Frequency Perfect Generated Audio: Final Report

Shruti Siva
Massachusetts Institute of Technology
Cambridge, MA, USA
shrutsiv@mit.edu

Felix Prasanna
Massachusetts Institute of Technology
Cambridge, MA, USA
fpx@mit.edu

Aarush Gupta
Massachusetts Institute of Technology
Cambridge, MA, USA
aarushg@mit.edu

***Abstract*—This paper presents Frequency Perfect Generated Audio (FPGA), a real-time autotuning system for the human voice. The system consumes 16-bit, 44.1KHz audio and produces autotuned audio of the same resolution after a delay of roughly 0.1 seconds. Popular autotune implementations involve computationally expensive signal processing algorithms — this work focuses on optimizing these algorithms onto an FPGA.**

The FPGA system contains several key stages of processing: I2S reception from an INMP441 [1] I^2S microphone, pitch detection, pitch correction, and output of processed audio (through FPGA speakers or over UART via USB). Each stage processes a 2048-sample "window" of audio. For maximal efficiency, the system pipelines computations across stages on a per-window basis. For efficient pitch detection and correction, we implement hardware-optimized versions of the Yin autocorrelation and time-domain PSOLA algorithms, respectively.

The code (HDL, testbenches, simulations, etc.) for the entire system can be found here.

***Index Terms*—audio, real-time, signal processing**

I. Safety

Our project revolves around generated audio, which sometimes sound harsh or unpleasant. We have taken care to reduce artifacts, pops, and static in our audio, however we still recommend general caution and the use of over-ear rather than earbuds headphones while testing.

II. Introduction

Autotune systems are a form of audio processing that receive audio samples and "tune" them. This involves shifting each note in the input to the closest note in a quantized scale, whether that be a standard Western scale corresponding to some key or any arbitrary ascending sequence of pitches. Such systems are frequently used to enhance audio recordings of human singing, either live or through post-processing [2].

Autotune systems, as audio processing systems, necessarily include computationally expensive signal processing algorithms such as Fourier transforms [3]. Many current state-of-the-art autotune systems perform two key steps to correct pitches: *pitch detection* and *subsequent pitch correction*.

The Yin autocorrelation algorithm (hereafter simply referred to as "Yin") is frequently used for pitch detection. Yin analyzes the periodicity of an audio signal using a modified autocorrelation approach [4]. The algorithm computes a difference function $d(\tau)$ for each possible time lag τ, quantifying how similar the signal is to a shifted version of itself. To enhance accuracy, Yin applies the cumulative mean normalized difference function (CMNDF), which highlights the true periodicity by normalizing $d(\tau)$ with the accumulation of previous values [5]. The true period of the signal is then estimated as the lag τ that minimizes this CMNDF, meaning shifting the signal by this period leads to minimal change in the signal.

A popularly used algorithm to preserve signal timbre while shifting pitch is the Time-Domain Pitch Synchronous Add and Overlap algorithm (hereafter referred to as "PSOLA"). PSOLA computes representative pitch periods for the signal, in a manner similar to the Yin autocorrelation algorithm [6]. Then, the spacing between the centers of these pitch periods is stretched or shrunk depending on the desired pitch shift. This adjustment is achieved through a sophisticated interpolation scheme, which is described in detail later in this paper. At a high level, this approach compresses or stretches the input signal to change its frequency while preserving its waveform shape.

This project focuses on taking these existing algorithms and signal processing techniques and efficiently optimizing them on an FPGA. Both Yin and PSOLA involve a high number of mathematical operations including additions, multiplications, divisions: just 1 second sampled at 44.1 kHz requires millions of operations to autotune! This problem is thus well-suited for an FPGA: through our efficient high-level system design as well as micro-optimizations, we build a low-latency autotune system performing these algorithms.

III. Design Choices

A. Parameter Selection

1) Audio Parameters: We process 16-bit 44.1KHz audio as this is a common encoding for high-quality audio that is also not extremely resource intensive to process. Firstly, at 44.1KHz, we have $\frac{10^8}{44{,}100} \approx 2267$ cycles to process one sample of audio and roughly $2267 \cdot 2048 \approx 4.64 \cdot 10^6$ cycles to process a full window of audio. Based on our calculations (see the following sections), this is an ample number of cycles to autotune one window. Secondly, because samples are 16 bits wide, we can sign/zero-extend samples and add fractional precision, while still keeping our bitdwith small enough for DSPs to do multiplications in one cycle.

2) Window Size: Audio is processed in windows — choosing this window size effectively is crucial for system design and performance. The first constraint imposed on our window size is due to the high number of arithmetic operations (especially multiplications and divisions) required by Yin. On a window of n samples, Yin requires $\frac{n(n-1)}{2}$ multiplications

and $O(n)$ divisions. Using a window size of 2048, Yin requires roughly 2.1 million multiplications and at most 2048 divisions per window. Due to this quadratic growth, our window size is bounded by the rate at which we can perform multiplications. In addition, our window size must also be small enough that the delay to receive and autotune one window is not disruptive, as our system is real-time.

Despite the constraints that Yin and our real-time characteristic impose, we also require a window size *large* enough for low frequency pitches to oscillate multiple times, which allows Yin to find local minima corresponding to those pitches in the $d(\tau)$ function. The lowest note in the classical bass range, E2 (≈ 82Hz), has a period of $\frac{44100}{82} \approx 538$ samples. Allowing this pitch to oscillate twice thus requires at least 1000 samples in our window.

After testing Yin in software, we found that a window size of 2048 produced sufficient pitch detection capabilities to detect low notes, while being feasible to implement on the FPGA. Furthermore, the delay in auto-tuning one window of this size is at most $\frac{2048 \cdot 2}{44100Hz} \approx 0.09s$.

B. Choice of Algorithms

A naive approach to pitch detection involves performing a Fast Fourier Transform (FFT) on the signal and taking the highest peak. However, taking the strongest FFT peak is not sufficient to detect the "heard" pitch of a signal due to intricacies of the human voice; subharmonics and overtones can often be stronger than the fundamental frequency.

A similarly motivated naive approach to correct the signal pitch would involve performing an inverse Fast Fourier Transform (IFFT), adjusting all frequencies to align with the quantized scale, and then transforming back to the time-domain using a FFT. While this method achieves pitch correction, it does not preserve the timbre of the input signal (i.e., its specific waveform shape) [7].

In addition to being algorithmically superior to FFT based approaches for our particular problem, Yin and PSOLA both operate completely in the time domain, making them more natural to implement on the FPGA.

IV. High-Level System Design

A system diagram is included in the appendix, which makes clear how all modules and I/O systems interact.

A. I/O

We use the INMP441 I^2S microphone to capture audio data. The microphone specification mandates an I^2S sequence using 64 cycles of `sclk` to read one sample from the microphone. At an audio rate of 44.1KHz, 2268 FPGA cycles (at 100MHz) pass between samples. Therefore, we use 36 FPGA cycles per `sclk` cycle, resulting in one sample every 2304 FPGA cycles, for an actual sampling rate of 43.4KHz.

To play back pitch-corrected audio directly from the FPGA, we use a PDM, which supports a frequency range large enough to convey the human voice. We chose this scheme rather than a PWM because the PWM's frequency range was severely limited to lower frequencies.

Finally, we also support transmitting autotuned audio from the FPGA via UART over USB. This allows us to play back autotuned audio as `.wav` files and collect results for testing. Because `pyserial` does not suport transmission of values larger than 8 bits, we use a module called `uart_turbo_tx` that takes a wider value (generallly 16-bit) as input, and sends it over several distinct UART transmissions. The receiver then reassembles the bytes into values. This allows us to stream 16-bit audio or pitch data over UART via USB.

B. Window Pipelining and Timing

In order to efficiently stream data through our system, we pipeline our algorithm on a per-window basis. The main modules are the I2S receiver, Yin module, and pitch correction module. These modules are perfect for pipelining as they shared limited hardware and operate in a sequential order on a given window. For reasons mentioned below, we also wait an extra window before actually outputting the signal produced by PSOLA.

C. BRAM Usage for Signal I/O

Due to the large window size of 2048, we use BRAMs to store all input signals as well as output signals produced by modules. An especially important aspect of this is how the PSOLA signal is outputted: PSOLA can actually produce windows with *less than* 2048 samples, which can cause issues with outputting a continuous stream of audio in real-time. To fix this, we store the PSOLA output in a BRAM, effectively used as a FIFO buffer, and wait an extra period at first before outputting signal so that the buffer always contains roughly 2048 samples. Note that PSOLA's output length can never be less than $2^{-(1/12)/2} \cdot 2048 \approx 1990$ nor more than $2^{(1/12)/2} \cdot 2048 \approx 2109$ since the maximum shift in period will be half a semitone. Along with the assumption that the distribution of output lengths across many periods will randomly vary and thus even out, we can reasonably assume that a fixed-size FIFO will neither overflow nor have much less than 2048 samples.

V. Pitch Detection: Yin Autocorrelation

A. Algorithm

In addition to computing the difference functions for each possible value of tau, YIN minimizes the *cumulative* difference (obtained by dividing this difference over the prefix sum of all previous differences up to and including the current difference), to mitigate the effect of resonance at the first formant. The fundamental period is either the period that minimizes this cumulative difference, or period that represents the first local minimum underneath a cumulative difference value of 0.1, to avoid subharmonic (octave) error from higher periods.

We adapted Yin for hardware by splitting it into two phases: difference function calculatation and CMNDF mininum finding.

B. Difference Function Calculation

We make use of two BRAMs: one to hold samples, the other to hold the intermediate difference calculations. The first phase occurs as the window is being streamed in. There are 2304 cycles before the next sample to compute the current sample's contribution to every difference function. This involves reading back a prior sample from BRAM (2 cycles), subtracting the two values (1 cycle), squaring the result (1 cycle with DSP), reading the current difference value for that τ from BRAM (2 cycles, started concurrently with subtraction), adding the squared value to the current difference (1 cycle), and writing back the result (1 cycle). In the worst case, with a period of 2048, this calculation will happen 2048 times. We pipelined this calculation over 2-cycle stages, so the total number of cycles required for n serial computations is $2n+4$. We chose $n = 512$ to safely remain under the cycle count of 2304. This requires four computations to be done in parallel. We sharded sample data across 2 BRAMS to use all four read ports on the same cycle, and used 4 DSPs. We chose to process 4 consecutive sample values in parallel, to take advantage of the fact that each sample would be unique mod 4 and the period value (the difference between the current sample index and that sample index) would be a different, but still unique, number mod 4. This allows for an intuitive addressing scheme for both the sample and diff BRAMs and an consistent way to mux between them.

C. Cumulative Difference/Minimum Finding

The next phase occurs after the current window has been streamed in and calculates the cumulative difference and minimum tau in a single loop. Because it uses the previous window's diff values as the next window is being streamed in, we require two more diff BRAMs, and will alternate reading and writing from each BRAM every window so the previous window's diffs are preserved for the duration of another window.

We first read the diff value from BRAM (2 cycles), add it to an accumulating prefix sum register (1 cycle), divide the diff value by this prefix sum (8 cycles), and compute one iteration of the min-finding algorithm (comparing this value to the previous minimum and updating the minimum tau if it is lower, with the early exit conditions specified in the algorithm). Since we have four ports available from these diff BRAMS, we perform these four calculations in parallel. Because the critical path is in the min-finding phase, we split it up into two cycles (one for the first two diffs, the other for the next two diffs). This requires 12 cycles per computation, for a total of 6144 cycles for 512 serial computations, which leaves. about 4.7 million cycles in the current window for the PSOLA algorithm to operate.

The bottleneck of this computation is the division. We wrote a fixed-point divider that utilizes a truncated version of the shift-and-subtract algorithm, taking advantage of the fact that the dividend will always be less than or equal to the divisor to improve division latency.

VI. Pitch Correction: PSOLA

We implement PSOLA in its standard form while also making a number of optimizations and simplifications, considering the trade-off between output quality and computational efficiency. A key optimization is that we omit the autocorrelation analysis generally required in PSOLA through a non-standard reuse of the Yin output.

A. Algorithm

The first step of performing pitch correction is to actually find the desired frequency shift; note that for autotuning, we quantize at the *semitone* granularity. Assuming a reference note of A4 at 440 Hz, corresponding to $1/(440\text{ Hz}) \times 44.1\text{ kHz} \approx 110$ samples, we find the closest value $110 \times \left(2^{1/12}\right)^i$ for $i \in \mathbb{Z}$ to the period P_Y inferred by YIN for the current window. (Note that $2^{1/12}$ corresponds to the ratio in periods of adjacent semitones.) We refer to this shifted period as P_S.

After calculating the shifted period P_S, PSOLA computes the shifted signal using the following algorithm. Note that this is for a *single window*, i.e. the pitch period is assumed to be constant. We denote the window length as W, which we choose as 2048.

1) Initialized `processed` as array of zeroes with length $W\frac{P_S}{P_Y}$. `signal` is the original signal for the window of length W.
2) $i \leftarrow 0,\ j \leftarrow 0$.
3) While $i < W - P_Y$:
   a) $\texttt{processed}[j-P_Y : j+P_Y] \leftarrow \texttt{signal}[i-P_Y : i+P_Y] * \texttt{window}$
   b) $i \leftarrow i + P_Y,\ j \leftarrow j + P_S$.

Note that `window` is the window function used for interpolation — for this implementation, we choose to use the Barlett window of length $2P_Y$

$$\texttt{window}_{\text{barlett}}(i) = \begin{cases} \frac{i}{P_Y} & \text{if } 0 \le i \le P_Y, \\ 2 - \frac{i}{P_Y} & \text{if } P_Y < n \le 2P_Y, \end{cases}$$

which peaks at index P_Y and linearly decays to zero on either side (i.e. a triangular window).

This implementation of PSOLA is heavily inspired by [8], with two key changes.

- We start processing windows centered at P_Y rather than at $P_Y/2$ as in [8], so we do not have to access a previous window of data. We found no significant differences in audio quality from this change.
- Some samples at the beginning and end of the output do not have peaks on both sides of them in the given window — the original PSOLA algorithm multiplies all original signal values by the interpolation window before adding them to the processed signal, but we only perform this multiplication when there are indeed peaks on both sides of a sample (i.e. corresponding to interpolation). This has the effect of not decreasing the amplitude of the samples at the beginings and ends of windows, which improved audio quality.

B. Hardware Implementation, Algorithm

PSOLA on the hardware level operates in two main phases.

1) Initial Computations: Compute the shifted period P_S, used as mentioned above in the PSOLA algorithm, and the inverse of the original period $1/P_Y$, which is repeatedly used for computing the Barlett window values (e.g. value at index $i < P_Y$ is i/P_Y. To find P_S we use a searcher module which iterates through ROM lookup table of "correct" periods until the closest one to P_Y is found and outputted. In parallel to this searcher, we use a fixed point divider to compute $1/P_Y$ over multiple cycles.

2) PSOLA Algorithm: After the initial computations are complete, we run the previously described PSOLA algorithm. We iterate over i and j, and then iterate over the signal values in each window of length $2P_Y$. The algorithm pseudocode describes multiplying the window array by the corresponding slice of the signal. In reality, we iterate over each index in the window, multiply the signal value at that index by the Barlett window function value, and add to the processed signal. We perform nearly all intensive operations sequentially due to the high number of cycles (roughly 5 million) available to us.

C. Hardware Implementation, I/O and BRAM Interfacing

Note that the above implementation is complicated by the fact that signal and output values are stored in BRAMs. To handle read and writes efficiently, we created a BRAM wrapper module around the PSOLA module: this module contains one BRAM which stores signal values and one BRAM for storing PSOLA output. We pipeline the interface between the signal and output BRAMs and PSOLA, as well as the logic within PSOLA, to avoid the two-cycle delay associated with BRAM reads and writes. As a result, the number of cycles this pipelined and optimized PSOLA implementation takes is roughly $2W$, where W is the window size: this is extremely low compared to the millions of cycles allowed for the computation!

Additionally, as mentioned earlier, we pipeline windows in our higher-level system, i.e. PSOLA will run on window i while window $i + 1$ is being streamed in. Thus, the signal BRAM in the wrapper module is actually of depth $2W$, where W is the window size: each input window is read into an alternating half of the BRAM while the other half (corresponding to the previous input window) is used as input to PSOLA. Reading in the signal values occurs fully in parallel with the PSOLA-related operations in the wrapper module to avoid high cycle delays. For PSOLA-related operations, there are two main phases: the core PSOLA module first runs until completion, after which the processed signal values stored in the output BRAM are iteratively outputted from a port in the BRAM wrapper module.

1) On-Board Optimizations: Several system components were added to the design to tackle the unique complications of PSOLA "on the board". First, audio data is read out from the serialization buffer 2048 times per window, which will start pulling garbage if fewer than 2048 samples are written back from PSOLA. This results in several audio discontinuities, or pops, which makes the output unpleasant to listen to. Instead of incrementing the read pointer at this point, we modify the buffer to play back a cache of previous audio values to smooth over the difference in window size until the write pointer catches up. In addition, we add a postprocessing step that replaces all audio output values less than a certain threshold (`0x00008000`) with the median possible audio value (`0x80000000`, for 32 bits) to remove any remaining discontinuities.

D. Additional Modules

Additional modules not covered in detail include an I^2S receiver, a 16-bit UART transmitter, a pipelined fixed-point divider, a specialized ring buffer for improving PSOLA output, and a "searcher" for finding the closest semitone to a given pitch.

VII. Evaluation

A. Software Simulations

The preliminary evaluation of our system design was through our software simulations. As mentioned previously, we wrote a full software simulation of our system before writing any RTL: this process allowed us to evaluate the correctness of our algorithms through the generated, autotuned audio.

B. Testbenching

One key component of evaluating our actual design written in System Verilog was testbench output. For instance, Yin and PSOLA were both testbenched in detail to ensure correctness. Yin values were compared to the software implementation. For PSOLA, the waveform produced by the hardware algorithm was plotted and compared to the original signal. This plot was also compared to that of the software implementation.

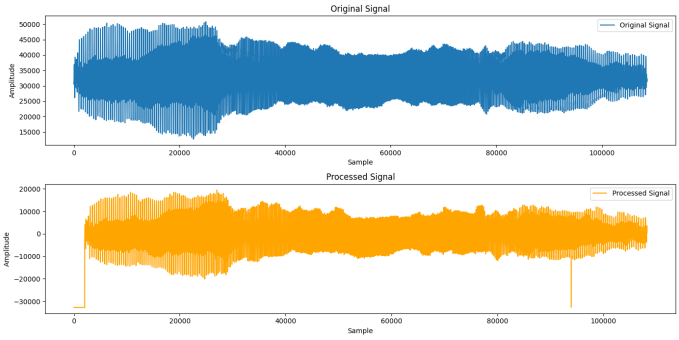

Fig. 1. Plots of a sample input waveform and the resulting output waveform produced by the PSOLA algorithm. The output is offset by one window — the first 2048 samples in the output audio are dummy placeholder values. Aside from one irregular spike, PSOLA clearly preserves the timbre of the input signal. Pitch-shifting is hard to observe from the waveform, but was clearly noticeable when the saved `.wav` file was played.

Generated plots such as the ones above allowed us to verify that the hardware PSOLA implementation preserved the timbre of input audio while also not producing any artifacts. Similar testbenching for wrapper modules and the fully integrated

system also allowed us to confidently evaluate our end-to-end pipeline as producing autotuned, timbre-preserving output audio.

C. Utilization

Our design uses 10 DSP blocks (8.33% of maximum), 6 18KB BRAM tiles, and 18 36KB BRAM tiles (28% of maximum). Aggressive pipelining allows us to minimize DSP utilization while achieving the compute density needed for Yin. Meanwhile, we use BRAM more heavily to duplicate signal values, allowing different modules to access windows concurrently. We also use BRAM to "smooth" over PSOLA output via a specialized FIFO. By rewriting PSOLA to output values at a consistent rate, we could eliminate this buffer. However, attempts at this made PSOLA much more difficult to test, as outputting samples at a slower rate dramatically increased simulation time.

D. Physical Evaluation

The most important aspect of our evaluation was in the actual physical system, which was especially important for I/O — previous evaluation used audio not recorded through a mic.

While some static was present, we were able to read high-quality audio signals from the microphone. The static present did not significantly interfere with the operations of the Yin or PSOLA algorithms.

Our system also produces relatively clean, autotuned audio, as is demonstrated in our final video. There are some minor artifacts such as "humming" in the produced audio, but the overall timbre of sound is quite similar to that of a human voice (individuals can be distinguished by their autotuned output). Additionally, the system clearly "snaps" each window of audio to the closest semitone, which is especially apparent when a pitch slide is sung into the microphone.

However, there were a few key limitations that our on-device system has.

1) *Popping:* There are occasional "popping" sounds, which manifest themselves especially prominently after sudden volume changes. The measures taken to cover these up produce metallic artifacts. We like these effects, because they sounds like T-Pain, but it is far from seamless.
2) *Filtration:* We attempted to process out noise and pops with a band-pass filter over the range of frequencies that Yin measures, using the Butterworth polynomial to determine the pole locations. This removed the popping, but filtered out a band of overtones that made the resulting audio partially lose its human timbre.
3) *System Robustness:* Our system is very sensitive to the exact amplitude and location of recording, and sometimes produces inconsistent results on input samples recorded in different ways (e.g. different microphone location or angle). This likely has to do with limitations in the clipping, filtering, and communication protocols which were used to optimize the on-device system.

Conclusion: While there are some artifacts and audio quality issues, Frequency Perfected Generated Audio (FPGA) works as intended in software, hardware simulation, and on the actual FPGA, autotuning input vocals in a timbre-preserving manner!

VIII. Implementation Insights and Reflection

Measure Twice, Cut Once: very early on in the project, we worked for several weeks writing software simulations of our algorithms. Firstly, this served to confirm that we understood the algorithms and were getting the expected results. Second, it is much faster to iterate in software versus hardware. Therefore, when we found algorithmic issues or had make algorithmic changes, we were able to to so easily. When it came time to writing RTL, we had a very clear idea of timing specifications, connections between modules, and exactly what calculations each module needed to perform.

Leverage `cocotb`: We heavily leveraged the fact that `cocotb` testbenches allow for full use of `python` functionality. An extremely useful aspect of this was saving testbench output as `.wav` files and playing it back to identify errors. Additionally, plotting produced waveforms was extremely useful in debugging PSOLA.

Write Software Simulations in a Hardware Friendly Way: While software simulations confirmed our algorithmic design, many implicit operations were not hardware-friendly. For example, using a single `numpy` function often translated to a nontrivial module and integration task in hardware. We found that writing very simple, clear simulations involving only basic operations, such as arithmetic and list indexing, allowed us to translate simulations to RTL more smoothly.

IX. Conclusion

A. Summary

In summary, Frequency Perfected Generated Audio is a real-time autotune system consisting of hardware-optimized versions of the Yin and PSOLA algorithms, using the I^2S and UART protocols for communication with an INMP441 microphone and an external audio sink, respectively. Key design choices include pipelining computations at a window level, choice of window and audio parameters, as well as BRAM design for signal storage. At a lower level, extensive pipelining, fixed-point representations, and other algorithm optimizations allowed Yin, PSOLA, and integration modules to run within the cycle limit for one audio window.

As thoroughly shown in our evaluation section, our system works as desired, producing a relatively clean, autotuned stream of audio based on the microphone input. Modules were also shown to work in software and `cocotb` simulation. As discussed in the evaluation section, our on-device system did have limitations in audio quality as well as produced audio artifacts in the output: future work could look to add further filtering and artifact-correction mechanisms to produce a cleaner output signal.

X. Contributions

All team members worked on high-level system design. Shruti worked on the Yin modules, Felix worked on I/O handling and end-to-end module integration, and Aarush worked on the PSOLA modules.

XI. Acknowledgments

We sincerely thank the course staff for their support in office hours and feedback on project deliverables. We also found Terry Kong's description of TD-PSOLA [8] particularly useful for deriving a hardware friendly implementation.

References

[1] InvenSense. *Omnidirectional Microphone with Bottom Port and I2S Digital Output*, 10 2014. Rev. 1.1.

[2] Charles Dodge and Thomas A. Jerse. Computer music: Synthesis, composition, and performance. *Schirmer Books*, 1997.

[3] Alan V. Oppenheim and Ronald W. Schafer. Discrete-time signal processing. *Pearson*, 2009.

[4] Alain de Cheveigné and Hideki Kawahara. Yin, a fundamental frequency estimator for speech and music. In *Journal of the Acoustical Society of America*, pages 1917–1930, 2002.

[5] Alain Cheveigné and Hideki Kawahara. Cumulative mean normalized difference function in yin algorithm. *JASA*, 2002.

[6] F. Charpentier and M. Stella. Pitch synchronous overlap and add method for pitch modification of speech. *ICASSP*, pages 19–24, 1986.

[7] Julius O. Smith. The inverse fft for audio pitch correction. In *AES Convention Paper*, pages 101–108, 2003.

[8] Terry Kong. Phase vocoder implementation with flwt and td-psola. https://web.stanford.edu/class/ee264/projects/EE264_w2015_final_project_kong.pdf, 2015.

XII. Appendix: System Diagram