

# 21M.385 Lecture Notes

---

## Lecture 2

### Audio files / Wave Files

- Wave files are recordings of audio data, stored as a linear set of samples (known as *PCM* – pulse code modulation). The Wave file header describes attributes such as:
  - Bit depth
  - Sampling rate
  - Number of channels
- Wave file data is the actual audio samples, generally uncompressed.
- Most audio files are stereo – storing 2 channels of audio (left/right).
  - This data is stored as *interleaved* samples.
  - A *frame* is a multi-channel sample. So, a one second stereo wave file, sampled at 44.1kHz has 44,100 *frames*, and 88,200 *samples*.
- MP3 files are compressed (usually about 10:1) using a lossy compression scheme. We can use conversion tools (like Audacity) to recreate PCM data from compressed audio.
- Most wave files represent CD-quality audio: 16 bit, 44.1Khz, stereo.

### Example: streaming

- To play stereo wave files, we have to configure Audio as stereo (last week was mono).
- Therefore, NoteGenerator must now provide interleaved audio data.
- We can play wave files as a stream or by preloading a buffer.
- WaveFileGenerator uses the built-in python module wave to read a wave file's header data and stream the samples themselves. Nice!
- Some notes:
  - wave's interface is in terms of frames, not samples
  - must convert from 16bit data to floating point data
  - pay attention to end of file condition
- Note how we deal with the *end condition*. If we ask wave for N frames of data, but we get back less than N, we know we have hit the end of the file.
- Press 'p' to play wave file. Note bug if 'p' is pressed twice.





### Playing snippets

- Playing a wave file from start to end is boring!
- Instead, let's identify snippets of audio and use those to create new kinds of music and interactions.
- Refactor WaveFileGenerator into two classes: WaveFile (knows how to read wave data) and WaveGenerator (asks WaveFile for the right window of data at the right time).
- Create a new class: WaveBuffer. It is similar to WaveFile in that it provides wave audio data. But, it keeps an in-memory copy of the data instead of streaming that data from disk.
- WaveBuffer and WaveFile are both classes that implement a WaveSource interface:

```
class WaveSource:
    def get_frames(self, start_frame, end_frame)
    def get_num_channels(self)
```
- This refactoring is a common thing to do when developing software systems. We break up the functionality of WaveFileGenerator into two pieces: an audio data provider, and an audio playback location manager. We can now supply an alternative method of serving up audio data and reuse the same playback system.
- Reasons to use audio streaming vs. in-memory audio buffer:

- Streaming: fixed memory usage for unlimited audio size. But assumes just one access pointer, or else HDD thrashing occurs.
- In-memory: fast access. No HDD constraints. But uses more memory.

### Creating Regions using Sonic Visualizer

- Get Sonic Visualizer from <http://www.sonicvisualiser.org>
- Open a Wave File
- Add a new regions layer (Layer->Add New Region Layer). There should now be 4 layers: global scroll/zoom control, time grid, waveform, regions: 
- Use up/down arrows or mouse wheel to zoom in/out
- Use the arrow tool  to select a region. You can use shortcut keys 1,2,3,4,5 to quickly pick a tool. Careful to get beginning and ending just right.
- Enable “constrain playback” and/or “loop playback”   to hear your selection.
- Once you like your regions, Edit->Insert Item at Selection. This will create a new region.
- Use Layer ->Edit Layer Data to edit regions. Make sure to give each region a unique name (label).
- You can save your session. It will make a .sv file.
- After you made your regions, File->Export Annotation Layer. Make sure to export as a text file.

### Reading the annotation layer with python

- Open a file using `open()`. Use `readlines()` to grab all the lines.
- For each line, you can either use the regular expressions module (`re`) or, look at the string functions for splitting strings. See `split()`. Sonic Visualizer outputs tab-delimited files.
- Remember that the data in the regions annotation files is in units of seconds.

### Looping

- An audio region can be looped if the start/end points are chosen correctly. Loop points can be
  - Butt-spliced
  - Cross-faded
- So far, we’ve triggered sounds by “fire and forget”. Each sound had a *predetermined* duration
- Looping audio will have infinite duration, so we must allow for a user-determined stop function (sometimes called `release()`).
- Careful about looping with respect to buffer filling. Do the math correctly to avoid gaps or popping. There will be a buffer at some point that must be filled from the end AND the beginning.

### Changing speed

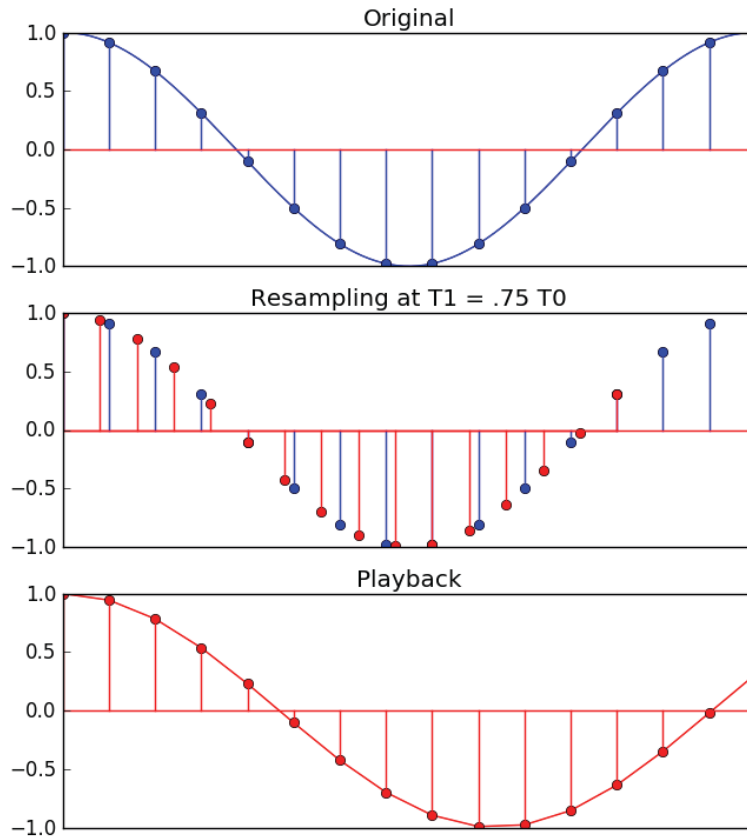
- Create a `SpeedModulator` - an audio generator that receives audio data and provides audio data. However, the number of input frames consumed is different from the output frames provided.
- For example, if a `SpeedModulator` consumes 300 samples, but provides 400 samples, the resultant audio will be slower by the ratio 3:4. See Figure below.
- In order to convert X audio samples into Y audio samples, we must *resample* the data.
- There are a few different resampling techniques: nearest neighbor, sample and hold, or interpolation. Interpolation can be linear or more complex (like cubic). Linear is good for our needs.

### Linear Interpolation

- In python, the function `np.interp()` is useful for interpolation of a time series. Read the numpy docs to remember how this function works.
- Careful about interpolation in stereo. You must split the channels, interpolate, and then re-interleave.

### Recording Audio

- We can use pyAudio for recording microphone data into a buffer as well.
- Similar to writing audio: on every `update()`, ask for number of frames available without blocking. Then grab the frames. Finally, convert from string to numpy array.
- We can playback the bits of recorded audio by creating another WaveSource object: `ArrayWave` – very similar to `WaveBuffer`.



A buffer of 16 samples needs to be filled. The desired modified speed ratio is 3:4.  
Use only 12 samples of input to create 16 samples of output.  
The resultant audio speed is changed by a factor of 3:4.

MIT OpenCourseWare  
<https://ocw.mit.edu>

**21M.385 Interactive Music Systems**  
Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.