FPGAutotune Pro
Final Report

Evan Andrews
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
evan_a@mit.edu

Alexander Sherstnev
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
ashrstnv@mit.edu

Benjamin Yonas
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
byonas03@mit.edu

*Authors contributed equally and are listed in alphabetical order.

***Abstract*—We present a real-time implementation of autotune entirely in hardware on an FPGA. Our design records audio input from a microphone, detects the fundamental frequency being sung, matches that frequency to the nearest note in the C major scale, and repitches the recorded audio to that note.**

I. Problem Definition and Theory

Autotune is an audio processing technique used to make vocals or instruments sound more pitch-accurate. Some versions of autotune perform pitch-correction in real time while others are used as post-processing tools. In our project, we autotune human singing in real time to match the nearest note of the chromatic scale in the key of C. The goal is to ensure the singer sounds noticeably more in-tune with autotune applied.

Implementing autotune consists of two main subproblems: detecting the frequency being sung, and adjusting the frequency of a sound sample. In order for the autotuned output to be perceived as real time, we aim to play back autotuned output with at most 50 ms of latency.

A. Pitch Detection

Human voice is not a pure sine wave, but rather a superposition of many waves called harmonics. The wave with the lowest frequency is known as the fundamental frequency or the 1st harmonic—this is the pitch we aim to detect and retune. The other waves are higher order harmonics with integer multiple frequencies of the fundamental (e.g. the 5th harmonic has 5 times the frequency of the fundamental).

We use the harmonic product spectrum algorithm [1] to detect the fundamental frequency. This algorithm operates on the frequency-space representation of an audio frame.

Let $a(\nu)$ be the amplitude of the frequency ν in the Fourier space. We first compute $b(\nu)$, which is large when ν is likely to be a fundamental frequency:

$$b(\nu) = \prod_{i=1}^{k} a(i \cdot \nu). \tag{1}$$

$b(\nu)$ represents the product of the amplitudes of the contributions by some frequency ν and its next k harmonics. We then choose the fundamental frequency $\nu_{\text{fundamental}}$ to be the frequency which maximizes b (see figure 1). We found that $k = 4$ gave very good results in our testing.

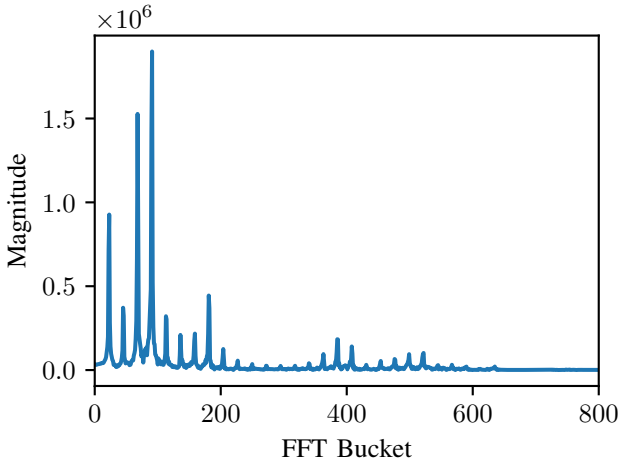

Fig. 1. A graph of the frequency space for a sample of human voice. The x axis is frequency and the y axis is the contribution by that frequency. Note the peaks occur at a fixed interval, at multiples of the fundamental frequency.

In our evaluation, we found this algorithm to be significantly more effective than simply scanning the frequency space for the first "peak" because noisy data often obfuscates peak positions.

B. Pitch Adjustment

The superposition of these harmonics and their relative magnitudes produces the timbre of the sound—the quality which differentiates unique voices. Therefore it is important to maintain these relative magnitudes and phases. The simplest way to adjust the pitch of audio while maintaining timbre is to speed it up or slow it down via resampling. A sine wave played twice as fast will have twice the frequency and twice the pitch. However, the duration of the sound is not preserved and phase discontinuities may arise when repitching sequential frames of audio.

To resolve these issues, we use a dynamic phase vocoder[1] [2]. We begin by grouping incoming samples into overlapping frames and multiply each frame by a Gaussian (also known as a Hanning window). The space between the start times of adjacent frames is called the hop size, and the duration of each frame is called the frame size.

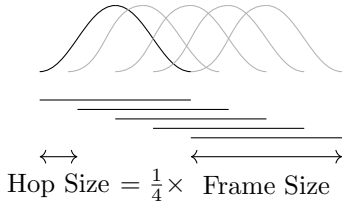

Fig. 2. An audio sample split into overlapping frames f_i, each multiplied by a Gaussian.

We can speed up or slow down each of these frames independently. Finally, we add the adjusted frames back together, with the original hop size. The resulting audio has approximately the same duration as the original audio (see figure 3), while each frame can be repitched by a different factor.

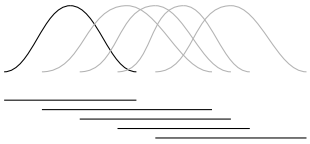

Fig. 3. Several overlapping audio frames, each sped up or slowed down by a different factor. The resulting audio has approximately the same duration as the input, despite the different scale factors. Note that the hop size is preserved.

This approach resolves the duration issue, but introduces a new problem. Since each frame is resampled at different rates, waves pick up a different phase shift across each frame. The result is that waves that were continuous in the original audio may be discontinuous if adjusted using this method. While testing, we found this to produce crackling artifacts.

To address this issue we must adjust the phase of each wave to reflect the phase shift produced by resampling each frame.

For some wave of frequency ν, let $\phi_n(\nu)$ be its starting phase at frame n. Normally, the phase shift accumulated between frames $n-1$ and n is $\phi_n(\nu) - \phi_{n-1}(\nu)$. Because of our retuning, however, the phase shift should be scaled by the tuning ratio R. A tuning ratio $R = 2$ means we expect twice the pitch, and so twice the phase shift.

Thus, we compute the adjusted phase for some wave of frequency ν on frame n to be

$$\phi'_n(\nu) = \phi'_{n-1}(\nu) + \big(\phi_n(\nu) - \phi_{n-1}(\nu)\big) \cdot R. \tag{2}$$

We then adjust the phase of each wave in the Fourier space by multiplying by $\exp i(\phi'_n(\nu) - \phi_n(\nu))$. This ensures that waves remain continuous across repitched frame boundaries.

[1]The cited article describes a "static" phase vocoder which tunes by a set resampling rate. Our algorithm differs by dynamically updating the resampling rate as we receive audio input.

C. Uncertainty Principle: Balancing Latency with Accuracy

To achieve "real time" playback, the frame size must be short enough for delay to be imperceptible while maintaining a frequency granularity fine enough to accurately choose the tuning rate R.

The frequency space generated by the discrete Fourier transform has a granularity g determined by the sampling rate $r = 16\,\text{kHz}$ and the frame size s. Note that the granularity g, which defines the bin sizes of the frequency space, is merely a function of the frame size in terms of time.

$$g = \frac{r}{s}. \tag{3}$$

In order to achieve $<$50 ms latency while leaving $\sim$10 ms for computation, $s < r \cdot 0.04\,\text{s} = 640$ samples. However, this produces a granularity $g = 25\,\text{Hz}$ which is too wide to accurately detect pitch. Even assuming instant computation, the theoretical best granularity we can achieve with this latency is 20 Hz—still too coarse to determine pitch.

To address this fundamental limitation, we maintain two separate audio buffers. The larger buffer is used to perform pitch detection, at a latency greater than 50 ms, while the smaller buffer is used to perform the actual adjustment, at a latency less than 50 ms. This strikes a balance between "real time" adjustment and accurate pitch detection.

II. Implementation

A. Evaluation in Python

To verify the quality of our method, we first implemented each module of our design in Python. This allowed us to test end-to-end how our modules would interact, and verify our algorithm produced high quality results.

In designing our software implementation, we took care to structure our Python similar to hardware. For example, we avoided passing arrays by value, and instead designed each module to operate in-place on a few large arrays that were passed as parameters to mirror the System Verilog we would eventually write.

Following the theory, we began by detecting the fundamental frequency of each frame. We verified that the harmonic product spectrum algorithm was effective by empirically testing against real audio. Figure 4 shows the result of our pitch detection and note matching on real data. The tuned note closely matches the detected fundamental—except for some discontinuities when the singer changes pitch. These are moments of silence in the audio where our algorithm struggles to detect a fundamental frequency among the noise. We found these spikes to be inconsequential in our testing as they occur very briefly when the audio has low amplitude.

Next, we implemented the phase adjustment module enabling us to verify the quality of our autotune end-to-end. We tested extensively on real audio to confirm that we produced high quality and consistent results.

Our full software implementation showed high quality results as a proof of concept, but had poor latency. Since we designed our algorithms carefully to be implementable in

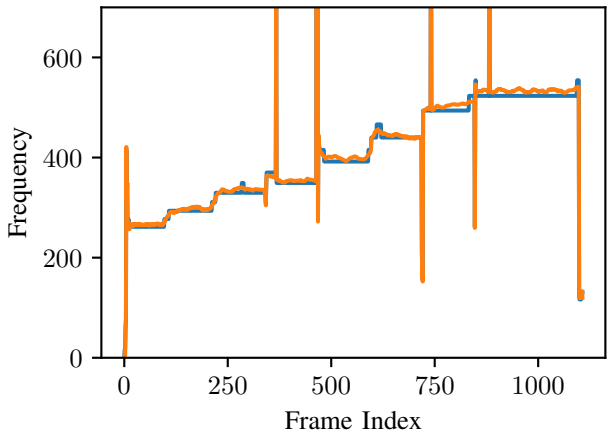

Fig. 4. The detected fundamental frequency (orange) and tuned note (blue) per frame (x-axis) in our software implementation. The audio recording features Ben Yonas singing the C major scale. The y-axis shows pitch represented in FFT bucket indices. Note the individual monotonically increasing scale degrees represented by the sustained lines.

hardware, we were confident we could achieve similar high-quality results with in real time on an FPGA.

B. Hardware Implementation

Once our software implementation was complete, we began implementing our algorithms in System Verilog. Figure 6 shows a block diagram of our hardware implementation. BRAMs, FFT modules, and I/O are highlighted. As the diagram shows, there are two main branches of our design: pitch detection and phase adjustment. We tackled these problems one at a time.

1) Audio I/O: Our initial step was to determine how we would record and playback audio on the device. We used the INMP441 MEMS microphone which communicates over the I^2S protocol. We then tested this in hardware by using pulse density modulation (PDM) to play back the recorded samples.

2) Pitch Detection: We first implemented the fundamental finder module, corresponding to the left branch of our block diagram (figure 6). We used an open source FFT implementation [3] which we configured to use 2048 samples at 16 bits of precision. This module gave us excellent throughput and good latency (< 5000 cycles).

We implemented the harmonic product spectrum using 48 bit multiplication, which proved to be the limit of what our FPGA could compute in a single cycle (using DSPs on a 100 MHz clock).

After computing which FFT bin contains our fundamental, we compute a weighted average with the adjacent FFT bins. Say we find bin i to contain the fundamental. Then we wish to compute the weighted $\mathcal{W}$ average of bins $i-1, i, i+1$:

$$\mathcal{W} = \frac{(i-1)F_{i-1} + iF_i + (i+1)F_{i+1}}{F_{i-1} + F_i + F_{i+1}}. \tag{4}$$

Once we computed the fundamental, we performed a linear scan in a lookup table (LUT) of C major notes to find the nearest match. In our initial testing, the fundamental and nearest note could be displayed on our FPGA's 7-segment display, which we used for quick debugging.

We were initially concerned about storing data in LUTs on the FPGA fabric instead of in BRAMs, but upon testing discovered that we had ample resources.

3) Resampling: Once our pitch detection was implemented and tested, we moved on to resampling. The algorithm we designed was inspired by PDM. A visualization of this algorithm is shown in figure 5. In our implementation we perform interpolation by considering each input sample four times. We then take the average of the four nearest samples in the output. This allows us to represent an output sample $\frac{1}{4}$, $\frac{1}{2}$, or $\frac{3}{4}$ between two input samples.

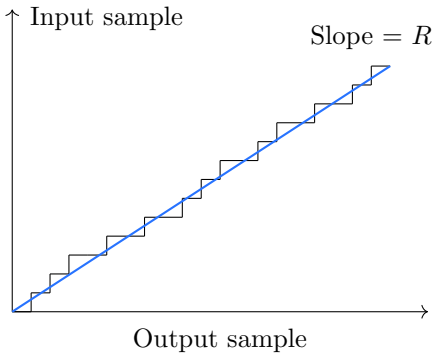

Fig. 5. A visualization of our resampling algorithm. One step on the x axis corresponds to one output sample, and one step on the y axis corresponds to one input sample. The straight line has slope R—the tuning ratio. The resampling algorithm takes vertically (to the next input sample) until it crosses the tuning line, then it takes a step horizontally (to the next output sample).

4) Phase Adjustment: Phase adjustment was the most challenging module to implement due to the complexity of the calculations involved. We chose to perform our calculations in polar coordinates for simplicity, and converted to and from Cartesian (real + imaginary) coordinates using the CORDIC algorithm. We used an open source implementation of the CORDIC algorithm [4] which we found simple and effective.

We represented our polar coordinates as two 16 bit numbers, a phase and an amplitude. To take advantage of modular arithmetic, we chose units for our phase such that $0 = 0$ and $2^{16} - 1 = 2\pi$. This simplified our calculations significantly. The CORDIC implementation we used allowed us to convert this form to and from 16 bit Cartesian coordinates efficiently.

Our phase adjustment formula (equation 2) depends on the previous original phase and adjusted phase for each wave. We decided to store this information in reprogrammable LUTs instead of BRAM. This gave us single-cycle access latency and made the implementation far simpler. Although it required more FPGA fabric to synthesize, we found this acceptable given our overall low resource usage.

5) Recombining Frames: Once the frame is resampled, it can finally be added to the output audio buffer and read out on the speaker. Frames must be added together since they overlap $\frac{3}{4}$ with each other. We achieved this by using a circular buffer

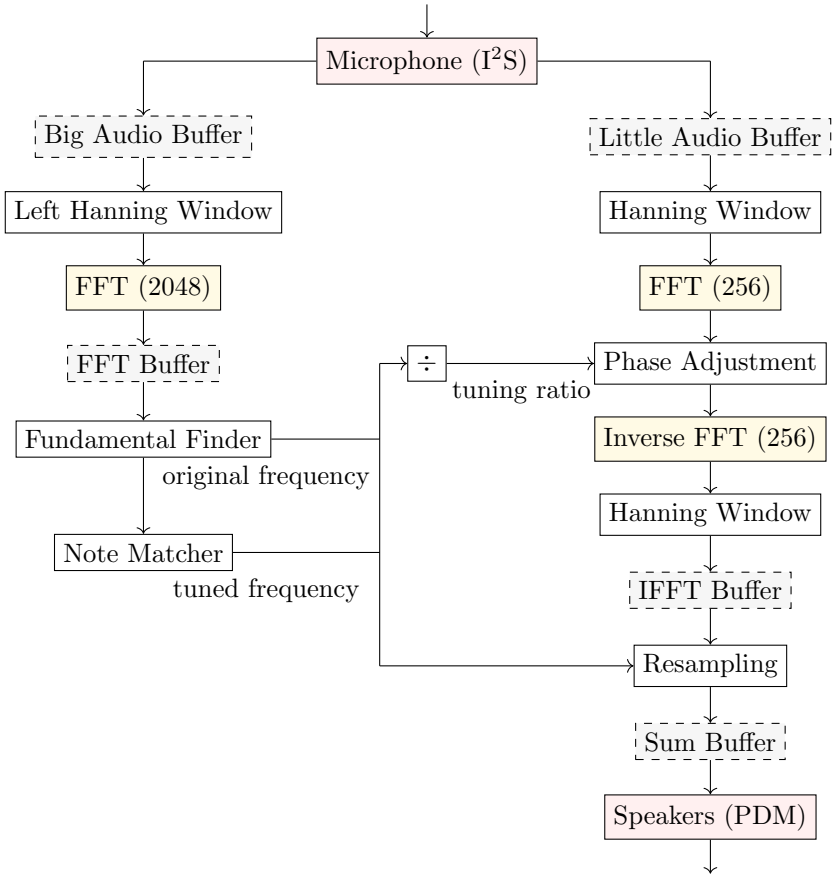

Fig. 6. A block diagram of our implementation.

equal to twice the length of one of the frames. This way, there is enough space for 4 samples to be superimposed on one another and played back correctly. The buffer is circular to avoid any excess BRAMs and keep audio output as simple as possible.

The final audio samples are passed into PDM to play out on the audio jack. We chose PDM instead of PWM to get higher fidelity for our 16 bit audio as the PWM lab audio did not produce sufficient audio quality for our standards.

C. Module Communication

One challenge in our hardware implementation was coordinating all of our modules. This is especially challenging in this project because there are tight deadlines for when work must be completed in order to achieve real time processing.

We decided to store most of the state of our system in top-level BRAMs rather than within modules. This made it simpler to test modules individually, and also made communications between modules simpler. Modules begin computation when a single-cycle "starting pistol" goes high, and output a ready signal when they finish. This eliminates the need to manage precise timing between modules and decouples their implementation.

Within modules, we attempted to achieve 1 sample per cycle throughput when possible. This allowed us to simplify inter-module communication further, by simply wiring the output and data ready signals of one module to the input and starting pistol signals of another. A good example of this in our block diagram is the communication between the FFT, phase adjustment, IFFT, and Hanning window modules—all of which have a throughput of 1 output per cycle.

III. Testing and Evaluation

Our initial goal and evaluation criterion was for autotuned singing to sound more on-tune than it would otherwise. Since this goal is subjective, most of our testing has been qualitative. We took advantage of plotting in both our software implementation and in our testbenches. We also referenced existing pitch detection software (TE Tuner) to ensure that our audio samples were being repitched properly.

We used Cocotb to create test benches for our individual modules. Due to limitations of Cocotb and the size of our top level, integration testing via simulation became impractical. Therefore, we wanted to ensure our unit testing of modules was thorough. Some test benches were straightforward and relied on asserts for expected values while more complex modules required different techniques. We were able to significantly leverage our Python implementation in these cases to compare outputs—either quantiatively or qualitatively. This

approach allowed us to carry our confidence from the Python implementation over to the hardware implementation.

For example, in the early stages of our hardware implementation we wanted to verify our FFT module worked as expected so we began by qualitatively comparing FFT outputs on recorded audio samples. As our tests became more elaborate, we began quantitatively comparing output between our Python and hardware implementations by checking the root mean square (RMS) of the difference with reasonable thresholds. Techniques like these were used for major modules including resampling, phase adjustment, and the Hanning window.

Finally, we implemented additional Python frameworks to facilitate our testing. For example, in order to tests BRAM-reliant modules, we implemented a simulated BRAM in Python which implemented the same interface, enabling us to easily verify modules which interacted with top level BRAMs.

IV. Bug Identification and User Interface

A. Bug Identification

Following the preliminary report of our autotune implementation, there were plenty of small bugs to address. For one, the sample accumulator was being improperly written to, resulting in high frequency noise destroying the audio quality. Additionally, we found bugs in the phase adjustment module which were very challenging to identify due to the more complex data manipulations present in the module. For example, our reconstruction of Hermitian symmetry was incorrect. As a result, we rewrote the module from scratch. Lastly we found bugs in the fundamental finding module which suffered from overflow in the accumulation of frequency products required for producing the harmonic product spectrum. To resolve this, we needed to increase our bit width for representations in the module.

B. User Interface

Finally, we added a nice interface which displays detected frequencies and notes being tuned to for the ease of identifying how well our hardware implementation works. This data is being transferred over UART to a computer, which is running the Python front end for the UI.

V. Conclusion

At the time of writing this final report, we feel that we have effectively addressed the technical challenges related to autotune. Given our testing in software and bug identification in hardware, we feel that our autotune module accurately identifies and retunes input audio to align with the C chromatic scale.

When looking back at how we implemented our design as a whole, there are definitely notable good approaches and practices that we upheld. Some good practices included developing comprehensive tests for most of our modules, which gave us the confidence of where to look during the hardware debugging stage of our development. Another great approach that we took was the implementation of a non-real time autotune implementation in python, which could be used to identify the signal processing limitations of our chosen sample and frame parameters.

Acknowledgment

The authors would like to thank Joseph Steinmeyer and Janette Park for their help during lab hours and for providing the hardware required for this project. Additionally, we would like to thank François Grondin for his inspiration to use the Phase Vocoder algorithm, and his excellent website detailing its implementation [2]. Furthermore, we thank Dan Gisselquist of Gisselquist Technologies for his excellent FFT [3] and CORDIC [4] modules. We also liked the epigraph at the end of his webpage: “He that withholdeth corn, the people shall curse him: but blessing shall be upon the head of him that selleth it.” [5]

References

[1] A. M. Noll, “Pitch determination of human speech by the harmonic product spectrum, the harmonic sum spectrum and a maximum likelihood estimate,” *Proceedings of the Symposium on Computer Processing in Communications*, vol. XIX, 1970.

[2] F. Grondin, “Guitar pitch shifter,” https://www.guitarpitchshifter.com/index.html.

[3] D. Gisselquist, “An open source pipelined fft generator,” https://zipcpu.com/dsp/2018/10/02/fft.html.

[4] ——, “Using a cordic to calculate sines and cosines in an fpga,” https://zipcpu.com/dsp/2017/08/30/cordic.html.

[5] King Solomon, *The Holy Bible*. King James, 1611, ch. Proverbs 11:26.