Student Name: Emil Sønderskov Hansen
Student ID: 20196042

Signal Processing for Interactive Systems
Mini-Project: Real-time YIN-Pitch detection for FMOD/Unity

AALBORG UNIVERSITY

1 Introduction

This paper introduces a real-time pitch detector based on the YIN algorithm. The pitch detector is developed in C++ and utilizes the FMOD Plugin API[1] to export a dynamic library (i.e., plugin) for the middleware FMOD for implementation in a 3D environment using the Unity game engine[2]. The Unity scene presents a 3D environment with an out-of-tune guitar. Users can adjust sliders to tune the guitar based on real-time pitch feedback (i.e., estimated fundamental frequency) from the plugin running in the background.

2 Estimation of fundamental frequency: Methods

An essential characteristic of sounds we perceive as music is pitch. Pitch is a perceptual phenomenon based on the fundamental frequency of the signal (meaning the shortest time interval where a signal repeats itself or τ) [2]. Hereby a periodic signal will obey $x_n = x_{n-\tau}$. If a signal is not perfectly periodic (which is most non-computer generated signals), the error (e_n) of the signal can be found by solving for 0, giving $e_n = x_n - x_{n-\tau}$ There are several methods for estimating the fundamental frequency of a signal, below is two described, and which is chosen for this implementation.

Autocorrelation Method: The Auto-Correlation Method is a widely used technique for pitch estimation and is particularly effective for perfectly periodic signals. The method involves calculating the auto-correlation function, which measures the similarity between a signal and a delayed version of itself (i.e., x_n and $x_{n-\tau}$) [2]. The method is derived using Mean Square Error (MSE). The final equation is: $R(\tau) = \frac{1}{N-\tau}\sum_{n=\tau}^{N-1} x_n x_{n-\tau}$. In autocorrelation, τ is often called the lag and aims to find the index n with the highest correlation. However, the autocorrelation method suffers from inaccuracies when a signal is not perfectly periodic (i.e., noise or variations in amplitude).

The Harmonice Method: The Harmonic Method is based on analyzing a signal in the frequency spectrum (i.e., its Fourier transform). A signals Fourier transform is given by $X(\omega) = \sum_{n=0}^{N-1} x_n e^{-j\omega n}$ [2]. Here $x(n)$ is discrete, however, $X(\omega)$ is a complex continuous function and needs to be discretized (e.g., by a Fast Fourier Transform). If we say that a signal is not perfectly periodic (as seen in the time domain above), the error of a signal in the frequency domain can be given by $e_n = x_n - \sum_{l=1}^{L} Acos(\omega ln + \phi)$, where L is the number of harmonics. Like the autocorrelation, the final equation is also derived using the MSE and is given by $\omega = \sum_{l=1}^{L} |X(\omega_0 l)|^2$ [2]. The downside of this method is that the amount of harmonics in the signal has to be known. This can differ greatly for different musical instruments, which makes the method difficult to use for different applications [2].

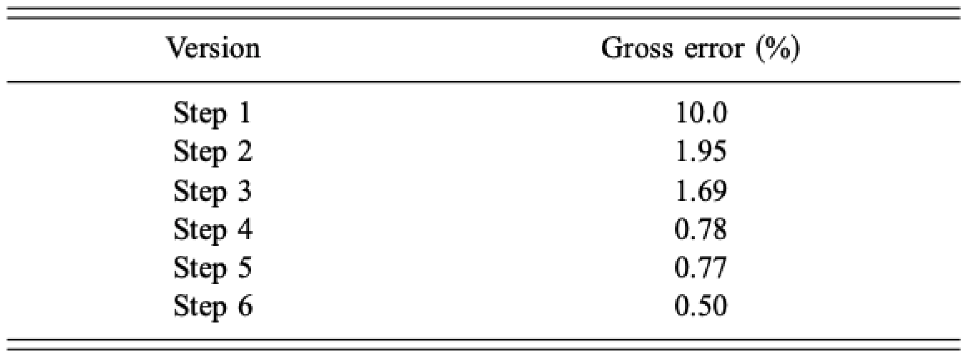
[1]https://www.fmod.com/docs/2.00/api/plugin-api-dsp.html
[2]https://unity.com/

The pitch estimator presented in this paper will rely on autocorrelation, as the amount of harmonic L will be unknown and should not restrict the use of the algorithm. To combat the inaccuracies of the autocorrelation method, it will however, be based on the paper *"YIN, a fundamental frequency estimator for speech and music"* by A. de Cheveigné et al [1], proposing several improvements (i.e., extensions) to traditional autocorrelation. The method is known as YIN.

3 YIN-Pitch detection and C++ implementation

3.1 Introducing YIN algorithm

The article introduces a method called YIN for the estimation of the fundamental frequency. This method combines autocorrelation and cancellation techniques to produce a more accurate estimation than solely using autocorrelation or even Fourier transform. The article presents a series of extra steps which decreases the gross error of the pitch detection algorithm (see figure 1). Because of the time frame of this mini-project and the desired result, this paper will solely look at the first 4 steps; however, as seen in the table, this provides a very low gross error. The whole section is based on the paper [1].



Version	Gross error (%)
Step 1	10.0
Step 2	1.95
Step 3	1.69
Step 4	0.78
Step 5	0.77
Step 6	0.50

Figure 1: The different steps of YIN and the error of pitch detection for each step

3.2 Step 1: Autocorrelation

The first step of the YIN algorithm is standard autocorrelation. The autocorrelation in YIN is described the same way as in section 1. However, they also slightly altered the equation, which is often used in signal processing for real-time purposes:

$$r(\tau) = \sum_{n=t+1}^{t+W-\tau} x_n x_{n+\tau}$$

In this case, the correlated signal is shifted forward (i.e., $n + \tau$), and the window size (i.e., the part of the signal to estimate) is subtracted by τ. This is beneficial for real-time purposes, as the buffer size of the system often defines the window size. The window size must therefore be subtracted by τ to not look for samples outside of the buffer. Based on this equation, a C++ method was generated, as seen below:

```cpp
float PitchDetector::calculateAutocorrelation(std::vector<float>& audioBuffer, int timeStep, int lag)
{
    float autocorrelation = 0.0f;
    for (int t =  timeStep; t < audioBuffer.size() - lag; t++)
    {
        autocorrelation += audioBuffer[t] * audioBuffer[t + lag];
    }
    return autocorrelation;
}
```

3.3 Step 2: Difference function

The next step improves the autocorrelation function. It firstly defines that the invariance of a periodic signal over a specific time shift (i.e., $x_n - x_{n-\tau} = 0$) is also the same when averaged over a window size and squared $\sum_{n=t+1}^{t+W}(x_n - x_{n-\tau})^2 = 0$. However, the averaged and squared signal becomes less sensitive to changes in amplitude, meaning it will perform better when a signal is not perfectly periodic. In contrast to the autocorrelation, it is now the lowest output which determines the time shift τ, which is estimated to be the fundamental frequency. The paper later defines the difference function in terms of the autocorrelation function:

$$d(\tau) = r(0) + r(t+\tau) - 2r(\tau)$$

By using this equation, the following C++ method was generated, as seen below:

```
float PitchDetector::differenceFunction(std::vector<float>& audioBuffer, int t, int lag)
{
    return calculateAutocorrelation(audioBuffer, t, 0) + calculateAutocorrelation(audioBuffer, t +
        lag, 0) - (2*calculateAutocorrelation(audioBuffer, t, lag));
}
```

3.4 Step 3: Cumulative mean normalized difference function

The difference function still has its flaws, as when no limit is set to the lower search range, it will often choose a dip at $\tau = 0$ or a big dip at the first harmonic. A solution to this problem is using the cumulative mean normalized difference function (CMNDF) which divides the difference function of τ by the average (from 1 to current τ). Additionally, the equation returns 1 when $\tau = 0$. When τ is not equal to zero, the equation is as follows:

$$CMNDF(\tau) = \frac{d(\tau)}{(1/\tau)\sum_{n=1}^{\tau} d(n)}$$

Based on this equation, the C++ method was generated, as seen below:

```
float PitchDetector::CMNDF(std::vector<float>& audioBuffer, int t, int lag)
{
    if(lag == 0)
        return 1;
    else
    {
        float sum = 0;
        for(int n = 1; n <= lag; ++n)
        {
            sum += differenceResults[n];
        }
        return differenceResults[lag] / sum * lag;
    }
}
```

This method requires the results from the difference function (i.e., from $0 : \tau_{max}$) to be saved in a vector beforehand.

3.5 Step 4: Absolute threshold

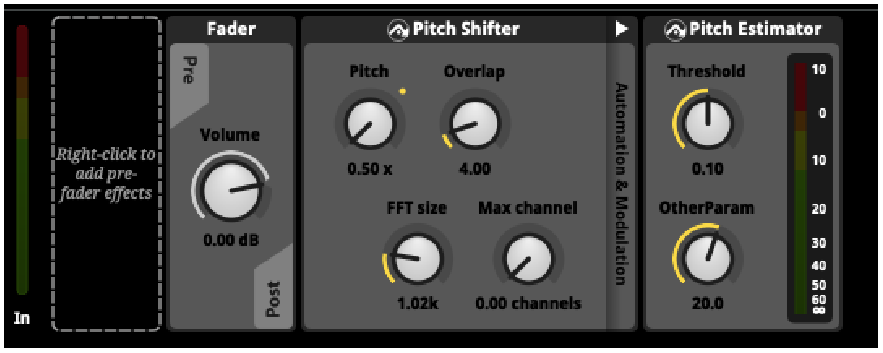
There is still one problem that has not been fixed by the CMNDF which is that the method will often choose a subharmonic, which causes a deeper dip, than the actual period. To solve this absolute threshold is used. Instead of choosing the τ giving the lowest value of the CMNDF, a threshold is used (often 0.1) and instead, find the smallest value of τ giving a minimum lower than the threshold. If no value is found, it will use the index with the lowest value from the CMNDF. This brings us to the final method in C++, which is the actual pitch detector method, and is generated as seen below:

```cpp
float PitchDetector::estimatePitch(std::vector<float>& audioBuffer) {
    bool pitchFound = false;
    float lowestCorr = 0.0f;
    int index = 0;
    int minLag = minBounds;
    differenceResults.clear();

    for(int i = 0; i<=maxLag;++i)
    {
        differenceResults.push_back(differenceFunction(audioBuffer, 1, i));
    }

    for (int lag = minLag; lag <= maxLag; lag++) {
        float cmndfVal =CMNDF(audioBuffer, 1, lag);
        if(cmndfVal < threshold)
        {
            index = lag;
            pitchFound = true;
            break;
        }
        if(cmndfVal < lowestCorr || lowestCorr == 0.0f)
        {
            if(lowestCorr != 0.0f && !pitchFound)
                pitchFound = true;

            lowestCorr = cmndfVal;
            index = lag;
        }
    }

    if(pitchFound)
        return static_cast<float>(sampleRate) / static_cast<float>(index);
    else
        return -1.0f;
}
```

As seen, the method tries to find the first CMNDF value under the threshold; if none is found, it uses the index (i.e., τ or lag) of the lowest value. To get the actual frequency, we divide the number of samples shifted (i.e. lag or τ) estimated to be the period of the fundamental frequency, with the sampling rate of the system.

4 FMOD and Unity implementation

The above methods from section 3 is part of a C++ class called **pitchEstimator**. This class is instantiated as an object in C++ called **plugin**, which implements callback methods from the FMOD API. The callback method provides ways to get the sample rate and buffer size of the system at creation, along with input buffers and output buffers at each process callback. The pitch detector object analyzes the input buffer and then passes this directly to the output buffer. The plugin class can then be exported as a dynamic library which can be used as a plugin in FMOD Studio and Unity (see figure 2 for the plugin in FMOD). All C++ code is attached to this delivery.



Volume 0.00 dB
Pitch 0.50 x
Overlap 4.00
FFT size 1.02k
Max channel 0.00 channels
Threshold 0.10
OtherParam 20.0

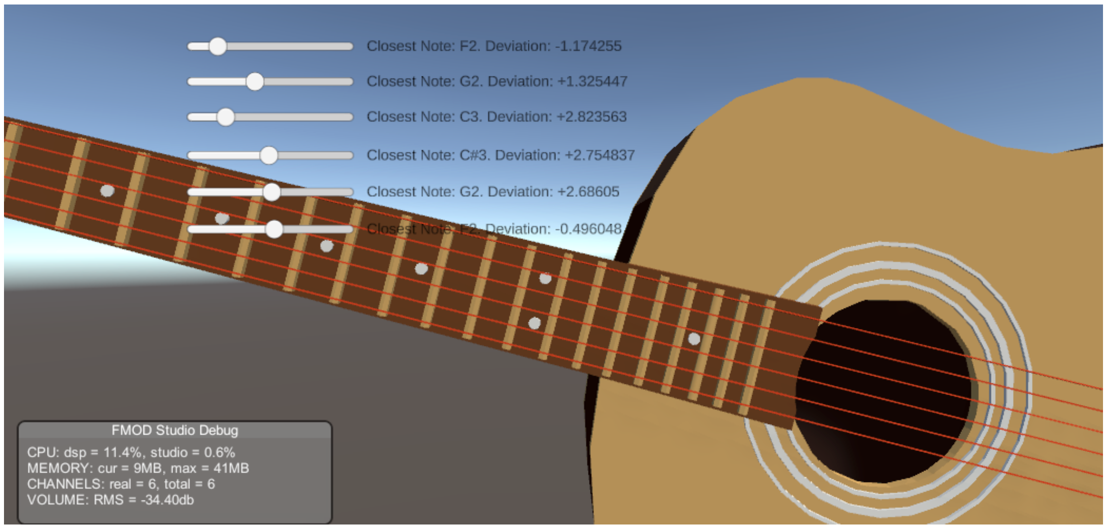
Figure 2: The plugin called ”Pitch Estimator” in FMOD STUDIO attached to an event along with a pitch shifter

In FMOD Studio, the plugin can be attached to events, analyzing the audio placed within the event. To get the frequency output of the pitch estimator, the DSP instance of each event instance must be loaded in Unity in a *C#* script. The method to achieve this can be found in the *C#* scripts attached to the delivery.

In Unity, a demo scene was generated with an out-of-tune guitar. The user can play the notes by dragging their mouse over a string. By adjusting sliders, the user can tune the guitar based on real-time feedback from the pitch estimator. The user can see which note the frequency is close to and how much the frequency deviates from that note (see figure 3). This is achieved by having a pitch shifter on each event controlled by the slider. The min and max value, along with the beginning value of the slider, is randomized.

5 Improvements, and future work

The algorithm works nicely for the case of tuning the guitar, however, some setbacks could be addressed in future work. Because of the real-time implementation, the maximum lag (i.e, τ_{max}) is limited to be bufferSize/2, as otherwise the window size seen in section 3.2, will increase outside of the size of the buffer. Because of this, it cannot estimate lower fundamental frequencies. E.g, in the case of a buffer size of 512 samples (i.e. $\tau_{max} = 256$) and a sample rate of 44100 Hz, the lowest frequency to estimate is $44100/\tau_{max} \approx 172hz$. In the case of a guitar, the notes E1, A, and D have frequencies below. In this implementation, A and D string is detectable by setting a sample rate of 22050 Hz, which is not ideal but helps with detection. A solution would be to not process a buffer for each process callback from FMOD, but instead add the buffer continuously to a vector, and wait until the size is large enough for lower frequencies to be detected.



```
Closest Note: F2. Deviation: -1.174255
Closest Note: G2. Deviation: +1.325447
Closest Note: C3. Deviation: +2.823563
Closest Note: C#3. Deviation: +2.754837
Closest Note: G2. Deviation: +2.68605
Closest Note: F2. Deviation: -0.496048
```

```
FMOD Studio Debug
CPU: dsp = 11.4%, studio = 0.6%
MEMORY: cur = 9MB, max = 41MB
CHANNELS: real = 6, total = 6
VOLUME: RMS = -34.40db
```

Figure 3: A screenshot from gameplay in unity. User can adjust sliders to tune guitar up or down, based on real-time feedback.

References

[1] Alain de Cheveigné and Hideki Kawahara. “YIN, a fundamental frequency estimator for speech and music.” In: *The Journal of the Acoustical Society of America* 111 4 (2002), pp. 1917–30. URL: `https://api.semanticscholar.org/CorpusID:1607434`.

[2] Mads Græsbøll Christensen and Mads Græsbøll Christensen. “12. Pitch Estimation”. In: *Introduction to audio processing*. Springer, 2019.

6 Appendix