Real-time pitch detection for singing using frequency-domain methods and the Web Audio API

Fredrik Wasström
Master's thesis in Computer Engineering
Supervisors: Antti Haimi, Kristian Nybom
Faculty of Science and Engineering
Åbo Akademi
2025

Abstract

Receiving correct feedback is vital for learning anything new. Learning to sing can be difficult for a variety of reasons, but knowing when one hits the right notes — and when one does not — is a good start. Computers have for a long time helped people with learning, be it through e-learning platforms, forums for connecting with experts, or lately, with large language models for the ability to almost converse with an expert at any moment. Computers have also been aiding in tuning instruments, such as guitars, for some time. This work explores frequency-domain methods and post-processing techniques for pitch estimation, to enable users to receive instant feedback on their singing. System architecture for such a pitch detector is then planned and outlined, and implemented using the Web Audio API for use on the web. The pitch detector is tested using audio from real choir singers for a reliable reference. The resulting pitch detector lays a good foundation by being accurate, but falls short in certain aspects.

Keywords: Pitch Detection, Audio Processing, FFT, Web Audio API, Harmonic Product Spectrum

Table of Contents

1 Introduction

To tune a guitar, someone with perfect pitch perception can simply pluck the string while turning the corresponding tuning peg and stop when the sound is correct. A person without perfect pitch would need to use a tuning fork or another sound source they know to be correct and compare the two until the difference is negligible. Both these processes are examples of pitch detection, one is absolute, and the other is relative. With the evolution of computers, this process has been made easier with both dedicated devices for tuning guitars and smartphone applications. In either case, some computational mechanism takes a sound as input and informs the user what the sound is or what needs to be done to correct it. How a computer can detect pitch has been explored in depth and has yielded many approaches. The pitch of a sound, how high or low it sounds, is tied to how fast a signal oscillates. Some methods look at the signal directly while others transform the signal in some way which makes certain methods of pitch detection easier to conceptualize and apply.

1.1 Motivator for the topic

For my bachelor's thesis, I created a web application which displays sheet music, plays the notes of the sheet music (with some limitations) and includes pitch recognition for grading the singer's ability to hit notes. Most of the application worked very well and everything I set out to implement worked well, except for the pitch detection. I used a neural network-based model called CREPE that ran as a part of a JavaScript library called ml5.js. I ended up with this approach because, from testing various pitch detectors, this one seemed to be the most reliable. It still was not very accurate and would often give back the wrong note which hurt the user's ability to learn. The focus of the work was on the entirety of the applications, and the scope combined with the lack of expertise on my part made me not explore the pitch detection part further.

1.2 Problem statement

The purpose of this work is to explore how the fast Fourier transform may be used for real-time pitch detection, and what additional techniques are required to achieve reliable pitch detection. The algorithm should be able to detect pitches from a microphone stream more or less in real time and should output precise notes. It should ideally be used in a web application, as this makes the application available on a wide range of devices but can also be used on-the-go, for example, for choir entrance tests or joint practice. The goal of this work is to create a reliable pitch detector that can be integrated in an existing application. The end user of that application should be able to trust the result of the pitch detection.

2 Introductory Fourier analysis

In the beginning of the 1800s, Joseph Fourier was working on a physics problem called the heat equation, a certain partial differential equation. He had the idea of expressing the original function as a sum of sine and cosine functions as these would integrate and differentiate easily, making the differential equation easier to solve. He eventually developed and introduced the idea of Fourier series, a way of expressing a function as a sum of trigonometric functions [1].

2.1 Fourier series

If a function has a period of 2π, the Fourier series takes the form

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n cos(nt) + \sum_{n=1}^{\infty} b_n sin(nt)$$

and states that a function can be represented by an infinite number of sine and cosine waves with different magnitudes and coefficients, plus some constant term. To compute the coefficients, the following formulae are used

$$a_0 = \frac{1}{\pi}\int_0^{2\pi} f(t)dt$$

$$a_m = \frac{1}{\pi}\int_0^{2\pi} f(t)sin(mt)dt$$

$$n_m = \frac{1}{\pi}\int_0^{2\pi} f(t)cos(mt)$$

which may be derived using the properties of a few integrals of trigonometric functions.

A sawtooth wave is perhaps the easiest to demonstrate the computations on, as it is simply $f(t) = t$ over the interval. The constant term is $\frac{1}{\pi}\int_0^{2\pi} tdt = 2\pi$. The first a_m term is the result of $\frac{1}{\pi}\int_0^{2pi} tsin(1t)dt = -2$, which may be done using integration by parts. The subsequent a_m sine terms repeat the exact same step with different values of m. For this function, all cosine terms are 0, since $f(t) = tcos(t)$ is an odd function with respect to the midpoint of the interval.

Thus, the Fourier series for this kind of sawtooth wave is

$$\pi + -2sin(t) - 1sin(2t) - \frac{2}{3}sin(3t) - \frac{1}{2}sin(4t)...$$

This approach, in a general sense, can be used to find the coefficients for other periodic functions.

2.2 Fourier transform

The Fourier transform is an operation that takes a function in time domain (a signal as a function of time, like a sound) and outputs a function in frequency domain, a kind of description for which kinds of sinusoids make up the original signal. Figure 1 shows a signal and its corresponding frequency-domain representation. The right diagram is a frequency-magnitude plot where the x-coordinate represents frequency and the y-coordinate represents the magnitude of the complex output of the transform. The peaks in the frequency domain shows that the signal is composed of the sinusoids $10sin(2\pi * 50t)$, $2sin(2\pi * 150t)$, $5sin(2\pi * 300t)$ and a significant amount of noise. In other words, the signal contains frequency components of 50Hz, 150Hz and 300Hz.

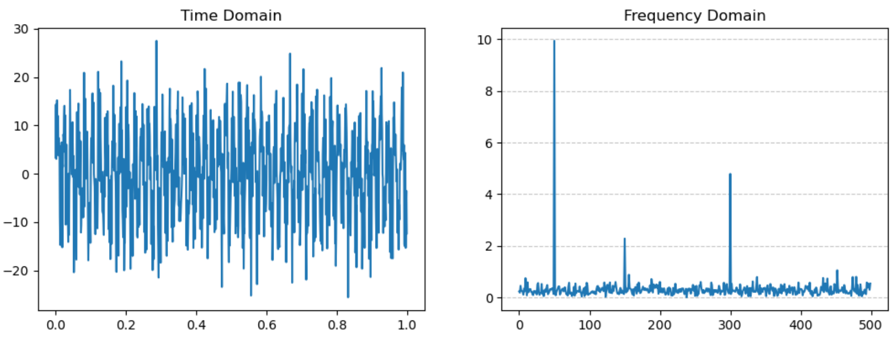

Figure 1: Some signal in the time-domain (displayed on the left) with added noise. Computing the transform with a discrete version of the Fourier transform reveals the main frequencies that make up the original signal in the time-domain (displayed on the right)

Even though the Fourier transform is an incredibly powerful function, its formula is compact.

$$\hat{f}(x) = \int_{-\infty}^{\infty} f(t)e^{-i2\pi xt}dt$$

It is worth noting at this point that *transform* refers both to the act of transforming the function between domains but sometimes also the output values are called the transform. The time-domain to frequency-domain transform is sometimes called the forward transform to emphasize the direction of the transform as the inverse (frequency-domain to time-domain) is also called a transform.

2.2.1 Fourier transform intuition

For the sake of simplicity, let f be a periodic function with a period of 2π. The following equations help explain how the Fourier transform works.

$$\int_0^{2\pi} sin(mx)sin(nx)dx = \int_0^{2\pi} cos(mx)cos(nx)dx = \begin{cases} 0 & m \neq n \\ \pi & m = n \end{cases}$$

and

$$\int_0^{2\pi} sin(mx)cos(nx)dx = 0$$

If $n = m$, $sin(nx), n \in \mathbb{Z}$ will interfere with $sin(mx), m \in \mathbb{Z}$, and integrating over the interval yields π. If $n \neq m$, integrating over the interval gives 0. When the signals are independent, when the definite integral equals 0, they are said to be orthogonal. When they are non-orthogonal, they correlate.

Intuitively, the Fourier transform is a function that checks correlation between another signal and all sinusoids. For example, transforming the signal $sin(440t)$, yields a 0 for every sinusoid, except $sin(440t)$, which means that $sin(440t)$ is part of the original signal.

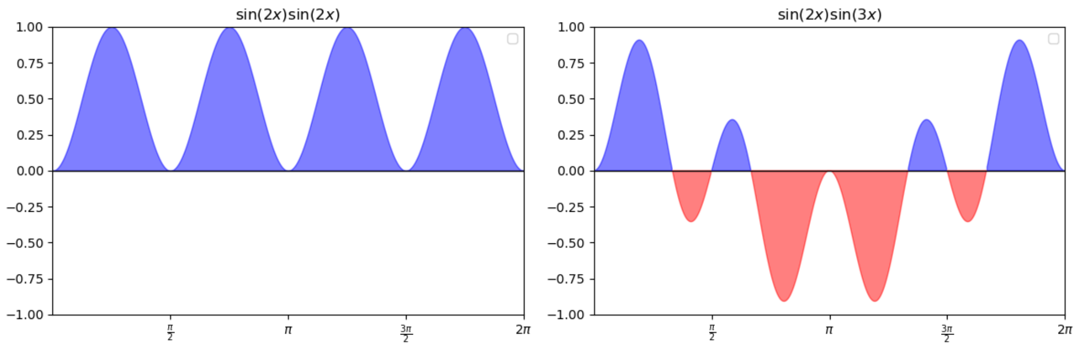

Figure 2: The graphs of $sin(2x)sin(2x)$ and $sin(2x)sin(3x)$ on the 0 to 2π interval. Matching n and m results in $sin^2(2x)$ which is always positive, resulting in a positive area whereas a mismatch causes equal positive and negative area, canceling out to 0.

2.3 Discrete Fourier transform

The Discrete Fourier transform (abbreviated as DFT), as the name implies, is the Fourier transform for discrete signals. Instead of integrating over the entire function domain, we sum the samples from the signal starting from the start of the signal at $t = 0$ to some $t = N$. The DFT for a signal x with N points is

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{i2\pi kn}{N}}$$

2.3.1 Matrix representation for the DFT computations

The DFT can be represented and computed by a matrix-vector multiplication.

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \\ \vdots \\ X(n) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^n \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2n} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{2n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^n & \omega^{2n} & \omega^{3n} & \cdots & \omega^{n^2} \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ \vdots \\ x(n) \end{bmatrix}$$

where $n = N - 1$ as the computation are zero-indexed and ω is the principle N-th root of unity $e^{2\pi i/N}$.

2.4 Inverse transforms

The time-domain signal can be used, transmitted and received, but is difficult to work with. The frequency domain, in contrast, is easy to work in, but makes little to no sense in many use cases. A C-major chord in the frequency domain cannot be played, for example. The forward Fourier transform allows the transformation from time to frequency domain, but once any modification (like high frequency filtering) is applied, the signal needs to be transformed back to the time domain to be useful. The operation that does this is appropriately called the inverse Fourier transform, IFT for short or IDFT, for the discrete variant.

The inverse Fourier transform is very similar to the forward Fourier transform:

$$f(t) = \frac{1}{\pi} \int_{-\infty}^{\infty} \hat{f}(\zeta) e^{i2\pi\zeta t} d\zeta$$

and the discrete inverse transform is

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{\frac{i2\pi kn}{N}}$$

the only difference is a normalization factor and negation of the exponent.

2.5 Computations

For pretty much any practical usage of the DFT, so many samples will be used that it's not feasible to compute by hand. The DFT computations thus need to be converted to an algorithm or similar programmatic construct a computer can understand and run.

2.5.1 Example DFT using the formula

The formula is technically easy to use both in manual computations and when writing software for it. Given a discrete input signal

$$x = [-0.01298834, 0.62287525, 0.64266088, 0.39309558, 0.55407458, \cdots]$$

which has a 100 elements, the DFT values can be computed with the formula

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{i2\pi kn}{N}}$$

. Starting with $k = 0$ $X_0 \approx -0.3215 + 0i$, a number with both the real and imaginary components close to 0, which indicates very little impact on the original signal. At $k = 5$ however, $X_k \approx 0.07725 - 24.5836i$ with a relatively large imaginary component. Repeating the same summation up to $k = 100 - 1 = 99$ the other values of X_k with relatively large imaginary components are at $k = 12$ and $k = 25$.

To obtain the frequency-magnitude plot, which simply will be referred to as the *spectrum* from here on, the magnitudes for each complex valued X_k are needed. The magnitude represents the length of the equivalent vector of the DFT values, and since the vector forms a right triangle, the length of the vector (or hypotenuse of the triangle) can be computed using the Pythagorean theorem $a^2 + b^2 = c^2$.

2.5.2 Computing the DFT programmatically

With the built-in complex data types and extensive mathematical and scientific computing libraries, implementing the DFT in Python is trivial. Leveraging Python's cmath library, the computations can be copied directly and 2 for-loops handles the summation and X_k indexing as shown below.

```
for k in range(N):
    for n in range(N):
        X[k] += signal[n] * cmath.exp(-2j * PI * k * n / N)
```

For a more primitive language like C, without complex exponentiation, the computations are still fairly straightforward to do due to Euler's formula that expands $e^{ix} = cos(x) + isin(x)$. The components can be separated into two separate data structures and so the imaginary unit can be dropped. A possible C implementation (without using a struct for complex numbers) is shown below.

```c
for(int n = 0; n < N; n++)
{
    real_components[k] += signal[n] * cos(-2*PI*k*n/N);
    img_components[k]  += signal[n] * sin(-2*PI*k*n/N);
}
```

These examples are purely illustrative, there is no reason to use the DFT in practice because there are algorithms that do the same transformation with significantly less computation, and the efficiency improvement grows with the input size.

2.5.3 Computing the inverse

Writing the inverse discrete Fourier transform in Python is as trivial as the DFT due to the cmath library. Like the formula, the algorithm is mostly the same, but with normalization and a negated exponent.

```python
for k in range(N):
  for n in range(N):
      signal[k] += filtered[n] * cmath.exp(2j * PI * k * n / N)
  signal[k] /= N
```

The inverse may be used after processing the frequency domain. For demonstration purposes, the signal has been filtered by removing any component that has a magnitude under a certain threshold. Figure 3 shows both the original signal and filtered signal in both time and frequency domain. One can observe less noise in the blue time-domain signal.

2.6 Fast Fourier transform

The DFT takes n^2 operations to perform as there are n outputs X_k and n amount of numbers to be summed. A fast Fourier transform (abbreviated as FFT) is any method that speeds up the computation of the DFT. This means that even if there is one Fourier transform and basically one DFT, there are multiple FFT algorithms. Arguably, very few users of the FFT cares about the implementation, they just expect the functionality of the DFT, but faster.

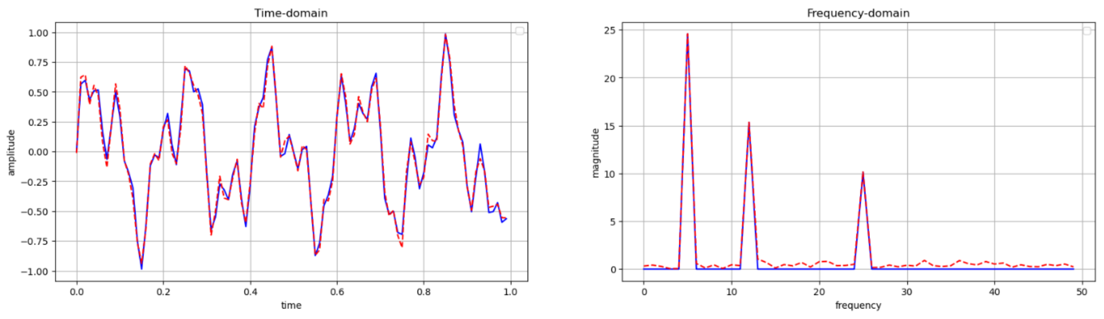

Figure 3: The time-domain and frequency-domain representations of the signal (red dashed line) and the filtered signal (blue solid line)

2.6.1 Time complexity

Time complexity refers to the asymptotic performance of an algorithm. Even though it often correlates with performance, it is ultimately theoretical and only considers the highest-order term, because it is the greatest contributor when the input size grows towards infinity. For example, consider an algorithm for multiplying 2 numbers. One such algorithm, commonly taught in elementary school, multiplies every digit of one operand with every digit of the other. The result of each multiplication is added to the previous, accounting for carrying, to form a single number. This process results in an array of numbers, each of which is a multiple of a power of 10. The final step of the algorithm is to sum these numbers. The first step takes N^2 operations, where N is the length of the operands (assuming they are the same length), and the second step takes N operations. This algorithm, thus, takes $N^2 + N$ steps but as N grows larger and larger, the N^2 term will contribute the most, so we say that the algorithm has a complexity of $O(N^2)$, which is referred to as *Big-O*. For example, multiplying two 1000-digit integers, the multiplication will take a million operations, and the summation will take 1000. Increasing the input size with 1 digit adds only 1 more addition, but it adds 1001 more multiplication steps.

Perhaps the most popular algorithms, the Cooley-Tukey FFT, has an algorithmic complexity of $O(n \log n)$, a significant speedup over $O(n^2)$ [2] [3]. Once in Big-O, the base of the logarithm does not matter, what matters is that it is logarithmic. An improvement from $O(n^2)$ to $O(n \log n)$ means that even with a mere 1000 samples, the speedup, purely in terms of time complexity, is approximately 100x. In a paper by James Cooley, he exemplified a computation with the Fourier transform on a dataset of 512,000 data points which he claims would see an approximately 12,800x speedup with the FFT [4]; 512,000 data points would take on the order of 262 billion operations to complete with an $O(n^2)$ algorithm which, with a modern personal computer, can take minutes to complete. An

$O(n \log n)$ algorithm takes a relatively measly 9.7 million operations and runs in milliseconds. This is akin to measuring the execution time of Quick sort to Bubble sort for an array of size 512,000 which indeed sees a speedup of this magnitude.

This speedup in execution time and theoretical time complexity improvement demonstrates the power of the FFT. Why this improvement is so important will, hopefully, be evident later. Essentially, the FFT is considered by some to be one of the most important algorithms of all time due to being able to perform very important heavy computations instantaneously.

2.6.2 The FFT algorithm

As previously mentioned, FFT is simply put any algorithm that computes the DFT *fast.* One of the easier to understand while still demonstrating the workings of the algorithm is the Radix-2 FFT. The Radix-2 FFT is easy to understand because it assumes the input size is a power of 2, meaning it's easy to recursively split into two. On a high level, the Radix-2 algorithm starts with splitting the input array into odds and evens. It then calls itself for the odds and evens, recursively splitting the array until it performs FFTs on arrays of length 1. On the way back up, it performs the DFT on smaller units of the original signal, utilizing the symmetry of roots of unity and values that it has already computed to perform multiple operations at once.

The following mathematical derivation of the Radix-2 FFT is adapted from [5]. It starts with splitting the DFT into two sums, one for the even indices and one for the odd

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} T^{2n} + \sum_{n=0}^{N/2-1} x_{2n+1} T^{2n+1}$$

where T_k is called the twiddle factor $e^{-\frac{2\pi ik}{N}}$. A common twiddle factor is factored out of the odd sum and the even and odd sums are denoted as E_k and O_k respectively.

$$X_k = E_k + T_k O_k$$

These functions E_k and O_k are periodic with a period $N/2$ which can be shown by showing that

$$e^{\frac{-2\pi in(k+N/2)}{N/2}} = e^{\frac{-2\pi in(k)}{N/2}}$$

which means that

$$E_k = E_{k+\frac{N}{2}}, O_k = O_{k+\frac{N}{2}}$$

and since this holds true for E_k and O_k, the following is also true

$$X_k = \begin{cases} E_k + T_k O_k, & \text{for } 0 \le k < N/2 \\ E_{k-N/2} + T_k O_{k-N/2}, & \text{for } N/2 \le k < N \end{cases}$$

This implies that the DFT can be computed in two layers (for lack of a better term), the lower $N/2$ terms and the upper $N/2$ terms. When computing the upper $N/2$ terms, i.e. when $N/2 \le k < N$, $X_k = E_{k-N/2} + T_k^{N/2} O_{k-N/2}$, which are the same O and E as the lower $N/2$ terms, just with an offset of N/2. The twiddle factor is raised to the power of N/2 because it too is in terms of k. We can thus split X_k into

$$X_k = E_k + T_k O_k$$

$$X_{k+N/2} = E_k + T_k^{N/2} O_k$$

which are the equations for the lower and upper $N/2$ terms. Finally, the twiddle factor is rewritten using the following property of complex exponentials

$$e^{\frac{-2\pi i(k+N/2)}{N}} = -e^{\frac{-2\pi i k}{N}}$$

which gives

$$X_k = E_k + T_k O_k$$

$$X_{k+N/2} = E_k - T_k O_k$$

Recall that

$$E_k = \sum_{n=0}^{N/2-1} x_{2n} T_k^{2n}$$

$$O_k = \sum_{n=0}^{N/2-1} x_{2n+1} T_k^{2n}$$

and that the DFT is

$$X_k = \sum_{n=0}^{N} x_n T^n$$

This means that when the element indices are adjusted, E_k and O_k are both DFTs essentially. To transform this purely mathematical notation into something that looks more like a complete algorithm, the core computations are implemented with a loop

```
do k = 0, N/2−1
    T = exp(−2*i * PI * k / N)
    X(k+1) = E(k+1) + T * O(k+1)
    X(k+1 + N/2) = E(k+1) - T * O(k+1)
end do
```

In Fortran, arrays are indexed starting from 1, this means that every array access needs an offset of 1. Before these computations, the evens and odds must be computed.

```
E = r2fft(input(1:N:2))
O = r2fft(input(2:N:2))
```

Any recursive algorithm also needs a base case to prevent the algorithm from looping indefinitely. In the case of the FFT, this is when the input array has a single value.

```
if (N == 1) then
    X = input
    return
end if
```

Putting everything together and adding variable declarations and initialization as well as the function declaration, a naive Radix-2 FFT could look like the following

```
recursive function r2fft(input, inverse_factor) result(X)
    complex, dimension(:) :: input
    complex, dimension(SIZE(input)) :: X
    complex, dimension(SIZE(input)/2) :: O, E
    complex :: T, i
    integer :: N, inverse_factor
    real :: PI

    PI = 3.14159265358979323
    i = cmplx(0, 1)
    N = size(input)

    ! Recursive base case
    if (N == 1) then
        X = input
        return
    end if

    ! Split
    E = r2fft(input(1:N:2), inverse_factor)
    O = r2fft(input(2:N:2), inverse_factor)

```

```fortran
    ! Combination step
    do k = 0, N/2-1
        T = exp(-2*i * PI * k * inverse_factor / N)
        X(k+1) = E(k+1) + T * O(k+1)
        X(k + N/2+1) = E(k+1) - T * O(k+1)
    end do

end function r2fft
```

This implementation assumes the input is a power of 2 but this is not handled anywhere. The program still runs, but to no surprise gives an erroneous answer. The algorithm can be visualized with a block diagram as shown in Figure 4.

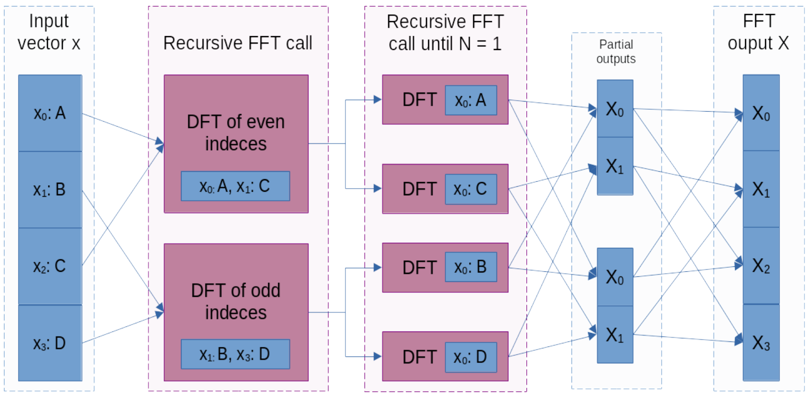

Figure 4: Diagram of a 4-point FFT. The process starts with splitting down the input vector to the recursive base case $N = 1$. After that the values are combined using the core computations of the FFT.

2.6.3 Time Complexity of Radix-2 FFT

The FFT is ubiquitous, and the algorithm has been explored in depth. Literature explaining and deriving the algorithm exists and following it to implement the algorithm in code is almost trivial. It is easy to overlook the ingenuity of the algorithm and be fooled by the simplicity of it when seeing it written down. Where the $O(n \log n)$ complexity comes from might not be entirely clear even when seeing the algorithm written out in code because, even though the divide-and-conquer method splits into two, multiple computations are done within each function call.

When taking a 4-point FFT, as shown in Figure 4, the last combination step definitely needs four computations. One can also observe that each of the 2-point FFTs (which is a subproblem of the 4-Point FFT) takes 2 computations. All in all, it takes eight computations to compute the 4-point FFT. The total number of computations for an 8-point FFT similarly is 8 plus 2 * number of computations of the 4-point FFT. As the FFT is a recursive algorithm, the total number of computations for an N-point FFT can be generalized to the following recursive formula

$$T(N) = 2 * T(N/2) + N$$

where N is assumed to be a power of 2 for simplicity. This recursive definition can be expanded until we get the recursive base case T(1). Since N is halved for every step, it takes precisely $\log_2 N$ expansions to reach the base case. Expanding the definition a few times (assuming N is sufficiently large)

$$T(N) = 2 * (2 * T(N/4) + N/2) + N = 4 * (T(N/4) + N) + N$$

$$= 4 * ((2 * T(N/8) + N/4) + N) + N = 8 * T(N/8) + N + N + N$$

$$= 8 * (2 * T(N/16) + N/8) + N + N + N = 16 * T(N/16) + N + N + N + N$$

reveals the pattern $T(N) = 2^k * T(\frac{N}{2^k}) + kN$, where k is the number of times the recursion is expanded to reach the base case. As $k = \log_2(N)$ the formula becomes $T(N) = 2^k * T(\frac{N}{2^{\log_2(N)}}) + kN$ which equals $T(N) = 2^k * T(\frac{N}{N}) + kN = 2^k * T(1) + kN$. As the 1-Point FFT performs no computations, $T(1) = 0$ and the final formula for the total number of computations for the N-point FFT is $N \log_2(N)$. In the context of Big-O, it only matters that the algorithm is logarithmic, the base won't matter in the same way that constants and linear scalars are dropped, which gives the well known O(NlogN).

2.6.4 Inverse FFT

Surprisingly, the IFFT is the FFT algorithm with $\omega = \frac{1}{n} e^{\frac{i2\pi kn}{N}}$, or in other words, the IFFT is the FFT with a negated twiddle factor, and a normalization added. In [6] it is shown that the IFFT algorithm can reuse the FFT logic, with a single line changed.

The Fortran code from earlier contained a inverse factor that was left unexplained. Below is an example of a simple abstraction for calling the forward FFT and inverse FFT but utilizing the same core r2fft function.

```fortran
function fft(input) result(X)
    complex, dimension(:) :: input
    complex, dimension(SIZE(input)) :: X

    X = r2fft(input, 1)
end function fft

function ifft(input) result(x)
```

```
    complex, dimension(:) :: input
    complex, dimension(SIZE(input)) :: x

    x = r2fft(input, -1)
    x = x / SIZE(input)
end function ifft
```

Both the FFT and inverse FFT are utilizing the same core Radix-2 implementation from earlier, but the inverse is computed with a negated twiddle factor based on the second argument. In the inverse FFT case, an additional normalization is also applied after the FFT computations are finalized.

This implementation can be tested empirically by first forward transforming a set of data points and then inverse transforming the transform and note that $x = ifft(fft(x))$ holds up to floating-point precision. When comparing a and $b = ifft(fft(a))$, all values were within $\epsilon = 1 * 10^{-30}$ when using quadruple-precision floating point variables and data structures.

Fortran is one language that has native complex arithmetic making it a very convenient language to implement complex number related algorithms in. In a language like C or JavaScript without such functionality, one way is to take the complex conjugate of the input, which is effectively equivalent to negating the twiddle factor. This can be verified using Euler's formula $e^{ix} = \cos(x) + i\sin(x)$. Cosine being an even function and sine being an odd function means that $e^{-ix} = \cos(-x) + i\sin(-x) = \cos(x) - i\sin(x)$ which is the complex conjugate of $\cos(x) + i\sin(x)$.

2.7 Applications of Fourier analysis

Fourier series were originally motivated by a differential equation in physics and even if it still is of great help for mathematicians solving similar problems, the true importance comes from the vast applicability of Fourier analysis combined with the significant performance improvement of the FFT. The Fourier transform allows the conversion between time and frequency domain which makes manipulation and analysis of arbitrary signals practical.

Signals are all around us, hidden in every day life. Merely accessing the internet using a wireless connection most likely involves the Fourier transform. The following chapter will briefly explore some of the applications of the Fourier transform to emphasize the importance of the idea.

2.7.1 Wireless communication

Several modern standards for wireless communication, like IEEE 802.11 WLAN (colloquially Wi-Fi), rely on Orthogonal Frequency Division Multiplexing (OFMD) [7]. In short, it is a way to pack discrete data (consecutive bits for example) compactly into a time-domain signal of orthogonal sinusoids. Orthogonal means that the signals do not interfere with each other and that integrating the product of several sinusoids over the period gives 0. This happens to be case when the angular frequency of two signals are two different integers.

OFDM needs a modulation scheme to convert data into sinusoid components. Quadrature Phase Shift Keying is one popular method, and it transforms the 4 combinations of 1's and 0's into 4 complex numbers, $\pi/2$ radians apart. There are different ways of assigning the values, for example, 00 may become 1, 01 becomes i, 10 becomes -1 and 11 becomes -i. These values are encoded in subcarriers which are then turned into the time domain using the IFFT in preparation for transmission. An FFT on the receiver's end transforms the transmission into frequency domain for demodulation. The frequencies and phases can then be extracted using the inverse of the modulation scheme to obtain the original data that made up the transmitted time-domain signal.

2.7.2 Multiplication

The FFT can be used to multiply two numbers, which after building a good understanding of the FFT, may feel wrong to someone learning about it. Not only can it perform multiplication, it can increase the efficiency of it. Multiplication is an algorithm that has a time complexity of $O(n^2)$ because each digit needs to be multiplied by every other digit. Using a fast Fourier transform, two numbers can be multiplied in $O(n \log n)$. The general gist of the method is to represent both numbers as polynomials, sampling them at a number of points, a number which must be large enough to uniquely define the polynomial, multiplying those points together, finding a new unique polynomial for the new set of points and converting the polynomial back to a number. That resulting number is the product of the two input numbers [6].

The sampling is what the FFT is responsible for. A polynomial in coefficient form can be written like $P(x) = \sum_{n=0}^{d} a_n x^n$. Sampling means that we evaluate the polynomial at some points. If the polynomial is evaluated at $\omega = e^{-\frac{i2\pi k}{N}}$, the resulting expression is $\sum_{n=0}^{d} a_n(\omega)^n = \sum_{n=0}^{d} a_n e^{-\frac{i2\pi kn}{N}}$ which is the DFT definition, where a is a discrete input signal with d points. This means the DFT is evaluating a polynomial at the nth roots of unity which also means that the

FFT can evaluate a polynomial at the nth roots of unity [8].

Even though this seems like a very convoluted process, for large numbers, this turns advantageous at inputs of length around 200 [8]. This means this method is practical in cryptography where it is fairly common to multiply extremely large integers.

In practice, it is not a trivial method to implement. The FFT (and IFFT) does indeed just work, but only for polynomials. The difficult part is converting back and forth between polynomial and integer. Each coefficient in a polynomial represents a digit in the integer, yet coefficients are not limited to the digits 0-9. There are multiple ways to do this conversion and one way involves modular arithmetic in conjunction with long addition. This method does not work for negative numbers, but an easy circumvention is to force both inputs to be positive and simply noting whether the output is negative. This can be done by checking if exactly one of the inputs is negative, in which case the output is negative.

3 Pitch detection using the Fourier transform

The purpose of this work is to study the Fourier transform for its viability as a pitch detector. The FFT for transforming a waveform into the frequency domain has been introduced and this chapter will focus on the properties of sound, introduce some musical terminology and how the FFT can be used to detect pitches from a time-domain signal.

3.1 Basics of sound

Sound physically is pressure propagation through a medium, a vibration which some things can produce, and some things can pick up. The word sound can be used to describe both the propagation itself and the phenomenon we feel when our ears react to the propagation. Headphones, strings of a guitar and vocal cords (together with the lungs) can produce sound and microphones and ears can pick up sound [9]. The propagation will have a certain strength at some point in the medium it travels through, which can be measured. This allows us to model sound as pressure change over time. At time t, the pressure at some point can be denoted as $f(t)$. This function over time can also be called a signal.

One of the simplest ways to create sound is connecting a speaker to a device that generates an analog current in the shape of a wave. The current, using an electromagnet, moves a membrane at the same frequency as the generator, causing the pressure difference around the membrane. This membrane displaces air at some rate which ears pick up as sound. For example, if the generator produces a 440Hz signal, the speaker's membrane will displace air at the same 440Hz. This displacement is propagated over air and is sensed by our ear as a pitch we call A4. Interestingly, the purity of the signal makes it sound harsh, and it is noise in the signal that gives warmth and beauty to the note. This is formally called timbre and it allows us to distinguish an A4 on a piano from an A4 on a guitar even though both are A4 sounds.

3.2 Music 101

Music terminology can be hard to define without circular definitions. There's also quite a bit of overlap with the same word meaning different things depending on context. A possible starting point would be defining the octave which is the difference between two sounds where one sound has twice the frequency of the other and is the largest interval in western music. If a sound oscillates at 440Hz, the sounds corresponding to 220Hz or 880Hz are an octave away.

3.2.1 Octaves

The name octave comes from the fact that it contains seven distinct base notes (the 8th forming the octave). This is called a heptatonic scale and when it sounds *nice*, it is called a diatonic scale (formally, it requires a certain structure of intervals). Diatonic scales include the familiar major (happy) and minor (sad) scales. The history and reasoning for the heptatonic scales in western music theory is not relevant for the work, but it is worth noting that it is not the only system for organizing sound.

Because the diatonic scales require a certain structure of intervals, the 7 base notes are not enough, so the octave needs to be divided further into what is called semitones. Semitone is typically defined as the smallest interval in western music and is typically 1/12th of an octave. Semitone is sometimes also used to talk about, not the interval, but the sounds that bound the interval. This non-interval semitone is sometimes also called a pitch and sometimes a note.

3.2.2 Notes and note names

Whle octave is used to describe an interval of 2 notes, we can define a first, second third etc. octave if we start from somewhere. A common way to label musical pitches is to use a letter to represent the note name and a number to indicate which octave it belongs to. This system is called Scientific Pitch Notation (SPN) and is used in Western music notation. It starts, not on an A, but on a C and depending on the convention used ends in either a B or an H. Figure 5 shows the octave bounds and how notes are ordered within the octave.

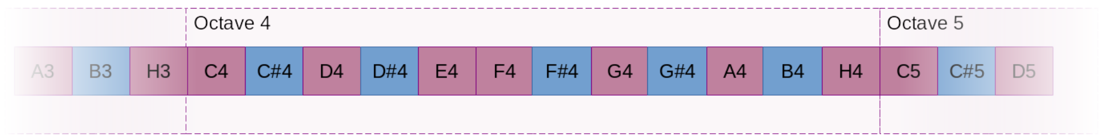

Figure 5: Notes of an octave. Base notes are marked with purple.

For example, the note which has twice (or half) the frequency of A4 forms an octave with A4. As the frequency is doubled, the other note is one octave higher than A4 so it is given the name A5. If the frequency is halved, it is named A3.

In western music, tuning typically starts from A4, which is defined in this system to have a frequency of 440Hz, and every other note is tuned relative to that. Tuning in this context essentially means finding the frequencies of the other notes so that everything sounds *right* (to avoid getting into unnecessary amount of music theory). A3, an octave lower than A4, will be half of 440Hz or 220Hz, and A5, an octave higher, will be double of 440Hz or 880Hz. The frequency for the other pitches can be approximated with equal temperament tuning, meaning that all semitones are equally spaced within octaves. This means that the frequency of any one pitch is exactly $2^{1/12}$ times the frequency of the previous one.

3.2.3 Flats and sharps

Flats and sharps are augmented notes. Sharps are denoted with a # symbol and flats with a ♭ symbol. Both flats and sharps are one semitone above the augmented note meaning there is quite a bit of overlap. For instance, a D#4 has the exact same pitch as a E♭4. The key difference between D#4 and E♭4 is context. It should be obvious, even to a person with no musical training, that when *playing in D-sharp*, there should be a D-sharp and not an E-flat even though they are the same pitch.

3.2.4 Terminology for readers

The base notes will be defined according to German/European music theory as

$$C, D, E, F, G, A, H$$

and B being $H\flat$. In this work, the term *note* will be used to describe the 12 distinct sounds within the octave because for the pitch detector, discerning the base notes and the sharps and flats is not of importance. Pitch will be used to describe the measurement/perception of how low or high a sound is.

3.2.5 MIDI numbers

For humans, it is easy to use the SPN labels for notes. C4, for example, is a simple pair of a letter and a number and gives users just enough information to know what it refers to. Computers on the other hand are different and instead have it much harder to analyze the string "C4" compared to pure frequencies. However the frequency value is often unnecessary because we usually only care about a discrete set of frequencies corresponding to notes, not all possible real valued frequencies. Musical Instrument Digital Interface (MIDI) is a technical standard for protocols, connectors and interfaces relating to music and provides among other things, simple integers for labeling notes. For instance, the first sound on a piano, A0, has the MIDI number 21.

3.3 Fundamental frequency estimation

Even though we can define a pitch to be a precise frequency, pitch is more of a sensation perceived by ears, and we can distinguish pitch from sound waves that are not simple oscillations. What we perceive as an A4 can contain a large amount of noise, or timbre. As it can be difficult to strictly define pitch, pitch detection may also be difficult to define. Pitch is, however, strongly tied to the fundamental frequency, which is the lowest perceived harmonic of a signal. Even though the timbre of a piano and a guitar are vastly different, which allows us to hear a difference between the two, if they play the same note, the fundamental frequency will be the same.

From a computational perspective, a possible definition is that pitch detection is the identification of the fundamental frequency. An additional computational aspect is to identify the corresponding note, because the sound names are more valuable to musicians than the frequencies expressed in hertz.

The fundamental frequency is typically the lowest frequency of a signal but, in some cases, we can hear a fundamental frequency that is not actually a part of the signal, a phenomenon appropriately called the missing fundamental. [10]. An example of this would be telephones that could not record below 300Hz. Male voices would still sound male, even though their fundamental frequency of around 150Hz was completely missing .

Mather 2006

3.3.1 The missing fundamental

The missing fundamental is in a sense an auditory illusion. It is a result of the constructive interference of harmonics that causes peaks with a period that is the greatest common divisor of the angular frequencies of the harmonics. This is shown in Figure 6 where the sum of several waves produce a wave with a prominent 110Hz oscillation. Even though the lowest frequency component is 220Hz, the prominent 110Hz oscillation is what we feel and hear.

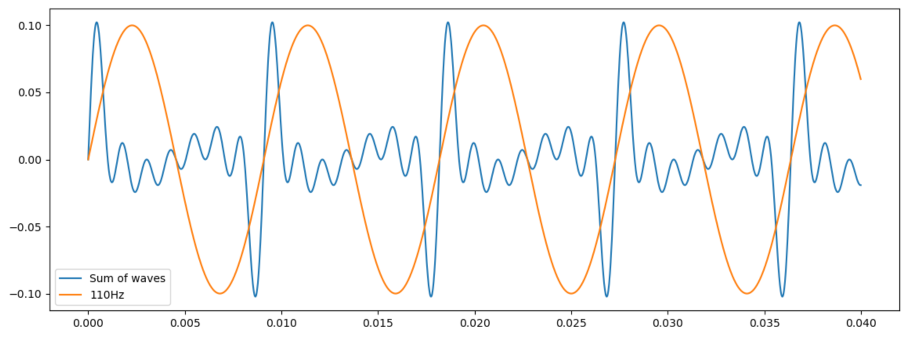

Figure 6: Constructive interference of harmonics creates the missing fundemental. Even though the blue signal only contains sinusoids with frequencies greater or equal to 220Hz, the most prominent part of the final signal has a period of 110Hz.

Pipe organs use this to their advantage. Creating extremely low sounds requires enormous pipes. Taking advantage of the missing fundamental allows relatively smaller organs to produce low sounds [11].

3.4 Pitch detection methods

Pitch detection methods typically fall into one of three categories: time-domain, frequency-domain, or statistical/machine learning methods. Some common time-domain methods include zero-crossing rate and autocorrelation. Zero-crossing rate is a method in which frequencies are extracted by keeping track of the rate at which the signal crosses the x-axis. Another is autocorrelation, in which the signal is compared to itself with a different phase.

This thesis will focus on a frequency-domain approach, where a Fourier transform is used to create a representation of the sound that is easier to analyze and process.

3.5 Transforming to frequency-domain

The first step in detecting pitch using the frequency domain is to transform the original signal into the frequency domain, and there are multiple ways to achieve that. The DFT and FFT were already introduced in the introductory Fourier analysis chapter, and as already established, some version of FFT should be used as the transformer since the DFT has terrible performance. How to apply the FFT is not yet clear, the background used dummy data to demonstrate the algorithm and its performance.

Disregarding the question of how to process binary audio files—even if the data were decoded—how should the data be processed? If the whole audio signal is given to the FFT, the spectrum will be a complete mess because real music or song contains more than a single note or chord.

3.5.1 Short-time Fourier Transform

The short-time Fourier transform (STFT), sometimes also called the windowed Fourier transform, is a method of analyzing a signal by looking at parts of the signal. The STFT is typically defined very similarly to the FT/DFT, but with a windowing function applied. The continuous STFT is usually defined like the following.

$$S(x(t), \tau, \omega) = \int_{-\infty}^{\infty} x(t) W(t-\tau) e^{-i\omega t} dt$$

where $x(t)$ is the input signal, and τ is the center of the window function W. The window function may be freely chosen, for example, a Gaussian window or Hamming window. Conceptually, this is computing the transform for a signal S_2 that is some signal S with a windowing function applied so that outside the window, the signal has 0 value. The window is then moved and the *next* value is computed. The discrete STDFT conversely computes the DFT for just some chunk of a discrete signal. Like the STFT, it then moves the window over some number of steps (called the hop length in discrete context) and repeats. If the hop length is shorter than the window size, the windows will overlap.

This sliding window introduces a new dimension to the output data—a time axis. This data constitutes a spectrogram and the typical way to visualize this data is with a heat map by compressing the 2D frequency domain to a column where intensity represents the magnitude value for a frequency. Figure 7 shows how the spectrogram is created.

It is worth pointing out that the STFT is not a special kind of Fourier transform but rather a method of applying the DFT/FFT (or continuous FT). This

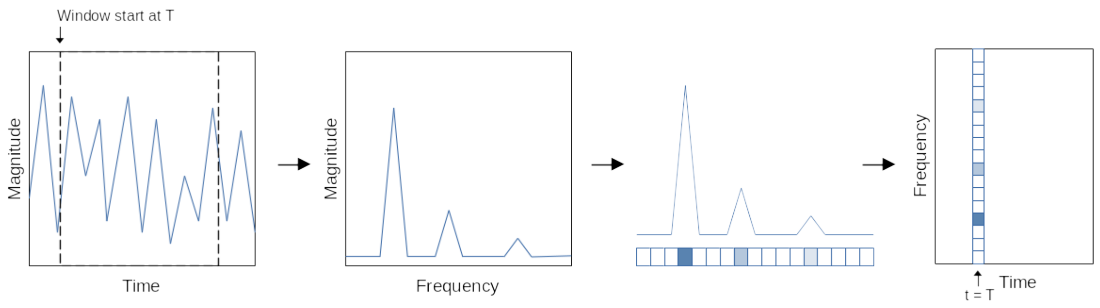

Figure 7: Diagram of how the STFT performs the DFT for one window which results in one column in the spectrogram. For ease of visualization, the frequency domain is typically visualized using color intensity for the magnitude.

means it exhibits the same behavior, but also that it has the problems, which will be introduced shortly.

For real-time processing, it is both essential and a somewhat redundant to talk about the STFT because the entire signal is never be available so there is no need for windowing. The data will be available as a stream (or chunks) and an FFT can be performed once a buffer is filled, after which the buffer is cleared. The buffer serves as the window, so as long as the transform is computed sufficiently often—which is necessary for real-time applications—the method, in a sense, implements a discrete STFT. The overlap, which is determined by the hop length, doesn't seem to be of much importance for the purpose of the estimation the fundamental frequency and makes little sense when the data becomes available with time. The overlap is important when considering note lengths and separation of repeated semitones [12], but does not seem to be a concern for just pitch detection. Consequently, the pitch detector will have a hop length the size of the window—or in other words, no window overlap.

3.6 Picking peaks from the spectrum

The Fourier transform is used to obtain the spectrum data, which can be used to plot which frequencies are part of the original signal. These frequencies that are part of the signal are called partials. Intuitively, for a lot of signals, anyone can look at the frequency-magnitude plot, choose peak, and claim that it is the fundamental frequency—and they would probably be right. Perhaps they are looking at the spacing of the largest peaks (if there are multiple) and, out of the peaks that fit the spacing, choosing the one that has the lowest frequency. This is a very straightforward approach and correct for some signals.

3.6.1 Harmonic Product Spectrum

Harmonic Product Spectrum (HPS) is an algorithm that works similarly to the intuitive approach. It is defined as follows:

$$Y = \prod_{r=2}^{R} X_r$$

where X_r denotes the signal X downsampled by r. Simply put, the HPS iteratively multiplies downsampled version of a signal. Downsampling in the frequency domain effectively compresses the frequency axis by the downsampling rate which means that previously integer multiples of some value kN end up in the same bins as N. When all the downsampled frequency-domain signals are multiplied together, correlated frequencies, or harmonics, form a very sharp peak [13]. This peak can be identified with a max function.

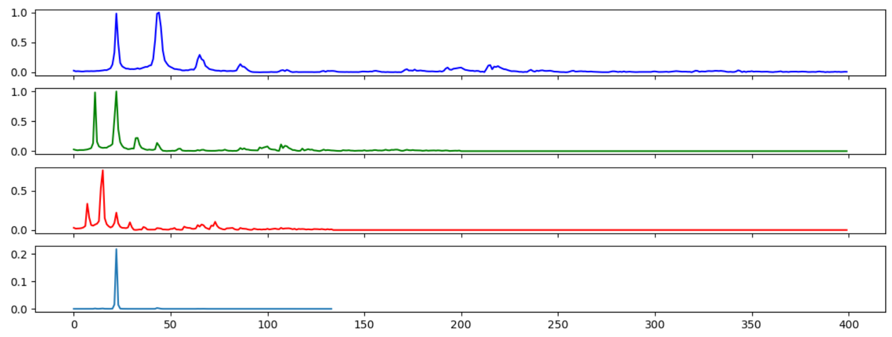

Figure 8: Plot of signal, downsampled versions of it and the final HPS. Harmonics align and form a peak.

HPS is a promising algorithm because it is simple and easy to compute, making it suitable for real-time analysis. As the HPS essentially looks at the ratio of the harmonics, it can find a missing fundamental. Even if the fundamental is part of the spectrum, it may not always be the strongest, in which case the HPS helps by enhancing it for easier discernment from noise.

The biggest problem with the HPS is that it does not perform well in the case of lacking harmonics [13]. This is likely not be a problem because human voices should be rich in harmonic content. In the more extreme case, where noise is dominating, the system would still compute the HPS and one largest peak would form somewhere, resulting in garbage data. A possible way to mitigate this is to check the spectral flatness (sometimes called tonality) of the signal and ignore the result if the flatness is too high. HPS is also prone to getting octaves wrong [14].

3.6.2 Constant False Alarm Rate

Constant False Alarm Rate (CFAR) is an algorithm that gives a dynamic threshold. It works by looking at some amount of reference cells that are beyond some gap around the target cell. For example, for cells 5, 6, 7, 8, 9, 10, 11, 12 and 13, the threshold given a gap size of 2 and reference cell count of 2 and the target is 9, the reference cells are 5, 6, 12 and 13, where as 7, 8, 10 and 11 are gap cells. The reference cells are then averaged (but other statistical methods can be used) and a bias may be applied. If the value of cell 9 is above the computed threshold, it is considered a target and noise if below. CFAR is typically used in conjunction with radar technology but could just as well be used to discern the partials in a spectrum.

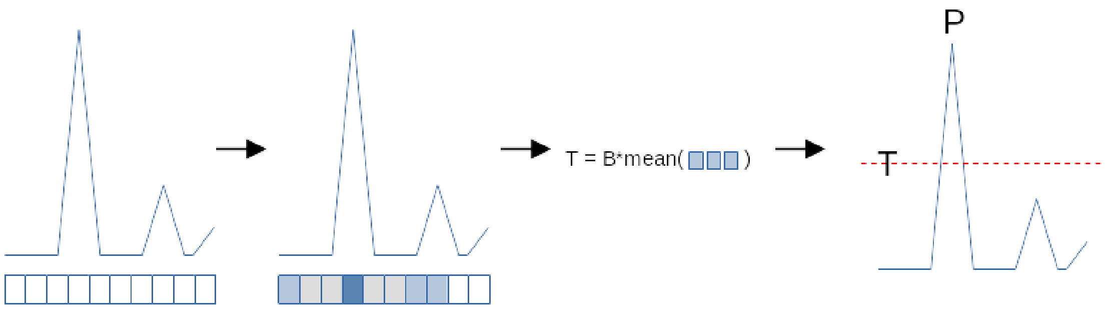

Figure 9: Diagram of the CFAR algorithm with 2 gap cells and 2 reference cells. The gap cells around the target point P are excluded but the reference cells in light blue are used for calculations. The reference cells are averaged which gives the threshold T for point P. As P is greater than T, P is regarded as a target.

To effectively use CFAR, the parameters must be carefully chosen. If the bias is too low, invalid targets may be considered targets and vice versa if the bias is too high. The gap and reference cells can be fairly flexible but not too large, as smaller harmonics may be considered noise compared to greater harmonics. The lowest pitch we want to estimate is around 50Hz and if we assume monophonic singing, the spectrum should mostly contain integer multiples of the fundamental frequency, which means that no peaks should ever be closer than 50Hz. With 4Hz bins, we can freely choose up to 12 gap and reference cells and not have a harmonic interfere with the CFAR computations of another [15].

As CFAR only finds the peaks, one problem with using it alone as a fundamental frequency estimator is that additional post processing is necessary. Since HPS already works well for extracting the fundamental frequency, it is chosen as the primary algorithm for the fundamental frequency estimator. CFAR could, however, be used to give a rough number for peaks, which may be used to dynamically adjust the number of iterations the HPS uses. If CFAR finds very few peaks or even just one as in the case of the electric piano in Figure **??**, the result could be used to completely change peak picker as HPS has issues with signals without harmonics.

3.7 Problems with the FFT

One problem with the FFT that [10] addresses is the size of the frequency bins. This is also noted for the STFT, limiting how short the STFT can be for achieving sufficient frequency resolution [12]. The FFT doesn't find precise frequencies, it finds bins of frequencies with the size S/N where S is the sampling rate and N is the FFT buffer size. As frequency grows exponentially (doubles every octave), higher notes are spaced further apart. This means that for lower notes, the frequency bins need to be significantly smaller as shown in Figure 10. Base singers may need to go very low, around E2 (87Hz) and in order to avoid notes in this range to fall into the same bin, the window size needs to be very small.

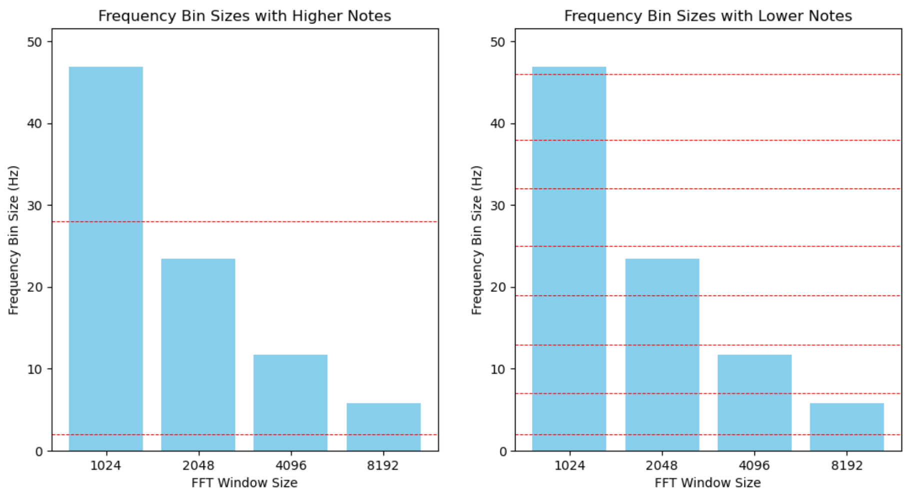

Figure 10: As the FFT window shrinks, different notes may fall into the same bin. Bars represent the bin sizes for various window sizes at 48kHz sampling rate and dashed lines show semitone intervals (the exact note values are irrelevant). For notes in the 4th octave, the window size can be smaller, but in the 2nd octave even 8192 samples is not enough to discern all pairs of semitones because the bins are larger than those semitone differences.

Base singers typically go as low as E2 and the difference between E2 and F2 is barely 5Hz which is smaller than the frequency bin of 8192 samples at 48kHz. They do not necessarily fall into the same bin but it would be safer to compute the FFT with a 16384 window size. This halves the frequency bin size and definitely accomodates differences even an octave lower. This introduces a significant amount of increased computation and is quite unnecessary as base singers do not realistically go much lower than E2. If the FFT implementation allows (many implementations require a power of 2), the window size should lie somewhere between 8192 and 16384 to lessen computations. For a 4Hz bin, $48000/x = 4 \iff x = 48000/4 = 12000$, 12000 samples would be enough to discern the lowest notes.

This introduces another problem, which has implications for real-time pitch detection which is that collecting 12000 samples at 48kHz takes 0.25 seconds, which arguably is not real-time anymore. As the bin size is $B = S/W$ means that $W = S/B$ and the latency (time taken to fill the FFT window) $L = W/S = \frac{S/B}{S} = 1/B$, the bin size and latency are inversely proportional, both can not be minimized. For a bin size of 4, it will necessarily take 0.25s to collect the FFT samples. This follows that naively transforming audio makes real-time pitch detection for base singers impossible because the detection latency will inherently always be too long with sufficiently fine frequency resolution.

3.8 Note lengths and timings

Note lengths are defined as fractions in relation to the beat. The beat is the recurring pulse of the music which relates to the tempo and the time signature. In common time (which is noted with 4/4), the beat is implied to have the same length as a quarter note. A whole note, thus, is sustained over 4 beats and similarly two eight notes can be played in a single beat. Music is in other words always written using relative lengths and the overall speed is defined by the author often using natural language. The musician should respect the relative note lengths, but they may or may not respect the pace set by the author.

How a piece of music is performed often deviates considerably from how it was written. There are several reasons for this. For one, sheet music is not an algorithm but closer to natural language. Much is up for interpretation, most notably as far as pace and dynamics are concerned. When a musician encounters the tempo marking, for example, *andante*, they just have to know and/or feel what it means because no consensus seems to exist about what *andante* means. Most sources give a range, but even the ranges are different between sources. Another term, *accelerando* simply means to accelerate. By how much or how quickly is not specified. Musicians may ignore them completely and even make them up, to introduce their own touch to the performance. In the simpler case, some sections may just be dragged as the musician or conductor wants. Figure 11 visualizes two time series using color intensity instead of magnitude. This is for demonstration purposes only and how they were created is not of importance for the moment. The upper time series is the definition of the piece as the sheet music states. The time series below is one performance of the piece. The graphics reveal how the pace in the rendition deviates from the sheet. It should be easy to see a strong similarity between the two. They are not quite identical, but their structure is identical, and they have similar substructures.

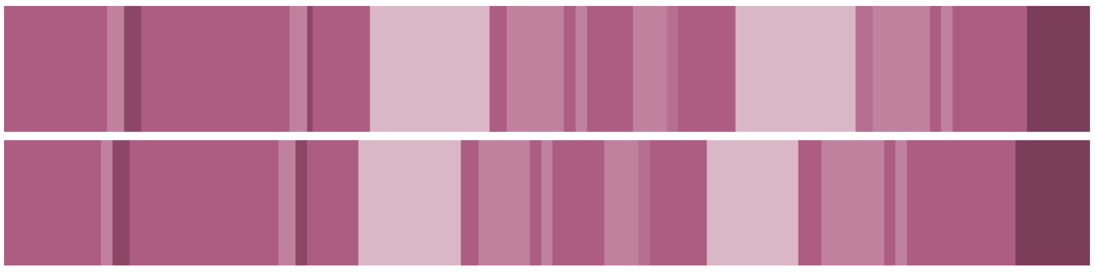

Figure 11: The time series of a piece as the sheet music suggests (above) and of a performance of said piece (below)

3.8.1 Time series similarity

The similarity of two time series can be measured with Dynamic Time Warping. Timing and DTW is important to understand because it helps understanding the computational problems in creating the pitch detector. As DTW works with a longer signal (as opposed to a single sample), it is not suitable for real-time detection. DTW is thus used for developing and testing the detector using recordings for consistency and conveniance and is only a tool for the analysis of the notes detected by the pitch detector.

3.9 Real-time monophonic pitch detection

The focus of this work is on real-time pitch detection, specifically for the purpose of rating and aiding a user in practicing their singing. To clarify, real-time means that the input of the system is a stream and the system processes small chunks as they come up and forget anything that it has already processed just like a human would listen to a short window in time, decide the pitch for that window and move on, not caring about the previous windows and not knowing anything of upcoming windows. The acceptable latency of *real-time* is contextual. For rendering, it should be around 33 milliseconds to achieve 30 frames per second, where as for real-time tracking in logistics, it is probably enough to update every minute or even hour.

What real-time means in this particular context can be debated and studied further, but the goal is just that it is fast enough so that the user can react to their mistakes. This is akin to a teacher telling a student that they are off to give the student instant feedback, as opposed to saying something like “there was a mistake in the 3rd note in the 11th bar” which may be little to no help for a beginner. According to humanbenchmark.com, the mean reaction time is 284ms based on their collected 81M benchmarks [16]. The Human Benchmark benchmark is based on visual stimuli, but it is noted that the reaction to auditory stimuli is slightly faster [17]. The goal for this work is significantly below the mean visual reaction time at around 100ms. In any case, even if humans can react to auditory stimuli faster than this, as the purpose is to give live feedback to the user on their own singing rather than having the user react to someone else’s singing, the user is limited by their reaction speed to the feedback (which is visual stimuli).

3.9.1 Zero-padding

One challenge with real-time pitch detection that emerged from the problem with the size of frequency bins was that for detecting lower base notes, the window size needed to be around 12000. However, collecting 12000 samples at 48kHz takes 0.25s, which arguably is not real-time anymore. It takes 0.33s if for any reason the window size needs to be 16384. If the window size is made smaller, the resulting FFT will have larger bins than what is necessary for detecting lower frequencies.

A popular approach to increasing frequency resolutions is to pad the end of the signal with zeros. Zero-padding makes the signal longer and while it necessarily introduces frequency components (more bins with a longer input signal), the peaks remain intact. A larger FFT window, while increasing computational complexity, allows for better frequency resolution. At 48kHz, 16000 samples are enough to achieve a 3Hz frequency bin. If out of these 16000, 10000 are zeros and 6000 legitimate data points, the FFT window can be ”filled” in a mere 125ms which is a more acceptable latency for real-time detection. Figure 12 shows how zero-padding affects the spectrum.

3.9.2 Reducing sampling rate and window size

As the bin size is the ratio of the sampling rate to the number of samples, $B = S/N$, reducing both N and S keeps the bin size small. If only the window size is reduced, the bins grow in size and frequency resolution is lost. With zero-padding fixing the latency issue, why not reduce the sample rate as well to ease computational load? Apart from the system limitations of the sampling rate, like the W3C web audio specifications only guaranteeing 8000 to 96000Hz, another

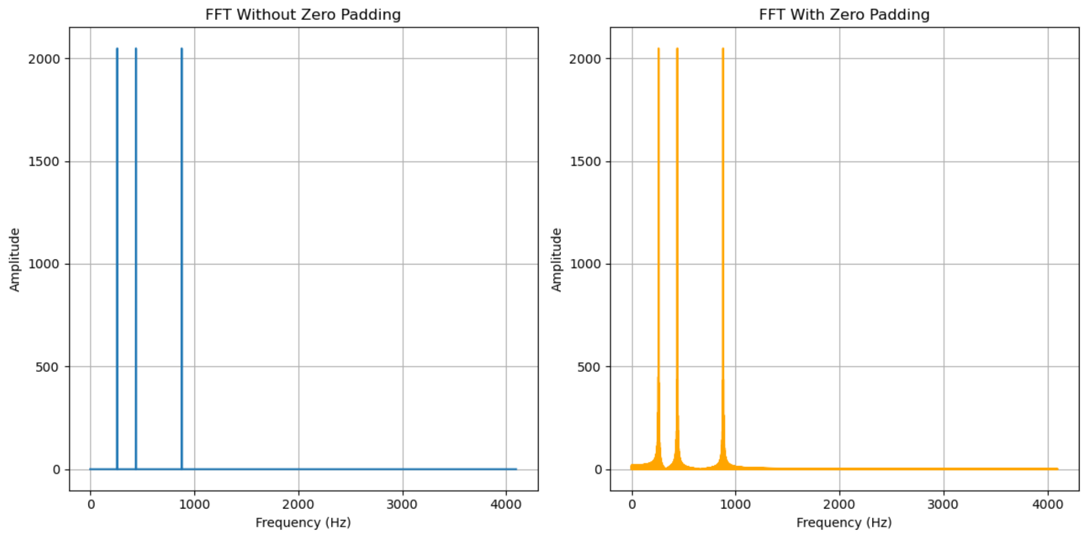

Figure 12: Spectrum contents of a signal consisting of 260, 440 and 880 frequency components. While the zero-padding introduces a small amount of frequency data around the peaks, the peaks themselves are not affected.

issue is aliasing. Aliasing, in short, is misrepresentation of higher frequencies as lower frequencies.

The Nyquist-Shannon sampling theorem states that there is no loss of information when sampling at $2B$ hertz if the signal contains frequency components that are less than B hertz. It, thus, implies that if the sampling rate is lowered, higher frequencies may be misrepresented. For example, if a signal is sampled at 44.1kHz, the biggest frequency component that can be represented without ambiguity is 22050Hz, which is also called the Nyquist frequency for the given sample rate. If the signal contained a 30kHz component, which would exceed the Nyquist frequency, it would be perceived as a 14.1kHz component. This can be computed using $|s - f|$ where s is the sample rate and f is the frequency component. Conceptually, one could imagine drawing the frequency range on a piece of paper and folding it along the Nyquist frequency. After the fold, the 30kHz and 14.1kHz components would end up in the same position.

As the pitch detector in this work has no reason to play back the sound, the waveform can be processed in almost any way as long as no crucial information is lost or skewed. Compared to the lowest male voices, there's seems to be less consensus on the highest female voice, but a pitch higher than G6 at around 1567Hz seems to be uncommon. A more typical is C6 at around 1046Hz. To be on the safe side, the ability to detect 1567Hz components can still be considered a valid requirement. If the pitch detector needs to detect a 1567Hz frequency, the sampling rate needs to be no smaller than 3134Hz. However, the human voice is rich in harmonics, if these are aliased, they will be presented as lower notes which in the best case would get the octave wrong and in the worst case get completely wrong notes.

The sampling theorem clearly states what we can and can not do with the signal, but it all depends on the harmonics of voices. Clearly, the sampling rate can not be 3134Hz because a significant amount of the relevant information will be in the harmonics above 1567Hz, but from limited testing, the harmonics certainly don't approach the 22050 Nyquist frequency. Even for female voices, the spectral content seems to be very weak above 10000Hz and almost indistinguishable from noise above 15000Hz. The sampling rate thus could probably be lowered to 30kHz or even 20kHz which would allow an FFT window size of 8192. This could be considered and explored if system resources were a bottleneck, but a modern laptop is performant enough to handle 16394-point FFT without slowing down the program.

3.10 Finding the nearest note

Western tuning is done in equal temperaments, meaning the octave is evenly divided in 12. Using A4 (440Hz) as a reference, one can calculate every semitone's frequency with a signed index from A4 using the following function.

$$f(i) = 2^{i/12} * 440$$

For example the -2nd semitone using the formula is approximately G4 which indeed is 2 semitones under A4. To avoid negative indices, the index is shifted by 69, the MIDI number for A4. To find the frequency of the MIDI number i the following function can be used.

$$f(i) = 2^{\frac{i-69}{12}} * 440$$

To find a MIDI number based on frequency, the function can be rewritten as a function of f.

$$M(f) = 12 * \log_2(f/440) + 69$$

While the MIDI numbers are discrete, the derived formula interpolates between the MIDI numbers. If the input is a frequency that does not have a corresponding semitone, like 450Hz, the result is $M(450) \approx 69.4$. This can be rounded to the nearest integer.

The function that will be used in the pitch detector is

$$M(f) = round(12 * \log_2(f/440) + 69)$$

3.10.1 Octave shift for tenors

Tenor's parts are written in a treble clef but sung an octave lower. This transposition is sometimes explicitly written with a little 8 under the clef sign, but in many cases this is not the case. This discrepancy is important to note when comparing the detected notes against the written notes. Some programs do encode the octave shift in the music XML when using the clef with the little 8 underneath. For instance, a C5 in the sheet music is in fact a C4 in the music XML, assuming the program that generates the music XML implements this feature (tested with MuseScore). However, if it is missing, the user would be getting errors when singing correctly because the system thinks the user is always an octave too low. A simple solution would be to simply let the user decide if the octave shift should be applied or not as it can be difficult—or even impossible—for a computer to figure it out without contextual metadata. The octave shift is global, so comparing notes correctly would simply involve subtracting 12 from the detected MIDI number before making the comparison.

3.10.2 Time keeping

How a piece is performed may deviate significantly from how it is written down by the author. If the purpose was offline processing, where the entire signal is given and the system processes it as fast as it can, DTW and other related processing tools may be used to rate the similarity of the performance and how it is written to conclude how accurate the user's singing was. Here, the input is the user's entire performance/practice, and they get back the complete analysis in one go. This won't necessarily take long because the performance penalty of the analysis is negligible.

However, since the goal is to give live feedback to the user, to analyze any note, the system needs to know at all times what note it should be expecting. One easy approach is to just force the user into a steady tempo and perhaps have a metronome to help them keep the tempo. This way, the system can iterate through all the expected notes at a constant pace, and compare whatever it has to whatever it gets. This is both easy to implement and beneficial for the user because even though there is artistic freedom in music, tempo and keeping relative note lengths correct is still a vital skill.

While the time series in 11 illustrate an important point about relative time in music, the application will solve this by forcing the user to keep a constant tempo which effectively aligns actual and expected notes, unless the user makes an error in time keeping. DTW will be used in testing the pitch detector because of recordings, that were provided for this work by Finlands svenska manssångarförbund (FSM), an alliance of finland-swedish men's choirs.

4 Implementation

The pitch detector will be made for usage on the web. The web is chosen for portability and ease of use. The end user, be it an individual wanting to learn to sing or a choir testing candidates, can use any mobile device with a microphone, be it a laptop, tablet or smartphone, and without installing anything test the accuracy of their singing or tune their guitar.

4.1 Web Audio API

As the pitch detector is intended to be used in the browser, for portability and ease of use, the Web Audio API will be extensively used. The purpose of the API is to allow developers control over audio processing functionality on browsers, by providing access to user audio devices, adding effects and more. The Web Audio API operates in an AudioContext, which can be thought of as an empty control flow graph. The graph is constructed using AudioNodes which fall into one of three categories: input or source nodes, modifier nodes and output nodes. The nodes are then connected to each other to form the graph.

The graph operates by passing blocks of data (called render quanta) between nodes, where one render quantum consists of frames of samples, where one frame contains one sample for every audio channel, at least according to the W3C Audio API 1.1 specifications. Render quantum seems to be a term reserved for the low-level processing of the data passing between nodes, and in reality is (at least on tested browsers) not an array of frames with a sample per channel but rather a set of channels with contiguous samples. Regardless of the internal memory layout of the blocks, important to note is that the blocks of data, which will be called render quanta for a lack of naming at API level, are 128 samples per channel in length.

4.1.1 Sources and destinations

The source nodes, as the name implies, are entry points for the audio control graph, they provide signals. Some of these include the OscillatorNode, a node which produces pure sinusoids, a MediaElementAudioSourceNode, which uses the media in an existing HTML audio element. As the purpose of the application is for the user to be able to record their own singing and have it analyzed for pitch correctness, the source in this case will be a MediaStreamAudioSourceNode, which provides a source signal from a MediaStream. The actual source will the method navigator.mediaDevices.getUserMedia that provides the MediaStream, but this is technically outside the AudioContext so it's not a source node. The output may be either the user's system's speakers, or another MediaStream. In this case, there will not be an output node because it is simply not needed. The output will be the result of the Fourier analysis.

4.1.2 Modifiers

The modifier nodes are the nodes that are neither sources nor destinations, they take in the signal from a node, performs some transform on the signal, and then hands it over to the next node, or nodes, in the graph. Some of these nodes apply effects, like reverb or gain, some can be used for visualizing audio, some can be used to split audio into separate channels for per-channel modification.

The available modifiers unfortunately do not help with the development of the pitch detector, so a lot of the processing would be done with Worklet nodes, a node that allows custom functionality. With this in mind, the simplest way is to not use the Audio API audio graph, but to just take the output from the microphone and in the simplest possible way turn it into data to be processed using regular procedural programming. The AnalyserNode has Fourier transform capabilities, but there does not seem to be a way to zero-pad it, which is integral to achieving both real-time processing and enough frequency resolution for base singers.

4.2 Pitch detector architecture

The identified steps of the pitch detector at the highest level are input, zero-padding, FFT, peak picking and some post-processing. It is a simple pipeline of collecting enough samples for the FFT from a microphone and then running analysis on the spectrum data. The only problem at this stage is converting the stream from the microphone to something that can be processed. There seems to be mainly two ways to go about implementing this kind of conversion. One way is to feed the stream from the microphone into a recorder and decode the binary data that becomes available with time. The other is to handle the raw audio data on the audio rendering thread by building a custom audio processor that handles the processing, whatever that may be.

As the Fourier transform and any peak-picking algorithm of choice works in the frequency domain, using the audio graph does not make much sense. The only thing the audio graph would be responsible for would be moving the render quanta between nodes, but this is a bit like using a chainsaw to slice bread. Figure 13 shows the proposed architecture which uses a bridge node to send the raw audio data from the rendering thread to the main thread. For every render quanta processed the data is passed through the node-processor port from the processor to the node in the main thread where it is passed from the node to a callback function. This callback function may then be used to accumulate the render quanta and start analysis.

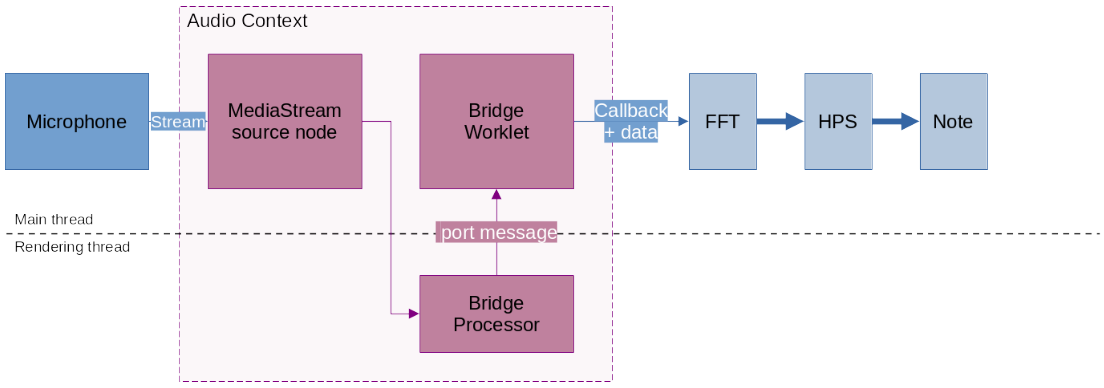

Figure 13: Architecture describing the pitch detector. Stream is converted to a data array by having a worklet node send the raw audio data through the node-processor port to be used on the main thread.

The proposed architecture offers flexibility without being overly complicated. The audio graph may be used to change the input from a microphone to a recording for validation and testing. It would also be possible to apply a real-time time-domain low-pass filter between the input node and the bridge node, which would allow downsampling and, thus, smaller FFT windows if computational power is a constraint.

4.3 Implementing the pitch detector

With a plan for the detector, the next step is to implement it. The implementation, based on the architecture can be roughly divided into 4 parts: recording an input, passing data between the threads, accumulating and performing analysis.

4.3.1 Audio input

As the application utilizes the audio graph, any one of the source nodes may be used to input data into the pitch detector. For development, a reliable source signal is of more value than my sad attempt at singing, so a series of mp3 files, some of which were provided by FSM were used with a MediaElementAudioSourceNode. The mp3 files provided by FSM were for some reason set up so that the dominant voice is alone on channel and the 3 others are on the other stereo channel. This is no problem for the audio node based application, we can simply put a channel splitting node between the source node and the detector bridge node, and connect the channel with the discrete part to the bridge.

```
this.nodes.channelSplitter.connect(this.nodes.bridge, 0);
```

4.3.2 Worklet nodes

Audio worklet nodes are a method of implementing custom audio modifier and sources (for example, a white noise generator). In this work, an AudioWorklet node will be used to implement the BridgeNode. Worklet nodes involve two parts: the node which is used in the audio graph, and the processor which is responsible for the behavior in the audio rendering thread. These are linked together.

The code snippet below shows how the BridgeNode is implemented.

```
export class BridgeNode extends AudioWorkletNode{
   constructor(audioContext, callback) {
       super(audioContext, 'BridgeProcessor');
       this.callback = callback;

       this.port.onmessage = (event) => {
           if (event.data instanceof Float32Array) {
               this.handlePortData(event.data);
           }
       };
   }

   handlePortData(data) {
       this.callback(data);
   }
}
```

The bridge node extends or inherits the AudioWorklet node class and methods are added. When the node receives data on its port, it just forwards it to a handler. The handler could do further processing on the data but for the moment only forwards it to the callback function in the pitch detector. The pitch detector callback function is the one responsible for accumulating samples, zero-padding etc.

As the BridgeNode doesn't override or specify anything new, it's really only a thin wrapper for an AudioWorkletNode which is evident by the constructor calling the super function, which invokes the parent class constructor. This just allows the user to create a BridgeNode as a opposed to a generic WorkletNode with a custom processor. This is arguably unnecessary, but can make the code nicer to read. The processor which is implicitly passed to the BridgeNode via the super constructor must be created as well.

```
export class BridgeProcessor extends AudioWorkletProcessor {
   process(inputs) {
        const input = inputs[0];
        const inputBuffer = input[0];
        this.port.postMessage(inputBuffer);
   }
}

registerProcessor('BridgeProcessor', BridgeProcessor);
```

As the purpose is not to actually process anything on the audio thread, but rather to utilize the flexibility of the audio graph and have processing done elsewhere, the processor simply sends everything it receives over the port to its node.

The only thing left to do is to instantiate the BridgeNode. What connects to it may be a microphone, an audio file, a gain node, a channel splitter, or anything else that may be useful for the pitch detection.

```
this.nodes.bridge = new BridgeNode(
  this.audioContext,
  this.bridgeCallback.bind(this),
);
```

4.4 Collecting samples

The bridge node's sole purpose is to move audio data between the audio rendering thread and the main thread. To do this, the node periodically triggers a callback function with one render quantum as an argument. The user can define their own callback function.

```
bridgeCallback(data) {
  this.analyze(data);
}
```

This also just forwards the render quantum to an analyze function to keep the code organized. In the analyze function is where accumulation happens and where analysis starts. For accumulating the samples, a ring buffer is used. When the function receives a render quantum, it first copies all the samples to a buffer with an offset. The write offset is then incremented by the render quantum size, a constant of 128.

```
// Copy buffer chunk to fft input vector.
for (let i = 0; i < RENDER_QUANTUM_SIZE; i++) {
    this.fftInputBuffer[i + this.fftBufferIteratorOffset] = data[
   i];
}
this.fftBufferIteratorOffset += RENDER_QUANTUM_SIZE;

// Perform all the analysis once enough samples.
if (this.fftBufferIteratorOffset >= FFT_TARGET_SAMPLE_SIZE) {
 // Analysis code...
 this.fftBufferIteratorOffset = 0;
}
```

When enough samples have been accumulated, a threshold which can be tuned, the analysis starts and the buffer iterator offset is reset. After the offset is reset, the new render quanta start overwriting old render quanta. In a limited sense, this functions as a ring buffer. Because the entire buffer is read at once, this provides all the ring buffer functionality needed for the accumulation.

4.5 Finding the fundamental frequency

When enough samples have been collected (set to 6000 while development and testing), the next step is to forward transform the signal into the frequency domain. The following function is called which runs a 16384-point FFT using a library called "fft.js" by GitHub user indutny, which claims to be "The fastest JS Radix-4/Radix-2 FFT implementation". The function is also responsible for creating the spectrum by calculating the magnitude of each complex data point using the Pythagorean theorem.

```
function getSpectrum(dataBuffer, fftWindowSize) {
   const square = (x) => x*x;
   const FFT = new FFTJS(fftWindowSize);

   const transform  = FFT.createComplexArray();
   FFT.realTransform(transform, dataBuffer);

   // Create spectrum from FFT transform values.
   const spectrum = new Float32Array(fftWindowSize);
   for (let i = 0; i < 16384; i++) {
       spectrum[i] = Math.sqrt(square(transform[2*i])+square(
   transform[2*i+1]))
       // spectrum[i] = (spectrum[i] > 10) ? 10 : spectrum[i]
   }

   return spectrum;
}
```

The output of fft.js is a single array with alternating magnitudes and phases, which is why the frequency-magnitude computations may look odd. The function is called from the main analysis method.

```
    const spectrum = getSpectrum(this.fftInputBuffer,
    FFT_WINDOW_SIZE);
```

The signal is implicitly zero-padded because the input array is instantiated with 0's and only a portion of them are assigned a value, precisely the first 6016 values, 6016 being the smallest multiple of the render quantum size (128) greater than the target sample size.

The spectrum is then passed on to a function which computes the HPS of the spectrum. The HPS algorithm is ported to JavaScript and looks like the following.

```
function hps(array, maxHarmonics) {
    const harmonicProductSpectrum = new Float32Array(array.length/
    maxHarmonics);
    harmonicProductSpectrum.fill(1); // Multiplicative identity
    for HPS.

    for (let harmonic = 1; harmonic <= maxHarmonics; harmonic++) {
        for (let i = 0; i < harmonicProductSpectrum.length; i++) {
            harmonicProductSpectrum[i] *= array[i*harmonic];
        }
    }

    return harmonicProductSpectrum;
}
```

The array is instantiated, not as 0's, but as 1's for the multiplicative nature of the HPS. This can be thought of as the identity array, which means that the act of copying the initial array may be done by the same loop as the other iterations, but with a downsampling factor of 1, or in other words, no downsampling.

4.5.1 Checking the spectrum flatness

While developing the system, it was revealed that the input waveform is not always very tonic and this gives a spectrum that has more noise than peaks. This seems to largely happen for two reasons. The first is the time between certain notes, when the singer either just pauses or takes a breath, resulting in complete noise. The second is that when a note is sustained and decays, it eventually loses its tonality, also resulting in noise. To prevent this, the application simply ignores the incoming data until the data is something that can be worked with.

In code, the spectral flatness is first computed with a fairly typical method, the ratio of the geometric mean and the arithmetic mean, as shown in the snippet below.

```
function spectralFlatness(signal) {
   const N = signal.length;
   const logSum = signal.reduce((acc, value) => acc + Math.log(
   value + 0.01), 0);
   const sum = signal.reduce((acc, value) => acc + value, 0);
   const geometricMean = Math.exp(logSum/N);
   const arithmeticMean = sum/N;

   return geometricMean/arithmeticMean;
}
```

If this value is under 0.6 (a value found empirically), the spectrum is deemed good and is given to the HPS. If the value is greater, the spectrum is ignored. The effect is that a note is assumed to continue until another clear note is received by the system. This mitigates the problems of noise due to silence or decay because the system just holds on to the last proper note it got.

4.5.2 Checking outliers

The presence of obvious outliers were also revealed in early testing while developing. There were times, when the pitch detector would for brief periods get MIDI numbers of 32 and 80, which are lower than the lowest male bass singers and higher than the highest female singers. These are filtered out in the same way as with non-flat spectra, by just ignoring the note.

4.6 Post-processing

Post-processing here refers to everything that is done after the fundamental frequency is found. Checking for flatness and the outliers could be considered post-processing methods, but they are done as earlier in the pipeline because it would be unnecessary to compute the HPS on a bad spectrum. Post-processing is strictly not necessary for pitch detection, but may help in simplifying analysis relating to the fundamental frequency, like scoring the accuracy of the user's singing. Post-processing also helps the user understand better what they may be doing wrong as pure frequencies mean very little to humans.

4.6.1 MIDI number to semitone name

A formula for the MIDI number was derived in the previous chapter. The derived formula cleanly translates into JavaScript, as shown in the following snippet.

```
 const midiNumber = Math.round(12*Math.log2(frequency/440) + 69);
```

The MIDI number is convenient, because it is a semitone, a simple integer and a standard, meaning it is easier to use for computation and analysis. However, for most people, the MIDI number means very little and anyone who has any background in music is already familiar with the existing notation. To convert the MIDI number to a note, two functions can be derived using known MIDI numbers and their corresponding note names.

The typical way to label a musical pitch is with SPN, where a number and a letter represents the octave and position within the octave. This is familiar to people with a background in music, and it would be the desired format for a note at the presentation layer.

As the notes repeat every octave and an octave consists of 12 semitones, the index that will be used to find the note within an octave is congruent to the MIDI number modulo 12. The octave number is the largest multiple of 12 strictly less than the number, which means it can be computed by rounding down the result of the MIDI number divided by 12. For some reason, the MIDI numbers misalign with the octave numbers, C0 being MIDI 12, the octave is shifted down one more step. The MIDI numbers, however, do align with the semitones as we start from C as C0 is the 12th MIDI number (which is congruent to 0 mod 12).

```
function getNoteName(midiNumber, scale) {
   const letter = scale[midiNumber % 12];
   const octaveNumber = parseInt(midiNumber /12) -1;

   return "" + letter + octaveNumber;
}
```

The semitones were previously defined, but those were only the semitones for certain diatonic scales. Most of the semitones will be identical across scales, but the sharps and flats must be adjusted to present the sung note's name appropriately. For example, for the E-flat minor key, the note name array would still start on C and would contain a D and an E, but the second element should be Db (b for flat) as the E-flat minor scale contains a D-flat instead of a C-sharp. At logic level, these are represented by the exact same MIDI number class, which is all the MIDI numbers 1 above a multiple of 12, but at the presentation layer, it makes a difference for someone able to read sheet music who would most likely react to the mismatch in notation. The reference material Finlandia is played in A-minor so the semitones for the A-minor key are hard coded in the application for testing purposes but parameterized for the note name function. In a proper application, the key could be inferred from the source (for example musicXML) and then the appropriate semitone name array is applied.

```
    const AflatScale = ["C", "Db", "D", "Eb", "E", "F", "Gb", "G"
  , "Ab", "A", "B", "H"]
    const noteName = getNoteName(this.currentDetectedNote,
  AflatScale);
```

4.7 Current detected note and sampling

The system continuously performs the FFT, checks for analysis criteria, performs HPS and post-processing and is completely independent of the pace of the reference. This means that the output of the system is essentially a stream as well. The proposed solution is outlined in figure 14 where the output stream is fed into a sampler to get a discrete sequence of notes.

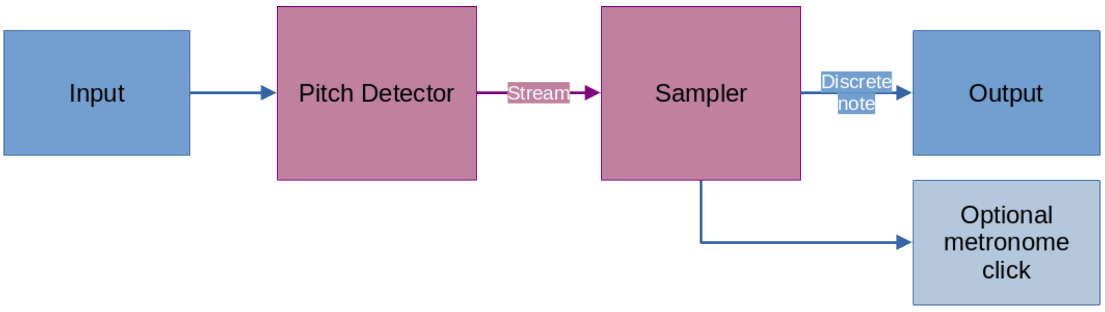

Figure 14: Diagram of how a sampler interfaces with the system

An optional metronome could be added by triggering a click from the sampler. The click should not be triggered on every sample as multiple samples would likely be taken on every beat.

The sampler subsystem is realized by continuously updating a single value in the pitch detector context and that value is checked by a function that is called periodically. For example, Finlandia's *allegro* was first assumed to be around 120BPM and then timed to be close to this. The shortest notes in Finlandia are 8th notes which means that the system should sample frequently enough to catch 8th notes. If sampling happened on 4th notes, the system could only check the first of a pair of 8th notes. A minute is 60000 milliseconds, so the sampling interval should at the very least be $\frac{60000}{(8/4)*120} = 250ms$. This is the slowest pace at which Finlandia must be sampled without losing information.

Note that 250ms is not a latency that is added to the latency of the pitch detector. Assuming the timing is set up correctly between the metronome and the sampler, the sampler would catch the note the user is singing and give feedback instantly, it just would not give more feedback for another 250ms. This is likely acceptable because if the user can hit a note, they can likely also hold it for 250ms. Sampling faster would likely just give redundant feedback, however, this could be explored further.

The values for these computations are once again hard coded. The snippet below shows how JavaScript is used to set up an interval that samples the continuously changing integer which represents the current detected note. The period of the beat is simply $\frac{60000}{T}$ which is eight times slower than the sampling period. One approach to creating the metronome would be to have the sampler count to eight, trigger an audible (or visual) cue, then reset the count.

```
const BPM = 120;
this.tempoInterval = setInterval(() => {
  this.accumulatedNotes.push(this.currentDetectedNote);
}, 60*1000/(BPM*2));
```

5 Results

As with many modern software projects, this system was not built with a waterfall methodology. This means that a lot of testing was done in parallel with development and minor problems have been mitigated. The results presented here is the state of the project when development halted, when the project is a minimum viable product with the biggest issues fixed.

5.1 Method of testing

Finlands-svenska manssångarförbundet (FSM) has kindly provided recordings of each of the parts of a TTBB choir's rendition of Finlandia. The notes for these were not provided, but sheet music of the same rendition was found online and manually verified to be correct. As a reminder, the system simply *listens* to the input stream and then just notes what the current detected note in time is. A separate time-keeper was added, which samples the detected note at regular intervals. As long as the sample frequency is small enough, no information is lost. However, as was discussed, a performance of a piece is seldom an exact instance of the definition (sheet music), especially in terms of tempo. Certain sections may be dragged out, for example, for whatever reason. Dynamic time warping (DTW) was proposed as a solution to comparing the performance and the reference for its ability to focus on parallels between substructures rather than absolute similarity.

For Finlandia, sampling at 8th notes is sufficient, because it does not contain shorter notes than that. As the current goal is just to check that a detected note is correct, the system needs to know what note it should expect at that moment in time. This means that the sheet music can be transcribed to a sequence of 8th notes or represented as a time series where each data point is a MIDI number equally spaced in time. Quarter notes become two 8th notes, half notes become four 8th notes and so on. Figure 15 visualizes the time series using color for amplitude.

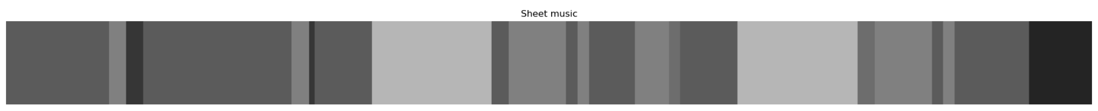

Figure 15: Sheet music of Finlandia as a time series, visualized with color for amplitude. To keep data points equally spaced in time, all notes are converted 8th notes.

Figure 15 is the same as the top part of Figure 11. The bottom part of that was a manually cleaned up version of the pitch detector's first effort, purely for demonstration purposes. DTW is only used to compare the recordings provided by FSM for a more proper test. In the real-time pitch detector, each note is directly tested as time goes on.

5.2 Analysis of HPS iterations

Each part of Finlandia was run through the system with 4, 5 and 6 for the number of HPS iterations. For these tests, the system is filtering out outliers and results of flat spectra. Figure 16 shows the comparison of one part. At a glance all versions are largely correct by looking at the substructures of the detected series and comparing them to the substructures of the reference. It's not obvious which time series did the best, but 6 iterations clearly did worst based on the amount of black in the diagram, which indicates a low note. Computing the error using DTW reveals that the error for 4 HPS iterations was 434, 352 for 5 iterations and 591 for 6 iterations. It can be hard to judge how good or bad a score of 352 is, but it clearly isn't optimal.

The same analysis can be done for the rest of the parts shown in Figures 17, 18 and 19.

Computing the average DTW error of the parts reveals that 5 iterations has the lowest error at 264.5. Again, hard to say how bad this score is, but the average

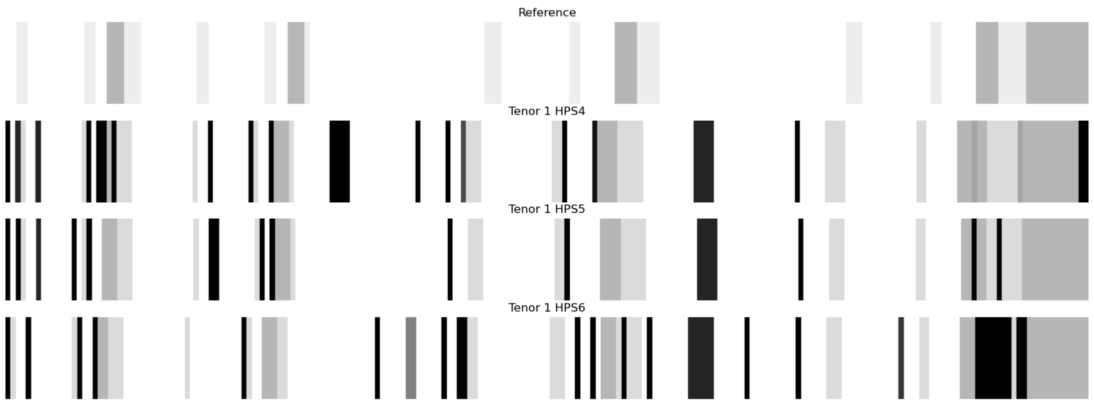

Figure 16: Comparison of the First Tenor recordings with different number of HPS iterations.

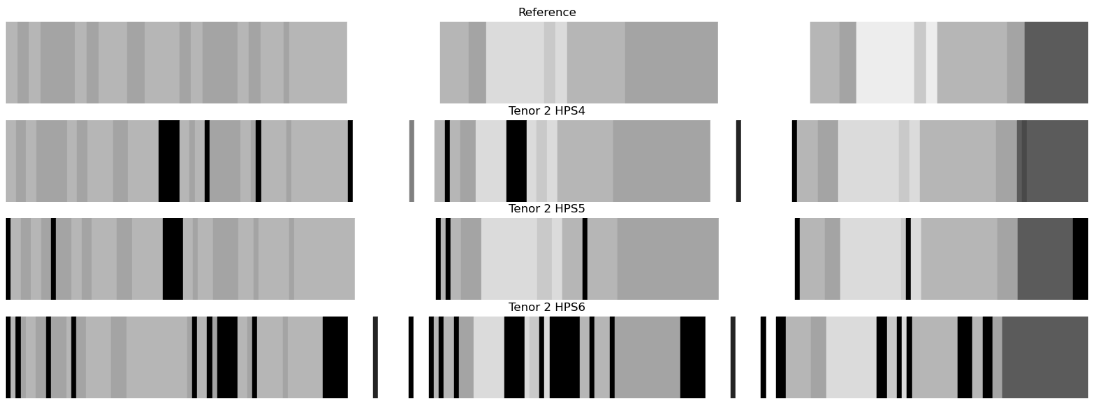

Figure 17: Comparison of the Second Tenor recordings with different number of HPS iterations.

for 6 iterations is 657.25 reveals at least that it's a big improvement. The average for 4 iterations is 299.75 which is still around 10% worse than 5 iterations.

5.2.1 Octave errors

After outlier and flat spectra filtering, the most common type of error happens when HPS folds the spectrum in a way where multiple peaks remain. Assuming a monophonic singing, these remaining peaks are still harmonics. The system naively selects the greatest peak from these which results may result in note being in the wrong octave. For the tenors the system would go consistently lower and higher for the basses. The problem likely lies in the number of HPS iterations as based on the diagrams, it can be observed that 4 iteration HPS performs very well in certain parts and 6 iterations performs well in others. This strongly suggests that the best approach would be to have some sort of Dynamic HPS that can adjust the number of iterations depending on the number of peaks. One approach could be the do 4 iterations by default after which more are done

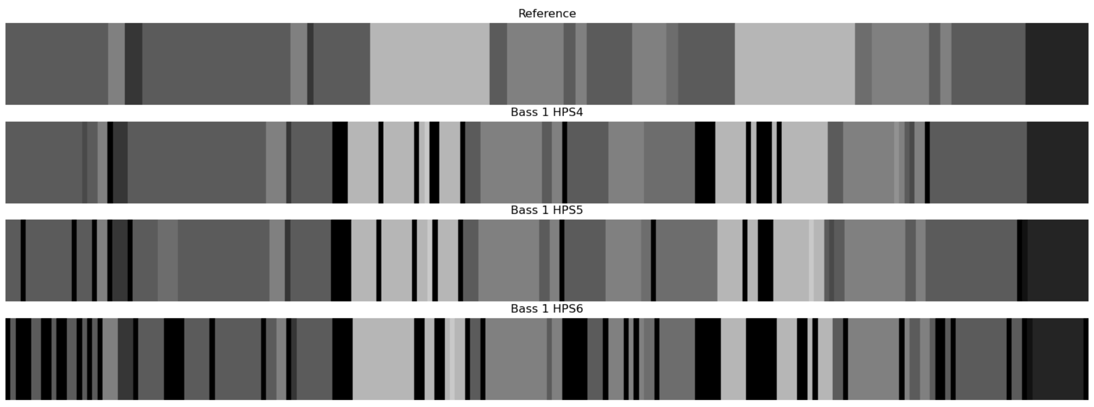

Figure 18: Comparison of the First Bass recordings with different number of HPS iterations.

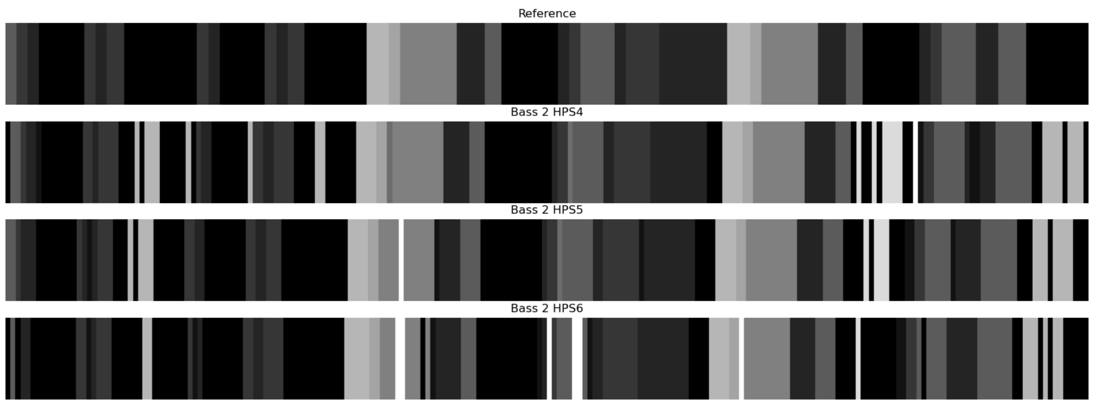

Figure 19: Comparison of the Second Bass recordings with different number of HPS iterations.

depending on the number of remaining peaks. CFAR would be a strong candidate for this approach.

Figure 20 is created by computing the remainder of the MIDI numbers modulo 12 to get octave indifferent notes. It clearly shows that most of the errors in the results regardless of HPS iterations are due to octave errors. In effect, it highlights the parts where the detector got the note wrong. This analysis is done to highlight the presence of octave errors, and it does not imply that the system could automatically correct octave errors by comparing notes mod 12, as this could lead to false positives if the user sung an octave wrong, which is important.

5.3 Using the detector as a guitar tuner

An attempt to use the pitch detector as a guitar tuner was made. The acoustic guitar was first intentionally detuned and then tuned back using the pitch detector. The results were mixed.

Octave errors were present for the guitar as well, but were not an issue as the

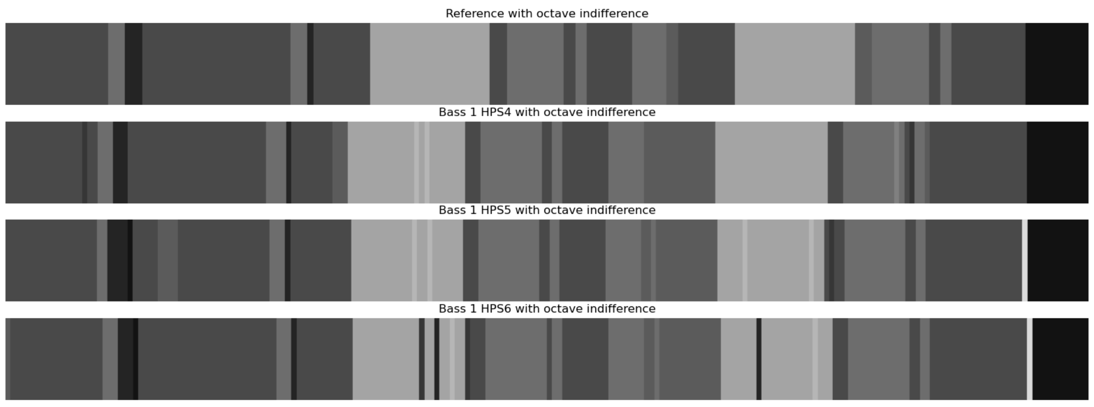

Figure 20: Comparison of HPS iterations with octave indifference.

guitar is typically tuned so that the highest open string produces an E3 and the lowest E2. Even though the pitch detector presented an E4, a user would most likely be satisfied with this.

Using the pitch detector as a tuner also showed that the detector responds quickly enough to be considered real time. Plucking the string repeatedly — or hard enough to sustain longer — allows for almost continuous turning of the tuning pegs. This allows for very convenient tuning.

The bin size is an issue for guitar tuning. The lowest open strings push the limit of what is possible with the current bin size of 3Hz. This bin size also means that there is a fairly wide range of "correct" pitches. After tuning with the pitch detector, the sound was compared to an online guitar tuner that uses reference pitches instead of pitch detection. The tuned guitar sounded flat and were off by perhaps 1 or 2 Hz. The guitar was once again detuned, but this time so that all open strings were too high. This time the guitar would sound a bit sharp. This implies that the bin sizes are simply too big for precise instrument tuning. That, however, would probably not be a problem for the goal of this particular pitch detector.

6 Conclusions

The purpose of this work was to explore frequency-domain approaches to pitch detection for use in real-time. It was constrained to focus on giving feedback specifically for male (TTBB) singers. Using the theory created by Joseph Fourier in conjunction with more modern signal processing techniques such as the FFT and Harmonic Product Spectrum, the system can process an incoming stream to continuously estimate the fundamental frequency. A sampler is used to convert the continuous estimate stream to a sequence of notes which can be compared offline with Dynamic Time Warping or in real-time by forcing the user to be on pace with the sampler. The system works relatively accurately for monophonic inputs but falls short in certain aspects. The most common type of issue is an octave error, when multiple peaks remain after the HPS and the pitch detector can't know for certain which peak to choose. This shows that pitch detection is easy to get working, but difficult to get right.

The product of this work is a JavaScript class and utility functions which could integrate in another application. The project repository contains a simple frontend that can be used to run the pitch detector. The pitch detector is built in a modular way that would allow customization, even if there is no proper API available. Future development could involve restructuring the code so that a user can overload two functions, one for the analyzing function and the other for setting up the audio graph. As seen with the audio recordings from FSM, simply changing the input node is insufficient, the user may need to modify the audio graph as well. The application in its current state is also not minimized or optimized in any way.

The results are mixed. The pitch detection works and gives accurate results most of the time. The most common error is one where the detected note is an octave (or two octaves, in some cases) off. This seems to be a limitation of HPS and testing with different number of multiplication iterations reveals that many of the octave errors could be mitigated with a version of the HPS that could adjust the number of iterations depending on the number of peaks in the spectrum.

Svensk sammanfattning

Inledning

En person med absolut gehör kan stämma en gitarr med att helt enkelt lysna på dess ljud och vrida på stämskruven tills ljudet låter rätt. En person som inte har absolut gehör måste lysna på en referens, t.ex. en stämgaffel, och jämföra ljuden. Hen hör om gitarren är lägre eller högre och kan justera ljudet i rätt riktning tills gitarren och referensen är samma. Båda fallen är exempel på tonhöjdsidentifiering. Detta är något datorer kan hjälpa med och därmed har man studerat och kommit fram till flera olika metoder. Tonhöjd har en stark anknytning till oskillations frekvensen på en ljudsignal. Vissa metoder fokuserar direkt på signalen medan andra omvandlar signalen till något som är lättare att jobba med.

Detta arbete fokuserar på Fourier transformen och and signalbehandlings metoder som krävs för tonhöjdsidentifiering. Systemet ska fungera i realtid och snurra i webbläsarmiljö så att det lätt kan integreras i webbapplikationer.

Fourier analys

Joseph Fourier jobbade en differentialekvation och tänkte att det skulle vara lättare att integera och derivera funktioner om de vore sinus eller cosinus vågot. Han började utforska tanken att representera funktioner som sinus och cosinus vågor och till slut utvecklade handet vi kallar i dag för Fourier serier. En generisk Fourier serie (på intervallet $[0, 2\pi]$ ser ut på följande vis

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n cos(nt) + \sum_{n=1}^{\infty} b_n sin(nt)$$

med detta menas att vilken som helst periodisk funktion på intervallet kan uttryckas som en kombination av olika sinus och cosinus vågor.

Fourier transformen, till skillnad från Fourier series, kan användas för icke-periodiska funktioner. Fourier transformen definieras ofta på följande vis

$$\hat{f}(x) = \int_{-\infty}^{\infty} f(t) e^{-i2\pi xt} dt$$

Fourier transformen av en signal är en ny signal i det så kallade frekvens domänet. Frekvens domän signalen beskriver vilka komponenter signalen innehåller och hurdana de är. Alternativt kan man se på Fourier transformen som en funktion som mäter korrelationen mellan en funktion och alla sinus- och cosinusvågor. Om inte en viss sinusvåg korrelerar med funktionen är den inte den komponenten en del av funktionen. Korrelationen fungerar pågrund av icke-ortogonalitet, alltså att integrering av $f(x)sin(nx)$ är π. Om funktion inte innehåller en viss komponent, kommer integreringen ge 0 för korreleringen mellan funktionen och komponenten.

Fourier transformen har en diskret variant som kallas ofta för DFT.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{i2\pi kn}{N}}$$

Denna funktion gör i princip samma som den kontinuerliga transformen men används för diskret data.

I frekvens domänet kan man lätt modifiera signalen. T.ex. filtrering av höga ljud kunde man lätt göra med att helt enkelt radera komponenterna. Frekvens domänet kan också användas för att lagra information för trådlös kommunikation. Man kan dock varken spela upp eller sända signalen i frekvens domänet. Man skulle måste omvandla signalen tillbaka till tids domänet. Detta gör man med inverstransformen.

$$f(t) = \frac{1}{\pi} \int_{-\infty}^{\infty} \hat{f}(\zeta) e^{i2\pi\zeta t} d\zeta$$

och den diskreta inverstransformen

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{\frac{i2\pi kn}{N}}$$

När man gör beräkningar med en dator används ofta en algoritm som kallas för Fast Fourier transform. Den kallas snabb för att den producerar samma resultat men gör det mer effektivt. Effektivitet är så hög att den verkligen är snabbare, speciellt för stora data mängder. FFT algoritmen noteras vara ca 12800 gånger snabbare än DFT algoritmen för 512 000 data punkter.

Tonhöjdsidentifiering

Tonhöjdsidentifiering handlar om att räkna ut vilken tonhöjd en uppspelad not är. Metoder går in i en av tre kategorier; tids-domän, frekvens-domän och statisk-ta/maskininlärnings metoder. En populär tids-domän metod är autocorrelation där signalen jämförs med en förskjuten version av sig själv.

Ljud är en tryckskillnad i ett medium och modelleras ofta som en våg. T.ex. en högtalare som spelar A4 trycker ut luft med en frekvens på 440Hz. En ren 440Hz signal kommer dock att låta hemsk och rå. Ett vackert ljud så som ett A4 på ett riktigt instrument innehåller mycket annat än bara 440Hz. Detta kallas klangfärg eller timbre.

Vad tonhöjd egentligen är är inte alltid klart. Man kan säga att tonhöjden är hur högt eller lågt en lyssnare anser att ljudet är. Man kan även säga att tonhöjden är grundfrekvensen i ett ljud. Även där uppstår ett problem där grundfrekvensen inte verkligen finns som en del av signalen. Detta kallas för *missing fundamental* och uppstår då harmonier bygger på varandra och bildar ett probminant ljud med en frekvens som är den största gemensamma delaren på harmoniernas frekvenser.

För tonhöjdsdetektering i frekvens domänet börjar man med att omvandla signalen med hjälp av en Fourier transform. Om man har ett längre ljud, kunde man använda en Short Time Fourier Transform (STFT) som kort sagt delar upp en signal i mindre fönster som kan analyseras skillt. I detta fall byggs en realtids detektor, vilket betyder att fönstren bildas naturligt, så en normal FFT används. I frekvens domänet visas vilka komponenter finns i signalen och härifrån ska en av dem väljas.

För processering av frekvens domänet finns det även flera algoritmer. Harmonic Product Spectrum (HPS) och Constant False Alarm Rate (CFAR) diskuteras. HPS nersamplar frekvens domänetoch multiplicerar iterationerna ihop vilket skapar ett nyt frekvens domän där harmonier förstärker den största gemensamma delaren. Denna algoritm är bra för den hittar missing fundamental om den saknas. HPS däremot fungerar väldigt dåligt om det finns få harmonier och kan i sådana fall ge fel svar. CFAR är en algoritm som hittar toppar i frekvens domänet med att skapa dynamisk tröskel med hjälp av att applicera statistiska metoder signalen. Om en punkt är ovanför tröskeln är den antagligen inte brus. HPS fungerar bättre för ändamålet men CFAR kunde hjälpa med att implementera en dynamisk HPS där iterationer ökas eller minskas beroende på hur många toppar CFAR har hittat.

Tonhöjdsdetektorn ska snurra i realtid. Vad en acceptabel latens är kontextuellt. För renderering är det 33ms för att uppnå 30FPS. För realtids följning på en beställning är en uppdatering per timme kanske tillräckligt. Eftersom syftet är att användaren ska få feedback direkt är latensen mest beroende på användarens egen reaktions tid. Att minimera latensen är svårt eftersom Fourier transformens frekvens resolution är inverst proportionell till mängden data punkter som samlas. Man kan kringå detta med *zero-padding* där man skapar artificiella datapunkter som inte har en inverkan på signalen.

Implementering

Tonhöjdsdetektorn implementeras för webben och därmed används Web Audio APIn mycket. Denna ger åtkom till användarens mikrofon och andra redskap som kan användas för audio processering. Om man får rå data från mikrofonen kan man enkelt processera det med bibliotek så som FFT.js. Hur man får rå data från mikrofonen är dock inte enkelt.

En lösning till detta är att använda sig av Web Audio APIns audio graf. Med denna kan man bygga upp en kontroll graf där noder skickar data till varandra. Noderna har olika roller; käll noder, modifierar noder och destinations noder. Käll noden för detta ändamål är en mikrofon nod och inga destinations noder används. Inga av de existerande modifierar noderna kan användas för tonhöjdsdetektorn. En AnalyserNode kör Fourier transformen, men denna kan man inte zero-padda, vilket måste göras för att få tillräcklig frekvens resolution i realtid.

Lösningen är att bygga en egen nod med hjälp av WorkletNodes. Tanken är att man bygger en nod som helt enkelt samlar rå data och ger det över till en normal JavaScript funktion. Detta ger även flexibilitet att använda grafen vilket gör det enkelt att validera och testa. För att testa systemet mot en ljudfil gäller det bara att byta ut mikrofon noden mot en audio fil nod.

Den implementerade tonhöjdsdetektorn är en pipeline där audio grafen samlar data, skickar det vidare till an analys funktion som kör FFT, HPS, en del post processering och till sist presenterar användaren med en not. Allt detta händer på ca 120ms.

Resultat

Tonhöjdsdetektorn testas med hjälp av ljudfiler som Finlands-svenska mansångarförbundet harförsett. Tonhöjdsdetektorn analyserar alla fyra stämmor av Finlandia och jämförs med notbladet.

Resultaten indikerar problem i HPS. Det vanligaste är att tonhöjdsdetektorn ger resultat som är fel med en oktav. Olika mängder iterationer för HPS testades även och slutsatsen är att en dynamisk HPS vore bra då fyra iterationer presterar bäst i vissa fall och 6 iterations bäst i andra fall.

Slutsats

References

[1] A. Bounchaleun, "An elementary introduction to fast fourier transform algorithms," 2019.

[2] S. Randhawa, "Analysing & implementing cooley tukey fast fourier transform algorithm," 09 2018.

[3] M. T. Heideman, D. H. Johnson, and C. S. Burrus, "Gauss and the history of the fast fourier transform," *IEEE ASSP Magazine*, vol. 1, no. 4, pp. 14–21, 1984.

[4] J. W. Cooley, "The re-discovery of the fast fourier transform algorithm," 1987, [Accessed 23-10-2024].

[5] M. Rozman, "Radix-2 fast fourier transform," https://www.phys.uconn.edu/~rozman/Courses/m3511_19s/downloads/radix2fft.pdf, 2019, [Accessed 12-02-2025].

[6] Reducible, "The fast fourier transform (fft): Most ingenious algorithm ever?" 2020, accessed: 2024-10-28. [Online]. Available: https://www.youtube.com/watch?v=h7apO7q16V0

[7] N. S. Gnanishivaram K., "Fft/ifft processor design for 5g mimo ofdm systems," https://warse.org/IJWCNT/static/pdf/file/ijwcnt04332014.pdf, 2014, [Accessed 04-04-2025].

[8] A. Emerencia, "Multiplying huge integers using fourier transforms," 2007, [Accessed 28-10-2024].

[9] W. P. Rossing T., Moore R., "The science of sound," https://api.pageplace.de/preview/DT0400.9781292055152_A24617782/preview-9781292055152_A24617782.pdf, [Accessed 04-04-2025].

[10] A. Gotsopoulos, "Computational challenges in pitch detection algorithms," https://gotsopoulos.com/files/Gotsopoulos-BachelorThesis.pdf, accessed 04-02-2025.

[11] Veritasium, "These illusions fool almost everyone," https://www.youtube.com/watch?v=Sn07AMCfaAI&t=1169s, 2024, [Accessed 04-04-2025].

[12] B. Evans, "A review of automatic music transcription low level processing techniques and the evaluation and optimisation of multiresolution fft parameters," https://eprints.hud.ac.uk/id/eprint/17816/1/Final_Thesis_-_November_2012.pdf, 2012, accessed 04-02-2025.

[13] P. McLeod, "Fast, accurate pitch detection tools for music analysis," 2008, accessed: 2024-11-22. [Online]. Available: https://www.cs.otago.ac.nz/research/publications/oucs-2008-03.pdf

[14] T. Smyth, "Harmonic product spectrum (hps)," 2019, accessed: 2024-11-22. [Online]. Available: http://musicweb.ucsd.edu/~trsmyth/analysis/Harmonic_Product_Spectrum.html

[15] M. Bruner, "How radars use cfar to detect targets," https://www.youtube.com/watch?v=BEg29UuZk6c, 2024, [Accessed 07-04-2025].

[16] H. Benchmark, "Reaction time statistics," https://humanbenchmark.com/tests/reactiontime/statistics, [Accessed 17-02-2025].

[17] J. Shelton and G. Kumar, "Comparison between auditory and visual simple reaction times," *Neuroscience & Medicine*, vol. 1, pp. 30–32, 01 2010.