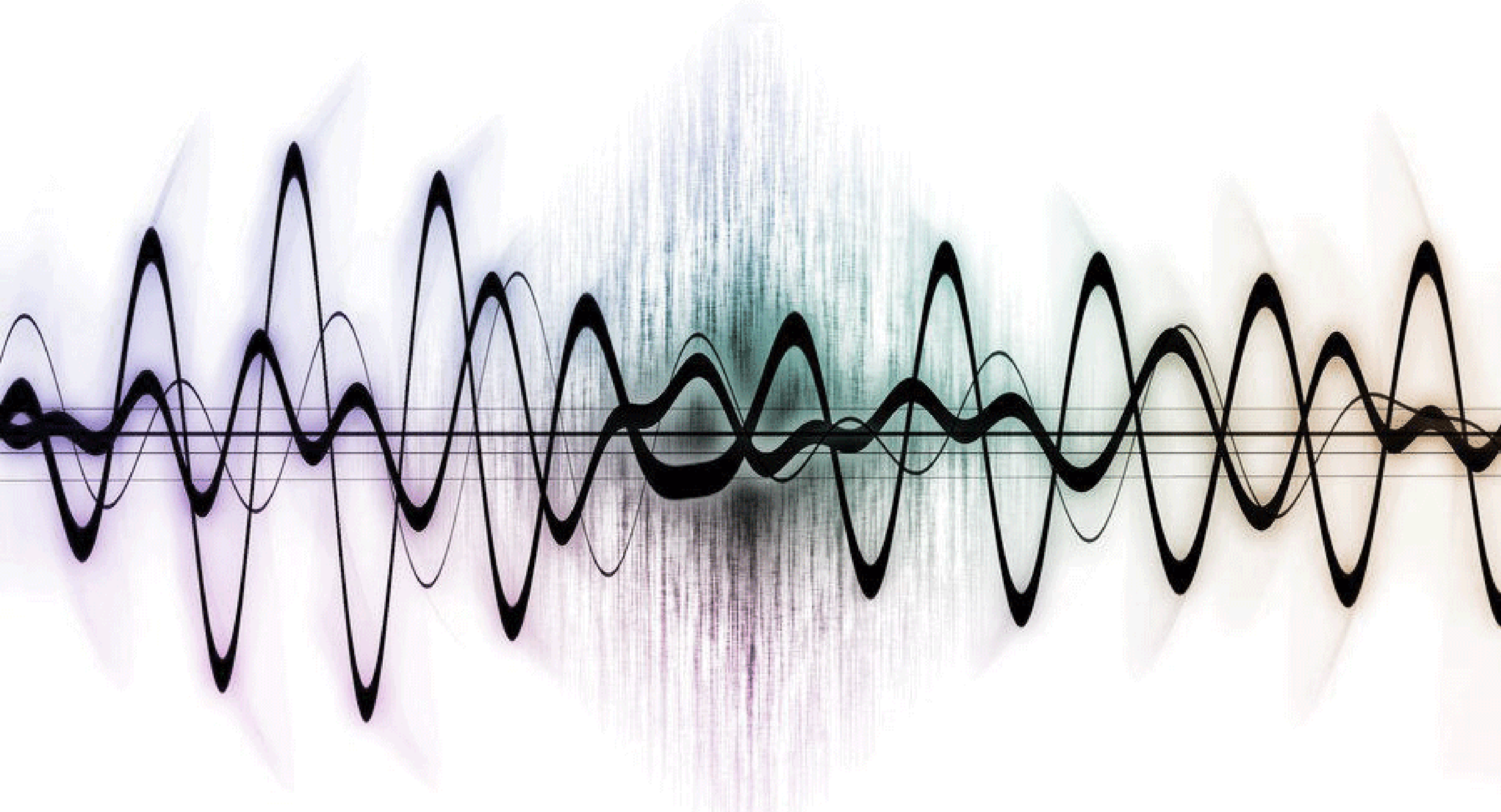
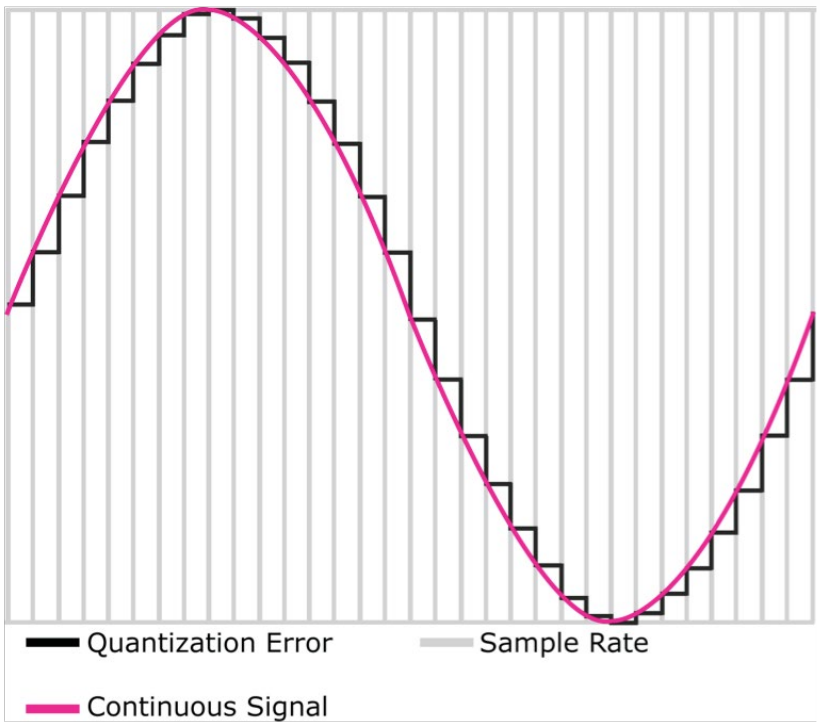


Tinkering Audio III: Construction of Melodies

Creative Computing: Tinkering – Lecture 10 – Michael Scott

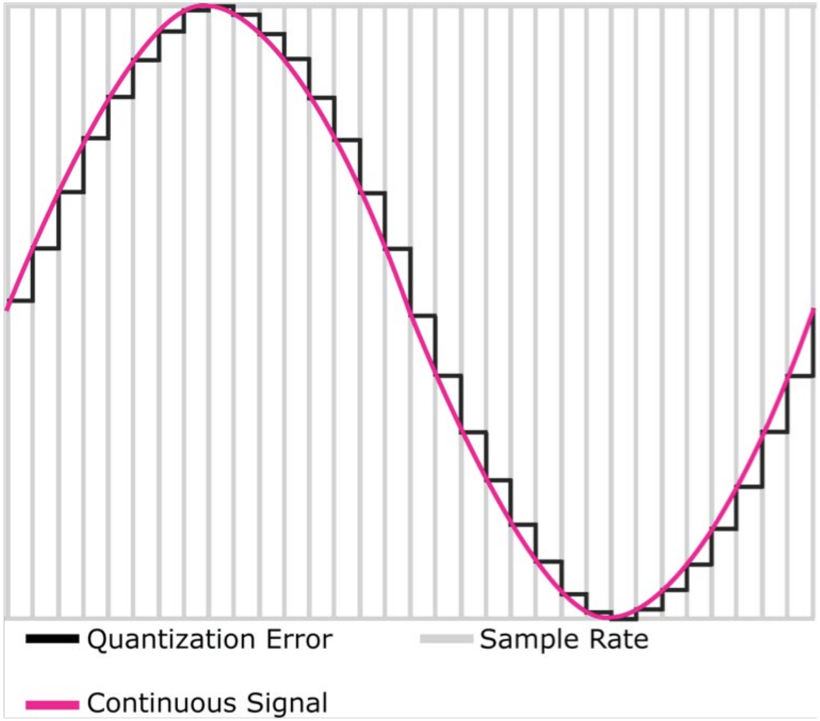
Recap on Last Week

- Bit-depth is important for representing displacement
  - Amplitude is displacement at a certain point (usually the crest and the trough)
  - Available range of volume often depends on the available hardware
  - Distance between two levels, however, depends on how bits are allocated



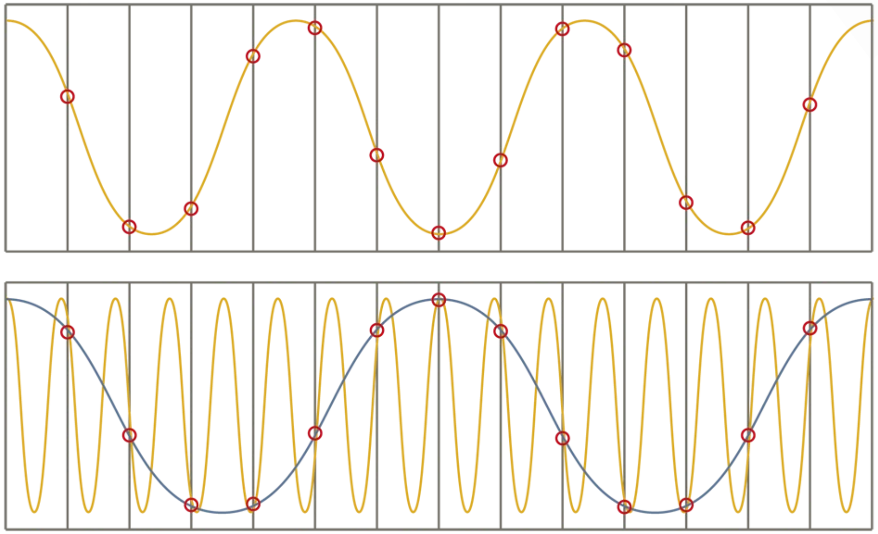
Recap on Last Week

- Bit-depth is important for representing displacement
  - Error arises when a displacement cannot be represented
  - Clipping occurs when maximum volume is exceeded, but this can be avoided with a sufficient bit-depth and normalization procedure



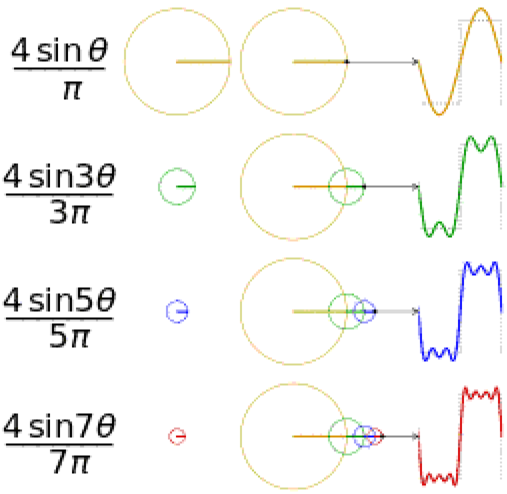
Recap on Last Week

- Sample rate is important for representing frequency
  - Aliasing occurs when sample rate is too low to represent the intended frequency
  - The wrong frequency is produced
  - Use Nyquist Theorem to determine the sample rate needed for a particular frequency



Recap on Last Week

- Signals can be combined through simple summation to create more complex signals
  - These sin() signals approximate a square wave

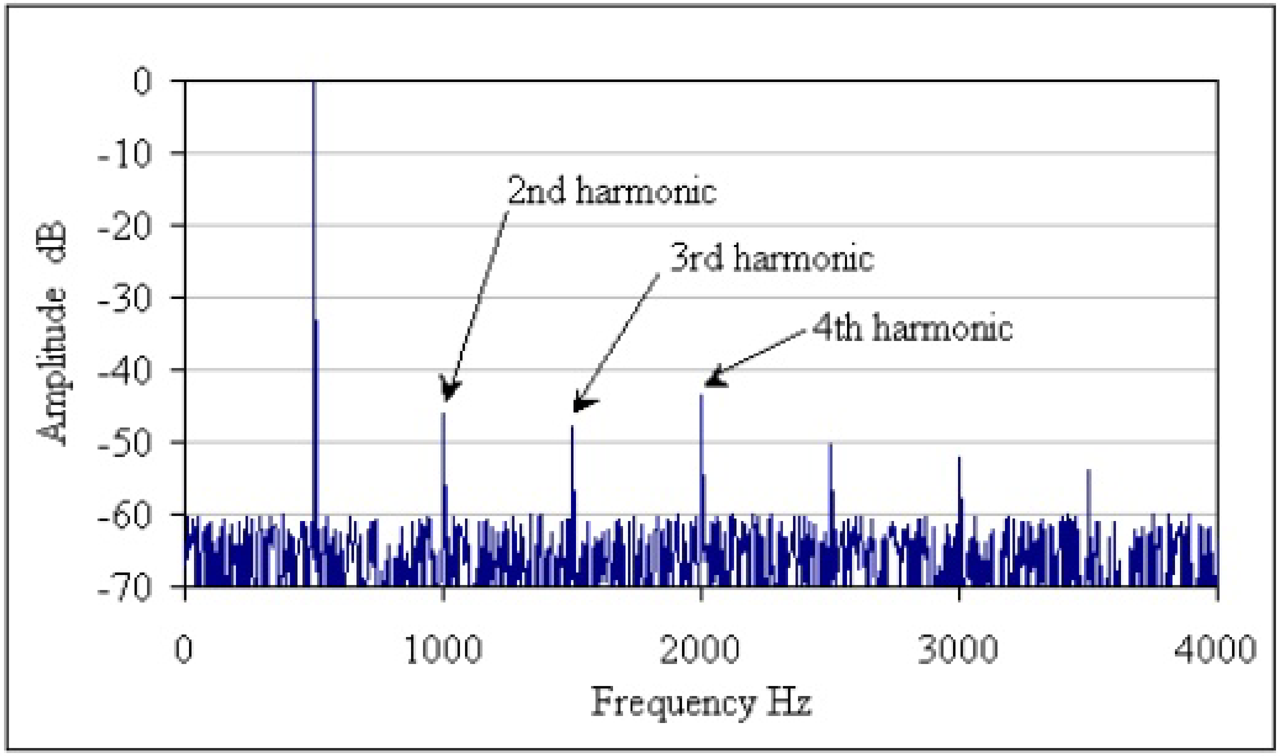


Recap on Last Week

- We can mix tones
  - We even know how to manipulate the volumes of the two sounds to adjust for overflow
  - We know how to normalize the tone in order to preserve the overall amplitude without clipping

- We can add sine (or other) waves together to create kinds of instruments / sounds that do not physically exist, but which sound interesting and complex

Recap on Last Week



The Harmonics of 500 Hz

Learning Objectives

By the end of this session, you will be able to:

- **Recognise** different forms of audio synthesis
- **Explain how** to implement ways to modify sounds including echoes, envelopes, and resampling
- **Explore** the parallel between sampling and scaling sound with sampling and scaling images
- **Write** a basic function to generate musical notes according to their frequency, amplitude, and ADSR envelope
- **Integrate** randomness using the random library

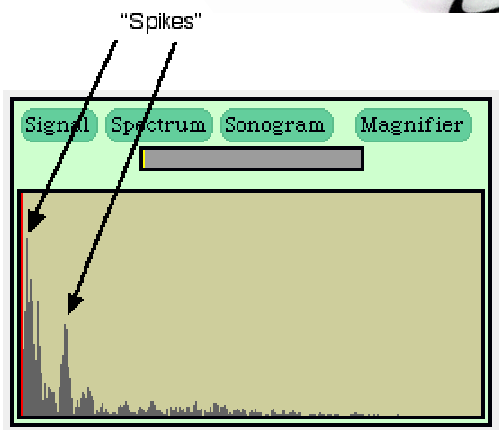
Tinkering Audio

SYNTHESISERS

Synthesizers

- There are several types of sound synthesis:
  - **Additive**: works by adding together several sine waves or other waves
  - **Subtractive**: works by generating harmonically rich waves and then filtering them to shape their frequency content
  - **Frequency modulation (FM)**: works by rapidly altering the frequency of a generated tone
  - **Physical modelling**: works by simulating the vibrations in real-world objects such as strings, wind instruments, even human vocal cords
  - **Sampling synthesizer:** works by modifying a pre-recorded sound to adjust the frequency and amplitude as well as add effects like echoes

Adding sine waves to make something completely new



- We saw earlier that complex sounds (like the sound of your voice or a trumpet) can be seen as being a sum of sine waves
  – Any sound can be made by summing sine waves – Fourier's Theorem
- We can *create* complex sounds by summing sine waves
- These are sounds made by mathematics, by invention, not based on anything in nature

Exemplar Square Wave

```
def square_wave(frequency, position):
    if math.sin(2 * PI * frequency * (position / SAMPLE_RATE)) > 0:
        return 1
    else:
        return -1
```

Adding Square Waves Together

```
marker = 0
def generate_tone(time_in_seconds, *freq):
    global marker
    audio_data = []
    for i in range(marker, marker + int(time_in_seconds * SAMPLE_RATE)):
        value = 0
        for j in range(0, len(freq)):
            value += square_wave(freq[j], i)
            value *= (MAX_VALUE / len(freq)) * VOLUME
            audio_data.append(value)
    marker += int(time_in_seconds * SAMPLE_RATE)
    return audio_data
```

Adding Square Waves Together

What's this?

```
marker = 0
def generate_tone(time_in_seconds, *freq):
    global marker
    audio_data = []
    for i in range(marker, marker + int(time_in_seconds * SAMPLE_RATE)):
        value = 0
        for j in range(0, len(freq)):
            value += square_wave(freq[j], i)
            value *= (MAX_VALUE / len(freq)) * VOLUME
            audio_data.append(value)
    marker += int(time_in_seconds * SAMPLE_RATE)
    return audio_data
```

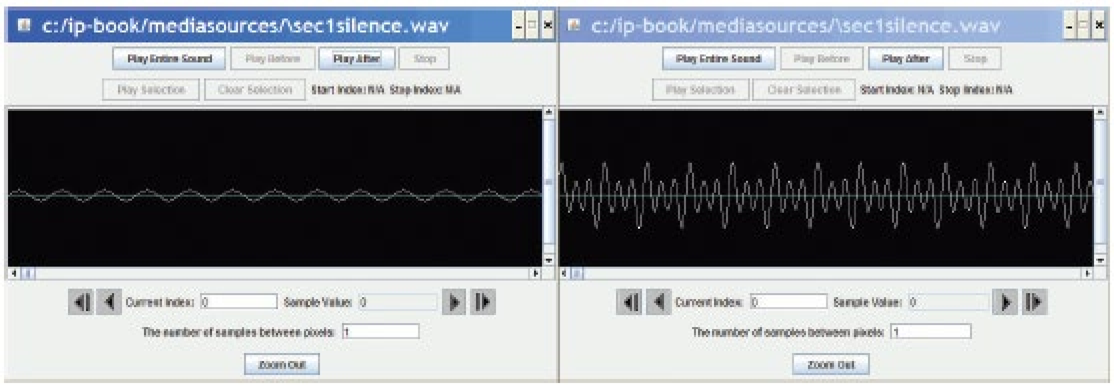
Adding Square Waves Together

```
marker = 0
def generate_tone(time_in_seconds, *freq):
    global marker
    audio_data = []
    for i in range(marker, marker + int(time_in_seconds * SAMPLE_RATE)):
        value = 0
        for j in range(0, len(freq)):
            value += square_wave(freq[j], i)
            value *= (MAX_VALUE / len(freq)) * VOLUME
            audio_data.append(value)
    marker += int(time_in_seconds * SAMPLE_RATE)
    return audio_data
```

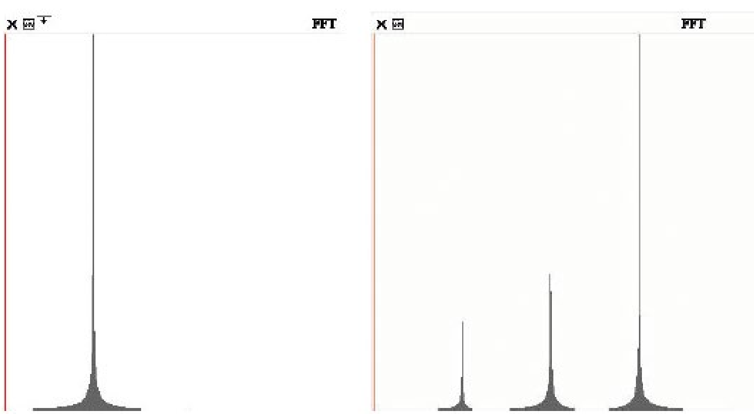
Why?

Comparing the waves

- Left, 440 Hz; Right, combined wave.



In Explorer



440, 880, 1320

In the Spectrum view in MediaTools

Tinkering Audio

ECHOES

A Function for Adding Two Sounds

```
def addSoundInto(sound1, sound2):
  for sampleNmr in range(0, getLength(sound1)):
    sample1 = getSampleValueAt(sound1, sampleNmr)
    sample2 = getSampleValueAt(sound2, sampleNmr)
    setSampleValueAt(sound2, sampleNmr, sample1 + sample2)
```

Notice that this adds sound1 and sound2 by adding sound1 ***into*** sound2

Adding Sounds with a Delay

```
def makeChord(sound1, sound2, sound3):
  for index in range(0, getLength(sound1)):
    s1Sample = getSampleValueAt(sound1, index)
    setSampleValueAt(sound1, index, s1Sample )
    if index > 1000:
      s2Sample = getSampleValueAt(sound2, index - 1000)
      setSampleValueAt(sound1, index, s1Sample + s2Sample)
    if index > 2000:
      s3Sample = getSampleValueAt(sound3, index - 2000)
      setSampleValueAt(sound1, index, s1Sample + s2Sample + s3Sample)
```

-Add in sound2 after 1000 samples

-Add in sound3 after 2000 samples

Note that in this version we're adding directly into sound1!

Creating an Echo

```
def echo(sndFile, delay):
  s1 = makeSound(sndFile)
  s2 = makeSound(sndFile)
  for index in range(delay, getLength(s1)):
    echo = 0.6*getSampleValueAt(s2, index-delay)
    combo = getSampleValueAt(s1, index) + echo
    setSampleValueAt(s1, index, combo)
  play(s1)
  return s1
```

This creates a delayed echo sound, multiplies it by 0.6 to make it fainter and then adds it into the original sound.

How the Echo Works

Top row is the samples of our sound. We're adding it to itself, but delayed a few samples, and multiplied to make it softer.

	100	200	1000	-150	-350	200	500	10	-500	-1000	-350	25	-10	1000	...		
Delay				100	200	1000	-150	-350	200	500	10	-500	-1000	-350	25	-10	1
* 0.6				60	120	600	-90	-210	120	300	6	-300	-600	-210	15	-6	6
+	100	200	1000	-90	-230	800	410	-200	-380	290	-344	-275	-610	890	...		

Tinkering Audio

RESAMPLING

How sampling keyboards work

- They have a huge memory with recordings of lots of different instruments played at different notes
- When you press a key on the keyboard, the recording closest to the note you just pressed is selected, and then the recording is shifted to exactly the note you requested.
- The shifting is a generalization of algorithms of doubling and halving frequency (as follows)

Doubling the frequency

Why +1 here?

```
def double(source):
  len = getLength(source) / 2 + 1
  target = makeEmptySound(len)
  targetIndex = 0
  for sourceIndex in range(0, getLength( source), 2):
    value = getSampleValueAt( source, sourceIndex)
    setSampleValueAt( target, targetIndex, value)
    targetIndex = targetIndex + 1
  play(target)
  return target
```

Here's the piece that does the doubling

How doubling works

sourceIndex = 0, 2, 4, 6, 8, 10, ...

100	200	1000	-150	-350	200	500	10	-500	-1000	-350	25	-10	1000	...

100	1000	-350	500	-500	-350	-10	...

targetIndex = 0, 1, 2, 3, 4, 5, ...

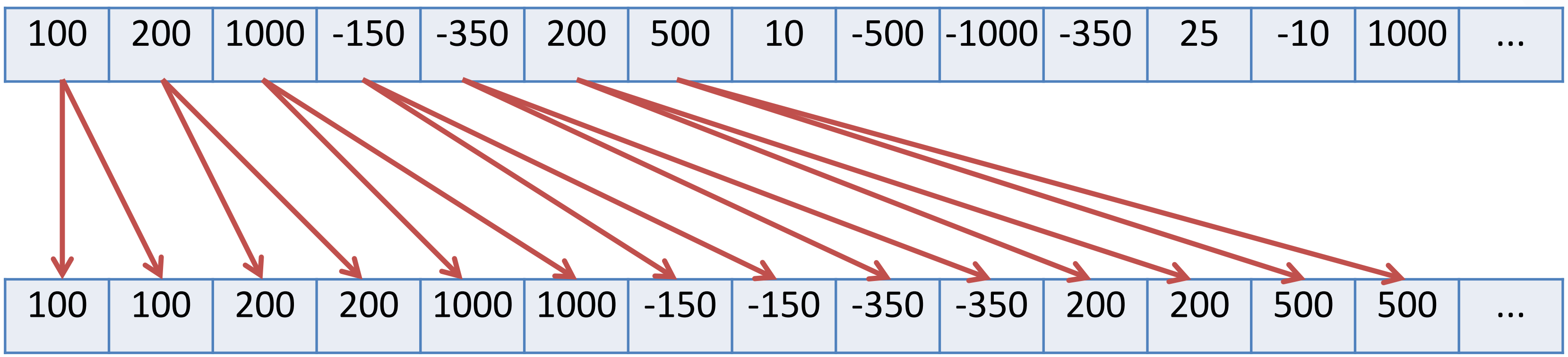
Halving the frequency

```
def half(source):
  target = makeEmptySound(getLength(source) * 2)
  sourceIndex = 0
  for targetIndex in range(0, getLength( target)):
    value = getSampleValueAt( source, int(sourceIndex))
    setSampleValueAt( target, targetIndex, value)
    sourceIndex = sourceIndex + 0.5
  play(target)
  return target
```

Here's the piece that does the halving

How halving works

sourceIndex = 0, 0.5, 1, 1.5, 2, 2.5, 3, ...
int(sourceIndex) = 0, 0, 1, 1, 2, 2, 3, 3, ...



targetIndex = 0, 1, 2, 3, 4, 5, ...

Can we generalize shifting a sound into other frequencies?

```
def shift(source, factor):
  target = makeEmptySound(getLength(source))
  sourceIndex = 0

  for targetIndex in range(0, getLength( target)):
    value = getSampleValueAt(source, int(sourceIndex))
    setSampleValueAt( target, targetIndex, value)
    sourceIndex = sourceIndex + factor

  play(target)
  return target
```

Why doesn’t it work?

- It works for shifting down, but not for shifting up

```
>>> hello = makeSound("helloworld.wav")
>>> lowerHello = shift(hello, 0.6)
>>> higherHello = shift(hello, 1.5)
You are trying to access the sample at index: 44034, but the last valid index
is at 44032
The error value is:
Inappropriate argument value (of correct type).
An error occurred attempting to pass an argument to a function.
 in file C:\Program Files
(x86)\JES\jes\python\jes\core\interpreter\__init__.py, on line 157, in
function run
 in file C:\Program Files
(x86)\JES\jes\python\jes\core\interpreter\__init__.py, on line 202, in
function execute
 in file <input>, on line 1, in function <module>
 in file C:\Users\Ed\Desktop\jes\comp120_8_07_shift_broken.py, on line 6, in
function shift
 in file C:\Program Files (x86)\JES\jes\python\media.py, on line 346, in
function getSampleValueAt
ValueError:
Please check line 6 of C:\Users\Ed\Desktop\jes\comp120_8_07_shift_broken.py
>>>
```

Three ways of fixing it: 1

```
def shift(source, factor):
    target = makeEmptySound(getLength(source))
    sourceIndex = 0

    for targetIndex in range(0, getLength( target)):
        value = getSampleValueAt( source, int(sourceIndex))
        setSampleValueAt( target, targetIndex, value)
        sourceIndex = sourceIndex + factor
        if sourceIndex > getLength(source):
            sourceIndex = 0

    play(target)
    return target
```

- What would shift(sound, 3) do?

Three ways of fixing it: 2

```
def shift(source, factor):
    target = makeEmptySound(getLength(source))
    sourceIndex = 0

    for targetIndex in range(0, getLength( target)):
        value = getSampleValueAt( source, int(sourceIndex))
        setSampleValueAt( target, targetIndex, value)
        sourceIndex = sourceIndex + factor
        if sourceIndex > getLength(source):
            break

    play(target)
    return target
```

- What would shift(sound, 3) do?

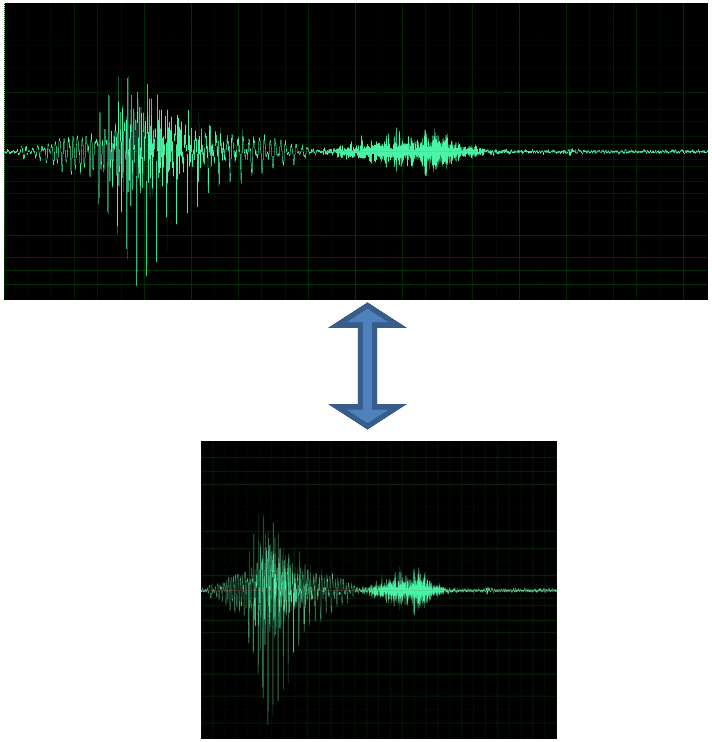
Three ways of fixing it: 3

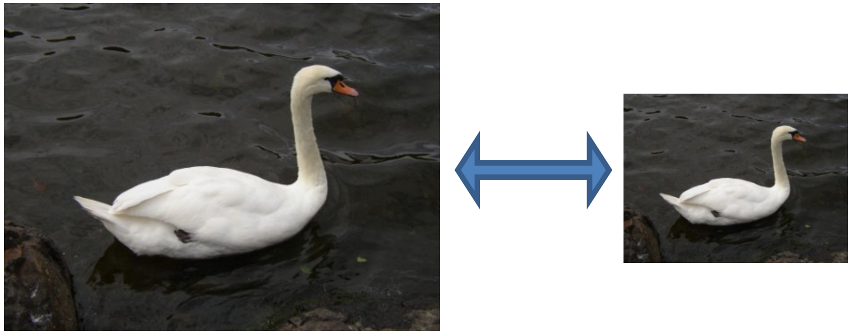
```
def shift(source, factor):
    target = makeEmptySound(int(getLength(source) / factor) + 1)
    sourceIndex = 0

    for targetIndex in range(0, getLength( target)):
        value = getSampleValueAt( source, int(sourceIndex))
        setSampleValueAt( target, targetIndex, value)
        sourceIndex = sourceIndex + factor

    play(target)
    return target
```

- What would shift(sound, 3) do?

Sampling as an Algorithm

- Think about the similarities between:
  - Halving the frequency of a sound
  - Scaling a picture up to twice the size
- Think about the similarities between:
  - Doubling the frequency of a sound
  - Scaling a picture down to half the size





Tinkering Audio

ENVELOPES

Envelopes

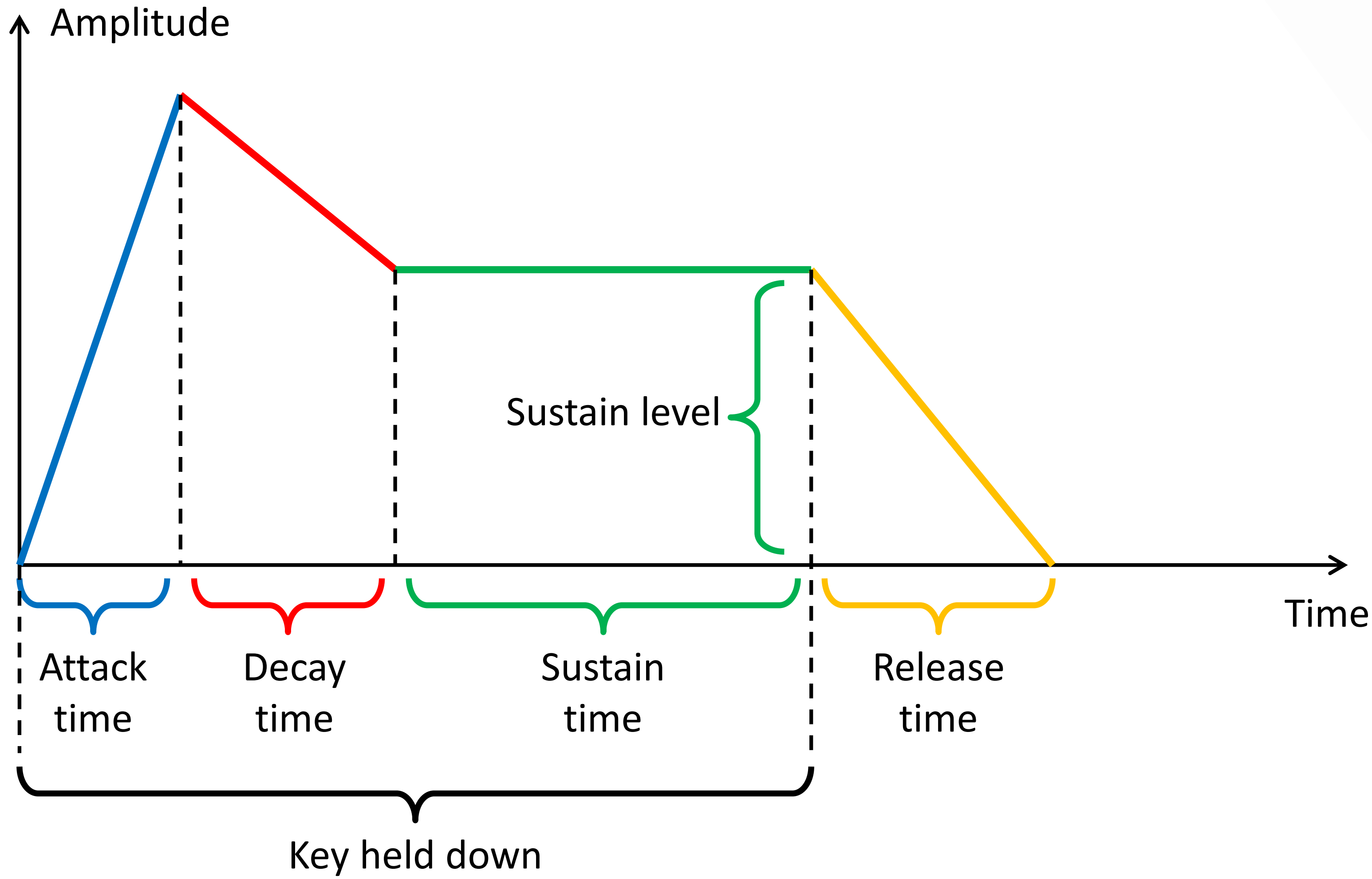
- Static sounds are not very interesting
- Almost all synthesizers allow the musician to manipulate **envelopes**
- Envelopes allow characteristics of the sound (volume, pitch, timbre etc) to vary over time

http://www.animations.physics.unsw.edu.au/jw/timbre-envelope.htm

ADSR envelopes

- The most common type of envelope is the **ADSR envelope**
  - **Attack time**: how quickly the sound reaches peak level when a note is played
  - **Peak level:** the peak amplitude that the note achieves (sometimes called attack level)
  - **Decay time**: how quickly the sound goes from peak amplitude to the sustain level after the initial peak
  - **Sustain level**: the amplitude that the sound maintains until the note stops playing
  - **Sustain time:** how long the sustain level is maintained (this can be a parameter, or can be determined by how long the musician holds the key down)
  - **Release time**: how quickly the note fades away when it stops playing

ADSR Envelope



ADSR Envelope

```
def adsr_envelope(audio_data,
             attack_level,
             attack_time,
             decay_time,
             sustain_level,
             sustain_time,
             release_time):

  number_of_samples = len(audio_data)
  attack_length = int(number_of_samples * attack_time)
  decay_length = int(number_of_samples * decay_time)
  sustain_length = int(number_of_samples * sustain_time)
  release_length = int(number_of_samples * release_time)

  # Attack
  p = 0
  for i in range(0, attack_length):
    audio_data[i] *= lerp(0, attack_level, p / attack_length)
    p += 1

  # Decay
  p = 0
  for i in range(attack_length, attack_length + decay_length):
    audio_data[i] *= lerp(attack_level, sustain_level, p / decay_length)
    p += 1
```

```
# Sustain
  for i in range(
    attack_length + decay_length,
    attack_length + decay_length + sustain_length
  ):
    audio_data[i] *= sustain_level

  # Release
  p = 0
  for i in range(
    attack_length + decay_length + sustain_length,
    attack_length + decay_length + sustain_length + release_length
  ):
    audio_data[i] *= lerp(sustain_level, 0, p / decay_length)
    p += 1

  # Gap
  for i in range(attack_length + decay_length + sustain_length +
release_length, number_of_samples):
    audio_data[i] *= 0  # Needs fixing if this happens

  return audio_data
```

Tinkering Audio

PARSING A NOTATION FOR TONES

Parsing a Notation

Parsing a Notation: ABC Notation

```
X:1
T:The Legacy Jig
M:6/8
L:1/8
R:jig
K:G
GFG BAB | gfg gab | GFG BAB | d2A AFD |
GFG BAB | gfg gab | age edB |1 dBA AFD :|2 dBA ABd |:
 efe edB | dBA ABd | efe edB | gdB ABd |
efe edB | d2d def | gfe edB |1 dBA ABd :|2 dBA AFD |]
```

Parsing a Notation: ABC Notation

- Lines in the first part of the tune notation, beginning with a letter followed by a colon, indicate various aspects of the tune such as:
  - the index, when there are more than one tune in a file (X:),
  - the title (T:),
  - the time signature (M:),
  - the default note length (L:),
  - the type of tune (R:)
  - and the key (K:).
- Lines following the key designation represent the tune. This example can be translated into traditional music notation using one of the abc conversion tools.

Further Notes on ABC

- http://abcnotation.com/examples

A Simpler Parser –
Key Concepts

- Reading notes and note durations from a string

- **Token** – a collection of symbols with a particular meaning (e.g., a note and its duration)

- **Delimiter** – the symbol that separates the tokens
(e.g., a blank space – remember this is still encoded in the computer!)

Useful Python operations for parsing

```
>>> s = "Hello world"
```

Splitting by delimiter

```
>>> s.split(' ')
['Hello', 'world']
```

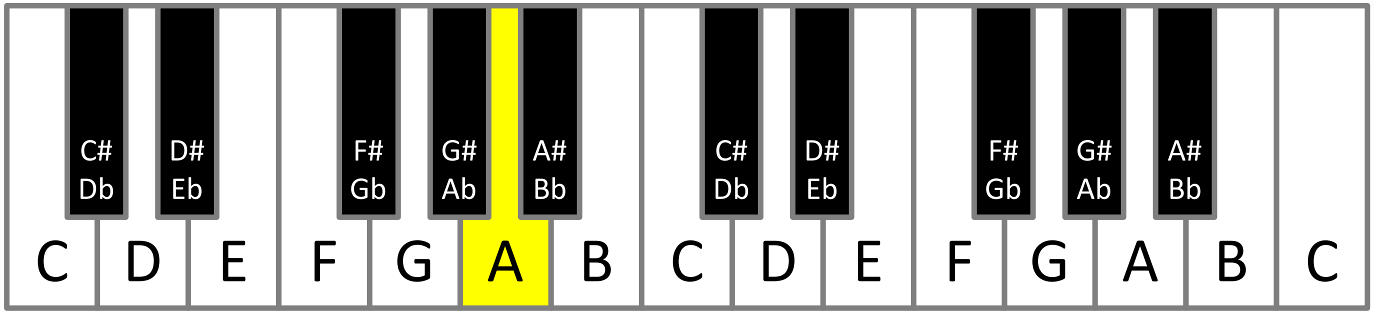
Getting a single character

```
>>> s[4]
'o'
```

Slicing

```
>>> s[3:7]
'lo w'
>>> s[:3]
'Hel'
>>> s[7:]
'orld'
```

Converting notes to frequencies



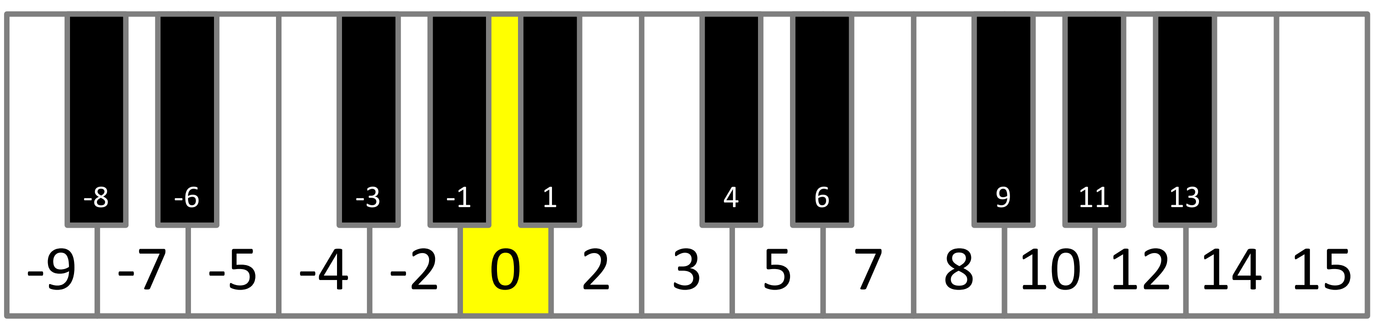
- The A above middle C on a piano is **440Hz**
- Notes **double** in frequency every **octave up**, and **halve** in frequency every **octave down**
  - A = 55Hz, 110Hz, 220Hz, 440Hz, 880Hz, 1760Hz, …
- 1 **octave** = 12 **semitones**
  - A, A#, B, C, C#, D, D#, E, F, F#, G, G#, A

Converting notes to frequencies

- Use this formula: $$f = 440 \times 2^{\frac{n}{12}}$$

```
frequency = 440.0 * 2.0 ** (note_number / 12.0)
```

- note_number is the **offset** of the note, in semitones, from A=440Hz



Randomly Choosing Notes from a List

```
>>> for i in range(1,5):
...         print random.choice([“C1", “D1", “E1",
                “F1", “G1", “A1", “B1",“C2",“D2“, “D3”])
...            print random.randrange(1, 8)
...
C1
4
E1
6
...
```

Tinkering Audio

SPLICING

Splicing

- Splicing gets its name from literally cutting and pasting pieces of magnetic tape together
  - Use generalisable clip and copy functions to construct sounds from other sounds

```
def clip(source, start, end):
  target = makeEmptySound(end - start)
  tIndex = 0
  for sIndex in range(start, end):
    value = getSampleValueAt(source, sIndex)
    setSampleValueAt(target, tIndex, value)
    tIndex = tIndex + 1
  return target
```

```
def copy(source, target, start):
  tIndex = start
  for sIndex in range(0, getLength(source)):
    value = getSampleValueAt(source, sIndex)
    setSampleValueAt(target, tIndex, value)
    tIndex = tIndex + 1
```

Splicing Sounds

- Splicing gets its name from literally cutting and pasting pieces of magnetic tape together
- Doing it digitally is easy (in principle), but painstaking
- The easiest kind of splicing is when the component sounds are in separate files.
- All we need to do is copy each sound, in order, into a target sound.
- Here's a recipe that creates the start of a sentence, “Guzdial is …” (You may complete the sentence.)

Splicing whole sound files

```
def merge():
  guzdial = makeSound(getMediaPath("guzdial.wav"))
  isSound = makeSound(getMediaPath("is.wav"))
  target = makeSound(getMediaPath("sec3silence.wav"))
  index = 0
  for source in range(0, getLength(guzdial)):
    value = getSampleValueAt(guzdial, source)
    setSampleValueAt(target, index, value)
    index = index + 1
  for source in range(0, int(0.1*getSamplingRate(target))):
    setSampleValueAt(target, index, 0)
    index = index + 1
  for source in range(0, getLength(isSound)):
    value = getSampleValueAt(isSound, source)
    setSampleValueAt(target, index, value)
    index = index + 1
  normalize(target)
  play(target)
  return target
```

How it works

- Creates sound objects for the words "Guzdial", "is" and the target silence
- Set target's index to 0, then let each loop increment index and end the loop by leaving index at the next empty sample ready for the next loop
- The 1st loop copies "Guzdial" into the target
- The 2nd loop creates 0.1 seconds of silence
- The 3rd loop copies "is" into the target
- Then we normalize the sound to make it louder

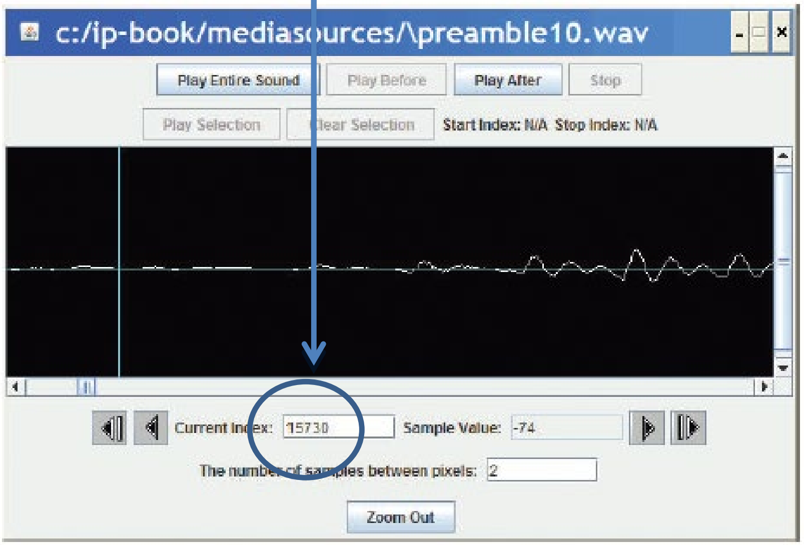
Splicing words into a speech

- Say we want to splice pieces of speech together:
  - We find where the end points of words are
  - We copy the samples into the right places to make the words come out as we want them
  - (We can also change the volume of the words as we move them, to increase or decrease emphasis and make it sound more natural.)

Finding the word end-points

- Using MediaTools and play before/after cursor, we can figure out the index numbers where each word ends
- We want to splice a copy of the word "United" after "We the" so that it says, "We the United People of the United Kingdom".

Word	Ending index
We	15730
the	17407
People	26726
of	32131
the	33413
United	40052
States	55510



The Whole Splice

```
def splicePreamble():
  file = getMediaPath("preamble10.wav")
  source = makeSound(file)
  target = makeSound(file)   # This will be the newly spliced sound
  targetIndex =17408      #  targetIndex starts at just after "We the" in the new sound
  for sourceIndex in range( 33414, 40052):  # Where the word "United" is in the sound
   setSampleValueAt(target, targetIndex,  getSampleValueAt(source, sourceIndex))
   targetIndex = targetIndex + 1
  for sourceIndex in range(17408, 26726):   # Where the word "People" is in the sound
   setSampleValueAt(target , targetIndex, getSampleValueAt(source, sourceIndex))
   targetIndex = targetIndex + 1
  for index in range(0, 1000):                #Stick some quiet space after that
   setSampleValueAt(target, targetIndex, 0)
   targetIndex = targetIndex + 1
  play(target)                                    #Let's hear and return the result
  return target
```

Tinkering Audio

RANDOMNESS

Random

```
>>> import random
>>> for i in range(1,10):
...     print random.random()
...
0.8211369314193928
0.6354266779703246
0.9460060163520159
0.904615696559684
0.33500464463254187
0.08124982126940594
0.0711481376807015
0.7255217307346048
0.2920541211845866
```

Useful random functions

- **random.uniform(a, b):** get a random **floating point** number between a and b
- **random.randrange(a, b):** get a random **integer** between a and b-1
- **random.choice(list):** choose a random element from a list
- **random.shuffle(list):** randomise the order of a list
- **random.seed(x):** can be used to make your program generate the same sequence of random numbers each time it is run
- Many others – see the documentation for details

White noise

```
import random

def makeNoise(amplitude, length):
  # Create a blank sound
  buildNoise = makeEmptySoundBySeconds(length)
  # Make some noise!
  for pos in range(getLength(buildNoise)):
    rawSample = random.uniform(-1, 1)
    sampleVal = int(amplitude * rawSample)
    setSampleValueAt(buildNoise, pos, sampleVal)
  return buildNoise
```

Tinkering Audio Assignment

- *Quality* **over** *quantity*
- **Avoid** feature creep and stick to *about* <u>**seven**</u> algorithms, such as this exemplar:

1st. tone generation ✓
2nd. tone combination ✓
3rd. audio splice and swap ✓
4th. audio envelopes and echoes ✓
5th. parsing tokens into audio ✓
6th. random audio generation ✓

- Counts across your groups, but of course some algorithms are better for sound effects (e.g. resampling) while others for melodies (e.g. parsing tokens into audio)