Assignment 1 Simulating a Piano

**Due: Monday 12th September by 4:59 pm  
Weight: 12%**

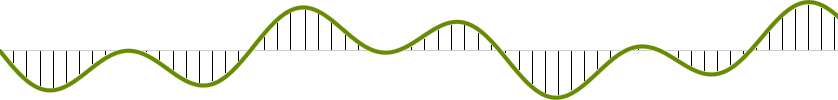
# Objectives

* Review interfaces, classes and their implementation in C#
* Use proper encapsulation when creating separate classes
* Use a unit test framework to write automated tests

# Background

The sounds that we hear in our day-to-day lives travel through the air as waves. When there is a disturbance in the air (like the pluck of a string or a clap of one's hands), the resulting pressure that is produced propagates through the air as a waveform which humans can hear.

A sound wave is an analog signal, which means that over time it is composed of infinitely many values. Obviously, computers have limited space, and therefore cannot store an infinite number of values in memory. To represent a sound wave using a computer, we sample it at specific intervals. If you think of a wave as a continuous function f(x), then sampling simply means choosing certain x-values at which to compute the function and storing the resulting y-values in a list. Therefore, to a computer, a sound wave is just a long list of numbers, where each number is sample of the wave at a certain time.



As you might guess, the more samples we have, the more accurate our sound will be. The **sampling rate** is how many times per second we choose to sample the wave. For CD audio, this value is 44100, which means that 44100 data values are encoded each second, which can be written as 44.1kHz. We know how general sound is represented, but what about notes? Each note in the modern chromatic scale is associated with a certain **frequency** (measured in Hz). The frequency of a note corresponds to the number of periods of that note's sound wave that occur each second. For example, the 49th key on a modern piano (A) has a frequency of 440 Hz. Similarly, the middle C has a frequency of about 262 Hz.

## Hammering a piano wire

Let's take a close look at what happens when we strike a piano wire, since that is what we will be simulating. At first, the wire is highly energized and it vibrates rapidly, creating a fairly complex (meaning rich in harmonics) sound. Gradually, due to friction between the air and the wire, the wire’s energy is depleted, and the wave becomes less complex, resulting in a "purer" tone with fewer harmonics. After some amount of time all the energy from the strike is gone, and the wire stops vibrating.

# The Karplus-Strong Algorithm

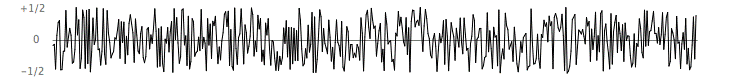
Now that we have a physical idea of what is happening in a struck wire, how can we model it with a computer? The [Karplus-Strong](https://en.wikipedia.org/wiki/Karplus%E2%80%93Strong_string_synthesis) algorithm played a seminal role in the emergence of physically modeled sound synthesis.

A piano wire is simulated as follows:

The frequency of the nth key on a piano is calculated as

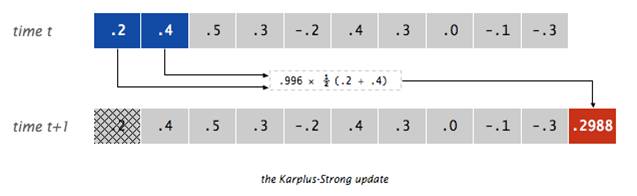
When the wire is initially struck, it contains a sequence of samples full of random noise. The number of samples in the sequence depends on the sampling frequency and the frequency of the note:

If the sampling frequency is 44100 Hz, and the frequency of the note is 440 Hz (for A), then there are 44100/440 or 100 samples (rounded to nearest integer). These 100 samples are initially filled with noise when the wire is struck. Noise is simulated by assigning random real numbers between -0.5 and +0.5.



The strike then causes a displacement which spreads wave-like over time. The Karplus-Strong algorithm simulates the vibration spreading like a wave: the algorithm repeatedly adds a new displacement sample at the end of the buffer that is the average of the first two samples, scaled by a decay factor, and deletes the first displacement sample from the buffer.

The diagram below gives a simplified example. For example, at time t=0, the wire is struck. So the N displacement samples (assumes N = 10) are filled with random numbers between -0.5 and +0.5.



At time t+1, a sample is added to the end, which is the average of the first two samples multiplied by a decay factor of 0.996. The first sample is removed. This is known as the “feedback mechanism”.

Notice that there are always N displacement measures at every time (the buffer does not change in length).

**Why does it work?** The two primary components that make the Karplus-Strong algorithm work are the ring buffer feedback mechanism and the averaging operation.

* The ring buffer feedback mechanism. The ring buffer, or circular array, models a wire tied down at both ends, in which the energy travels back and forth. The length of the array determines the fundamental frequency of the resulting sound. The feedback mechanism reinforces the fundamental frequency and its harmonics. The energy decay factor (.996 in this case) models the slight dissipation in energy as the wave makes a roundtrip through the string.
* The averaging operation. The averaging operation serves as a gentle low-pass filter (which removes higher frequencies while allowing lower frequencies to pass, hence the name). Because it is in the path of the feedback, this has the effect of gradually attenuating the higher harmonics while keeping the lower ones, which corresponds closely with how a piano wire sounds.
* Some intuition. A more intuitive, but somewhat less precise explanation of the algorithm is the following. When you hammer a piano wire, the middle of the wire bounces up and down wildly. Over time, the tension on the string causes the string to move more regularly and more gently until it finally comes to rest. High frequency wires have greater tension, which causes them to vibrate faster, but also to come to rest more quickly. Low frequency wires are looser, and vibrate longer. If you think about the values in the array as positions over time of a point in the middle of a string, filling the buffer with random values is equivalent to the string bouncing wildly (the pluck). Averaging neighboring samples brings them closer together, which means the changes between neighboring samples becoming smaller and more regular. The decay factor reduces the overall amount the point moves, so that it eventually comes to rest. The final kicker is the array length. Low notes have lower frequencies, and hence longer ring buffers (44,100 / N is bigger if N is smaller). That means it you will have to go through more random samples before getting to the first round of averaged samples, and so on. The result is it will take more steps for the values in the buffer to become regular and to die out, modeling the longer reverberation time of a low wire.

# Instructions

In this project, a simulation capable of producing different piano notes will be created. A console application will provide a text-based user interface, where the user can select which key they want to press using the keyboard. A separate library (DLL) will be created to contain the algorithm that generates the piano note to be played. Ensure your code is added to a private git repository. Generate an appropriate gitignore file for the repository.

## Scaffolding

The following projects will need to be created, see below for details on what they contain:

* A console application called KeyboardPiano
* A class library called PianoSimulation
* An MS Test Unit test project called PianoSimulationTests

**Ensure you use the –framework argument when calling dotnet new and specify a netcoreapp3.1**

## Piano Class Library

Create a class library called PianoSimulation. This library will contain all the code needed compute the result of the Karplus-Strong algorithm and return the value to be played the audio system. Add the provided IRingBuffer and IMusicalString interfaces to this project.

### CircularArray class

Write class CircularArray which implements IRingBuffer. Note that the CircularArray uses an array internally. The constructor of the CircularArray has one parameter, the length of the internal array. **Hint**: use a variable to track where the front of the buffer is at any time: don’t move all the elements of the array.

Write unit tests in the PianoSimulationTests project to validate the functionality of the class. Note, to run unit tests call dotnet test

### PianoWire class

Write class PianoWire class which implements the IMusicalString interface. Note that the PianoWire contains a Circular array. The constructor of the PianoWire takes two parameters: the note’s frequency and the sampling rate. These parameters are used to create a CircularArray of the appropriate size.

Write test cases to test your PianoWire class.

### Piano Class

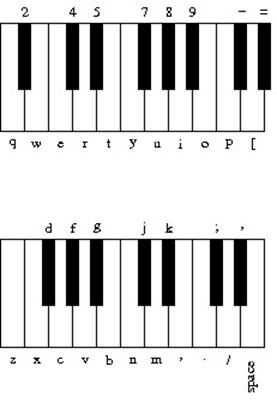
The Piano class implements the IPiano interface. Our piano has different characters that correspond to keys (or wires) in a piano. It encapsulates a List<IMusicalString> representing the wires in the piano. Each wire is associated with a key.

The constructor:

public Piano(string keys = "q2we4r5ty7u8i9op-[=zxdcfvgbnjmk,.;/' ", int samplingRate = 44100)

takes two parameters.

The keys parameter represents the character mapping to the piano keys. The string above maps to these piano keys:



* Loop through every character in the keys string – each one is associated with a note whose frequency is calculated as where n is index of the character. Note, this is for a shorter piano without a full 88 keys. Pay attention to any rounding or implicit casting to integers in your implementation.
* Based on the frequency of the note and the sampling rate, instantiate a PianoWire and add to the List.
* The Piano object has two methods:

public void StrikeKey(char key) Strikes the piano key (wire) corresponding to the specified character. Note, use the IndexOf method to find the index of the character within the string, which should be the same as the index in the List

public double Play() Goes through all the piano wires and returns the sum of their samples. This allows multiple notes to be played at the same time.

Note that the unit tests of the Piano class depend on the unit tests of the previous classes passing.

## Console Application

Create a console application called KeyboardPiano that provides a user interface for the user to select which key they want to play on the piano. Ensure that the interface has a readable menu, that the user can easily exit. The user should be able to list all the strings in the piano with their associated keys, strike a single key on the piano by typing in the console.

Use the provided Audio.cs file to play sound through the speakers. Audio.cs is dependent on the NAudio package, which is an external library. Use dotnet add package NAudio to add a reference to that external package. Note, this will update the KeyboardPiano.csproj to reference the NAudio package.

Additionally, you will need to add a reference to the piano class library PianoSimulation, to use your simulation in your console application.

To play a key on the piano, create a piano object and an audio object. After striking a key, call the Audio class’s Play method, where it takes the result of the pianos Play method. Note you will need to loop over the Audio class’s play method repeatedly to get a good result. Loop the sampling rate multiplied by 3.

# Deliverable

Create a private Gitlab repo to track your code. Add your instructor as a maintainer to the repo (section 1: swetha411, section 2: ddubois1). When developing, feel free to use the feature branch model.

The deliverable will consist of a Merge Request (Pull request) from your main branch to a branch called Submission. Note, in this case you are merging from main into the Submission branch you have created. There must be no commits on the Submission branch, and the change list should show all the code you have modified in main. Submit the URL for the Merge Request to Moodle.

# Resources

* Dotnet cli <https://docs.microsoft.com/en-us/dotnet/core/tools/>
* Dotnet add package <https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-add-package>
* Dotnet add reference <https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-add-reference>
* MS Test Asserts <https://docs.microsoft.com/en-us/visualstudio/test/using-the-assert-classes?view=vs-2022>

# Attribution

This assignment is based on a [Nifty Assignment](https://introcs.cs.princeton.edu/java/assignments/guitar.html) introduced by Kevin Wayne (Princeton University) and refined by Stuart Reges (University of Washington), David Reed (Creighton University), Benedict Brown (University of Pennsylvania), and John DeNiro (U of California, Berkeley). Large segments were copied directly.