGen.last** method to read the output of the impulse, the value set by **Impulse.next** does not appear as the output until after the next sample boundary. In this case, there is a consistent 1::samp offset between setting .next and reading that value using .last)*

[example]

```
Impulse i => dac;
while( true ) {
    1.0 => i.next;
    100::ms => now;
}
```

We'll use the Impulse class to play a wav file one sample at a time. Then we'll

manipulate the samples before we play them.

Enter and run this code. Save it as impulsePlay1.ck.

```
// impulsePlay.ck
// use Impulse to play one sample at a time from a sound file
Impulse imp => dac;

// read wav file into SndBuf
SndBuf buf;
me.sourceDir() + "/music.wav" => string fname;
fname => buf.read;

buf.samples() => int numSamples;

function void playNormalSpeed()
{
    0 => int ix;
    while( 1 )
    {
        buf.valueAt( ix ) => imp.next;
        1::samp => now;
        ix++;
        //
        if ( ix > numSamples )
            break;
    }
}

playNormalSpeed();
```

Twice as Fast

Twice as fast is equivalent to playing every other sample which plays the sound one octave higher. Enter and run this code. Save it as impulsePlay2.ck

```
function void playNormalSpeed()
{
    0 => int ix;
    while( 1 )
    {
        buf.valueAt( ix ) => imp.next;
        1::samp => now;
        ix++;
        //
        if ( ix > numSamples )
            break;
    }
}

function void playDoubleSpeed()
{
    0 => int ix;
    while (1)
    {
        if ( ix % 2 == 0 ) // chuck mod function %
        {
            buf.valueAt( ix ) => imp.next;
            1::samp => now;
            ix + 2 => ix;
            if ( ix > numSamples )
                break;
        }
    }
}

playNormalSpeed();
playDoubleSpeed();
```

Half Speed

Playing every sample twice which plays the sound one octave lower. Add this code and run it. Save it as impulsePlay3.ck

```
function void playHalfSpeed()
{
    0 => int ix;
    while(1)
    {
        for ( 0 => int n; n < 2; n++ )
        {
            buf.valueAt( ix ) => imp.next;
            1::samp => now;
        }
        1::samp => now;
        ix++;
        if ( ix > numSamples )
            break;
    }
}

playNormalSpeed();
playDoubleSpeed();
playHalfSpeed();
```

Play Kiss Kiss

Remain seated. Add this code and run it. Save it as impulsePlay4.ck. After several repetitions it start to sound like "KissKiss" to me.

```
function void playKissKiss()
{
    2000 => int ix;
    0 => int tmp;
    while(1)
    {
        buf.valueAt( ix ) => imp.next;
        1::samp => now;

        ix + 4000 => tmp;
        // check for wrap around
        if ( tmp  > numSamples )
            tmp - numSamples => tmp;

        buf.valueAt( tmp ) => imp.next;
        1::samp => now;
        ix + 2 => ix;

        if ( ix > numSamples )
            2000 => ix;
    }
}

playNormalSpeed();
playDoubleSpeed();
playHalfSpeed();
playKissKiss();
```

Phasors

A phasor is a rotating vector. View this web page.

http://science.sbcc.edu/~physics/flash/optics/phasors4.swf

Phasors In ChucK

The Phasor object in ChucK is used to generate the phase values of a sine wave at a given frequency. The ChucK documentation describes the phasor like this.

[ugen]: Phasor

- *phasor - simple ramp generator (0 to 1)*
- *can be used as a phase control.*

(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) - *set duration of the ramp in each cycle. (default 1.0)*

All Phasor methods are defined as READ/WRITE. Read and write methods use a slightly different syntax. For example, if you want to set (write to) .freq you'd use this form:

```
441 => aPhasor.freq;.
```

If you want to get (read from) .freq you'd use this form:

```
aPhasor.freq.freq() => float frq;
```

You can find the default settings for each of the Phasor parameters with this short program.

```
Phasor phsr => blackhole;

<<< "phsr.freq", phsr.freq() >>>;
// <<< "phsr.sfreq", phsr.sfreq() >>>;
<<< "phsr.phase", phsr.phase() >>>;
<<< "phsr.sync", phsr.sync() >>>;
// <<< "phsr.width", phsr.width() >>>;
```

Output should be:

```
[chuck](VM): sporking incoming shred: 1 (phasorTest.ck)...
phsr.freq 220.000000
phsr.phase 0.000000
phsr.sync 0
```

Errors In Phasor Documentation

phasor.sfreq and phasor.width are undefined. You'll get an error if you try to use them.

Phasors and Phase Increment

The phasor frequency is equivalent to the phase_increment we used in Octave in Lab 6. A phasor frequency of 1.0 Hz will index every sample position at the default sample rate. Here's an example.

Using Impulse and Phasor To Create A Sine Wave

Enter and run this code. Save as phasorSine.ck

```
// phasorSine.ck

Impulse imp => dac;
Phasor phsr => blackhole; // get samples without sound

441 => phsr.freq;
0.3 => float A;

now + 1::second => time later;
while (now < later )
{
      A * Math.sin( 2 * Math.PI * phsr.phase() ) => imp.next;
      1::samp => now;
}
```

LFO (Low Frequency Oscillator)

Another common use of the phasor is to create an LFO the modulates the amplitude of a waveform. Try this. Experiment with different beatsPerSecond values.

```
// phasorLFO.ck

SinOsc s => dac;
0.5 => s.gain;
// mtof does MIDI to Frequency conversion
// there's also an ftom()
Std.mtof( 60 ) => s.freq;

// Low Frequency Oscillator LFO
Phasor lfo => blackhole;
1.5 => float beatsPerSecond;
beatsPerSecond / 2 => lfo.freq;

// should hear 3 beats
now + 2::second => time later;
.4 => float A;
while ( now < later )
{
    A * Math.sin( 2 * Math.PI * lfo.phase() ) => s.gain;
    10:: samp => now;
}
```

Graphing a Phasor

We'll use the SinOsc object in ChucK to graph the phase of the sine wave.

[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*

http://chuck.cs.princeton.edu/doc/program/ugen.html

Create this short program in miniAudicle to print out the phase samples for three periods of a 441 Hz sine wave.

```
SinOsc s => blackhole;

44100 => int SR;
441.0 => s.freq;

// this is a little more than three periods
( SR/s.freq() + 10 )::samp + now => time later;

// print phase to Console Monitor window
while (now < later )
{
    <<< s.phase() >>>;
    1::samp => now;
}
```

Copy all phase data from the Console Window and paste it into TextWrangler.

```
0.000000
0.010000
0.020000
0.030000
```

```
0.040000
0.050000
0.060000
0.070000  etc.
```

Save the file in the m208Lab7 folder as sinePhase_441Hz.txt.

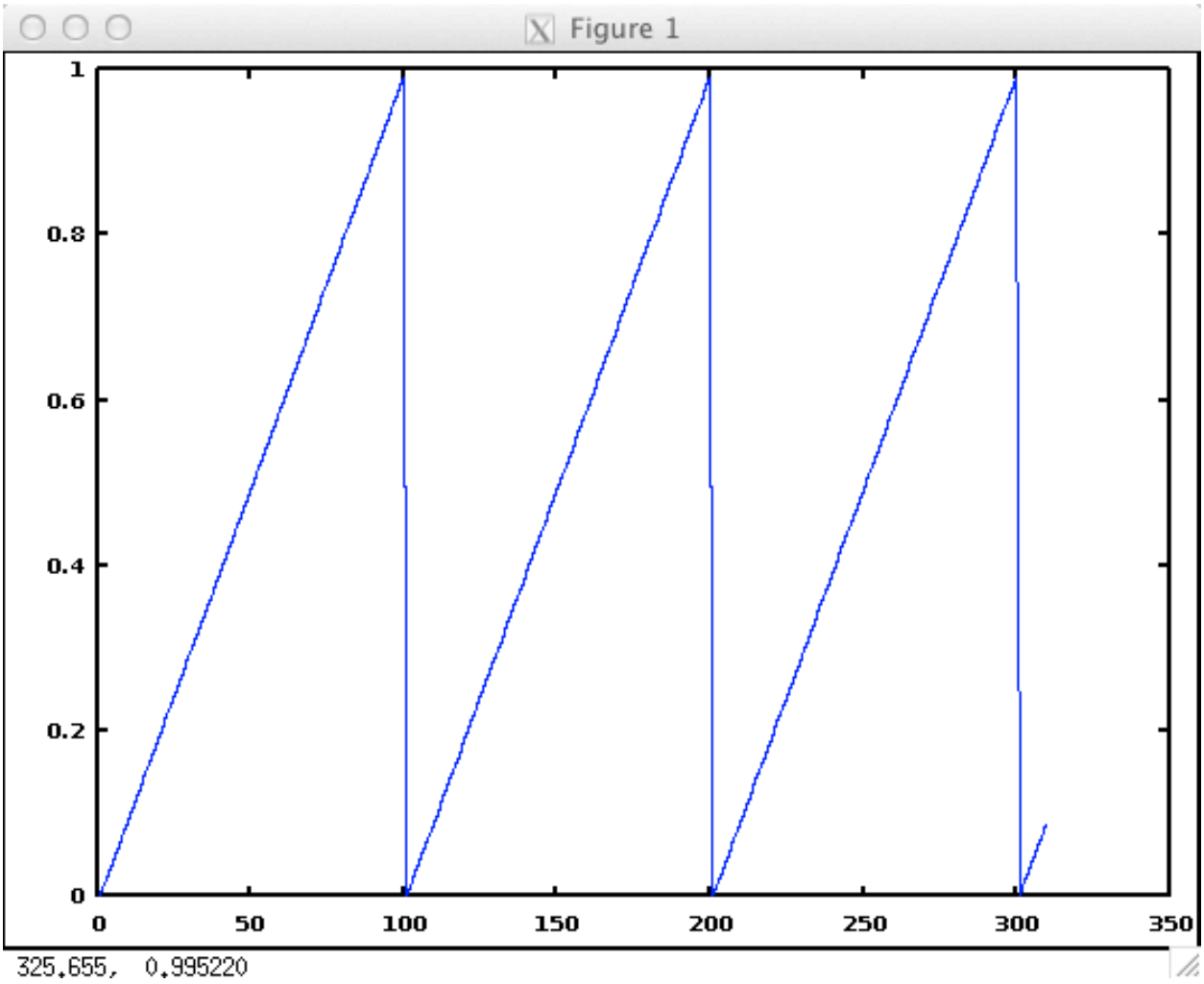
Open Octave

Open Octave and cd to the ~/Desktop/m208Lab7 folder and enter these commands.

```
octave-3.4.0:1> cd /Users/je/Desktop/m208Lab7
octave-3.4.0:2> phs = load( "-ascii", "sinePhase_441Hz.txt" );
octave-3.4.0:3> phs
phs =

   0.00000
   0.01000
   0.02000
   0.03000
   0.04000
```

Plot The Phase In Octave

```
octave-3.4.0:10> plot(phs);
```



The X axis units are samples at the rate of 44100 samples in one second. The Y axis units are radians where 2π radians is represented as 1.0 or one full rotation of a vector around the unit circle. The plot shows the period repeats itself every 100 samples, which represents a frequency of 44100/100 = 441 Hz.

Unipolar And Bipolar Waveforms

The phasor is similar to the sawtooth wave used in many synthesizers. The difference is that the phasor's amplitude values range from zero to one, while the sawtooth's values range from minus one to plus one. Waveforms whose amplitude values range for 0 to 1 are called unipolar waveforms. Waveforms whose amplitude values range from –1 to + 1 are called bipolar waveforms.

Repeat With A Phasor

Modify the code to use a Phasor instead of a SinOsc.

```
// phasor441.ck

1::second / 1::samp => float SR;
<<< "SR", SR >>>;

Phasor phsr => blackhole;
441.0 => phsr.freq;

SR / phsr.freq() => float samplesPerPeriod;
<<< "samplesPerPeriod", samplesPerPeriod >>>;

( samplesPerPeriod * 3 + 1 )::samp + now => time later;

while ( now < later )
{
    <<< phsr.phase(), "" >>>;
    1::samp => now;
}
```

Run it and you'll get exactly the same values.

Phasor at 1 Hz Equals the Sample Rate

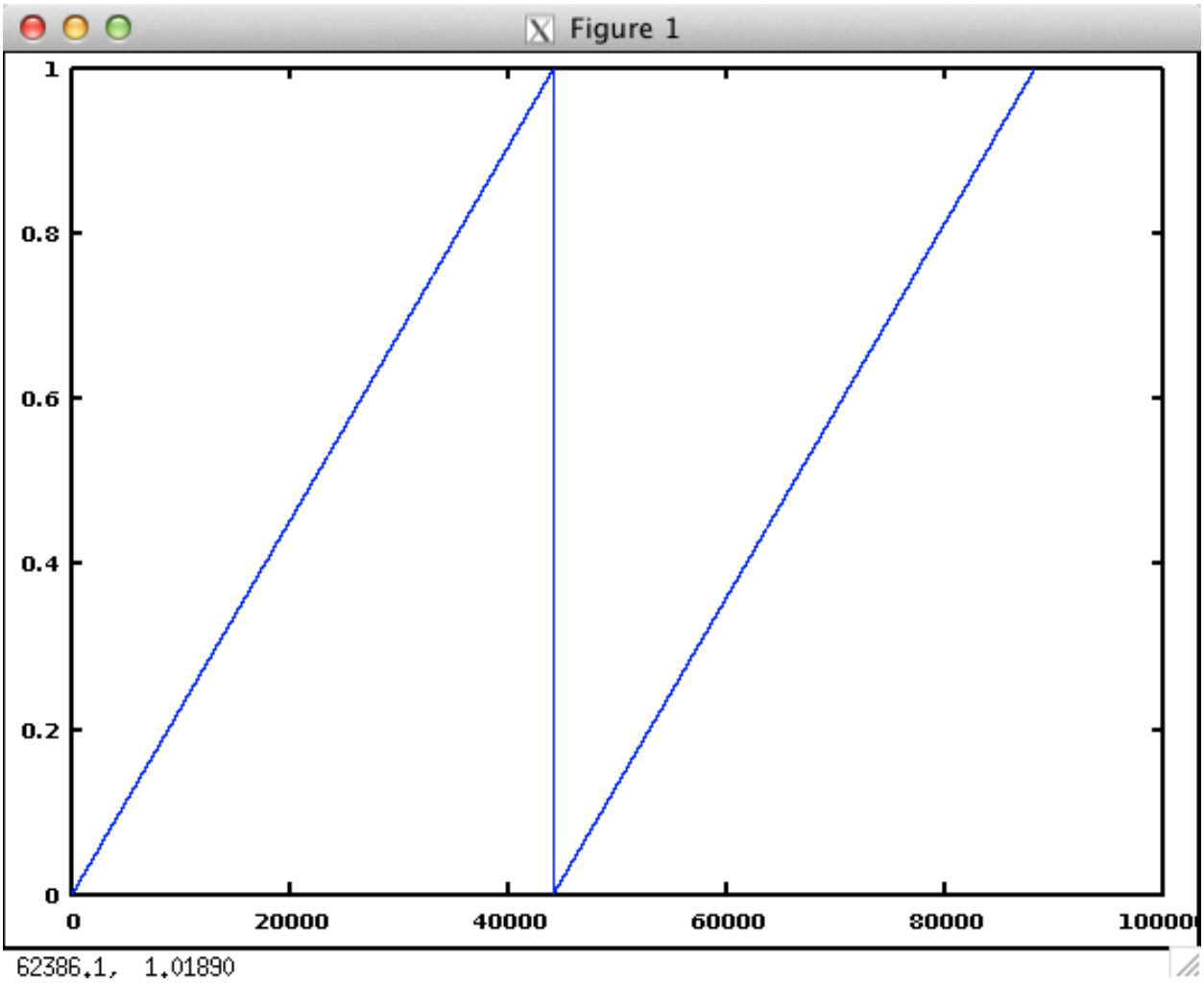
This code also demonstrates how ChucK can read and write from a file.

```
// phasor_1Hz
// demonstrates reading and writing to a file

Phasor phsr => blackhole;
1.0 => phsr.freq;

// open a text file for writing
FileIO fio;
// open for write
fio.open( "phasor_1Hz.txt", FileIO.WRITE );

// guarantee file was opened
if( ! fio.good() )
{
    <<< "can't open file for writing" >>>;
    me.exit();
}

// let phasor run for two seconds
44100 => int SR;
2 * SR => int numSamples;
numSamples::samp + now => time later;

0 => int samplesWritten;
// output phase to file
while (now < later )
{
    // write to file
    fio <= phsr.phase() <= IO.newline();
    1::samp => now;
    samplesWritten++;
}
```

```
// phasor_1Hz
// demonstrates reading and writing to a file

Phasor phsr => blackhole;
1.0 => phsr.freq;

// open a text file for writing
FileIO fio;
// open for write
fio.open( "phasor_1Hz.txt", FileIO.WRITE );

// guarantee file was opened
if( ! fio.good() )
{
    <<< "can't open file for writing" >>>;
    me.exit();
}

// let phasor run for two seconds
44100 => int SR;
2 * SR => int numSamples;
numSamples::samp + now => time later;

0 => int samplesWritten;
// output phase to file
while (now < later )
{
    // write to file
    fio <= phsr.phase() <= IO.newline();
    1::samp => now;
    samplesWritten++;
}
```

Plot In Octave

```
octave:5> phs = load( "-ascii", "phasor_1Hz.txt" );
octave:6> plot( phs );
```



Wavetable Iterpolation with Phasors

The following code demonstrates the truncate method of interpolation. It also uses three overloaded debugPrint functions.

```
// phasorPlay.ck
Impulse imp => dac;

// read wav file into SndBuf
SndBuf buf;
me.sourceDir() + "/music.wav" => string fname;
fname => buf.read;

// get number of samples in the wav file
buf.samples() => int numSamp;

// calculate sample rate
1::second / 1::samp => float SR;

// DEBUGING FUNCTIONS
1 =>  int DEBUG;
function void debugPrint( string str, string str2 )
{
    if ( DEBUG )
        <<< str, str2 >>>;
}

function void debugPrint( string str, int num )
{
    if ( DEBUG )
        <<< str, num >>>;
}

function void debugPrint( string str, float num )
{
    if ( DEBUG )
        <<< str, num >>>;
}

debugPrint( "fname:", fname ); // prints string string
debugPrint( "numSamp:", numSamp );  // prints string int
debugPrint( "Sample Rate:", SR );  // prints string float

// Use a phasor to control playback speed
Phasor phsr => blackhole;
0.75 => phsr.freq;
float fpos;
int ipos;


while ( 1 )
{
    phsr.phase() * SR => fpos;
    // truncate method
    Math.floor( fpos ) $ int => ipos;
    buf.valueAt( ipos ) => imp.next;
    1::samp => now;
}
```

Random Number Functions In Chuck

ChucK documentation lists two sets of random functions; one set in the Std library and another in the Math library. In order to use functions from these libraries you prefix the method name with either Std. or Math. The Std random functions are deprecated so we'll be using the Math library. Here's the documentation for the four random functions.

[function]: int **random** ( );

- *generates random integer between 0 and Math.RANDOM_MAX*
- *(NOTE: Math.random*() functions use a different, superior random number generator than the Std.rand*() functions)*

[function]: int **random2** (int **min**, int **max**);

- *generates random integer in the range [min, max]*

[function]: float **randomf** ( );

- *generates random floating point number in the range [0, 1]*
  (NOTE: this is different semantics than Std.randf(), which has the range [-1,1])

[function]: float **random2f** (float **min**, float **max**);

- *generates random floating point number in the range [min, max]*

http://chuck.cs.princeton.edu/doc/program/stdlib.html

Example 1 - Math.random

Returns an integer.

```
0 =>  int count;
do
{
    Math.random() => int num;
    <<< "Math.random:", num >>>;
    count++;
} while ( count < 10 );

<<< "The largest random integer is: ", Math.RANDOM_MAX >>>;

```

The numbers are large and not real useful for digital audio.

```
[chuck](VM): sporking incoming shred: 1 (RandomTest1.ck)...
Math.random: 151680496
Math.random: 847770076
Math.random: 876700941
Math.random: 470241155
Math.random: 1662294721
Math.random: 841780504
Math.random: 1188298021
Math.random: 1965591597
Math.random: 1188229007
Math.random: 206308855
The largest random integer is:  2147483647
```

Example 2 - Math.random2

Returns an int within the range of the two parameters low, high.

```
1 => int count;
int guess;
<<< "I'm thinking of a number from 1-10" >>>;
do
{
    Math.randomZ(1,10) => guess;
    <<< "Try", count, "Math.randomZ = ", guess >>>;
    count++;
} until ( guess == 8 );
<<< "You're right it was ", guess >>>;
<<< "And it only took", count-1, " tries." >>>;

```

Another example of how computer music sounded in 1950's movies.

```
SinOsc s => dac;
0.2 => s.gain;
while( true )
{
    Math.randomZ( 200, 1000 ) => s.freq;
    100::ms => now;
}
```

Example 3 - Math.randomf

Returns a float in the range 0.0 to 1.0.

```
0 => int count;
while ( count < 10 )
{
    Math.randomf() => float flowt;
    <<< "Math.randomf:", flowt >>>;
    count++;
}

```

Another example.

```
SinOsc s => Gain g => dac;
0.3 => s.gain;
while( true )
{
    Math.randomf() => g.gain;
    250::ms => now;
}
```

Example 3 - Math.random2f

Returns a float within the range low to high.

```
<<< "I'm thinking of a number from 1-10 with 6 decimal places!" >>>;
1 => int count;
float guess;
do
{
    Math.random2f(1.0,10.0) => guess;
    <<< "Try", count, "Math.random2 = ", guess >>>;
    count++;
} until ( guess > 5.55 && guess < 5.56 );
<<< "\nYou're in the ball park ", guess >>>;
<<< "It was between 5.5 and 5.6" >>>;
<<< "And it only took", count-1, " tries." >>>;

```

Another example using one of ChucK's built in STK (Synthesis ToolKit) instruments.

[ugen]: **Mandolin** (STK Import)

extends StkInstrument

(control parameters)

- **.bodySize** - (float , READ/WRITE) - *body size (percentage)*
- **.pluckPos** - (float , READ/WRITE) - *pluck position [0.0 - 1.0]*
- **.stringDamping** - (float , READ/WRITE) - *string damping [0.0 - 1.0]*
- **.stringDetune** - (float , READ/WRITE) - *detuning of string pair [0.0 - 1.0]*
- **.afterTouch** - (float , WRITE only) - *aftertouch (currently unsupported)*
- **.pluck** - (float , WRITE only) - *pluck instrument [0.0 - 1.0]*

(inherited from StkInstrument)

- **.noteOn** - (float velocity) - *trigger note on*

- **.noteOff** - (float velocity) - *trigger note off*
- **.freq** - (float frequency) - *set/get frequency (Hz)*
- **.controlChange** - (int number, float value) - *assert control change*

Try this example.

```
1  Mandolin m => dac;
2  Std.mtof(67) => m.freq; // G above middle C
3
4  while (true)
5  {
6      // randomize the pluck position
7      Math.randomf() => m.pluckPos;
8
9      // Play it
10     m.pluck(1);
11
12     150::ms => now;
13 }
```

A Musical Example

Line 2 - add NRev a reverb effect
Line 5 – 0 is no reverb, 1.0 is 100% reverb, known in other software as wet/dry mix
Lines 8-11 define standard quarter note, eighth note, and 16th note values at a tempo of 120
Line 15 create an array of notes.
The notes are enclosed in brackets.
@=> is used for array assignment
notes[] delcares notes to be an array
elements of the array can be indexed: n[0] = 55, n[2] = 60, n[7] = 72, n[8] is an error
Line 19 select a random index from the notes[] array
Line 22 mtof() is MIDI to Frequency (there's also ftom)
Lines 27-33 define probabilities in percentages that each of the three note values will occur.

```
// Add a reverb to the mix
Mandolin m => NRev reverb => dac;
// change the reverb wet/dry mix
// 0 is no reverb, 1.0 is full reverb
0.2 => reverb.mix;

// quarter note in milliseconds, Tempo = 120
500 => float t4;
t4 / 2 => float t8;
t8 / 2 => float t16;
dur millis;

// create an array of pentatonic scale notes
//G   A   C   D   E   G   A   C
[ 55, 57, 60, 62, 64, 67, 69, 72 ] @=> int notes[];

while (true)
{
    // choose a random index into the notes array
    Math.random2( 0, notes.cap()-1) => int ix;
    // select that note
    Std.mtof( notes[ix] )  => m.freq;

    // set a random plucking position
    Math.randomf() => m.pluckPos;

    Math.random2(1,100) => int guess;
    if ( guess < 11 )  // 10% chance
        t4::ms => millis;
    else if ( guess > 70 ) // 30% chance
         t8::ms => millis;
    else
        t16::ms => millis;  // 60% chance

    // Play it
    m.pluck(1);


    millis => now;
}
```

Experiments

Change the tempo of the quarter note.
Transpose the notes an octave lower Std.mtof(notes[ix] - 12);
Run multiple shreds.

You can also randomize
.bodySize
.stringDamping
.stringDetune

- **.bodySize** - (float , READ/WRITE) - *body size (percentage)*
  - **.pluckPos** - (float , READ/WRITE) - *pluck position [0.0 - 1.0]*
  - **.stringDamping** - (float , READ/WRITE) - *string damping [0.0 - 1.0]*
  - **.stringDetune** - (float , READ/WRITE) - *detuning of string pair [0.0 - 1.0]*
  - **.afterTouch** - (float , WRITE only) - *aftertouch (currently unsupported)*
  - **.pluck** - (float , WRITE only) - *pluck instrument [0.0 - 1.0]*