EECS 1710
Programming for Digital Media

Lecture 16 :: Working with Audio 2

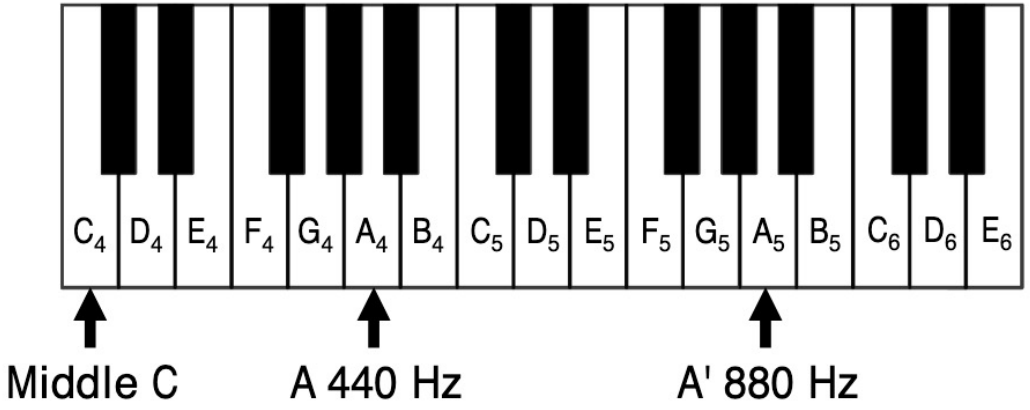
YORK UNIVERSITÉ UNIVERSITY

Equal-Tempered Music Scale

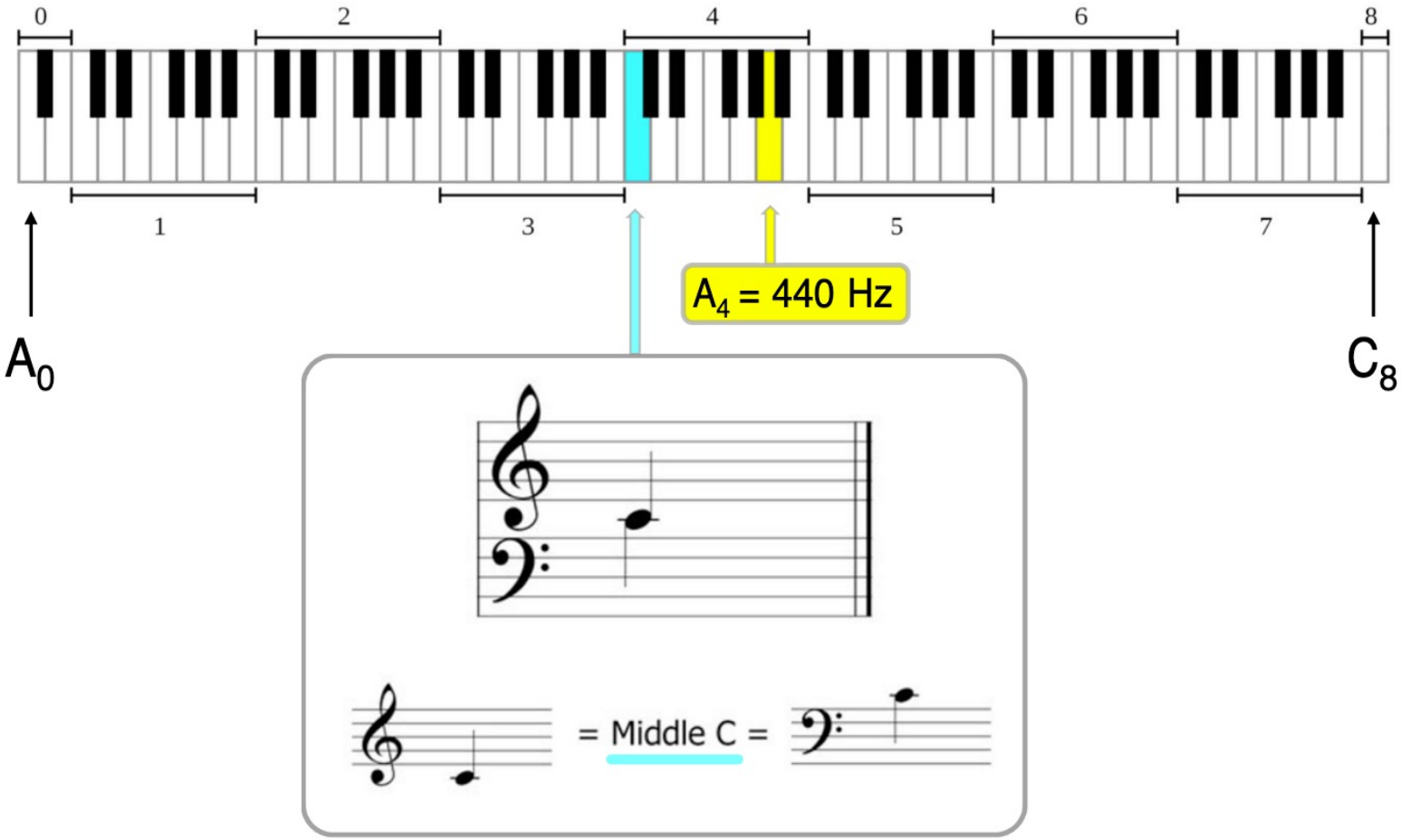
The harmonic system used in most Western music is based on the "equal-tempered scale", as typified by the pattern of white and black keys on a piano. The frequency of each note on a piano keyboard is related to its neighbor by the following formula:

$$f_{n+1} = f_n \times 2^{1/12}$$

where note $n + 1$ is "one semitone" above note n. The factor $2^{1/12}$ ensures that two notes separated by twelve steps in the equal-tempered scale differ in frequency by a factor of $2^{12/12} = 2$. This interval is known as an "octave". To allow musicians to travel and perform with different orchestras in different countries, a standard evolved and was adopted in the early 1900s to ensure instruments were in tune with each other. The standard specifies that "A above middle C" has a frequency of 440 Hz (see below). With this reference point, and with the $2^{1/12}$ frequency factor between adjacent notes, the frequency of any note on any instrument was standardized.

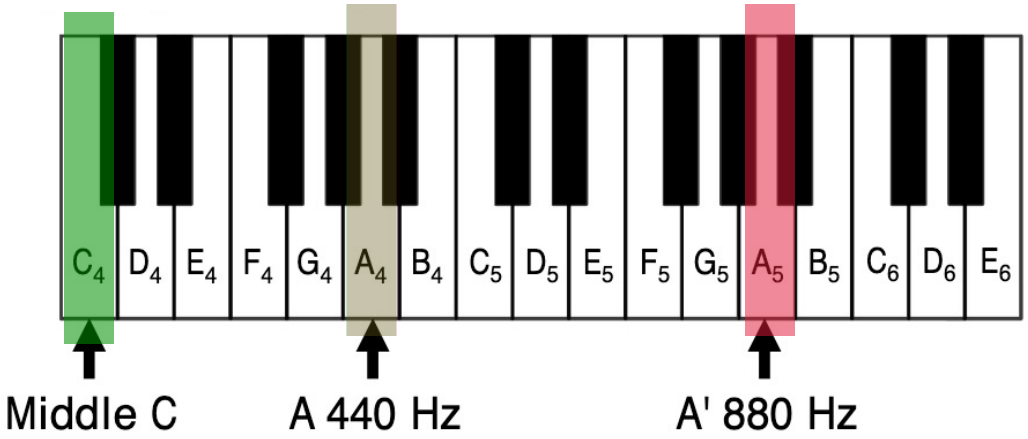


Musical Notes on a Piano



Lab 5

- Involves generating specific tones relating to notes on a keyboard/piano
- Beginning with a reference tone (say 440Hz)
  - if you know the note you want (octave relative to reference octave, and position of note (relative to reference note)
  - you can calculate the frequency of the note you want to generate
  - you can use that frequency value as your variable to generate the right sinusoid… which when played back should sound like the correct tone



$$f_{n+1} = f_n \times 2^{1/12}$$

$f_{desired} = f_{ref} * 2^{s/12}$

$f_{desired} = f_{A5} = f_{A4} * 2^{+12/12}$

$= 440 * 2$

$= 880Hz$

← A5 is +12 semi-tones from A4 (i.e. up one octave)

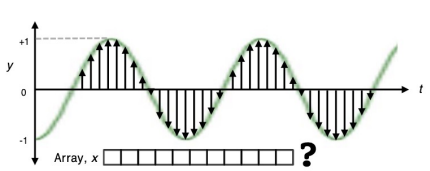
$f_{desired} = f_{C4} = f_{A4} * 2^{-9/12}$

$= 440 * 0.5946$

$= 261.63 Hz$

← C4 is -9 semi-tones from A4 (i.e. down 9/12 octave)

YORK UNIVERSITÉ UNIVERSITY



$$y(t) = \sin(2\pi f t), \qquad t = \frac{i}{SR}$$

where y is an array of samples representing a musical note
t is time in seconds
$y(t)$ is the amplitude at time t (between -1 and +1)
f is frequency in cycles per second or Hz
i is an index into the array ($i = 0, 1, 2, \ldots n\text{-}1$)
SR is the sampling rate (number of samples/second)

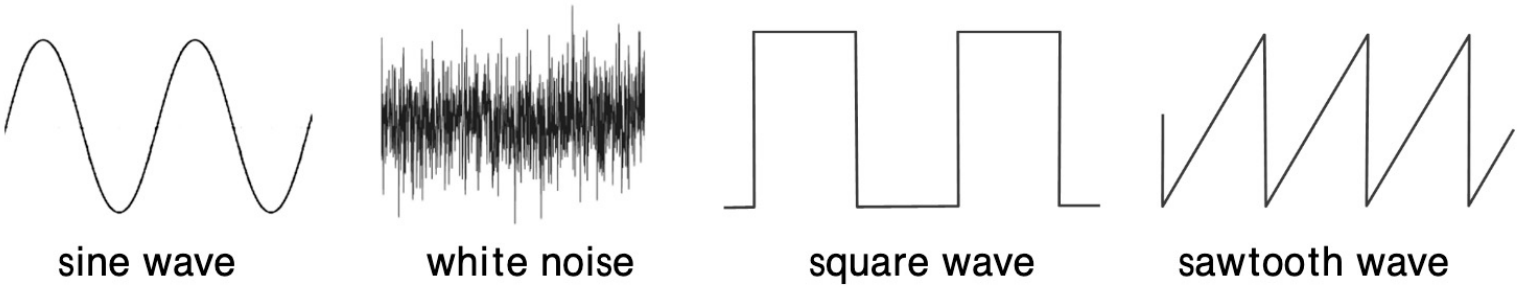
```
import processing.sound.*;

// CREATING A SIMPLE TONE
// (pure sine/cosine waveform)

AudioSample sample;

void setup() {
  size(640, 360);
  background(255);

  // Create an array of sinusoid y(t)
  int sampleRate = 44100;    // number of samples per cycle = SR
  float freq = 440;          // replace with freq relative to ref freq!

  float[] sinewave = new float[sampleRate];

  for (int i = 0; i < sampleRate; i++) {
    sinewave[i] = sin(TWO_PI*freq*i/sampleRate);  // formula for y(t) above right
  }

  // Create audiosample from data, set framerate
  sample = new AudioSample(this, sinewave, sampleRate);

  // Play the sample in a loop (but don't make it too loud)
  sample.amp(0.5);      // sets to half amplitude
  sample.loop();        // audio buffer setup to play over and over in loop
}

void draw() {
}
```

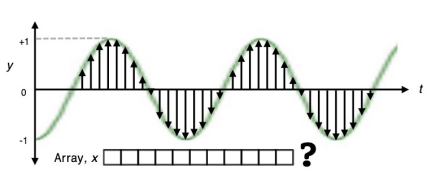
YORK UNIVERSITÉ UNIVERSITY

Standard Waveform Patterns

- Typical setup (uncompressed audio):
  - SR = 44,100Hz (44,100) samples for one second of audio
  - Each sample is a real value between -1 and +1
  - We can control amplitude using `.amp()` method

  - Common Waveform Patterns:





$$y(t) = \sin(2\pi f t), \qquad t = \frac{i}{SR}$$

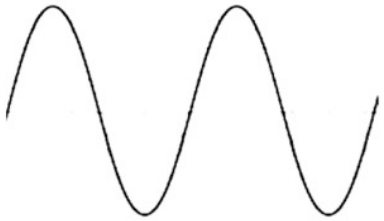
where y is an array of samples representing a musical note
t is time in seconds
$y(t)$ is the amplitude at time t (between -1 and +1)
f is frequency in cycles per second or Hz
i is an index into the array ($i = 0, 1, 2, \ldots n\text{-}1$)
SR is the sampling rate (number of samples/second)

```
import processing.sound.*;

// CREATING A SIMPLE TONE
// (pure sine/cosine waveform)

AudioSample sample;

void setup() {
  size(640, 360);
  background(255);

  // Create an array of sinusoid y(t)
  int sampleRate = 44100;   // number of samples per cycle = SR
  float freq = 440*pow(2,12.0/12);   // freq = 12 semitones above 440 = 880Hz
  freq = 440*pow(2,-9.0/12);         // freq = 9 semitones below 440 = 261.63Hz

  float[] sinewave = new float[sampleRate];

  for (int i = 0; i < sampleRate; i++) {
    sinewave[i] = sin(TWO_PI*freq*i/sampleRate);  // formula for y(t) above right
  }

  // Create audiosample from data, set framerate
  sample = new AudioSample(this, sinewave, sampleRate);

  // Play the sample in a loop (but don't make it too loud)
  sample.amp(0.5);      // sets to half amplitude
  sample.loop();        // audio buffer setup to play over and over in loop
}

void draw() {
}
```

YORK UNIVERSITÉ UNIVERSITY

Sine Wave (revisited)

$$y(t) = \quad (\sin\ 2\pi f t)$$

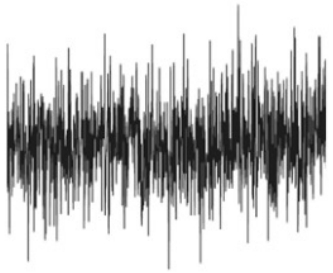


sine wave

```
float[] genSineWave(float freq, float sr, int numSamples) {

  // assume numSamples = duration * sampleRate;  (e.g. 5 sec = 5*44100 samples)

  float[] waveform = new float[numSamples];

  for (int i = 0; i < numSamples; i++) {
    // if i goes beyond sampleRate samples... can repeat sinewave
    waveform[i] = sin(TWO_PI*freq*(i%sr)/sr);
  }

 return waveform;

}
```

(i%sr) repeats sine wave if the numSamples is larger than the sampleRate

White Noise

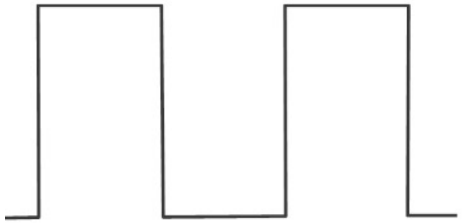


white noise

```
float[] genWhiteNoise(int numSamples) {

  // uniform white noise just uses random values on (-1,+1) interval
  // when played back, it sounds like radio static

  float[] waveform = new float[numSamples];

  for (int i = 0; i < numSamples; i++) {
        waveform[i] = random(2)-1;    // generates random on (0-2) then -1
                                      // so final value is on (-1,1)
  }

  return waveform;
}
```

Square Wave

$$y(t) = \text{sgn}(\sin\ 2\pi f t)$$



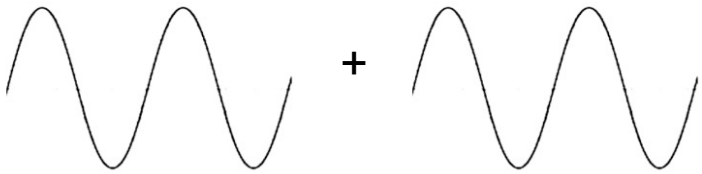
square wave

This means:
sgn(x) = -1 if x<0
sgn(x) = +1 if x>=0

```
float[] genSquareWave(float freq, float sr, int numSamples) {

  float[] waveform = new float[numSamples];
  float sineValue;

  for (int i = 0; i < numSamples; i++) {

    sineValue = sin(TWO_PI*freq*(i%sr)/sr);

    // if positive, set to +1, if negative set to -1
    if (sineValue>=0)
      waveform[i] = 1.0;
    else
      waveform[i] = -1.0;

   return waveform;
}
```

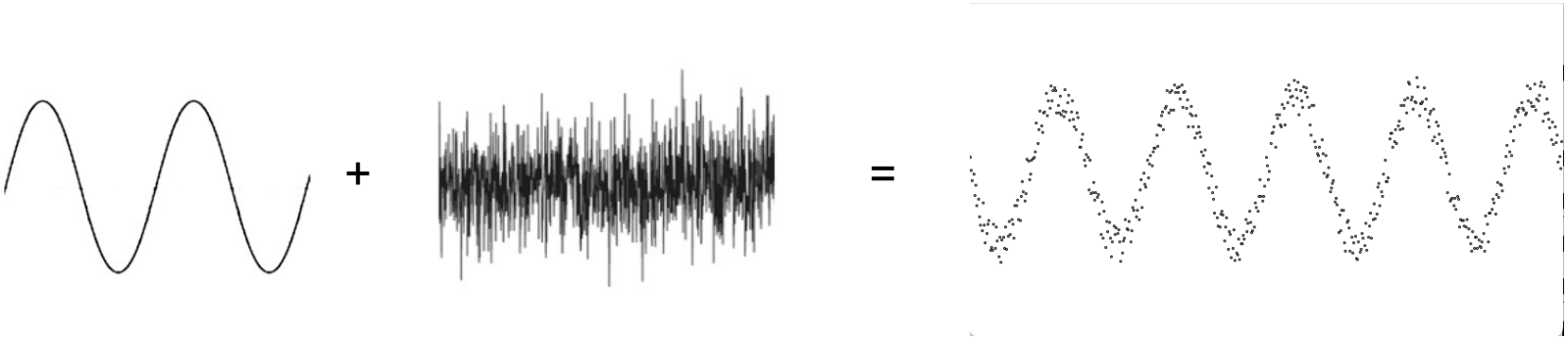
(i%sr) allows numSamples to be > sr, but the sine wave just repeats

Adding waveforms



```
float[] addWaveforms(float[] w1, float a1, float[] w2, float a2) {

 // assumes both waveforms are same size arrays!!
 // a1 used to scale w1, a2 used to scale w2   (both between 0-1)

 float[] wResult = new float[w1.length];

 if (w1.length == w2.length) {

   // add them
   for (int i=0; i<wResult.length; i++) {
     wResult[i] = a1*w1[i] + a2*w2[i];
   }
 }

 return wResult;   // will be zeros if diff lengths
}
```

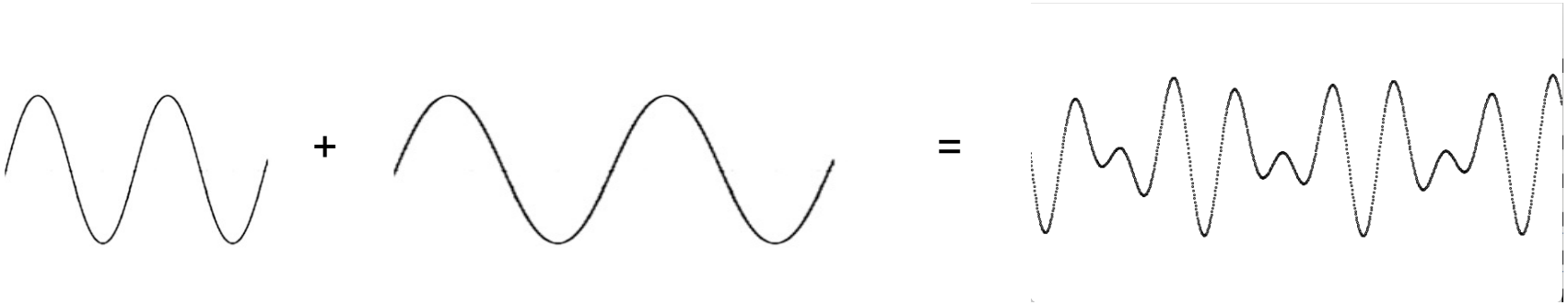
Making a tone noisy (note + scratchiness)



```
float[] waveform1 = genSineWave(440,sampleRate,numSamples);
float[] waveform3 = genWhiteNoise(numSamples);

waveform = addWaveforms(waveform1,0.8,waveform3,0.2);

displayWaveform(waveform,500);
```

Note, mostly waveform1 with a little bit of noise

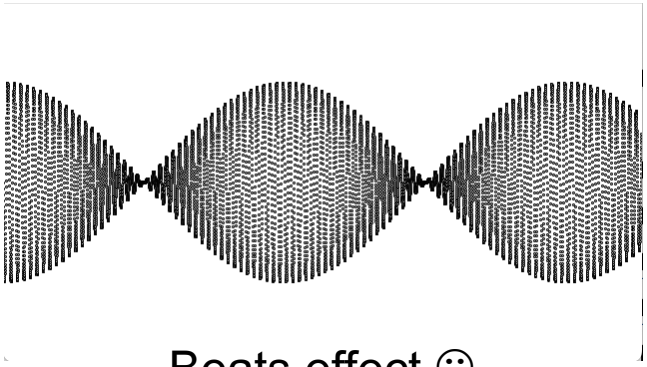
Adding tones of different frequencies?



```
float[] waveform1 = genSineWave(440,sampleRate,numSamples);
float[] waveform2 = genSineWave(300,sampleRate,numSamples);

waveform = addWaveforms(waveform1,0.5,waveform2,0.5);

displayWaveform(waveform,500);
```

Really close frequencies?

```
float[] waveform1 = genSineWave(440,sampleRate,numSamples);
float[] waveform2 = genSineWave(430,sampleRate,numSamples);

waveform = addWaveforms(waveform1,0.5,waveform2,0.5);

displayWaveform(waveform,10000);
```

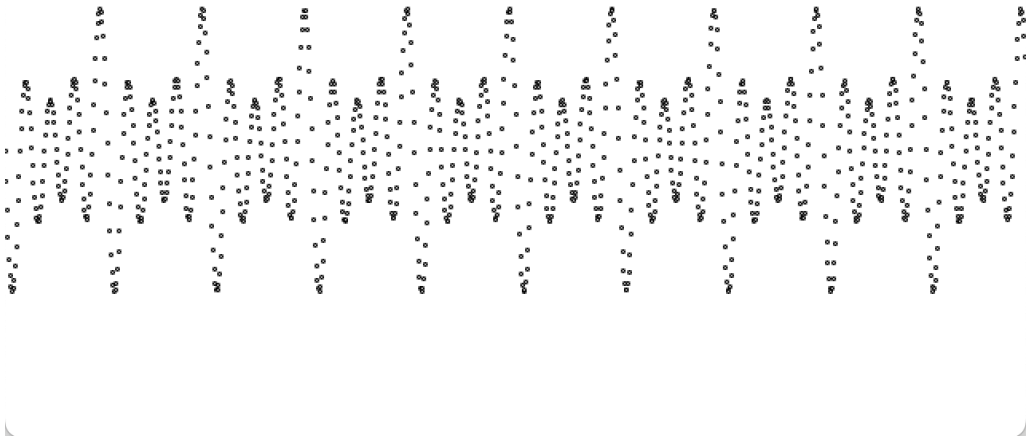


Beats effect ☺

Harmonics?

Tones (sine waves) that have frequencies that are integer multiples of one another

```
float[] waveform1 = genSineWave(440,sampleRate,numSamples);
float[] waveform1b = genSineWave(440*2,sampleRate,numSamples);
float[] waveform1c = genSineWave(440*3,sampleRate,numSamples);
float[] waveform1d = genSineWave(440*4,sampleRate,numSamples);

waveform = addWaveforms(waveform1,0.5,waveform1b,0.5);
waveform = addWaveforms(waveform,0.5,waveform1c,0.5);
waveform = addWaveforms(waveform,0.5,waveform1d,0.5);
```



Similarly, chords are combinations of tones (some combinations sound good, some don't)

(Amplitude) Articulation

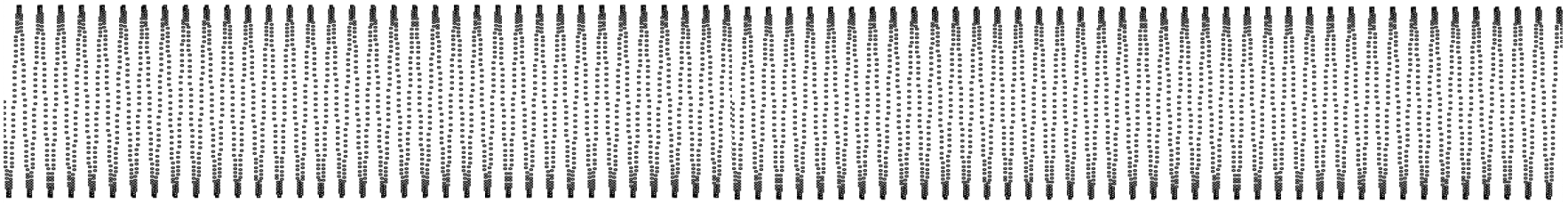
- Three articulation modes:
  - LEGATO (long, like an organ; this is the default)
  - STACCATO (short – amplitude goes to 0 after 0.1 second)
  - DECAY (fading, like a piano or plucked string)

Think about how you would modify addWaveforms() to achieve these effects on a single waveform

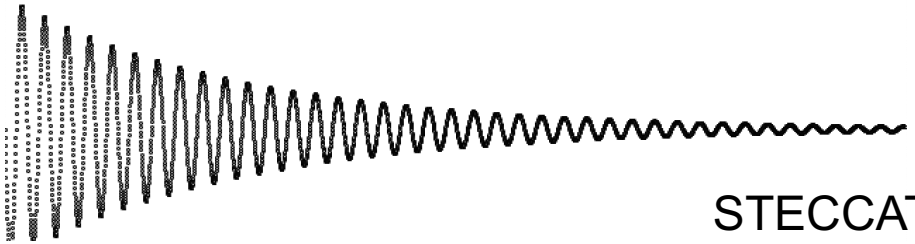
Could ”modulate” waveform (multiply by a value that varies)

The value could get smaller (decay): `amp *= 0.99;`
Or could decay/grow linearly: `amp = amp*(1- i/numSamples);`
Or could decay exponentially: `amp = exp(-1.0*i/K);  // e^-i/K`

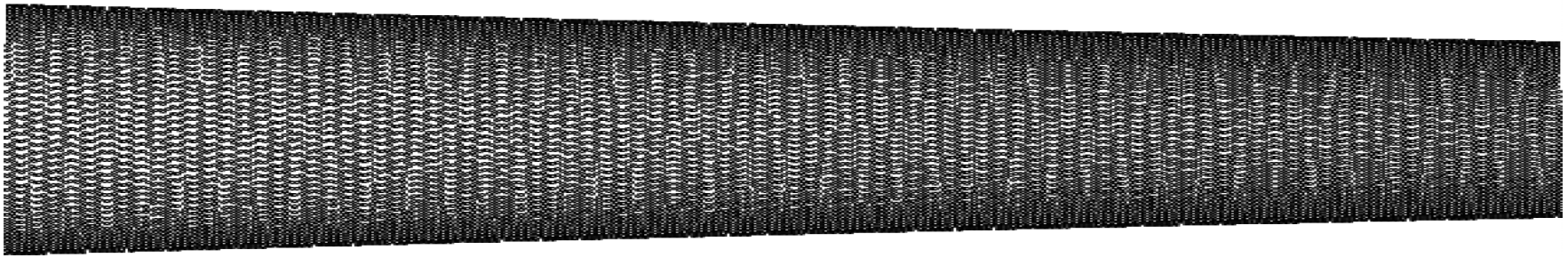
Articulation



LEGATO



STECCATO



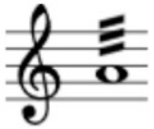
DECAY

Frequency articulation

Chirps ~ Rise/Fall of Freq (lab 5)

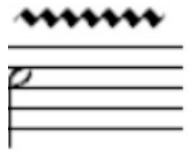
Tremolo

- Tremolo is a periodic change in amplitude.
- Achieved by multiplication: sin(A) * sin(B), where A is the note, and B is the tremolo



Vibrato

- Vibrato is a periodic change in frequency.
- Two additional variables are needed:
  - fVib → frequency of the vibrato
  - aVib → amplitude of the vibrato (change in frequency of the note)



More in lab 5

Digression – more objects & basic file IO

Some other useful reference types:

Data

Composite

`Array`	An array is a list of data
`ArrayList`	An `ArrayList` stores a variable number of objects
`FloatDict`	A simple table class to use a `String` as a lookup for a float value
`FloatList`	Helper class for a list of floats
`HashMap`	A `HashMap` stores a collection of objects, each referenced by a key
`IntDict`	A simple class to use a `String` as a lookup for an int value
`IntList`	Helper class for a list of ints
`JSONArray`	A JSONArray is an ordered sequence of values
`JSONObject`	A `JSONObject` is an unordered collection of name/value pairs
`Object`	Objects are instances of classes
`String`	A string is a sequence of characters
`StringDict`	A simple class to use a `String` as a lookup for an `String` value
`StringList`	Helper class for a list of Strings
`Table`	Generic class for handling tabular data, typically from a CSV, TSV, or other sort of spreadsheet file
`TableRow`	Represents a single row of data values, stored in columns, from a `Table`
`XML`	This is the base class used for the Processing XML library, representing a single node of an `XML` tree

ArrayLists

- Like arrays, but a lot more convenient!
  - keeps elements in a sequence (like arrays) but can grow
  - includes methods to add, sort, find max, reverse, shuffle etc…

- IntList → dynamic/resizable array of ints
- FloatList → dynamic/resizable array of floats
- StringList → dynamic/resizable array of Strings

- ArrayList → use if you want a list of any type of object
  (e.g. like an ArrayList of PVector)

StringList

In fact most methods common to all ArrayLists

Methods	
`size()`	Get the length of the list
`clear()`	Remove all entries from the list
`get()`	Get an entry at a particular index
`set()`	Set an entry at a particular index
`remove()`	Remove an element from the specified index
`append()`	Add a new entry to the list
`hasValue()`	Check if a value is a part of the list
`sort()`	Sorts the array in place
`sortReverse()`	A sort in reverse
`reverse()`	Reverse the order of the list
`shuffle()`	Randomize the order of the list elements
`lower()`	Make the entire list lower case
`upper()`	Make the entire list upper case
`array()`	Create a new array with a copy of all the values

IntList

In fact most methods common to all ArrayLists

Methods	
`size()`	Get the length of the list
`clear()`	Remove all entries from the list
`get()`	Get an entry at a particular index
`set()`	Set the entry at a particular index
`remove()`	Remove an element from the specified index
`append()`	Add a new entry to the list
`hasValue()`	Check if a number is a part of the list
`increment()`	Add one to a value
`add()`	Add to a value
`sub()`	Subtract from a value
`mult()`	Multiply a value
`div()`	Divide a value
`min()`	Return the smallest value
`max()`	Return the largest value
`sort()`	Sorts the array, lowest to highest
`sortReverse()`	Reverse sort, orders values from highest to lowest
`reverse()`	Reverse the order of the list elements
`shuffle()`	Randomize the order of the list elements
`array()`	Create a new array with a copy of all the values

```
final int MAX_ITEMS = 10;
String [] inventory = new String[MAX_ITEMS];
int numItems = 0;


inventory[numItems++] = “banana”;
inventory[numItems++] = “stick”;
inventory[numItems++] = “BFG”;
inventory[numItems++] = “abomb”;
inventory[numItems++] = “magic potion”

// output inventory
println(“You currently have ” + numItems + “ items:”);
for (int i=0; i<numItems; i++) {
        println(inventory[i]);
}
```

```
StringList inventory = new StringList();

inventory.append(“banana”);
inventory.append(“stick”);
inventory.append(“BFG”);
inventory.append(“abomb”);
inventory.append(“magic potion”);

// output inventory
println(“You currently have ” + inventory.size() + “
items:”);
for (int i=0; i<inventory.size(); i++) {
        println(inventory.get(i));
}
```

No need for fixed size or tracking num elements etc.

If we have more than 10 elements, array will be an issue

```
// output inventory
println("You currently have "
         + inventory.size() + " items:");
for (int i=0; i<inventory.size(); i++) {
  println(inventory.get(i));
}

// reverse order
println();
inventory.reverse();
println("Reversed: you currently have "
         + inventory.size() + " items:");
for (int i=0; i<inventory.size(); i++) {
  println(inventory.get(i));
}

// sort (alphabetically)
println();
inventory.sort();
println("sorted: you currently have "
         + inventory.size() + " items:");
for (int i=0; i<inventory.size(); i++) {
  println(inventory.get(i));
}
```

```
You currently have 5 items:
banana
stick
BFG
abomb
magic potion

Reversed: you currently have 5 items:
magic potion
abomb
BFG
stick
banana

sorted: you currently have 5 items:
abomb
banana
BFG
magic potion
stick
```

```
// output inventory
println("You currently have "
         + inventory.size() + " items:");
for (int i=0; i<inventory.size(); i++) {
  println(inventory.get(i));
}

// reverse order
println();
inventory.reverse();
println("Reversed: you currently have "
         + inventory.size() + " items:");
for (int i=0; i<inventory.size(); i++) {
  println(inventory.get(i));
}

// sort (alphabetically)
println();
inventory.sort();
println("sorted: you currently have "
         + inventory.size() + " items:");
for (int i=0; i<inventory.size(); i++) {
  println(inventory.get(i));
}
```

```
You currently have 5 items:
banana
stick
BFG
abomb
magic potion

Reversed: you currently have 5 items:
magic potion
abomb
BFG
stick
banana

sorted: you currently have 5 items:
abomb
banana
BFG
magic potion
stick
```

Reading in simple text files

colours.txt

```
black 0 0 0
white 255 255 255




red 255 0 0
blue 0 0 255


green 0 255 0


grey 128 128 128


darkgrey 50 50 50
lightgrey 200 200 200
```

Can use a method directly:

```
Strings[] lines = loadStrings(filename);
```

filename (e.g. colours.txt) has to exist within the sketch folder

Example (read and remove blank/empty lines)

```
String[] readTextFile(String fileName) {

  String[] lines = loadStrings(fileName);
  StringList content;                                  // an arraylist of strings

  println(fileName + " has " + lines.length + " lines");
  if (!(lines.length>0)) return null;

  content = new StringList();  // instantiate empty StringList
  int empty = 0;
  int text = 0;

  for (int i=0; i<lines.length; i++) {

    if (!(lines[i].isEmpty()||lines[i].isBlank()) ) {
       content.append(lines[i]);
       text++;
    }
    else {
      empty++;
    }
  }
  println("-> there were " + empty + " empty lines");
  println("-> there were " + text + " non-empty lines");
  return content.toArray();
}
```

YORK UNIVERSITÉ UNIVERSITY

”parsing” the input file…

```
The file: colours.txt has 16 lines
-> there were 8 empty lines
-> there were 8 non-empty lines

colours.txt contains:
          black 0 0 0
          white 255 255 255
          red 255 0 0
          blue 0 0 255
          green 0 255 0
          grey 128 128 128
          darkgrey 50 50 50
          lightgrey 200 200 200
```

Example
(process the lines → using split on each)


```
void setup() {
  size(600, 800);

  String[] colourList = readTextFile("colours2.txt");
  println("\ncolours.txt contains: ");
  float sX = 100;
  float sY = 100;

  for (int i=0; i<colourList.length; i++) {
    println("\t" + colourList[i]);

    // for each colour... set a stroke colour, and draw colour in that colour
    String[] tokens = split(colourList[i], ' ');
    String colName = tokens[0];
    int colrgb = color(int(tokens[1]), int(tokens[2]), int(tokens[3]));

    stroke(colrgb);
    fill(colrgb);
    textSize(128);
    text(colName, sX, sY);
    sY += 100;
  }
}
```

YORK UNIVERSITÉ UNIVERSITY

Reading in simple text files

colours.txt

```
black 0 0 0 110.2
white 255 255 255 202.123




red 255 0 0 289.412
blue 0 0 255 334.98


green 0 255 0 431.5


grey 128 128 128 550.756


darkgrey 50 50 50 600
lightgrey 200 200 200 150.21
```

Format of the file has to be known

e.g. could use 4th number for positioning Text labels in y direction

Can have first line read and processed to figure out how to read the rest of the file (more on this next lecture)

YORK UNIVERSITÉ UNIVERSITY

Example
(let file determine y positions of text labels)


```
void setup() {
  size(600, 800);

  String[] colourList = readTextFile("colours2.txt");
  println("\ncolours.txt contains: ");
  float sX = 100;
  float sY = 100;

  for (int i=0; i<colourList.length; i++) {
    println("\t" + colourList[i]);

    // for each colour... set a stroke colour, and draw colour in that colour
    String[] tokens = split(colourList[i], ' ');
    String colName = tokens[0];
    int colrgb = color(int(tokens[1]), int(tokens[2]), int(tokens[3]));

    stroke(colrgb);
    fill(colrgb);
    textSize(128);
    text(colName, sX, sY);
    sY = float(tokens[4]);
  }
}
```

YORK UNIVERSITÉ UNIVERSITY