Audio Process Design & Implementation
Practical 10 – Introducing MySynth

1 Introduction

This semester's practicals will investigate the nature of synthesis and control through the development of professional audio plugins, using the MySynth code template, incorporating the *Synthesizer Toolkit (STK)*, a C++ code library for DSP, developed by Stanford's *Center for Computer Research in Music & Acoustics (CCRMA)*.[1] Your understanding of practical synthesis techniques will be assessed as part of the second assignment, as well as the exam.

The STK code library is a collection of ready-made C++ objects that you can include in your own projects, providing you with a basic collection of DSP building blocks (filters, signal generators, basic effects, etc.), which has been extended with additional objects and features specifically for the APDI module, and which you can combine together to make synthesisers based on practically any type of synthesis. Objects in the code library broadly correspond with the different types of nodes of flow diagrams presented in the lectures and course notes. Similarly, you can also think of each code object as fulfilling roles similar to the basic synthesis objects used to construct a Max patch. Compared to Max:

- Instead of creating and linking components visually, you will insert and link objects using C/C++ code, drawing and developing on the programming techniques learnt in the first semester (and also the first year).

- While prototyping synthesisers is a little more cumbersome than Max, plugins written in C/C++ code typically perform better, and can be run inside most music programs, such as Logic, SoundTrack, Cubase, AU Lab, etc. This not only makes it easier to use your synthesisers and effects in your own music, but makes them easier to distribute (or sell) for use by others – though please read the note overleaf, concerning commercial development and use or distribution of STK, JUCE, AU and UWE code.

- While Max is popular with artists and sound installations, the C/C++ concepts and techniques taught on this module more closely reflect industry-standard practices within professional audio and music software development.

This first practical introduces you to the new *MySynth* Xcode/Visual Studio template project and the technologies it is built on, including STK and JUCE.[2] You will then extend the basic synthesiser demo, based on a single sine wave signal generator, by using additive synthesis to create saw and square wave signal generators, combining multiple sine wave generators to build up the complex tone of these waves. To do this, you will be introduced to several concepts surrounding *classes* and basic concepts of *object-oriented programming (OOP)*, which allow you to bundle up each complex tone's multiple sine wave generators into a single code object, which can then be used to replace the demo's original "sine" object.

---

[1] *Synthesiser Toolkit (STK)* / CCRMA - https://ccrma.stanford.edu/software/stk/
[2] *Jules' Utility Class Extensions (JUCE)* / Raw Material Software - http://www.juce.com

1.1 *Object-Oriented Programming (OOP)*

Object-oriented programming (OOP) is based on packaging up all the related variables, constants, functions, and processes relating to a single object in your synthesizer into a single object in your code, called a *class*. You can then use (and re-use) that object in your code without worrying about the low-level details of how it works internally. For example, when we think about synthesizers, it's easier to think in terms of "filter", "oscillator", or "modulator", rather than the multitude of individual transistors and resistors or individual lines of DSP code they're made of. The class becomes a new data type (like integers or floats, but more 'complex') that you can create and place in your processing code, much like any other variable. For example, once you've written a filter class, you can create as many instances of that filter as you like – e.g. using an array of filter objects to create a filter bank.

This process of grouping related items is called *abstraction*, because it hides away the low-level details within a higher-level container and helps you see the bigger picture more clearly. Within a program, there can be many levels of abstraction, created to make the code easier to understand and maintain. For example, you might create a class to represent a filter bank, which would itself contain multiple filter objects. Indeed, some of STK's more complex objects themselves include other STK objects as sub-components.

1.2 The *MySynth* Xcode / Visual Studio Project

Replacing last semester's template projects for developing AU/VST effect plugins (*MyEffect)*, a new template, called *MySynth*, is provided for developing synthesiser plugins, which you will be able to load as software (or "virtual") instruments in compatible plugin hosts (*Logic*, Cubase, REAPER, Premiere Pro, etc.).

MySynth works as a mini-plugin, in the same a way as *MyEffect* – that is, it connects to a host AudioUnit or VST plugin, called *Synth Plugin* (.component or .dll file), which handles the low-level details of midi input and audio output, talks to the plugin host (DAW), and provides tools for crafting UIs or handling input. Like *MyEffect*, a configurable UI has been provided for you, enabling you to focus on synthesis (audio processing) and control (handling MIDI).

This architecture allows your code to be written in a pure C++, without any OS- or API- (or even JUCE-) specific code, in a single, unified format, in a format suitable for the underlying host system, allowing you to write plugin code that can be run on multiple platforms without a re-writing. Hence, while *MySynth* runs as an AU synth plugin on Mac and a VSTi plugin on windows, it is possible to adapt the code for any plugin format or OS (e.g. RTAS, iOS, etc.).

A Note on Commercial or Open-Source Development using *MyEffect*/*MySynth*

When using other people's code (libraries, templates, wrappers, etc.), you must always check the license to find out what it can be legally used for. Depending on the license, you may have to pay a fee to use the software commercially, or may be obliged to open source it (make your source code available publically) before distributing it, sometimes even when giving it away. For example, JUCE can be used for free if you share your code (make it open source), but commercial uses (without open-sourcing) incur a license fee.

You are free to share, distribute and sell your *MyEffect*/*MySynth* plugin binaries without restriction, and show the code privately to prospective employers, but please email Chris Nash (chris.nash@uwe.ac.uk) if you wish to distribute the code publicly.

2 Getting Started

NB: The following instructions (and workarounds) are similar to those for the previous MyEffect project.

1. From Blackboard, download the *MySynth* project template (under "Learning Materials"). Copy the `MySynth.zip` file to a local folder and double-click it to expand its contents.

   Note: Always be sure to copy work to safe, permanent storage - your APDI folder plus an external drive and/or cloud-based storage (e.g. OneDrive). See Practical 3 for tips.

   The resulting folder contains two sub-folders:

   - `Build` – containing the executable binaries (including *Synth Plugin*, *Plugin Host*, and your compiled *MySynth* mini-plugin), plus any temporary build files.
   - `MySynth` – containing all the source code, resource files, and code projects. This folder contains **all your work**, and is the one you need to back up or submit for assignments.

2. In the `MySynth` folder, open the appropriate project file `MySynth.xcodeproj` (Mac/Xcode) or `MySynth.vcxproj` (Windows/Visual Studio), by double clicking it.

   - The project consists of five main code files: `SynthPlugin.cpp/h`, `SynthPlugin.cpp/h` and `SynthExtra.h`.
   - To test our plugin, a dedicated program called "Plugin Host" is provided. The program should install and run automatically when you build and run your code. It should also handle configuration and installation of the *Synth Plugin* and your *MySynth* mini-plugin.

3. Build and run the project (⌘R in Xcode; F5 in Visual Studio). After a short delay, you should see your plugin appear, running in *Plugin Host*. Click a piano key to check the audio is working.

4. If your plugin does not appear, try the following troubleshooting steps:

   - After compiling, check in the `Build` folder to make sure the `MySynth.bundle` (Mac) or `MySynth.dll` (Windows) files have been produced. This is the mini-plugin that links with the `Synth Plugin.component` (AU) or `Synth Plugin.dll` (VST) plugins.
   - Running the project installs / launches the *Plugin Host*, which in turn automatically installs and/or runs your plugin. On Windows, everything runs from the `Build` folder. On Mac, plugins need to be installed to the system folder, which *should* happen automatically when you first run your code, but you might see errors (see inset, below).
   - Otherwise, check the *Discussion Board* on Blackboard and/or report any problems (always including details of your system and any error messages) to the module leader.

5. The plugin UI includes a virtual piano keyboard, which can be controlled by clicking keys with the mouse or using the computer keyboard. The middle row of keys are the white notes, with black notes above:

```
  [W] [E]     [T] [Y] [U]     [O] [P]
[A] [S] [D] [F] [G] [H] [J] [K] [L] [;]
```

   The plugin should also respond to input from attached MIDI controllers. If available, check that the controller's keys also trigger note playback.

***Mac only**: The tools for Xcode development are distributed using a disk image (dmg) file. Double click this file to "insert" the disk, and the follow the instructions inside it. After copying* Plugin Host.app *to your Applications folder, run the app from that folder by right-clicking it and selecting "Open" – not by double-clicking it. This will give you the option to proceed through the security warning, and will also authorise the Synth Plugin that gets installs. There will be a short pause as it installs the plugin and you may need to re-run the app.*

To check the install in Finder, press Shift-⌘G and enter: `~/Library/Audio/Plug-Ins/Components` (note the ~ tilde). Verify that `MySynth.bundle` and `Synth Plugin.component` are both present. If they are, you might need to either rebuild or re-run the project, or even restart the computer to refresh your audio plugin cache.

2.2 How *MySynth* Works

In the code project, a lot of the internal workings and functionality (e.g. note management) are handled automatically, so that you need typically only focus on five main source code files to develop your synthesiser:

- `SynthNote.cpp/h` – contains the implementation code for generating sound for each individual note (*MyNote*) and handling related MIDI input.
- `SynthPlugin.cpp/h` – respectively define / declares the code for the wider synthesiser / plugin (*MySynth*), including UI event handlers and any additional audio processing applied to the synthesiser's entire output (e.g. global effects).
- `SynthExtra.h` – provides a space for you to develop your own functions and objects to use anywhere in your plugin (i.e. in *MyNote* or *MySynth*).

Within these files, your plugin is divided into two main types of component – a *Synth* class (`MySynth`), used to represent the synthesiser as a whole, and a *Note* class (`MyNote`), used to represent individual notes:

The MySynth Object (SynthPlugin.cpp/h)

This object represents the synthesiser *as a whole*, containing and managing a set (array) of multiple note objects (as member objects) to support polyphonic playback. When it receives MIDI input, it automatically assigns a new note object and updates the array, disconnecting it when the note is no longer needed. Whenever the plugin host (e.g. Logic, Cubase) asks the plugin for audio data, *MySynth* individually requests the audio from each active note, and mixes them together. The summed audio can then be further processed to add additional, global effects (EQ, reverb, etc.) before being returned to the host and played aloud. In this sense, *MySynth* is very much like *MyEffect*, using *MyNote*s to generate audio input.

In `SynthPlugin.cpp`, these processes are largely handled by the following functions, editing and extending the bodies of which will be the focus of later practicals and the assignment:

- `MySynth::MySynth()` – the constructor: called once, when the synthesiser is first created. Used to setup the initial configuration of the synth (initialising variables, etc.).
- `MySynth::postProcess()` – called *after* individual notes have been synthesised and mixed, providing an opportunity to apply global effects to the final output. As an effect, you should notice a similarity to *MyEffect*'s `process()` function from previous practicals – and your previous effect code will work with only a few name edits. By default, the provided code simply passes the data through without changing it.

The MyNote Object (SynthNote.cpp/h)

Most synthesis efforts will focus on generating an individual note's audio. Each ***MyNote*** object provides the sound for a single note, based on the corresponding MIDI input, which is passed on from the synthesiser. It can be thought of separately, as a *monophonic* synthesiser, without worrying about other notes or how they are managed. It receives relevant MIDI *Note On/Off* messages as well as MIDI *Control Change* messages, which are processed to determine how the sound of the note starts/ends and evolves over time. Then, whenever the synthesiser asks for the audio data, *MyNote* will generate the next block of audio for its note.

- `MyNote::onStartNote()` – called when the note is first triggered (e.g. in response to a MIDI *Note On* message). Used to set the initial configuration of the note and initialise the note's member variables and objects (e.g. DSP components), based on the requested pitch and velocity, which are passed as arguments to the function.
- `MyNote::onStopNote()` – called when the synthesiser requests the note to end (e.g. in response to a MIDI *Note Off* message from the user), and used to manage the end of a note. By default, this function returns `true`, telling the synthesiser that it should terminate (and delete) the note immediately. However, it is often desirable to release the note more gracefully – for example, have the note slowly fade away, after its key is released, rather than abruptly cut to silence. In such cases, this function is used to reconfigure the note's variables or objects to execute the release, and then returns `false`, to tell the synthesiser not to terminate the note just yet. It is then up to the note object to signal the actual note end, using the `process()` function (see below).
- `MyNote::process()` – called *regularly* to process a block of audio for an individual note, used to synthesise and process the note's audio data. This function will be the focus of much of your programming, directly determining the fundamental sound of your synthesiser. As with other process functions (such as `MyEffect::process()`, as well as `MySynth::postProcess()`), this function processes audio data in blocks (or buffers). However, the data provided to the function (`**outputBuffer`) contains no input audio (silence), and only acts as a receptacle for the audio data you generate. Thus, the code structure, while largely similar to before, writes directly to the output buffer without having to read an input. Note, however, the return value, which is used to signal to the synthesiser whether the note should continue (`true`), or whether it can be deleted (`false`). Situations where you might want to prematurely terminate a note (without waiting for a *Note Off*) include one-shot sounds that don't recycle or repeat (e.g. drum hits). See also `onStopNote()` (above), for other scenarios where you need to manually control the termination of a note.

2.3 A Simple Synthesiser

By default, the *MySynth* code project is configured to provide a very simple synthesiser, based on a single sine wave generator. The sine wave is generated using a DSP component, called `Sine`, provided by the STK code library. The library provides many basic DSP components, including other types of signal generators, as well as filters, basic synthesis models, and effects processors, which we will discover, combine and extend during these practicals. However, all these DSP and code objects are designed to be used in a manner similar to `Sine`, which the template project demonstrates the use of:

- In SynthNote.h, a `Sine` object, named `signalGenerator`, is added to the synthesiser by declaring it as a member variable of `MyNote`. The sine wave generator's own implementation (within the STK library), defines and handles much of its own initialisation and setup automatically, without you needing to manually initialise it. Note, however, how it is added to the note object rather than the synthesiser object, as each individual note requires its own sine wave generator (at its own frequency).
- When a note is triggered, the `MyNote::onStartNote()` function is used to handle the `pitch` and `velocity` requested by the original MIDI *Note On* message. The pitch is converted to a fundamental frequency using a mathematical expression, which the `Sine` object is then set to use via its `setFrequency()` member function. Other DSP

objects will have similar member functions to configure their use, depending on their audio processing role. Note how both `fFrequency` and `fLevel` are stored as shared class variables so that they can be used (and changed) later – in the latter case to scale the amplitude of our output to control the volume of the sine wave.

- Finally, in the `MyNote::process()` function, we simply call the `Sine` object's `tick()` function for each sample (i.e. within the `while` loop), to get successive values of the sine wave, which are scaled by `fLevel` and placed in the output buffer.

Note how, compared to previous practicals, all the calculations for managing the phase position and calling the `sin()` function are hidden (in the STK source files), making the code easier to read and allowing you to focus on synthesis design. This demonstrates one of the key advantages of *object-oriented programming*, allowing you to abstract away complexity.

Before continuing, have a play with the synthesiser using either keyboard, and observe the various real-time analysis plots in the plugin's UI, which includes a waveform display (time domain), spectrum plot (frequency domain) and sonogram (time and frequency domain). While these don't have the detail or re-configurability found in packages like Amadeus Pro or Audacity, they offer some insight into how your synthesised sound is made up, and provide a useful diagnostic tool for tailoring your plugin to specific timbres or identifying bugs.

If you are unsure about any aspect of the template, or have questions about how the example code works, feel free to ask the module tutors.

3 Adding More Harmonics

In this exercise, you will add harmonics to the sine wave to create a more complex tone. As you should have noticed in the visual plots, sine waves only have a single frequency component. However, we can layer several sine waves to create a harmonic series, which are not heard as separate, distinct pitches, but instead grouped into one pitch and heard as a new timbre. This process of adding sound waves together is an example of *additive synthesis*. The following sections will walk you through the synthesis of both sawtooth and square waves, while also teaching you how to create new DSP and code objects from existing ones.

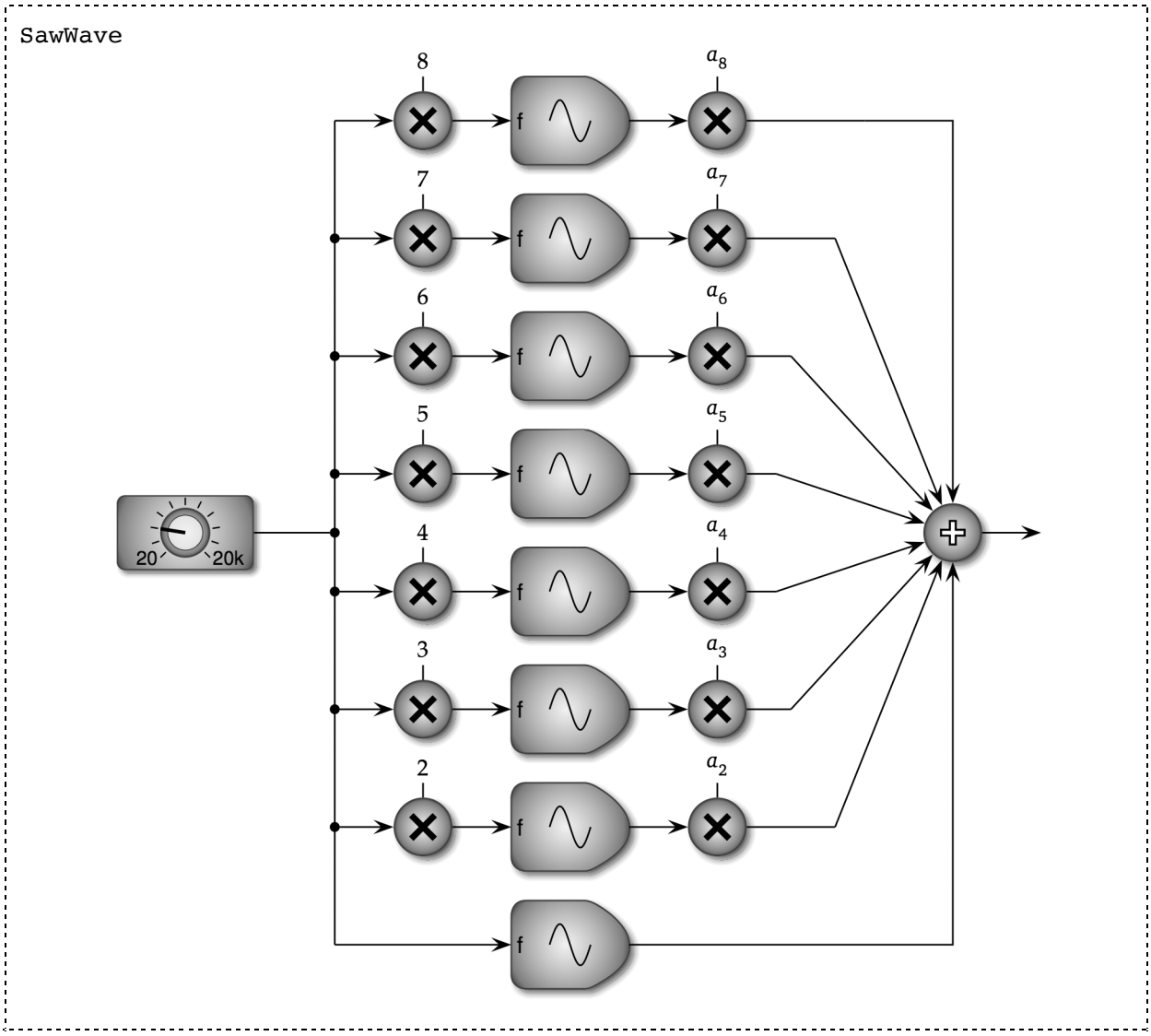
3.1 Creating a SawWave Object [BASIC]

Section 2 highlighted the advantages of grouping code components together in separate objects, in order to abstract and break down the complexity of the overall synthesiser. While we could add extra harmonics by using an array of `Sine` objects in `MyNote`, a better approach is to create a new object to represent our new tone, and place all the variables, initialisation code, and audio processing inside it. Then we can simply use this new object in `MyNote`, in place of the existing `Sine` object, without changing any existing code in `SynthNote.cpp`.

```
class ClassName
{
public:
    void memberFunction(){
        // code to execute
    }

private:
    int memberVariable;
};
```

The example code shown (inset, left) illustrates the general form of a class declaration, for an imaginary class, named `ClassName`, containing a private member variable, called `memberVariable` (an integer that cannot be accessed outside the class), and a public member function, `memberFunction()` (a function that can be called outside the class, but can itself access the class' private member variable).



The above block diagram shows all the processing functionality that our object will encapsulate, effectively bundling all the components into a single processing block that takes a frequency input and provides an audio signal output. In this respect, creating an object is much like creating a sub-patch in Max.

Create an object to represent a new sawtooth signal generator:

1. In `SynthExtra.h`, declare a new class called `SawWave`, with labels for public and private sections. Do not forget the braces { } or semi-colon (;) at the end of the class!
2. To keep track of the number of harmonics to generate, add the following compiler macro above the class declaration:

   ```
   #define MAX_HARMONICS 8
   ```

   Now, the compiler will automatically substitute the number 8 whenever it sees `MAX_HARMONICS` – so that in the future, if we want to change the number of harmonics, we only have to change this macro, rather than everywhere we'd used the number '8'.
3. Our complex tone will be built by layering multiple sine tones, representing each harmonic. Add an array of `Sine` objects in the class' *private* section of your object:

   ```
   Sine harmonic[MAX_HARMONICS];
   ```

4. We want our class to be a drop-in replacement for the `Sine` object, so it must mimic the same functions, and provide similar functionality. Declare and define the following functions (which are used in `SynthNote.cpp`), inside your new class. Some tips are provided to help you code the bodies of each function, but in each case, you will generally want to use a `for` loop to iterate over each of the `Sine` objects:

   `void reset()`

   This function is used to set the phase position to the start of the wave. Since your wave is simply a combination of sine waves, your version simply needs to call the corresponding `reset()` function on each of the `Sine` objects it contains – i.e. `harmonic[0]`, `harmonic[1]`, `harmonic[2]`, and so on …

   `void setFrequency(float frequency)`

   This function is used to set the frequency of the tone. A sawtooth wave comprises the *fundamental frequency* (which corresponds to the perceived pitch) plus all the remaining frequencies in the corresponding *harmonic series* (i.e. 1x, 2x, 3x...) Your version of this function needs to set the frequencies of each of `Sine` object, to correspond with the frequencies of the harmonics (as pictured in the diagram).

   `float tick()`

   This function generates the audio by calculating the next audio sample. It is called repeatedly by the synthesiser, returning one floating-point sample value at a time. Your version of this function will call the corresponding `tick()` functions of each `Sine` object, adding and appropriately scaling each returned value (representing each harmonic), mixed into a single floating-point value. To model the decreasing harmonic amplitudes in the sawtooth, scale each harmonic by 1.0 / *n*, where *n* is the number of the harmonic, beginning at 1 for the fundamental.

5. Your sawtooth signal generator is now ready to be used. In the declaration of `MyNote` (in SynthNote.h), simply change the type declaration of the `signalGenerator` object to use your `SawWave` class type, rather than the `Sine` type.
6. Run the plugin and (initially, with a low volume to protect your ears from potential bugs) audition your saw wave generator. Be sure the oscilloscope (waveform display) shows a roughly saw-like shape, and that the spectrum and sonogram (frequency plot) show a regular series of decaying harmonics.

3.2 From Saw to Square [BASIC]

Once you have a saw wave object, converting it to generate a square wave is easy. Since both are based on arranging multiple frequency components in a harmonic series, you can re-use most of the code you have already written:

1. In `SynthExtra.h`, copy and paste the `SawWave` class code, to create a duplicate below the existing declaration, and rename the new class `SquareWave`.
2. The principal difference between saw and square waves is that square waves contain only odd harmonics. Thus, you can approximate a square wave simply by altering the `tick()` function to set only odd harmonics (i.e. skipping the even harmonics).
3. Change your `signalGenerator` object, in `MyNote`, to use your `SquareWave` class.
4. As before, audition the sound made by your new plugin and observe it in the oscilloscope to check if the wave has a square shape.

5. You may notice the sound is not as bright or harsh as a typical square wave. This is because we are only generating eight harmonics, and thus the higher frequency harmonics are not present. However, the sound does resemble that of a well-known musical instrument. If you recognise the timbre, make a note of the instrument:

6. Now, return to the definition of `MAX_HARMONICS` (in SynthExtra.h), and change it to 16. Then rebuild and re-run the plugin, to see how the sound and visual look of the wave has changed. It should now sound brighter and more recognisable.

3.3 From Copying to Subclassing [CORE]

Copying and editing the sawtooth object to create a square wave object is easy and straightforward, but results in a lot of duplicated code. To avoid this, we can use another powerful concept of object-oriented programming, called *inheritance*. This allows us to tell the compiler that the new object we are creating is mostly based on an existing object, so that we are only required to write definitions for new parts and/or parts that have changed. The new object becomes a *subclass* of the original – also referred to as the *child* and *parent* class, respectively – and we can add functions to replace (or *override*) the original's.

To adapt your implementation of `SquareWave` and inherit code from `SawWave`:

1. Change the first line of the class to read:

```
class SquareWave : public SawWave
```

2. Delete the functions that do not change – `reset()` and `setFrequency()`. These will be inherited from the parent `SawWave` class.
3. Remove the entire `private` section, including the array of `Sine` objects, which are also inherited from the parent `SawWave` class.
4. Finally, a small change is required in the parent `SawWave` object. A parent class' `private` variables and functions are inaccessible from outside the class (even for child classes), so `SquareWave`'s replacement `tick()` function will not be able to access the `Sine` objects. Instead, we must put these objects in a `protected` section, enabling access from child classes such as `SquareWave`, but not from other unrelated classes. To do this, change the `private:` label to a `protected:` label, in the `SawWave` class.

Your plugin should now build and run exactly as before. However, notice how much more concise the code is. Moreover, if you were to later improve (or find a bug in) one of the shared functions (e.g. `reset()`), you only need to fix it in one place, as the fixed version will be inherited by derived classes automatically. As software projects (including synthesisers) grow in complexity, such techniques help keep the code manageable and easier to maintain.

(Advanced/Optional) Classes should be logically designed to make code as easy to read as possible. We exploit the similarity of saw and square waves in our two objects, but their relationship is slightly misrepresented: a square wave is not "a class of" saw wave – rather, they are both types of additive synthesiser based on the harmonic series. It may thus be more logical to have a parent class (e.g. called `Additive`*) from which both signal generators are then derived, and which supplies the common or default functions to its children.*

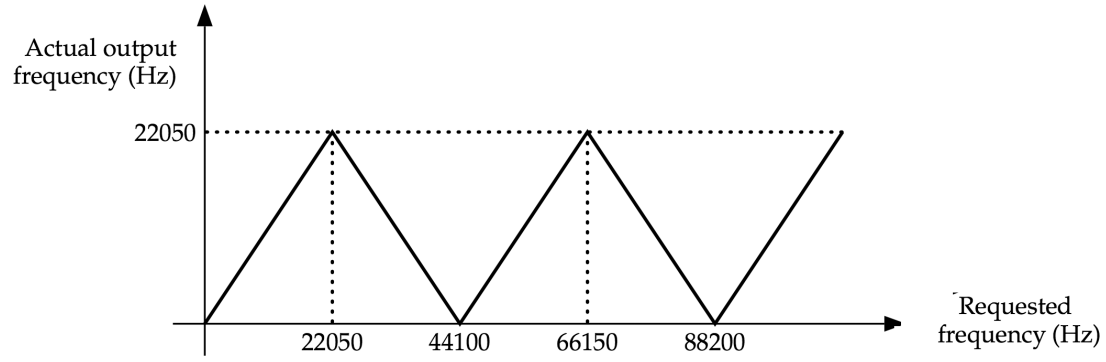
To add further clarity, we can label the functions we are overriding (in the parent), and those that override them (in the child). Functions we expect to be overridden are called ***virtual*** *functions, and it is good practice to label them as such. The following example makes clear the relationship (and can also help the compiler flag mistakes in your code!) …*

```
class Additive {
    virtual float process();
};
```

```
class SawWave : public Additive {
    float process() override;
};
```

4 Aliasing and Band-Limiting [CORE]

Aliasing occurs when frequency components are introduced to a digital audio stream above the Nyquist frequency (equal to half the sample rate). As frequency partials rise beyond this limit, they "wrap around" (the frequencies are reflected around the Nyquist frequency) and start falling in frequency instead of rising. When these hit 0Hz, they wrap around again and start rising. At a sample rate of 44100Hz, the effect is as follows:



4.1 Aliasing Effects

In the signal generators we have developed, there is nothing preventing the frequency of the harmonics going beyond the Nyquist limit, and frequency artefacts (*aliasing*) may be heard, especially at higher notes:

1. Using either your `SawWave` or `SquareWave` classes, set the number of `MAX_HARMONICS` to 16 (to make it easy to produce aliasing) and then build and run the plugin.
2. First, play notes by moving up the virtual keyboard with the mouse, listening for the change in tonal quality. At a certain point, the frequency partials start being reflected around the Nyquist limit. The frequencies onto which they are reflected are no longer integer multiples of the fundamental, so will sound more dissonant. The more harmonics that are reflected, the more dissonant and less musical the tone.
3. Now, switch to the `Spectrum` plot and again move gradually up the keys of keyboard using the mouse, watching the peaks of the harmonics. The built-in plot shows the entire frequency range, from 0Hz up to the Nyquist limit (e.g. 22050Hz).
    - As you move beyond E4, you should see more and more upper harmonics become reflected, falling in frequency instead of rising.
    - Later, as you move beyond F#5, the reflected harmonics fall below 0Hz and are again reflected, beginning to rise again.
    - As you approach the higher registers (above C6), the dissonance of the aliasing noise increasingly overwhelms the pitch of the original musical tone.

4.2 Band-limited Oscillators [CORE]

To avoid aliasing effects, signal generators should avoid generating harmonics above the Nyquist frequency. Our signal generator objects calculate the frequencies of a note's harmonics, and assigns them to individual `Sine` objects in `setFrequency()`. Before we add the output of each `Sine` object to the overall mix in `tick()`, we can check a harmonic's frequency and act accordingly (e.g. choosing not to add it to the mix, if it's above the Nyquist limit):

1. In the `tick()`, use the `getSampleRate()` helper function to work out the Nyquist limit and assign it to a temporary local variable.
2. Update the code that mixes the harmonics together so that only harmonics with frequencies below the Nyquist limit are included in the mix. Use the `Sine` object's `getFrequency()` to find out what frequency it's set to.

   Notice how `setFrequency()` *and* `getFrequency()` *functions are used to set the frequency of a* `Sine` *object. If you look at the definition in* `Helpers.h`*, you'll see that the variable used to actually store the frequency (*`frequency`*), is "protected" and inaccessible to the outside. This ensures that* `Sine` *retains control over its internal state – and that all frequency changes are run by the class'* `setFrequency()` *function, which updates the actual variables used to generate the sound.*
3. Build and run the plugin, and check that the sound is no longer aliased. Compare the quality of the tone to that before, using both your ears and the `Spectrum` plot.

It should be noted that while the inharmonic frequencies of aliasing generally give rise to dissonance and unpleasant timbres, inharmonic frequency components (partials) make an important contribution to the timbre of most acoustic sounds and real-world instruments, and can significantly increase the range and richness of tone colours possible through synthesis, as we shall investigate in future practicals.

The STK library also includes band-limited sawtooth and square wave oscillators for you to use in your synthesisers. While the oscillators developed during this practical are accurate anti-aliased digital oscillators, the STK versions use a more advanced and efficient technique, designed to model characteristics of analog synthesisers, including some inharmonic qualities. To use them, edit `MyNote` to use either a `Saw` or `Square` object, in place of the existing classes.

5 Finally

Make sure that you archive a copy of all the source files edited today, or backup the entire `src` folder. Use zip compression to create archives (.zip files) of each practical/exercise.

It is also recommended that you re-read this practical handout before the next session, to reinforce the concepts introduced, as they will form the basis for future sessions.