MUMT 618: Week #2

1 Time-Varying Delay Effects

A variety of common audio effects are implemented using variable-length delay lines. In this section, we analyze several of these techniques, many of which can be seen as simulating physical or non-physical variations of sound sources and/or receivers. Note that a discussion of artificial reverberation will be "delayed" to a later section.

1.1 Flanging

- Flanging involves the summing together of a signal and a time-varying delayed version of itself.
- The input-output relationship for a flanger is given by:

  $$y[n] = x[n] + gx[n - M[n]],$$

  where $M[n]$ is the time-varying length of a delay line and g is the "depth" of the flanging effect. A flanger block diagram is shown in Fig. 1.

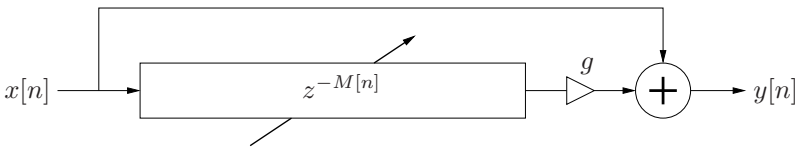

Figure 1: A digital flanger block diagram.

- Because the delay-line length, $M[n]$, must change continuously and smoothly through time, it is necessary to make use of an interpolating delay line.
- At any instant in time, the flanger is equivalent to a feedforward comb filter, which has a frequency response as shown in Fig. 2.
- For $g > 0$, there are M peaks in the frequency response, centered about the frequencies $\omega_k = 2\pi k/M, k = 0, 1, \ldots, M - 1$. Between these peaks, there are M notches at intervals of f_s/M Hz.
- As M changes over time, the peaks and notches of the comb response are compressed and expanded. The spectrum of a sound passing through the flanger is thus accentuated and deaccentuated by frequency region in a time-varying manner.
- The delay-line length of a flanger is typically modulated by a low-frequency oscillator (LFO). Oscillator waveforms are typically sinusoidal, triangular, or exponential.
- For a sinusoidally varied delay,

  $$M[n] = M_0 \cdot (1 + A\sin[2\pi f nT]),$$

  where f is the flanger "rate" in Hz, A is the "excursion" (maximum delay swing), and M_0 is the average delay-line length that controls the average notch density.

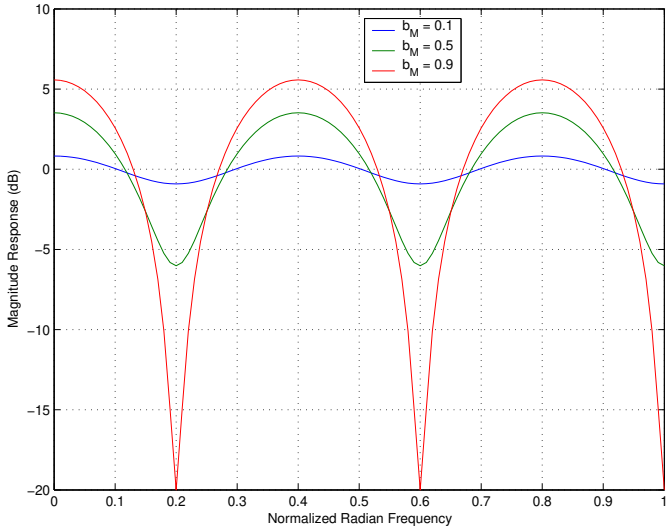

Figure 2: Magnitude response of a feedforward comb filter with $M = 5$, $b_0 = 1$, and $g = b_M = 0.1$, 0.5, and 0.9.

- For values of $-1 \leq g \leq 0$, the peaks and notches of the comb filter trade places. In practice, g is normally contrained to the interval $[0, 1]$ and the option of sign inversion is provided by a "phase inversion" switch.
- In the inverted mode, a notch is located at zero frequency. As a result, bass response will likely be weakened.
- Some flangers also implement feedback (in addition to the delayed feedforward path), which introduces spectral peaks and notches as previously described for feedback comb filters.
- The following MSP patch implements a flanger.

1.2 Phasing

- A *phaser*, or *phase shifter*, is similar to a flanger in that it sweeps notches through the spectrum of an input signal. But while a flanger provides only uniformly spaced notches, a phasor can modulate the frequencies of *non-uniformly* spaced notches.
- A phaser is implemented with allpass filters instead of delay lines, as shown in the block diagram of Fig. 3.

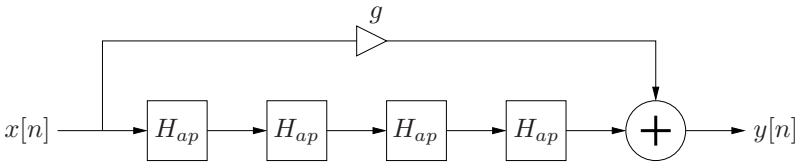

Figure 3: A digital phaser block diagram.

- Second-order allpass filters are particularly convenient to use because each can control a separate notch frequency and bandwidth. Second-order allpass filters have a transfer function given by

$$H(z) = \frac{a_2 + a_1 z^{-1} + z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}},$$

where

$$\begin{aligned} a_1 &= -2R\cos(\theta) \\ a_2 &= R^2, \end{aligned}$$

R is the radius of each pole relative to the z-plane unit circle (R=1), and the pole angles are $\pm\theta$. The pole angle can be interpreted as $\theta = \omega_c T$ where ω_c is the frequency and T is the sampling period.

- The phaser will have a notch wherever the phase of the allpass chain is at π (180 degrees). This happens close to the complex-conjugate pole pair angles.
- The instantaneous frequency response of a phaser created using 4 second-order allpass filters with notch frequencies set at 300, 800, 1000, and 4000 Hz and R = 0.9, 0.98, 0.8, and 0.9 is shown in Fig. 4.

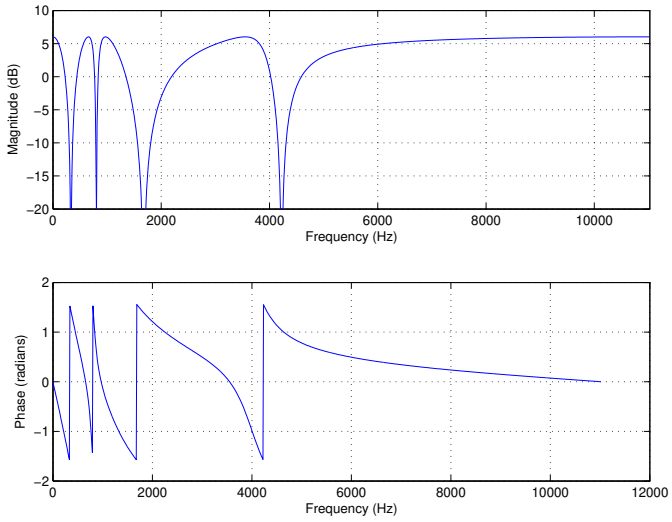

Figure 4: Instantaneous frequency response of a phaser created with 4 second-order allpass filters and notch frequencies set at 300, 800, 1000, and 4000 Hz.

- The depth of the notches can be varied together by changing the feedforward gain parameter g.
- To achieve the time-varying "phasing" effect, the notch frequencies are modulated with a periodic signal. Note that only a single filter coefficient need be changed in each allpass section to accomplish this.

1.3 Doppler Effect

- The Doppler effect occurs when a sound source and listener are moving relative to one another. When the source and listener move toward each other, the sound is perceived to increase in frequency. When the source and listener move away from each other, the sound is heard to decrease in frequency.
- The Doppler effect can be used to enhance the realism of simulated moving sound sources.
- The amount of frequency shift is given by:

$$f_l = f_s \frac{c + v_{ls}}{c - v_{sl}},$$

where f_s is the frequency of the source at rest, f_l is the frequency perceived by the listener, v_{ls} is the speed of the listener toward the source (zero if not moving), v_{sl} is the speed of the source toward the listener (zero is not moving), and c is the sound speed.

- We can simulate time-varying source and listener velocities using a time-varying digital delay line with separate read and write pointers. The write pointer corresponds to the source signal and the read pointer corresponds to the listener. If the source position is moving toward the listener, the write pointer increment should be changed from 1 to $1 + v_{sl}/c$. Likewise, if the listener is moving toward the source, the read pointer increment should be changed from 1 to $1 + v_{ls}/c$.
- Interpolated reads from a delay line (fractional delay lengths) were previously discussed. Interpolated writes are referred to as *de-interpolation*.
- Because Doppler shift is dependent only on the *relative* motion of a source and listener and because the de-interpolation process is generally more complicated to implement than interpolation, it is best to change only the read pointer increment when possible.
- A continuously varying delay can be implemented with a "growth parameter" $g = -v_{ls}/c$ such that the read pointer is incremented by $1 + g$ at each time step.
- Multiple read pointers can simulate multiple moving listeners.
- Multiple write pointers can simulate multiple moving sources.
- It is not possible to use a single delay line to simulate both multiple moving listeners and multiple moving sources. In this case, the number of necessary delay lines is the lesser of the number of listeners or sources.

1.4 Chorus Effect

- The simulation of many apparent similar sound sources occuring nearly in unison from a single, actual source is referred to as a chorus effect.
- The chorus effect is achieved by applying various independent modifications to multiple copies of a given input signal and summing them together to create the output signal.
- Signal modifications can include delay, frequency shift, and/or amplitude changes.
- Time-varying delay lines can simulate changes in delay and frequency shift.
- A single multi-tap interpolating delay line with modulated ouput tap locations can efficiently and effectively implement the chorus effect.
- The following MSP patch implements a chorus effect. Better results can be achieved with more "copies" of the input signal.
- The `rand~` MSP object produces interpolated noise at a subsampled rate, as specified by the input argument. This provides a noise like signal with energy only at frequencies below this rate.

1.5 Pitch Shifting

- The Doppler effect causes a source signal to appear as though it has been pitch shifted. Pitch shifting of an input signal can thus be implemented with time-varying delay lines as described above.
- Because the read pointer of a pitch shifter is incremented at a constant non-integer rate, the read pointer will eventually "catch up to" or "fall back into" the write pointer location. To avoid discontinuity issues caused when the read and write pointers cross, a multiple read pointer cross-fade system can be used.

2 Synthesis ToolKit Introduction

The *Synthesis ToolKit in C++ (STK)* is a set of open source audio signal processing and algorithmic synthesis classes written in C++. STK was designed to facilitate rapid development of music synthesis and audio processing software, with an emphasis on cross-platform functionality, realtime control, ease of use, and educational example code. STK currently runs with "realtime" support (audio and MIDI) on Linux, Macintosh OS X, and Windows computer platforms.
In this course, we will make use of STK to gain hands-on experience implementing simple acoustic and audio algorithms.

2.1 Introduction

- STK provides a set of C++ classes that help you create audio signal processing programs.
- STK makes use of object-oriented programming methodology and hierarchy.
- STK does not specify particular rates (audio and/or control) or scheduling systems. It is up to the programmer to design a framework that best addresses a given processing context.
- STK provides support for either single-sample computation or vectorized computations.
- A short STK tutorial, as well as class documentation, is provided at http://ccrma.stanford.edu/software/stk/.
- In the following notes, it is assumed that STK will be compiled from a unix command-line interpreter (shell) such as provided by the OS-X Terminal application or MSYS in Windows.
- Students will need to find a text editor for creating and editing programs. The text editor in the Eclipse IDE is a good option.

2.2 Installation and Compiling (Macintosh OS-X and Windows)

- For this course, we will compile programs that make use of STK classes by linking against a static library (`libstk.a`).
- The latest release of the STK distribution can be downloaded from http://ccrma.stanford.edu/software/stk/download.ht
  Alternately, the most up-to-date version, though perhaps not fully tested, can be downloaded from https://github.com/thestk/stk
- Download STK and put it into a working directory of your choice. In the following notes, it is assumed that the STK distribution is located in the `~/Desktop/tmp/stk/` directory and that a static library has been compiled in the `src` subdirectory. This can be done by typing `./configure` and then `make` from within the `stk/` directory (using a Terminal window).
- Under these conditions, we can compile a simple C++ program file `myprogram.cpp` in the working directory (`~/Desktop/tmp/`) and link against the STK library as follows (further libraries may be necessary depending on the particular program):

  ```
  g++ -std=c++11 -o myprogram -Istk/include/ -Lstk/src/ myprogram.cpp -lstk
  ```

- In the previous compile statement, the `-o` flag specifies a name for the resulting binary (which will be called `myprogram` in this example). The `-I` flag specfies a directory in which to search for header files, the `-L` flag specifies a directory in which to search for libraries, and the `-l` flag specifies a library name to link to (`stk` in this example).
- On a computer running Windows, the same approach as outlined above can be used with the MinGW compiler and the MSYS shell. Another, perhaps easier option, may be to use the Windows Subsystem for Linux (WSL), though I have not tried that yet.

- If attempting to compile with an IDE, such as Visual Studio Code, it will be critical to setup the proper preprocessor definitions and libraries for your system:
  - Non-realtime: __LITTLE_ENDIAN__
  - Realtime OS-X: __LITTLE_ENDIAN__, __MACOSX_CORE__; frameworks: CoreFoundation, CoreAudio, CoreMIDI; libraries: pthread
  - Realtime Windows Direct Sound API: __LITTLE_ENDIAN__, __WINDOWS_DS__; libraries: pthread, dsound, ole32, winmm
  - Realtime Linux ALSA API: __LITTLE_ENDIAN__, __LINUX_ALSA__; libraries: pthread, asound

2.3 Inheritance

Nearly all STK classes inherit from the Stk abstract base class, which provides common functionality related to error reporting, sample rate control, and byte swapping. Several other base classes exist that roughly group many of the classes according to function as follows:

- `Generator`: source signal unit generator classes [`Blit, BlitSaw, BlitSquare, Envelope, ADSR, Asymp, Noise, Modulate, SineWave, SingWave, Granulate`]
- `Filter`: digital filtering classes [`OneZero, OnePole, PoleZero, TwoZero, TwoPole, BiQuad, Fir, Iir, FormSwep, Delay, DelayL, DelayA, TapDelay`]
- `Function`: input to output function mappings [`BowTable, JetTable, ReedTable`]
- `Instrmnt`: sound synthesis algorithms, including physical, FM, modal, and particle models
- `Effect`: sound processing effect classes [`Echo, Chorus, PitShift, LentPitShift, PRCRev, JCRev, NRev, FreeVerb`]
- `WvOut`: audio file and streaming output classes [`FileWvOut, RtWvOut, InetWvOut`]
- `WvIn`: audio file and streaming input classes [`FileWvIn, FileLoop, RtWvIn, InetWvIn`]

2.4 A Simple Example: Single Sample Computations

- Audio and control signals throughout STK use a floating-point data type, `StkFloat`, the exact precision of which can be controlled via a typedef statement in the file `Stk.h` (the default is a *double*).
- Most STK classes and instruments expect input/output values in the range +/-1.0.
- The following program will generate 20 random floating-point (StkFloat) values in the range -1.0 to +1.0:

```
#include "Noise.h"
using namespace stk;

int main()
{
  StkFloat output;
  Noise noise;

  for ( unsigned int i=0; i<20; i++ ) {
    output = noise.tick();
    std::cout << "i = " << i << " : output = " << output << std::endl;
  }

  return 0;
}
```

2.5 A Simple Example: Vectorized Computations

- The `StkFrames` class provides a general "mechanism" for handling and passing vectorized, multi-channel audio data.
- The example from above could be performed as a "vectorized" computation in the following way:

```
#include "Noise.h"
using namespace stk;

int main()
{
  StkFrames output(20, 1);   // initialize StkFrames to 20 1-channel frames
  Noise noise;

  noise.tick( output );   // perform vectorized computation
  for ( unsigned int i=0; i<output.frames(); i++ ) {
    std::cout << "i = " << i << " : output = " << output[i] << std::endl;
  }

  return 0;
}
```

- The `StkFrames` class also provides data interpolation functionality.
- While vectorized operations can be advantageous in some contexts, the following examples will only use single-sample computations.

2.6 "Ticking"

- The sample computations performed by each STK class are generally implemented in a function called `tick()`.
- Most STK unit generators take and/or produce monophonic data. Thus, the `tick()` function normally takes a single sample input value and/or outputs a single sample output value.
- Overloaded versions of the `tick()` function exist to allow vectorized computations.
- In a few cases, STK classes may input or output multi-channel data (ex., `WvIn`, `WvOut`, and their subclasses).
- In a multi-channel context, the `tick()` functions taking or returning `StkFloat` arguments output average values for all channels or input a single value to all channels.
- The `tick()` functions taking `StkFrames` arguments, on the other hand, expect multi-channel input/output data in the form of an `StkFrames` variable.

2.7 Audio File Input/Output

- The `FileWvIn` and `FileWvOut` classes can be used to read or write data from/to a soundfile.
- Files of type RAW, WAV, AIFF, SND (AU), and MAT (Matlab) containing 8-, 16-, and 32-bit integer, as well as 32- and 64-bit floating-point, data are supported.
- The `SineWave` class provides sinusoidal oscillator functionality. `SineWave`'s internal table is statically allocated and shared amongst multiple instances of the class.

```
// sineosc.cpp STK tutorial program

#include "FileWvOut.h"
#include "SineWave.h"
using namespace stk;

int main()
{
  // Set the global sample rate before creating class instances.
  Stk::setSampleRate( 44100.0 );
  int nFrames = 100000;

  SineWave input;
  input.setFrequency( 440.0 );

  FileWvOut output;
  try {
    // Open a 16-bit, one-channel WAV formatted output file
    output.openFile( "hellosine.wav", 1, FileWrite::FILE_WAV, Stk::STK_SINT16 );
  }
  catch ( StkError & ) {
    exit(0);
  }

  // Single-sample computations
  for ( int i=0; i<nFrames; i++ ) {
    try {
      output.tick( input.tick() );
    }
    catch ( StkError & ) {
      exit( 1 );
    }
  }

  return 0;
}
```

- The classes responsible for soundfile support are `FileRead` and `FileWrite`.
- The `FileRead` and `FileWrite` classes offer an efficient means for querying audio file parameters because they do not read/write data during instantiation. Further, they never allocate internal memory. Rather, file data is read/written directly to/from a user-provided `StkFrames` object from/to the file.
- The `FileWvIn` and `FileWvOut` classes provide a "tick-level" interface to the `FileRead` and `FileWrite` classes.
- `FileWvIn` provides a variable read rate (including negative rates) using linear interpolation.
- The class `FileLoop` is a subclass of `FileWvIn` and provides looping functionality for any of the supported soundfile types mentioned above.
- When opening a file, the `FileWvIn` class automatically adjusts the data "read" rate based on the current STK sample rate and the file rate. For example, if the sample rate is currently 44100 Hz and the file rate is 22050, the read rate will be set to 0.5. If you need to set an explicit read rate for a given `FileWvIn` instance, it is safest to change it *after* the file has been opened.

2.8 STK Filters

- STK provides a variety of classes for digital filtering, all inheriting from the `Filter` abstract base class.
- The class `Iir` provides generalized infinite impulse response filtering functionality similar to that of the filter function in Matlab.
- The class `Fir` provides generalized finite impulse response filtering functionality and is slightly more efficient than using the `Iir` class when only a feedforward structure is needed.
- All filter classes in STK inherit from the `Filter` class, including `Fir, Iir, OneZero, OnePole, PoleZero, TwoZero, TwoPole, BiQuad, Delay, ...`
- The `BiQuad` class provides functionality for resonance filtering, as used below.

```
#include "FileWvIn.h"
#include "FileWvOut.h"
#include "BiQuad.h"
using namespace stk;

int main( int argc, char *argv[] )
{
  FileWvIn  input;
  FileWvOut output;

  if ( argc != 2 ) {
    std::cout << "usage: " << argv[0] << " file" << std::endl;
    std::cout << "    where 'file' is an input soundfile to process" << std::endl;
    exit(0);
  }

  try {
    // Load the input file
    input.openFile( argv[1] );
  }
  catch ( StkError & ) {
    exit(0);
  }

  // Set the global STK sample rate to the input file sample rate.
  Stk::setSampleRate( input.getFileRate() );

  // Reset the input file reader increment to 1.0
  input.setRate( 1.0 );

  try {
    // Define and open a 16-bit, one-channel WAV formatted output file
    output.openFile( "filter2", 2, FileWrite::FILE_AIF, Stk::STK_SINT16 );
  }
  catch ( StkError & ) {
    input.closeFile();
    exit(0);
  }

  BiQuad filter;
  filter.setResonance( 1000.0, 0.999, true );  // set resonance to 1000 Hz, pole radius = 0.999, ar
```

```
  StkFrames frame( 1, 2 );                    // one frame of 2 channels

  int i;
  i = input.getSize();  // in sample frames
  while ( i-- >= 0 ) {

    try { // single frame computations
      frame[0] = input.tick();
      frame[1] = filter.tick( frame[0] );
      output.tick( frame );
    }
    catch ( StkError & ) {
      goto cleanup;
    }
  }

 cleanup:
  input.closeFile();
  output.closeFile();

  return 0;
}
```

2.9 Realtime Audio and MIDI Input/Output

- The STK-independent class `RtAudio` provides realtime audio input and output support for a variety of computer operating systems and APIs, including OS X (CoreAudio, Jack), Linux (OSS, ALSA, and Jack), and Windows (DirectSound and ASIO).
- The STK-independent class `RtMidi` provides realtime MIDI input and output support for a variety of computer operating systems and APIs, including OS X (CoreMIDI), Linux (ALSA sequencer, Jack), and Windows (WinMM).
- The class `RtAudio` provides callback functionality for use in a realtime synthesis context.
- The STK classes `RtWvIn` and `RtWvOut` provide audio input/output functionality in the form of blocking function calls. This works fine for simple audio examples but it is not robust for "consumer-level" applications, especially on systems for which the underlying audio API is based on callbacks (Macintosh OS-X, Linux Jack, and Windows ASIO).

2.9.1 Blocking Functionality

- Below is an example realtime audio output program that uses the `RtWvOut` class in a blocking context.

```
// rtsine.cpp
#include "SineWave.h"
#include "RtWvOut.h"
#include "Envelope.h"
#include "ADSR.h"
#include "FileWvOut.h"

using namespace stk;

int main()
{
```

```
  // Set the global sample rate before creating class instances.
  Stk::setSampleRate( 44100.0 );
  Stk::showWarnings( true );
  int nFrames = 100000;
  int releaseCount = (int) (0.9 * nFrames);
  float rampTime = (nFrames - releaseCount) / Stk::sampleRate();

  try {

    SineWave sine;
    RtWvOut dac( 1 );  // Define and open the default realtime output device for one-channel playba

    /* If you wanted to also create an output soundfile (also uncomment output.tick() below)
       FileWvOut output;
       output.openFile( "rttest", 1, FileWrite::FILE_WAV, Stk::STK_SINT16 );
    */

    /* Use ADSR */
    ADSR env;
    env.keyOn();
    env.setAllTimes( rampTime, rampTime, 0.7, rampTime ); // Attack time, decay time, sustain level


    /* Or use a linear line segment envelope (and comment-out the ADSR lines above)
       Envelope env;
       env.keyOn();
       env.setTime( rampTime ); // Attack and release time
    */

    sine.setFrequency( 440.0 );

    // Single-sample computations
    StkFloat temp;
    for ( int i=0; i<nFrames; i++ ) {
      temp = env.tick() * sine.tick();
      dac.tick( temp );
      //output.tick( temp );
      if ( i == releaseCount ) env.keyOff();
    }
  }
  catch ( StkError & ) {
    exit( 1 );
  }

  return 0;
}
```

- This example can be compiled on a Macintosh OS-X system with the following syntax, assuming the file `rtsine.cpp` is in the working directory as described above:

```
g++ -std=c++11 -Istk/include/ -Lstk/src/ -D__MACOSX_CORE__ rtsine.cpp -lstk -lpthread -framework Co
```

- This example can be compiled on a Windows system (using MinGW and MSYS) with the following syntax, assuming the file `rtsine.cpp` is in the working directory as described above:

```
g++ -std=c++11 -o rtsine -Istk/include/ -Lstk/src/ -D__WINDOWS_DS__ rtsine.cpp -lstk -lpthread -lds
```

2.9.2 Callback Functionality

- Below are two examples of realtime audio output using the `RtAudio` class in a callback context.

```
// rtex1.cpp
//
// Realtime audio output example using callback functionality.

#include "RtAudio.h"
#include <iostream>
#include <cmath>

struct CallbackData {
  double phase;
  double phaseIncrement;

  // Default constructor.
  CallbackData()
    : phase(0.0), phaseIncrement(0.0) {}
};

int sin( void *outputBuffer, void *, unsigned int nBufferFrames,
         double, RtAudioStreamStatus, void *dataPointer )
{
  // Cast the buffer and data pointer to the correct data type.
  double *my_data = (double *) outputBuffer;
  CallbackData *data = (CallbackData *) dataPointer;

  // We know we only have 1 sample per frame here.
  for ( int i=0; i<nBufferFrames; i++ ) {
    my_data[i] = 0.8 * std::sin( data->phase );
    data->phase += data->phaseIncrement;
    if ( data->phase > 2 * M_PI ) data->phase -= 2 * M_PI;
  }

  return 0;
}

int main()
{
  unsigned int nBufferFrames = 256;  // 256 sample frames
  unsigned int sampleRate = 44100;
  unsigned int nChannels = 1;
  double frequency = 440.0;
  CallbackData data;
  RtAudio dac;

  // Setup sinusoidal parameter for callback
  data.phaseIncrement = 2 * M_PI * frequency / sampleRate;

  // Open the default realtime output device.
  RtAudio::StreamParameters parameters;
  parameters.deviceId = dac.getDefaultOutputDevice();
  parameters.nChannels = nChannels;
```

```
  try {
    dac.openStream( &parameters, NULL, RTAUDIO_FLOAT64, sampleRate, &nBufferFrames, &sin, (void *)&
  }
  catch ( RtAudioError &error ) {
    error.printMessage();
    exit( EXIT_FAILURE );
  }

  try {
    dac.startStream();
  }
  catch ( RtAudioError &error ) {
    error.printMessage();
    exit( EXIT_FAILURE );
  }

  char input;
  std::cin.get( input ); // block until user hits return

  // Stop the stream.
  try {
    dac.stopStream();
  }
  catch ( RtAudioError &error ) {
    error.printMessage();
  }

  return 0;
}
```

- This example can be compiled on a Macintosh OS-X system with the following syntax:

```
g++ -std=c++11 -Istk/include/ -Lstk/src/ -D__MACOSX_CORE__ rtex1.cpp -lstk -lpthread -framework Cor
```

- The previous example can be easily modified to take realtime input from a soundcard, pass it through a comb filter, and send it back out to the soundcard as follows:

```
// rtex2.cpp
//
// Realtime audio input/output example with comb filter using callback functionality.
#include "RtAudio.h"
#include <iostream>
#include "Delay.h"
using namespace stk;

int comb( void *outputBuffer, void *inputBuffer, unsigned int nBufferFrames,
          double, RtAudioStreamStatus, void *dataPointer )
{
  // Cast the buffers to the correct data type.
  double *idata = (double *) inputBuffer;
  double *odata = (double *) outputBuffer;
  Delay *delay = (Delay *) dataPointer;

  // We know we only have 1 sample per frame here.
```

```
  for ( int i=0; i<nBufferFrames; i++ ) {
    odata[i] = idata[i] + 0.99 * delay->lastOut(); // feedback comb
//    odata[i] = idata[i] + delay->tick( idata[i] ); // feedforward comb
    odata[i] *= 0.45;
    delay->tick( odata[i] );  // feedback comb
  }

  return 0;
}

int main()
{
  unsigned int nBufferFrames = 256;  // 256 sample frames
  unsigned int sampleRate = 48000;
  unsigned int nChannels = 1;
  RtAudio adac;

  Delay delay( 10000, 10000 );

  // Open the default realtime output device.
  RtAudio::StreamParameters oParameters, iParameters;
  oParameters.deviceId = adac.getDefaultOutputDevice();
  iParameters.deviceId = adac.getDefaultInputDevice();
  oParameters.nChannels = nChannels;
  iParameters.nChannels = nChannels;
  try {
    adac.openStream( &oParameters, &iParameters, RTAUDIO_FLOAT64, sampleRate, &nBufferFrames, &comb
  }
  catch ( RtAudioError &error ) {
    error.printMessage();
    exit( EXIT_FAILURE );
  }

  try {
    adac.startStream();
  }
  catch ( RtAudioError &error ) {
    error.printMessage();
    exit( EXIT_FAILURE );
  }

  char input;
  std::cin.get( input ); // block until user hits return

  // Stop the stream.
  try {
    adac.stopStream();
  }
  catch ( RtAudioError &error ) {
    error.printMessage();
  }

  return 0;
}
```