THE RTSTFT FRAMEWORK

Jackson Kaplan

Abstract

RTSTFT is a library written in pure C, and includes a phase vocoder algorithm with several novel ways for users to manipulate the algorithm directly to achieve highly unique sounds. It also includes a small built-in command line tool with its own language, rt_cmd, to allow for precise control of these manipulations. At its simplest usage, RTSTFT can be utilized much like an audio equalizer or filter, but its strengths lie with how much access to the underlying algorithm is given to the user. Not only is it an extremely competent pitch shifter, capable of handling complex harmonic content at certain settings, but it also deigns to break into as-of-yet unexplored territory in experimental music production.

This paper is intended to not only outline the specifics of RTSTFT, but provide an in-depth and hopefully intuitive explanation of all the underlying mathematics, as I found that there are very few beginner-friendly resources for learning about advanced DSP[1] algorithms such as this one. As such, this paper is intended to be as educational as it is technical, and I hope that any who come across this paper find it useful. I insist that anyone trying to learn from what I've written here contact me with any questions, as I believe the knowledge I've employed in the creation of RTSTFT should be easily accessible to everyone without needing to spend hours familiarizing oneself with academic DSP symbology.

Acknowledgements

RTSTFT is a work inspired by the growth in my musical abilities inspired by many of my peers and professors at CalArts. In particular, I'm incredibly thankful to Hanna for helping my grow my taste in music and discover the types of sounds that inspired my to create RTSTFT, and to Tracey for being the expert sonic experimentalist I needed around to actually feel capable of tackling the types of sounds I wanted to recreate. A huge thanks to John Tejada and John Baffa for helping me solidify my knowledge of musical creativity and mix engineering, without which I wouldn't have even known where to begin when trying to design my own audio processors. And finally thanks to Jacob Penn for the monumental amount of assistance in restructuring the architecture of RTSTFT's codebase, bringing it from an over-engineered monstrosity to a streamlined system that can actually accomplish the goals it was created for.

[1] Digital Signal Processing

Background & History

Motivation and philosophy behind creation

Though I have been playing music since I was an adolescent, I only felt like I came into my own when I started exploring digital music production. As time has gone on, I've found myself drawn more and more to extremely experimental variants of sound design, to a point where in the past several years I've started to feel limited by the tools available to digital musicians. As I've grown as a programmer, I've begun to realize that there is a whole world of possibilities that isn't being explored by most plug-in designers. The vast majority of digital audio plug-ins seek only to emulate their analog ancestors, which in my view, is deeply near-sighted and fails to take advantage of the boundless possibilities that exist in the realm of digital audio. Analog audio processors are often revered for imparting a more "natural" and "warm" sound, and in this, they invariably excel—to the point that it becomes futile for digital processors to attempt to compete with them. Instead of trying to chase such a fever dream, I am of the opinion that far more resources should be devoted to finding new ways that digital audio manipulation can generate results that would be otherwise impossible. One avenue that has not been explored by the industry is by giving users much more direct control over these algorithms, as the few experimental plug-ins that do exist tend to hide most of the technicalities of their operation for fear of seeming too complex or intimidating.

This is the core of the intention behind RTSTFT. It gives unadulterated control over almost every facet of its underlying DSP, and tries to do so in a way that is still accessible to users who aren't experts in the field. It's a very difficult balance to strike, but without even an attempt at it, music producers and sound designers might never know the world of possibilities that exists for them but has been kept out of their reach. Not only is RTSTFT meant to be accessible, but it is also ideally meant to be informative. The more users know about what's going on under the hood, the more they'll be able to intentionally bend audio in new and exciting ways. Further, it's secondarily founded on the idea that many skills that come as second nature to programmers should be universal to everyone, without needing to dig through pages on pages of unfriendly manuals and forum posts just to get a handle on the basics. The main example of this that RTSTFT is intended to convey is the usage of the command line. The wealth of tools and time-saving capabilities that a computer's command line has should be something that everyone has access to, but unfortunately the command line itself serves as a barrier to wider-spread adoption with its bizarre syntax that can come across as completely alien to the uninitiated. With a (hopefully) friendly command line interface custom-made to act as a friendly introduction to using the command line, it is my hope that some users will take RTSTFT as a jumping-off point to learning how to use their own computer's command line, a skill that I see as a necessity so that people can spend less time doing battle with programs that have complicated graphical interfaces and more time actually completing their tasks and doing what they actually intend to.

RTSTFT is hopefully the first in a line of many audio tools built to make users more comfortable with the more technical side of the software they use, in hopes that it may unlock creativity and greatly increase the range of possibilities that musicians and sound designers have when making their art.

History of spectral audio

Throughout the history of recorded audio, there have always been tools to modify the incoming sound in order to have it fit the artists' and producers' preferences. The earliest of these were circuits that were already heavily employed in fields that had little to do with audio; electrical mechanisms like filters and amplifiers were extremely important in maintaining the growing power grids of the era. Even synthesizers, often seen as the first step in the advent of electronic music, were still fundamentally analog, and thus shared the same limitations as the other audio tools available to musicians in the studio. It was only with the advent of computers did decades, if not centuries, of signal processing theory in the realm of academics become viable for use in music production. One of the first examples of a digital machine being used in the studio was Eventide's H910 Harmonizer(*H910 – The Halls of Valhalla*, 2010). Though there were technically analog devices capable of doing this(*Pitch Shifters, Pre-Digital – The Halls of Valhalla*, 2010), they involved complex systems of multiple tape heads rotating with a slight offset. The H910 offered an extremely simple and intuitive interface to achieve pitch-shifting without the hassle of such a delicate analog system, and its popularity grew exponentially soon after its release. However, the era of digital music didn't begin until two important milestones were reached: first, the ability of mainstream computers to be able to handle large quantities of audio, and second, the release of Antares Auto-Tune.

Andy Hildebrand was an electrical engineering PhD who was contracted by Exxon in the 1990s to develop software that would allow them to more accurately decipher seismic data in order to better locate underground oil reserves.("The Mathematical Genius of Auto-Tune," 2016) However, as a classically-trained musician, Hildebrand soon realized that these same techniques could prove to be useful to music producers as well. He went on to develop Auto-Tune, a digital pitch corrector that completely changed music as we know it today. Utilizing an algorithm known as the phase vocoder (which lies at the core of RTSTFT as well), Auto-Tune is able to change the pitch of an incoming audio signal, and can do so automatically by detecting the original pitch using a technique known as autocorrelation. Since the release of Auto-Tune, there have been countless audio plug-ins that utilize spectral techniques such as the phase vocoder to alter audio in ways that analog devices are simply incapable of achieving. However, many of these digital algorithms, and certainly all the spectral ones, would not exist if not for the work of French mathematician Jean Baptiste Fourier.

Though Fourier's influence on modern audio processing is debatably greater than any other individual contributor, his work had nothing to do with sound; instead, his mathematical theories were focused on the behavior of heat.(Jessop, 2017) Fourier's discoveries were looked upon with skepticism in his own time, but are now regarded as one of the foundations of modern mathematics. Without getting too technical, the core of Fourier's contributions is the idea that *any* periodic signal can be broken down into a series of harmonic sinusoid waveforms. The function that allows this decomposition has since been termed the Fourier transform, and it now serves as the foundation for countless fields, ranging from music production and engineering to the study of quantum physics. As we'll see, though the math can seem a little daunting, it is extremely elegant in its core ideation, and in application it presents a massive space of possibilities for experimental sound design that I believe have barely been explored.

Mathematics

For this section explaining the math behind RTSTFT, I will be using *extremely* explicit notation, and explaining concepts that may seem trivial or unnecessary to those with more experience with the maths used. I do so in order to create an explanation that I wish had existed when I was learning all of this on my own. The technical parts of this paper are intended for a anyone with only a fuzzy recollection of highschool level calculus, and insist upon instilling the reader with a deep understanding of the actual mechanisms and reasoning behind the math instead of just the computations alone. This does result in much lengthier, wordier explanations, but I hope it will also lead to the paper being more accessible to anyone learning these concepts on their own.

An Overview of the Fourier Transform

For some set of N input values x_{input}, the Discrete Fourier transform (DFT) will return an array of values x_{output} at an index k defined as[2]:

$$x_{output}[k] = \sum_{n=0}^{N-1} x_{input}[n] \cdot e^{\frac{i2\pi kn}{N}}$$

Note that this set of input values, or "signal" as it will be hereon referred to, is zero-indexed, hence taking the sum from 0 to N-1. This equation may certainly seem a little daunting, but we can break it down a bit by noting that it shares the form of Euler's formula, $e^{ix} = \cos x + i \sin x$. Applying this, we get the following:("Discrete Fourier Transform," 2022)

[2]For those who may not know: the sigma symbol $\sum$ simply indicates adding up the values of an expression for all integers in the given range. As such, $\sum_{n=0}^{N-1} f(n)$ simply indicates determining the output of $f(n)$ for all n from 0 to $N-1$, and then summing all these values together.

$$x_{output}[k] = \sum_{n=0}^{N-1} x_{input}[n] \cdot (\cos(\frac{i2\pi kn}{N}) + i\sin(\frac{i2\pi kn}{N}))$$

Though this seems no less complicated, it tells us something very fundamental about the equation: it is returning a value that lies on some circle where the x-axis is real and the y-axis is imaginary. The DFT imagines the input signal as some addition of many different harmonic sine waves, where the lowest frequency sinusoid has a frequency of 1/N samples, and the highest a frequency of 2 samples. The inner portion of the equation, $x[n] \cdot \frac{i2\pi kn}{N}$, can be interpreted as the complex "response" of a given frequency $\frac{1}{k}$ at a given sample n, with the sum of all these complex numbers representing the response

What matters is, when we take the absolute value and the angle of a complex number at some index k of x_{output} (corresponding to taking the radius and the angle of the complex number on its imaginary circle), the values we get back are the amplitude and the phase of the sinusoid with the frequency $\frac{1}{k}$ samples.

Let me state that again in simpler terms, for emphasis: ***the DFT decomposes any input signal into a set of sinusoid waves, and gives us the phase and amplitude of each of those sinusoids***. This mathematical operation is not an estimation: it is 100% accurate and completely reversible. Even for people familiar with the DFT, the significance of this cannot be overstated: being able to decompose a signal like this, turning its time-domain information into frequency-domain information, is an invaluable tool in almost any field of mathematics that deals with periodic signals. This explanation still might not suffice for many, as even I still have trouble fully linking the variables in the equation to what's actually going on under the hood. I insist you go to 3blue1brown's YouTube Channel (3Blue1Brown, 2018) and take a look at his video on the DFT; I find the visualizations to be extremely helpful in building a deeper understanding of how the DFT really works.

The DFT lies at the core of countless more complex signal processing algorithms, such as the ones that lie at the heart of RTSTFT: the phase vocoder, and the short-time Fourier transform upon which it depends.

The Short-Time Fourier Transform

The short-time Fourier transform (STFT) is a method of extracting data about how the phase and frequency distribution of a signal changes over time. This is achieved by taking overlapping sections of an incoming signal, which is called *windowing*, and performing a Fourier transform on each of these windowed frames. Practically, this is generally done using a fast Fourier transform (FFT), which is an optimized version of the normal DFT that takes advantage of symmetries in the math that can be ignored when working with real-valued input signals.

The STFT is often performed on overlapping frames of signal, with the factor of the overlap $F_{overlap}$ determining how much the samples frames overlap. This overlap distance, defined as the frame size N divided by $F_{overlap}$, is referred to as the “hop,” as it is the distance the algorithm hops to get from one frame to the next. As an example, an STFT with $F_{overlap} = 4$ and $N = 1024$ would take a 1024-sample frame from the input signal every 256 samples. A long input signal of 4096 samples, when run through this STFT, would return 13 windowed frames, each sampled 256 samples apart from each other.

These overlapping frames will also have a *windowing function* applied to them.(Götzen et al., 2000) This reduces the power of samples at the edges of the frames, and tends to increase the precision of the STFT. The Hanning window is the most commonly used of these:

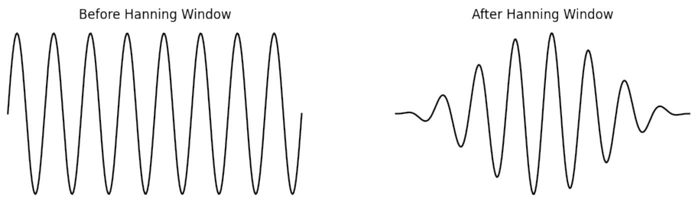

Figure 1: Sinusoid Hanning windowing

Though the STFT is excellent for producing visualization and other analyses of long or real-time signals that would defy easy analysis via a single FFT, their real power becomes apparent when their tiered nature is leveraged to perform complex manipulations on the input signal that would otherwise be impossible.

For a signal that is comprised of M overlapping frames of size N, any frame $m \in \{0, ..., M-1\}$ in the array of analysis frames A[3] is defined by the following equation:

$$A_m[k] = (\sum_{n=0}^{N-1} x_{input}[n] \cdot e^{\frac{i2\pi kn}{N}}) \cdot w[n]$$

for the output fra x_{frame} in the input sequence x_{input}, and the windowing function $w[n]$ for the given sample n.

To reconstruct the input signal, one need only add the frames back together in the same order they were extracted, an operation defined as such:

$$x_{output}[n] = \sum_{i=0}^{m-1} = A_{i,(n-\text{hop}_a)} \cdot w[n] \cdot (u[n - i * \text{hop}_a] - u[n - i * \text{hop}_a - N])$$

[3]We denote these frames as "analysis" frames as they represented the analyzed amplitude and phase information from the incoming signal. This distinction will be very important when we start modifying the frames using phase vocoding.

where hop_a is the hop size calculated as $\text{hop}_a = \frac{N}{F_{overlap}}$, and $u[n]$ is the unit step function, defined as:

$$u[n] = \begin{cases} 1 & n \geq 0 \\ 0 & n < 0 \end{cases}$$

In this context, the unit step function is simply used to excluded any of the frames that aren't overlapping the current frame m. This entire reconstruction operation is known as "overlap-adding," as it is quite literally summing the overlapping frames back together. Note how we also re-apply the windowing function during the overlap-add: during the extraction phase, the window is used as it helps increase the resolution of the FFT, while the windowing during the overlap-add ensures a smooth transition between overlapping frames.

Next, we'll look at how using an hop size during overlap-adding that is different from the one used during frame extraction can allow for pitch-shifting and time-stretching, as well as how the phase vocoder corrects the phase issues that arise from doing so.

Phase Vocoder

The phase vocoder is an algorithm that allows for frequency-domain manipulation of signals while keeping the time domain constant. Most likely, you'll know that speeding up audio, like when you change the rotational speed of a record player, changes the pitch. What a phase vocoder can do is change the overall time of a signal while keeping the pitch constant, or change the pitch while keeping the time constant. The way it does this is by ensuring that the phases of each individual sinusoid, or "bin", aligned between the frames. In short, a phase vocoder does this by taking the phase of each bin and adjusting it using the phase of the same bin from the previous frame.

Time-stretching with an STFT can be accomplished by simply overlap-adding the windowed frames with a different hop size—e.g., the same 1024/256 STFT mentioned above could make its input audio twice as long by simply re-layering the frames 512 samples apart as opposed to the original 1024. To pitch-shift, one need only resample this audio to its original length; for instance, an audio clip stretched to twice its length could be resampled to its original length to shift its pitch up one octave. We can call this two hop sizes hop_a (the *analytical* hop size) and hop_s (the *synthetic* hop size). The difference between these two sizes can be referred to as the "stretching factor," which will be notated as S in this paper.

Unfortunately, using this technique when overlap-adding introduces some issues. Namely, the STFT does not perfectly account for changes in frequency between frames when said frequency exists somewhere between bins: this is called the "phase coherence" problem.(Srinivas et al., 2017) Imagine a very simple STFT that has bins evenly spaced 1Hz apart (we'll assume these frames are exactly 1 second long, for simplicity). If one frame possesses a sinusoid at 1.8Hz, while the subsequent one changes in pitch to 2.2Hz, both of these sinusoids will be "detected" by the 2Hz bin. The issue arises from the fact that the 2Hz bin "assumes" a perfect 2Hz sinusoid, which will fail to accurately represent the phase difference between the two frames if the distance between them is changed, such as in a time-stretch/pitch-shift operation.

This is where the phase vocoder comes in. Using the phase information from the current bin and the previous bin, it calculates the "true" frequency that is being detected by each bin, which eliminates the phase coherence problem (Srinivas et al., 2017).

Phase Vocoder - Mathematical Preface

Before beginning, in the spirit of being unnecessarily explicit in the technical explanations in this paper, allow me to clarify several things that were difficult for me to grasp as someone who didn't have much exposure to academic-level DSP symbology.

In the following equations, there are two symbols that you may find intimidating: the Greek letters ϕ and ω (phi and omega). ϕ is used to denote a phase angle, i.e. the current "position" of a waveform, where as ω is the angular frequency, which is essentially just the cyclic frequency multiplied py 2π radians. For instance, if we look at the following sine wave:

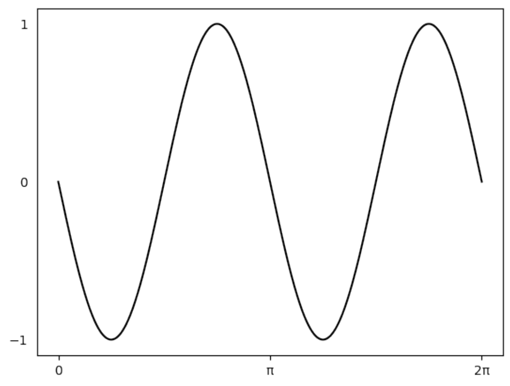

Figure 2: Sine wave with angular frequency $\omega = 4\pi$ and phase angle $\phi = \frac{\pi}{2}$

First off, we notice that this sine wave cycles twice over the course of its period, which gives it an angular frequency of $\omega = 2 * 2\pi = 4\pi$. We also note that the waveform initially moves downward, implying that it has been shifted over (as a normal sine wave initially proceeds upwards). As such, we say its phase angle, ϕ, is equal to the amount it has been shifted to the left. In this case, the sinusoid begins at the point that would've corresponded to $\frac{\pi}{2}$ radians on the un-shifted waveform, giving this waveform a phase angle $\phi = \frac{\pi}{2}$.

The other main confusion I found in the notation surrounded the difference between the mathematics provided and the actual implementation in the code. A musician's instinct would be to qualify these separate waveforms as frequencies relative to their sampling rate, e.g. some sinusoid that cycles once over 1024 samples at a sample rate of 44.1kHz should be considered mathematically as an 46.06Hz wave. However, mathematically, the Fourier transform is agnostic of our real-world audio frequencies, and instead only cares about the angular frequency *relative to sample rate*; i.e., the FFT is only concerned with how many times a waveform cycles over the number of samples it's being performed. For example, the sinusoid present in the first bin of our 1024-sample FFT from above will not be represented in the math as 46.06Hz, but instead simply as a wave with $\phi = 2\pi$.

One rather simple but vital piece of information is that frequencies are spread evenly across FFT bins, but for any FFT, half of the bins represent useless values in the context of audio bins, as the frequencies are not spread from 0 to the max frequency, but instead 0 to the *sample rate*. Technically, all the bins above the bin N/2 represent negative frequencies as the Fourier transform is bidirectional, but in the context of real-valued audio samples these can safely be ignored.[4]

The frequency for any FFT bin k is $\frac{k \cdot \text{sample rate}}{N}$. Because the FFT is agnostic of what the real-world frequencies are, we can choose arbitrary values for sample rate as long as we keep our math consistent. In order to keep the amount of mathematical operations to a minimum, it is advantageous to calculate time in terms of frames, as opposed to seconds. By this definition, the lowest FFT bin would have a frequency of 1 cycle per frame, or an angular frequency of $\phi = \frac{2\pi\text{rads}}{\text{frame}}$.

This may be a little daunting, but I believe the information provided above will be enough for anyone with the patience to puzzle it out for themselves—I simply wished to illustrate concepts that I felt were not made clear in the resources I used, to the point where my math was faltering simply because I didn't understand the difference between, for instance, audio frequency and angular frequency.

With that out of the way, we can now get into the math of the phase vocoder itself.

[4]This is because for real-valued Fourier transforms, the bins are symmetric around the N/2 bin. This is one of the mathematical symmetries that is used to implement fast Fourier transform algorithms.

Definition

The main goal of the phase vocoder is to calculate the *true* frequency that is being detected by each bin, not just the "ideal" frequency we obtain as $\omega_{ideal}[k] = 2\pi k \frac{\text{rads}}{\text{frame}}$.[5] To start this process, we must first find the phase shifts for each bin between the current frame and the previous frame. Please refer to the excellent web article (Grondin, n.d.) in my sources for some great visualizations of this process, although note that the math here is done a little differently. The phase shift, $\Delta\phi$, is simply the difference in ϕ between two bins of two adjacent frames, so we can determine it as such:

$$\Delta\phi_m[k] = \phi_m[k] - \phi_{m-1}[k]$$

where $\phi_m[k]$ indicates the phase of bin k for frame m.

We can now calculate the deviation of the real, measured frequency from the ideal frequency by first calculating an "ideal phase shift" as $\Delta\phi_m^{ideal}[k] = \frac{\omega^{ideal}[k]}{F_{overlap}}$, which is the amount of phase offset one would expect if it had the ideal angular frequency.[6] Using this, we can find the *deviation* of the real frequency from the ideal frequency, $\Delta\omega$, as the difference between the calculated and the ideal phase shifts:

$$\Delta\omega_m[k] = \Delta\phi_m[k] - \Delta\phi_m^{ideal}[k]$$

Using this value, we can obtain our estimate of the true frequency as:

$$\omega_m^{true}[k] = \Delta\omega_m[k] + \omega^{ideal}[k]$$

Because the FFT only deals in wrapped phases, i.e. phases that are constrained to the range $[-\pi, \pi]$, we must ensure that this true frequency is also appropriately constrained to this range. We can do this by defining a simple wrapping function(Lim, 2007):

$$\text{wrap}(\phi) = \phi - \text{round}(\frac{\phi}{2\pi}) \cdot 2\pi;$$
$$\omega_m^{wrapped}[k] = \text{wrap}(\omega_m^{true}[k])$$

And there we have it! We have an accurate, wrapped estimation of the true angular frequency[7], which we can then employ to correct the phase coherence issues by calculating a new, synthetic phase ϕ_s for each bin like so:

[5]These ideal frequencies are identical for any all frames, hence the omission of the m index.

[6]This is because of the fact that the distance between two frames is simply $\frac{\text{frame size}}{F_{overlap}}$, and that our time units are measured relative to frames as opposed to seconds.

[7]One vital caveat here is that our $\omega^m_{wrapped}[k]$ is not in terms of $\frac{\text{rads}}{\text{frame}}$, but rather $\frac{\text{rads}}{\text{overlap}}$. We could get the estimation of the true frequency in $\frac{\text{rads}}{\text{frame}}$ simply by multiplying $\omega_{wrapped}$ by

$$\phi^s_m[k] = \text{wrap}(\phi^s_{m-1}[k] + \omega^{wrapped}_{m-1}[k] \cdot S)$$

Note how we multiply our wrapped frequency estimate by the stretching factor—this is why the phase vocoder is so vital. Without it, we would have absolutely no way of making a good estimate where the phase of this bin will be when it is stretched, because all we get are the "ideal" frequencies obtained from the FFT. Furthermore, this is why we use synthetic phase and true frequency from the previous frame instead of the current one, as we need to account for how much we've stretched previous frames in our current calculation. Because this is a recursive definition, we need to define an edge case for the first bin that we use: simply, we use the unadjusted analytical phase as the synthetic phase and the wrapped true frequency for the first frame. We also need to wrap this value once more, as ϕ^s will be fed back into the STFT for an inverse transform

After this, we can convert the phase and amplitudes back to their complex representations, perform the inverse FFT, and then overlap-add the frames using the synthesis hop size hop_s:

$$x_{output}[n] = \sum_{m=0}^{M-1} = O_{m,(n-\text{hop}_s)} \cdot w[n] \cdot (u[n - m * \text{hop}_s] - u[n - m * \text{hop}_a - N])$$

where $O_{m,n}$ is represent the sample at index n of the synthetic frame m produced by the phase vocoder. This produces our time-stretched audio signal, fully phase-corrected! As stated before, this can be used to shift the pitch by linearly interpolating the stretched signal back to the size of the input signal, which trades shift in time for a shift in pitch.

The maths above are a little different from any of the sources I found, but they much more closely resemble how the calculations are done in the actual algorithm. As such, I felt it useful to explain it this way as it can be a little confusing as to how doing the math with time in terms of seconds or samples gives the same result as having your time be in terms of FFT frames. With these definitions in place, and hopefully with the reader's understanding of Fourier transforms and phase vocoders improved, we can take a look at some of the implementation details of the RTSTFT library.

Implementation

RTSTFT is written in pure C, with the intention of being adaptable to almost any purpose. In its early stages, it was tested as a purely offline processing library, achieving impressive speeds thanks to the inclusion of the PFFFT (Pretty Fast

$F_{overlap}$, but that would be an unnecessary computation as we need the angular frequency in terms of the overlap length for the final calculation anyways.

FFT) library for the requisite Fourier transforms.(Pommier, n.d.) Because of the inclusion of this PFFFT, RTSTFT was not able to meet ANSI C standards, however this is likely not an issue as this standard is only required by exceptionally old or specific hardware. It is not only a phase vocoder implementation, but also provides end users with several novel ways of manipulating the STFT and phase vocoder algorithms, hence the creation of a companion plugin for RTSTFT was created to allow musicians and producers to harness the high level of detailed control that the library allows: rtstft_ctl. Somewhat contrary to its name, RTSTFT is not intended to be a particularly low-latency DSP processor; instead, it attempts to give real-time control over parameters that are generally only available in offline algorithms, if at all. The following implementation breakdown will focus solely on RTSTFT, as rtstft_ctl is mostly just wrapper code intended to allow clean GUI access to the library's functionalities. Unfortunately, there is not a functioning test suite written for RTSTFT, but rtstft_ctl could be profiled to get a very good idea of RTSTFT's real-world performance. I have not noted any significant performance dips unless using large FFT frame sizes, i.e. 8192 or higher.

Architecture

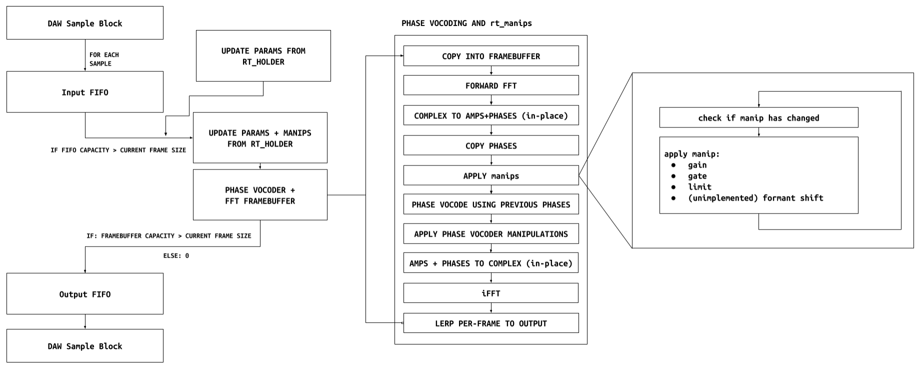
DAW Sample Block
FOR EACH SAMPLE
Input FIFO
UPDATE PARAMS FROM RT_HOLDER
IF FIFO CAPACITY > CURRENT FRAME SIZE
UPDATE PARAMS + MANIPS FROM RT_HOLDER
PHASE VOCODER + FFT FRAMEBUFFER
IF: FRAMEBUFFER CAPACITY > CURRENT FRAME SIZE
ELSE: 0
Output FIFO
DAW Sample Block
PHASE VOCODING AND rt_manips
COPY INTO FRAMEBUFFER
FORWARD FFT
COMPLEX TO AMPS+PHASES (in-place)
COPY PHASES
APPLY manips
PHASE VOCODE USING PREVIOUS PHASES
APPLY PHASE VOCODER MANIPULATIONS
AMPS + PHASES TO COMPLEX (in-place)
iFFT
LERP PER-FRAME TO OUTPUT
check if manip has changed
apply manip:
gain
gate
limit
(unimplemented) formant shift

Figure 3: RTSTFT implementation overview

The core data flow of RTSTFT is fairly standard as spectral processors go. The host program will pass an array of audio samples as floating-point numbers, which RTSTFT will ingest one-by-one[8] into a custom FIFO[9] data structure. Whenever the FIFO has enough samples to constitute an FFT frame, N samples are read from the FIFO and placed into the frame buffer array. To facilitate overlapped reading, only hop_a samples are then purged from the FIFO, such that the next frame is read from the appropriate position in the audio stream.

[8]Initially, RTSTFT was designed to read entire blocks of samples at a time, but was later changed to work sample-by-sample to reduce overall complexity and increase tolerance to unpredictable measures taken by the host.

[9]First-In, First-Out

Once inside the frame buffer, the audio data is first transformed using the PFFFT real-only forward transform, and subsequently has its complex components converted into amplitudes and phases. RTSTFT then undergoes its first batch of manipulations, which meddle with the amplitudes of the bins; these will be described in detail in the next section. Following the amplitude manipulations, phase vocoding proceeds as described above, with the exception of some phase-based manipulations. Finally, the frames are inverse transformed and then overlap-added into RTSTFT's second FIFO that accumulates processed samples.

The one major innovation of RTSTFT is the way it approaches the final overlap-adding and interpolation step. I don't claim to have invented the following procedure, but I did develop on my own and didn't find any mention of a similar tactic in any sources I came across.

Normally, a phase vocoder-based pitch shifter will time-stretch all available audio, and then interpolate the full stretched signal back down to its original size. In real-time audio, this is not practical, as it would involve a *third* FIFO to hold the stretched audio, and only interpolate after all frames that overlap with the current frame have also been processed. This would result in a twofold increase in latency, which is not ideal for a library intended to be used in music production. Instead, RTSTFT interpolates *each frame* on its way into the output FIFO, which is mathematically sound as the initial phase angle of each bin is conserved through the process of interpolation, and any rounding errors with respect to the window amplitudes of the samples are minimal. At this point, the audio is fully processed and simply awaits reading at the output FIFO, which pops off individual samples at the same rate that the input FIFO ingests them.

RTSTFT also presents an outward-facing API to allow for host programs to change its settings in real-time. To account for the host dispatching these changes on a different thread than the one responsible for audio processing, RTSTFT has systems in place to ensure that its settings aren't meddled with while it's in the process of manipulating a frame, as this could result in disastrous effects. Most settings are stored in a volatile buffer intended to take multiple writes from many threads; this buffer only transfers its data to the "official" settings buffer once at the beginning of every FFT frame's processing. For more complicated changes, such as a change in the FFT size or overlap factor, RTSTFT has a crude thread lock implementation, which prevents any audio I/O from occurring while it interpolates its per-bin settings to the new FFT settings.[10]. However, the effects of this lock are often barely noticeable, as the code is generally quick enough to finish these settings changes before the next chunk of audio needs to be processed.

Next, we'll explore the heart and soul of RTSTFT: its manipulation engine.

For more information, take a look at RTSTFT's source on github..

[10]In rtstft_ctl the plugin also notifies the host to bypass its processing functionality while this lock is active

Manipulations

RTSTFT presents two types of manipulations to the user: global phase-based manipulations, and per-bin amplitude manipulations. These could easily be expanded on in the future, but I'll just describe what currently exists in the implementation.

The pitch setting is the same stretching factor S as seen in the phase vocoder definitions section of this paper, determining the amount by which the frames are moved apart in their overlaps and to what extent they are stretched during interpolation. Retention and Phase Mod affect how much the phase from the previous frame affects the current frame. Phase Chaos introduces random variation to the final phase, which can be leveraged to produce a chorus-like effect. These parameters are applied to the calculation of the synthetic phase like so:

$$\phi^s_m[k] = \text{wrap}(\phi^s_{m-1}[k] \cdot \text{retention} + \omega^{wrapped}_{m-1}[k] \cdot \text{pitch} \cdot \text{phase mod} + \text{rand}() \cdot \pi)$$

where rand() produces some random value from the range $[-1, 1]$.

These phase manipulations generally produce extremely garbled or highly resonant sonic results, which can often be grating but can occasionally produce some extremely interesting experimental sounds. However, the pièce de résistance of RTSTFT is undoubtedly its per-bin manipulations. Consisting of Gain, Gate, and Limit, these manipulations allow you to define a gain multiplier for each bin, a gate amplitude which defines the minimum amplitude a bin must have or else be set to 0, and a limit amplitude that will clip the maximum amplitude of the bin. These settings can be set with high precision using the rt_cmd language built into RTSTFT (which we'll look at in a second), but can also be used extremely intuitively with rtstft_ctl's graphical spectral display. When the GUI doesn't give the sufficient level of precision however, generally that's the end of the story for most audio plugins, but for RTSTFT, rt_cmd provides that extra dimension of control.

rt_cmd

rt_cmd is the native command line language built directly into the RTSTFT library. Unfortunately, it is not nearly as fleshed out as I would have like it to have been, but as I developed the rtstft_ctl plugin, I began to realize how most operations are much simpler to undertake from the GUI as opposed to the command line, especially in the context of music production. That said, I did leave in a command line so that anyone who felt the need to get extremely surgical with rtstft_ctl may do so by leveraging the rt_cmd language.

Syntax

rt_cmd follows an exceptionally simplistic syntactic structure, which can be summed up as:

```
COMMAND [FLAGS[FLAG ARGUMENTS...]...] COMMAND ARGUMENTS...
```

To get familiar with it, we should take a look at a few examples.

To limit the amplitude of bins 25 through 60 (inclusive) to ± 0.5, where amplitude is measured in the normal DSP convention of $[-1, 1]$:

```
limit 0.5 25-60
```

To apply (roughly) the same limiting using decibels (dBFS):

```
limit -b -6 25-60
```

Apply gain to bins 200-400 utilizing an exponential curve, with the curve ranging from -12dB to -6dB:

```
gain -b -e -2 -12 -6
```

Here we can see the two main flags currently employed in rt_cmd: `-b` and `-e`. The `-b` flag simply indicates that the parser should expect values in decibels, which is often a much more intuitive unit to use for music producers. `-e` produces an exponential curve, taking in three values: the curve strength (which should be in the range $[-10, 10]$), the value at the beginning of the curve, and the value at the end of the curve. The curve equation is defined as $f(x) = x^{2^a}$, where a is the curve power. This curve is then use to interpolate between the start and end values of the curve. See the source for details on exactly how this is accomplished.

Parsing

Though it behaves like a command-line tool, rt_cmd's parser is actually much more akin to that of a full-blown language compiler. Though this is overkill for its current state, it is intended to be infinitely and easily extensible, allowing for the language to grow naturally with the rest of the library as additional functionality is added. For a full picture of how rt_cmd parses its commands, I'd recommend taking a look at the source, but the general flow of execution goes like this:

1. A string of characters is passed into rt_cmd's parser object, which separates the string into raw tokens solely by separating them by whitespace.
2. The type, or "flavour" of these tokens is determined by the lexer—e.g. integer, float, etc.
3. Given the command designated by the first token, a corresponding function is executed which takes the organized and lexed tokens and passes them into the appropriate RTSTFT API function.

Thought not the most computationally efficient system, modern computers can handle string operations like these with ease, which is only compounded by the fact that the command parser will almost always be running on a different thread.

Future Optimizations

Utilizing the PFFFT Library, RTSTFT runs extremely fast due to the library's usage of both ARM Neon as well as x86 SSE/AVX SIMD intrinsics. For those unfamiliar, SIMD (Single Instruction Multiple Data) instructions are implemented in some processors to be able to perform a single operation on multiple pieces of data, for instance multiplying 8 floating point numbers together in pairs. RTSTFT's algorithm (outside of PFFFT) has not been implemented to take advantage of such instruction sets.

Most of the amplitude manipulations involve a conditional check for every single bin (e.g., the gate manipulation checks if the bin amplitude is less than the corresponding gate value before setting the bin amplitude to 0). This could be implemented in SIMD with commands such as ARM's Neon `VCLE` instruction, or the SSE `mm_cmple_ps` instruction, followed by SIMD multiply; the limit manipulation could be even simpler with Neon's `VMIN` or SSE's `mm_min_ps`.

It may be possible to vectorize the linear interpolation step of the RTSTFT algorithm, but it would likely be prohibitively difficult to implement compared to the small boost in performance it would gain.

One of RTSTFT's biggest issues at the moment is its horrific worst-case processing performance: while ingesting samples, RTSTFT runs extremely quickly, but the moment there are enough samples to create a full FFT frame, the *entire* phase vocoder algorithm is executed in one fell swoop. For larger FFT sizes, this could very easily halt the audio thread as the processor struggles to finish the computations in time. There are several possible remedies for this:

1. Execute the phase vocoder algorithm on a worker thread whenever an FFT frame is complete.
2. Subdivide the phase vocoder algorithm into separate chunks of work to be executed one-by-one during different cycles.

Though the threaded option it is the best solution on paper, it would increase the necessary complexity of RTSTFT exponentially, as managing the synchronization of the worker with the audio thread would involve an extremely fault-tolerant system of checks and fallbacks. The second option, though still requiring an increase in complexity, would save on implementation time as it doesn't introduce a threadsafety concern. For instance, once an FFT frame is ready to be read, it could be digested and forward-transformed during once cycle, manipulated during the second cycle, phase vocoded during the third, and then inverse transformed and marked as ready for reading during the fourth. These operations would all still occur sequentially, keeping the DSP of RTSTFT self-contained. At current, this worst-case performance problem only seems to be an issue with FFT sizes of 8192 or higher, most of which can be ameliorated by the user by utilizing offline rendering to ensure a clean signal.

In all likelihood, all of these optimizations would only be pursued if RTSTFT were ever rewritten with the intent of creating a commercial product, but I will leave them here for any in the open source community who would like to create a higher-performance fork of RTSTFT.

Conclusion

RTSTFT is the culmination of years of experience in music and sound design, born of a desire to pull back the curtain on the fascinating world of DSP that is often hidden from musicians and producers. Though simple relative to many other existing plug-ins, it is hopefully the first in a line of many audio processors that try to give users full control over their audio, and embolden sonic creativity in ways that were previously unthinkable. I implore any readers of this paper who have their own ideas for novel applications of DSP algorithms to contact me directly, as it is my goal to inspire a new wave of software that truly takes advantage of the wealth of possibilities digital audio has to offer.

Appendix

Contact

Please feel free to reach me through email via jacksonkaplan@alum.calarts.edu, or find my social media linked through my website, ancientjpeg.github.io.

Code

The core RTSTFT library can be found at its github repo located at https://github.com/ancientjpeg/RTSTFT, and the companion plugin rtstft_ctl can also be found on GitHub located at https://github.com/ancientjpeg/rtstft_ctl with instructions to build from source.

Glossary of terms

$$\text{Frame size} = N \qquad \text{Frame count} = M$$

$$\text{Sample index} = n \qquad \text{Frame index} = m \qquad \text{Bin index} = k$$

$$\text{overlap factor} = F_{overlap} \qquad \text{stretching factor} = S$$

$$\text{input sequence} = x_{input} \qquad \text{output sequence} = y[n]$$

$$\text{array of analysis frames extracted via STFT} = A$$

$$\text{array of synthetic frames assembled after phase vocoding} = O$$

$$\text{analysis hop} = \text{hop}_a = \frac{N}{F_{overlap}}$$

$$\text{synthetic hop} = \text{hop}_s = \text{round}(S * \text{hop}_a)$$

References

3Blue1Brown. (2018). *But what is the Fourier Transform? A visual introduction.* https://www.youtube.com/watch?v=spUNpyF58BY

Discrete Fourier transform. (2022). In *Wikipedia.* https://en.wikipedia.org/w/index.php?title=Discrete_Fourier_transform&oldid=1081065184 Page Version ID: 1081065184

Fourier transform. (2022). In *Wikipedia.* https://en.wikipedia.org/w/index.php?title=Fourier_transform&oldid=1085783068 Page Version ID: 1085783068

Götzen, A., Arfib, D., & Bernardini, N. (2000). *Traditional(?) Implementation of a phase vocoder: The tricks of the trade.* 37–44.

Grondin, F. (n.d.). *Guitar Pitch Shifter - Algorithm.* Retrieved May 10, 2022, from http://www.guitarpitchshifter.com/algorithm.html

H910 – The Halls of Valhalla. (2010). https://valhalladsp.wordpress.com/tag/h910/

Jessop, S. (2017). The Historical Connection of Fourier Analysis to Music. *The Mathematics Enthusiast*, *14*(1-3), 77–100. https://doi.org/10.54870/1551-3440.1389

Lim, K. A. (2007). *An Open-Source Phase Vocoder with Some Novel Visualizations.* 39.

Pitch Shifters, pre-digital – The Halls of Valhalla. (2010). https://valhalladsp.wordpress.com/2010/05/04/pitch-shifters-pre-digital/

Pommier, J. (n.d.). *Jpommier / pffft / pffft.c — Bitbucket.* Retrieved May 11, 2022, from https://bitbucket.org/jpommier/pffft/src/master/pffft.c

Srinivas, N., Amara, M., & Kumar, P. K. (2017). *Implementation of Pitch Shifter using Phase Vocoder Algorithm on Artix-7 FPGA.* 8.

The Mathematical Genius of Auto-Tune. (2016). In *Priceonomics.* https://priceonomics.com/the-inventor-of-auto-tune/