Design and Run Real-time Spectral Processing on the Web with Faust

Shihong Ren, Stéphane Letz, Michel Buffa, Laurent Pottier, Yang Yu

Design and Run Real-time Spectral Processing on the Web with Faust

Shihong Ren
Shanghai Conservatory of Music, SKLMA, China; Univ Jean Monnet, ECLLA Lab, France
shihong.ren@univ-st-etienne.fr

Stéphane Letz
Yann Orlarey
Univ Lyon, GRAME-CNCM, INSA Lyon, INRIA, CITI, EA3720, 69621 Villeurbanne, France
{letz, orlarey}@grame.fr

Michel Buffa
Univ Côte d'Azur, I3S Lab, INRIA, France
michel.buffa@univ-cotedazur.fr

Laurent Pottier
Univ Jean Monnet, ECLLA Lab, France
laurent.pottier@univ-st-etienne.fr

Yang Yu
Shanghai Conservatory of Music, SKLMA, China
yuyang@shcmusic.edu.cn

ABSTRACT

Web-based spectral processing with the Web Audio API is a challenging task that requires efficient and flexible tools. It involves Fourier transform utilities and frequency-domain data manipulations. In this paper, we present a novel framework for designing and running real-time spectral processors on the web, using FAUST as the programming language and its web-based toolchain. Our framework is inspired by Max's `pfft~` paradigm, which allows users to create custom spectral processors from streamed spectral data. Using FAUST language and suitable FFT tools, the designed algorithm can be compiled to WebAssembly modules that are executed in an Audio Worklet. We demonstrate the potential of our framework by showing some examples of spectral effects and synthesizers that can be easily designed and executed on the web.

1 Introduction

The maturity of the Web Audio standard and its implementation has enabled more complex online audio applications to be developed. Thanks to the DSP customization feature provided by Audio Worklet and WebAssembly, we can design and run more sophisticated DSPs for the web, such as spectral processors based on Short-Time Fourier Transform (STFT).

The native AnalyserNode from the Web Audio API includes a Fast Fourier Transform (FFT) implementation for retrieving real-time spectral data on demand. It returns magnitude per frequency bin via the `getByteFrequencyData` or `getFloatFrequencyData` method, providing a straightforward way for real-time spectral analysis, typically used for web-based spectrum visualization. Nevertheless, it has significant limitations that make it impractical for applications requiring a more comprehensive and accurate spectral processing and analysis tool. First, an FFT analysis generally produces a complex numerical value for each frequency interval, from which magnitude and phase information can be inferred, whereas the AnalyserNode does not return phase information, resulting in incomplete frequency domain data. Second, the window function using by the FFT analysis is not controllable, it is hard-coded onto a Blackman window. Third, for scenarios where users need to analyze successive windows such as with the STFT, there is no way to precisely control the timing of each analysis as the function call time on the main thread is not reliable. In addition, no window overlap mechanism is implemented in AnalyserNode. Fourth, the node does not provide any inverse FFT (IFFT) method. Without phase information, users are unable to reconstruct or modify the input signal with the given data and API.

Essentia.js [5, 4], Meyda [8], aubiojs[1] are web-based spectral processing libraries that extract audio features in real time. Due to the limitation of the AnalyserNode, they all ship with a different FFT implementation for audio analysis. It's convenient to use these libraries to retrieve musical information, but they don't provide any audio-to-audio spectral processors or synthesizers from spectral data. Currently, few frameworks can be found for musicians to design spectral processing algorithms. Traditional tools for audio developers such as Matlab, Python and JUCE are not usable for the web environment. Cycling '74's RNBO now supports exporting spectral processors (analysis and synthesis) to the web target thanks to its `fft~` and `ifft~` RNBO objects.[2] This feature is impressive and friendly to Max users, but Max is not free and the designing feature is only available on Max desktop software. Additional development is also required for running the exported DSP on the web.

Our work aims to create a fully web-based workflow for designing and running real-time spectral processors, includ-

Web Audio Conference WAC-2024, March 15–17, 2024, West Lafayette, IN, USA.

[1] https://github.com/qiuxiang/aubiojs
[2] https://rnbo.cycling74.com/learn/using-the-fft

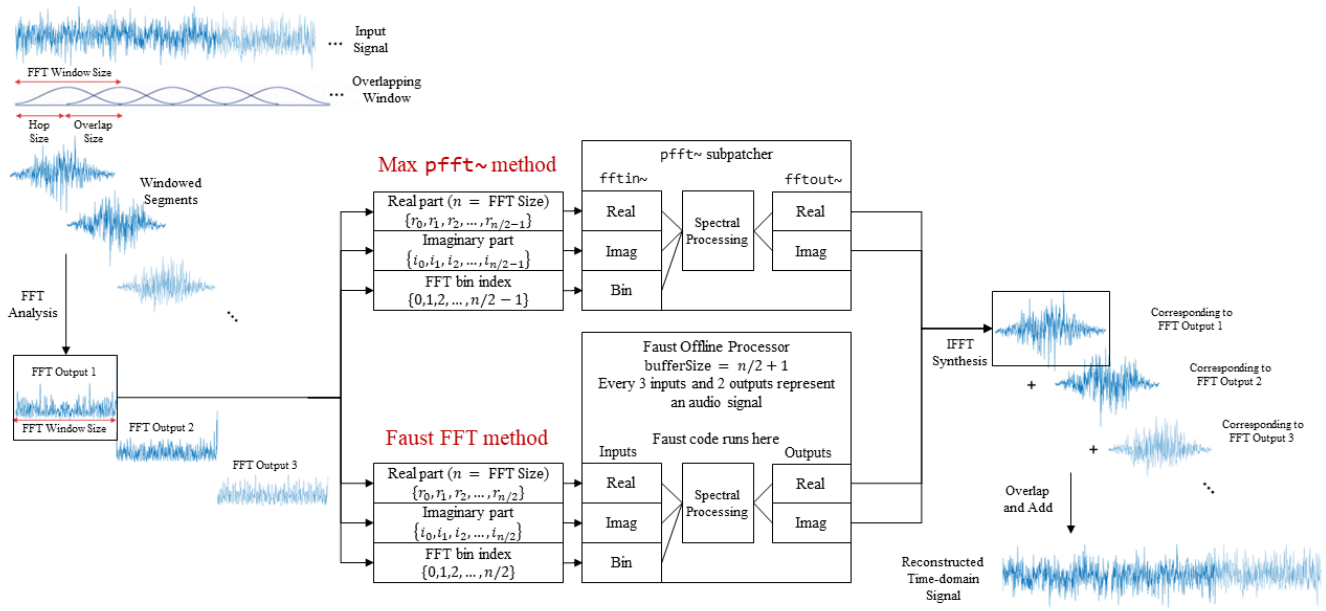

Figure 1: Real-time spectral processing paradigms: Max's `pfft~` compared to Faust FFT.

ing FFT-based analysis and synthesis. We use Max's `pfft~` as a reference paradigm and FAUST [7] as the designing language. FAUST is a functional programming language for DSP that supports exporting to a variety of platforms and standards. Since 2014, the FAUST compiler and its runtime were available on the web, allowing developers to compile FAUST DSPs to JavaScript-compatible binary code and dynamically run the DSP within the browser [2]. FAUST IDE was developed in 2019 using modern web technologies such as WebAssembly and AudioWorklet, offering various testing, debugging and audio visualization features, allowing connecting to different kinds of audio/MIDI inputs and outputs [10]. A new version of the FAUST WebAssembly compiler with JavaScript API: `faustwasm` was released in 2022 [9]. It provides JavaScript wrapper for FAUST DSPs that can be AudioWorklet or standalone classes for both web and Node.js environments. Our work is based on `faustwasm`, adding an adapter for a FFT/IFFT module that is used before and after the FAUST DSP, and streaming the spectral data bin by bin as the DSP's inputs and outputs. We will first introduce the paradigm in §2, then present our implementation in §3. A few examples and their usage in the FAUST IDE and JSPatcher [11] will be demonstrated in §4, followed by a discussion in §5.

2 Paradigm

A typical spectral processor needs to first convert windowed time-domain signal to frequency-domain data frames using FFT, then process the frames and reconvert them to time-domain signal using IFFT. Max is a graph-based visual programming language (VPL) that implements a system dedicated for this type of spectral processing with the `pfft~` object. It is widely used for musical project and has an interesting and viable design for a web-based environment to adopt.

As Figure 1 shows, the `pfft~` object allows users to create and load special subpatchers that manipulate frequency domain signal data. It performs STFT on the incoming time domain signal with specific FFT size, overlap factor and window functions. Hanning, Hamming, Blackman, triangle, and square window functions are available. For each FFT window, it sends the spectrum (frequency domain data) to the subpatcher's input and takes its output as the modified spectrum for an IFFT to reconstruct the time domain signal.

In order to give more precise data on the spectral change over time, the object supports overlapping the FFT frames. For example, with an FFT size of 1024 and 2 overlaps, the object will perform once an FFT of 1024 samples each 512 samples. The object performs by default a real FFT which is faster and returns half of the frequency spectrum. Since the FFT calculation parameters are defined by the user, this `pfft~` object will adapt all the audio object in its loaded subpatcher to a specific rate which is different from the audio sampling rate. The subpatcher loaded by the `pfft~` object needs to be specially designed for spectral processing. In this patcher, users can get the spectrum of the input signal using the `fftin~` object and send the processed spectrum using the `fftout~` object (see Figure 2).

`fftin~` object has three outlets that can be connected with other audio objects, they give spectral information of a given audio channel. The leftmost 2 outlets put out a stream of real and imaginary numbers for the bin response for each sample of the FFT analysis, the third outlet puts out the current FFT bin index. These numbers are ordered

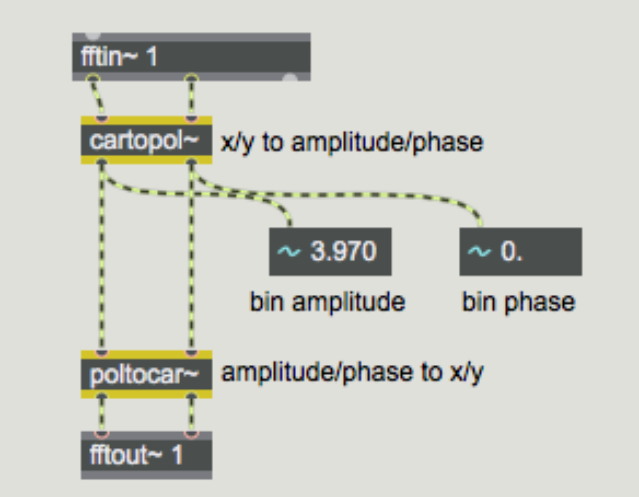

Figure 2: An example patcher of Max's `pfft~`.[3]

by their FFT bin for each frame. With this mechanism, the `fftin~` and `fftout~` the users can easily identify which bin the complex number (real/imaginary pair) belongs to. Additional information about current running FFT analysis such as the overlap factor and the FFT size can be acquired in the subpatcher using the fftinfo object.

Max provides several spectral processing examples with `pfft~`. Implementations of `pfft~` subpatchers for getting and manipulating amplitude and phase data of each bin, creating FFT-based filter, synthesis, delay, reverb and simple time-stretching are shown in the documentation. These examples are implemented with regular Max Signal Processing (MSP) objects which gives us hints about designing web-based spectral processing system: it is possible to use time-domain-oriented DSPs in a frequency domain context for spectral processing if the DSP algorithm is carefully and correctly designed.

3 Implementation

Our aim is to implement a system that allows doing FFT/IFFT and insert a DSP component between the two transforms inside an AudioWorklet. In this work, we are facing two major challenges: one is to minimize the latency between the input and the output with a correct buffering mechanism, another is to maximize performance by avoiding buffer copying.

The buffering issue need to be addressed at the AudioWorklet processor's level, where the process method is called each audio rendering block, with 128-sample-long input and output buffers provided. Since the FFT size can be bigger than the audio buffer size of 128, we need a larger buffer for storing the input audio signal before these samples can be FFTed. Similarly, another larger buffer is needed for storing the IFFTed audio and for adding overlapped windows before the audio output. We implemented these two buffers as *ring buffers* [1] to avoid data moving, read/write pointers are stored for each buffer.

Based on a WebAssembly version of both FFTW and KissFFT libraries[4] developed by j-funk, we rewrote a JavaScript API that can be more easily used in an AudioWorklet environment. These two libraries are likely the fastest web-based FFT/IFFT implementations according to his benchmark.[5] In our version,[6] the FFT forward/reverse (IFFT) methods support passing a callback as the parameter instead of the input buffer. The callback's parameter is the internal input buffer of the FFT module that allows users to fill in with the callback function body (e.g., fill with the data from the input ring buffer in our case). This approach can avoid one extra buffer copy.

The provided AudioWorklet also supports other FFT libraries as long as the given FFT library conforms the interface definition which includes the constructor, the forward/reverse methods that accept a callback or a `Float32Array` as parameter, and the dispose method for memory clean up. We also designed hooks that are used for interpreting the FFTed array. For example, for a real FFT of size n, FFTW stores the result — real and imaginary parts — as:

$$\{r_0, r_1, r_2, \ldots, r_{\frac{n}{2}}, i_{\frac{n+1}{2}-1}, \ldots, i_2, i_1\}$$

The FFTed array has the same length as its input if n is power of 2. The array contains all the necessary information about its input since $i_0 = i_{\frac{n}{2}} = 0$.[7] However, KissFFT outputs differently, which is:

$$\{r_0, i_0, r_1, i_1, r_2, i_2, \ldots, r_{\frac{n}{2}}, i_{\frac{n}{2}}\}$$

The array has a length of $n + 2$. So, the hook allows users to customize the interpretation according to different FFT implementation, and transform the FFTed array as following three arrays for spectral processing, then transform back after the processing for the IFFT:

$$\{r_0, r_1, r_2, \ldots, r_{\frac{n}{2}}\}, \{i_0, i_1, i_2, \ldots, i_{\frac{n}{2}}\}, \{0, 1, 2, \ldots, \frac{n}{2}\}$$

We also noticed that Max's `pfft~` drops a pair of the last complex number $\{r_{\frac{n}{2}}, i_{\frac{n}{2}}\}$ which represents spectral information at the Nyquist frequency. This is probably for keeping the buffer length power of 2 but can produce inaccurate IFFT result.

After the FFT analysis of each buffer, we use a FAUST DSP as the spectral processor to modify or generate the spectrum for the IFFT synthesis. The three arrays that represent the current spectrum will be passed to the FAUST DSP as signal of three input channels.

The FAUST DSP is the core of this process. It is a `FaustOfflineProcessor` generated by the `faustwasm` module with a buffer size of $\frac{n}{2} + 1$. By its initialization, we will detect following special DSP parameters and set them to a constant value, in order to pass the information about the current FFT setup:

- `fftSize`: current FFT size.
- `fftHopSize`: current FFT hop size per frame, correlated with the FFT overlap factor.

[3]Image from Max documentation: https://docs.cycling74.com/max8/tutorials/14_analysischapter04.

[4]https://github.com/j-funk/fftw-js and https://github.com/j-funk/kissfft-js

[5]https://github.com/j-funk/js-dsp-test/

[6]https://github.com/Fr0stbyteR/fftw-js and https://github.com/Fr0stbyteR/kissfft-js

[7]https://www.fftw.org/fftw3_doc/The-Halfcomplex_002dformat-DFT.html

A FAUST DSP for spectral processing will have a group of real, imaginary, index triplet as 3 frequency-domain inputs for each time-domain input channel to receive spectral data. For each time-domain output channel, we assume that the FAUST DSP will output 2 frequency-domain channels containing real and imaginary values of the current bin. For example, a 2-in 2-out bypass FAUST spectral processor can be written as

- verbose version:

```
process(real_1, imag_1, bin_1, real_2, imag_2, bin_2) =
    real_1, imag_1, real_2, imag_2;
```

- simplified version where we omitted the last (`bin_2`) input as it is not used in the processor:

```
process = _, _, !, _, _;
```

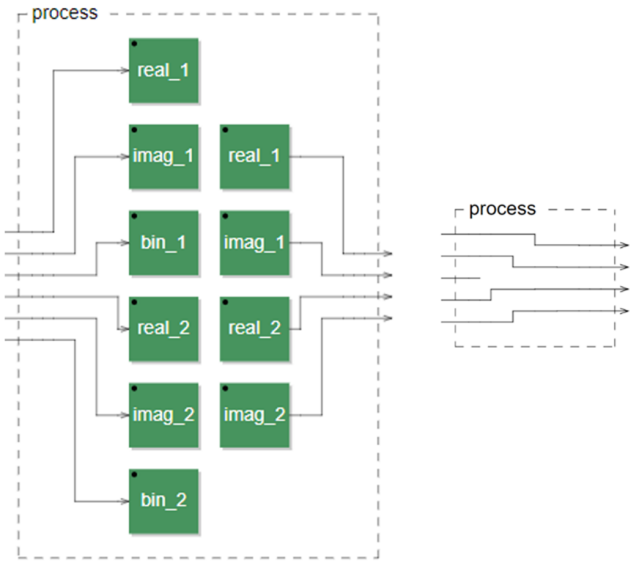

Figure 3: 2-in 2-out bypass Faust spectral processor (left: verbose version, right: simplified version).

Figure 3 shows the architecture of the FAUST spectral processor inside an AudioWorklet processor.

We use AudioWorklet parameters to control the FFT type: `fftSize`, `fftOverlap`, `windowFunction`, and `noIFFT`. The `fftSize` and the `fftOverlap` parameters will instantiate the FFT analyzer or configure it upon any parameter value change. The `windowFunction` parameter receives a number as the index of a pre-defined window functions list. The `noIFFT` parameter will toggle a special synthesis mode after the spectral processing. This mode is also present in Max by putting the keyword `noIFFT` as a parameter of the `fftout~` object, which bypasses IFFT and outputs raw spectral data. This mode can be used for audio analysis scenarios.

As a new usage of the FAUST language, we added an option in a new version of the FAUST IDE[8] to enable this FFT processing mode and to specify FFT parameters. Users can test, visualize and debug the FFT processor written in FAUST in real time with the environment (see Figure 1).

In 2021, we developed a web-based graph editor for designing FAUST DSP for the JSPatcher web application within an object called `pfaust`. It allows users to create patchers by connecting FAUST functions, compile and run the DSP interpreted from the patcher directly in the application. In this work, we added two new objects `faust-fft` and `pfaust-fft` in JSPatcher for users to design spectral processors using FAUST code or patcher and run it in real time with other data and objects. This object shares the same interface with pfaust object which contains a subpatcher that is interpreted as a FAUST DSP. In `pfaust-fft`'s case, the DSP is a spectral processor instead. Some examples are demonstrated in the next section.

[8]Experimental branch available on https://faustide.shren.site/

4 Examples

For the proof of the concept, we built in JSPatcher some simple spectral processing examples using the new `pfaust-fft` mechanism.

4.1 Spectral Gain

To start with, a simple gain controller in a spectral processor will multiply both real and imaginary part by the gain factor (Figure 4):

```
gain = hslider("gain", 1, 0, 1, 0.01);
process = *(gain), *(gain);
```

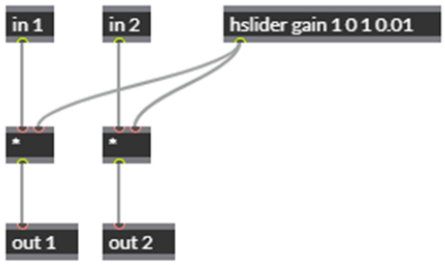

Figure 4: Simple spectral gain in Faust.

4.2 Spectral Filter

Using the third channel, we will be able to identify the current bin index and its center frequency and create FFT filters by changing the gain factor according to the bin index. For example, a basic high-shelf filter can be written as (Figure 5):

```
gain = hslider("gain", 1, 0, 1, 0.01);
cut_bin = hslider("cut_bin", 1024, 0, 1024, 1);
process(real, imag, bin) = real * gain_bin, imag *
     gain_bin with {
    // Check if the bin is lower than the cut_bin
    low = bin < cut_bin;
    // If lower, use the parameter value, else 1
    gain_bin = (low == 0) * gain + low;
};
```

We can declare a special DSP parameter called `fftSize` that can be read to get the actual FFT type information and make the cutoff parameter Hertz-based (Figure 6):

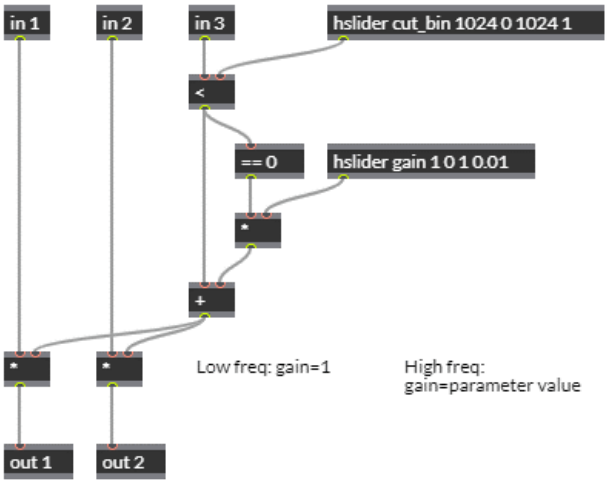

Figure 5: Simple FFT filter in Faust.

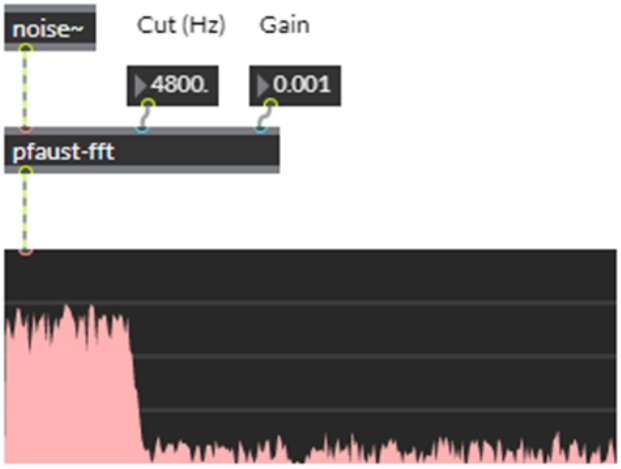

Figure 6: Spectral high-shelf filter running in JSPatcher.

```
import("stdfaust.lib");
gain = hslider("gain", 1, 0, 1, 0.01);
// cut_bin = hslider("cut_bin", 1024, 0, 1024, 1);
// High-shelf cutoff frequency in Hz
cut = hslider("cut", 440, 0, 24000, 0.1);
// global variable set by the processor itself
fftSize = hslider("fftSize", 1024, 2, 16384, 1);
// FFT bin index of the cutoff frequency
cut_bin = cut / (ma.SR / fftSize);
// ...
```

4.3 Spectral Convolution

For more sophisticated spectral processor, we will need to use both amplitude and phase information about the current FFT bin. The conversion between the complex numbers and the amplitude/phase information can be done with a cartesian-polar coordinates converter (Figure 7):

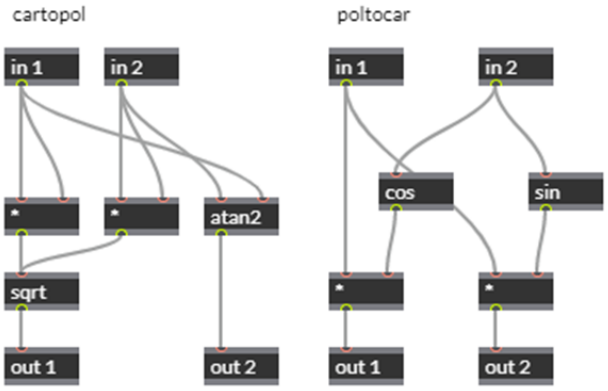

Figure 7: Cartesian-polar coordinates converter in Faust.

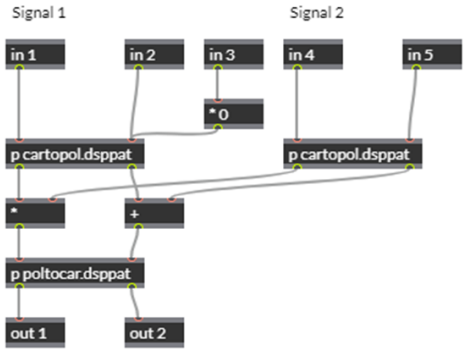

Figure 8: Spectral convolution in Faust patcher.

```
// cartesian to polar
cartopol(x, y) = x * x + y * y : sqrt, atan2(y, x);
// polar to cartesian
poltocar(r, theta) = r * cos(theta), r * sin(theta);
```

With the converter, cross synthesis and convolution effects can be easily implemented. Figure 8 shows how a spectral convolution looks like in a spectral FAUST patcher, where cartesian-polar coordinates converters are implemented in subpatchers.

4.4 Spectral Centroid Analyzer

With `noIFFT` option enabled, the DSP can output analysis result as audio signal. The design of the DSP is a little bit trickier as the result needs to be locked with each FFT frame. In most of the cases, a sample-and-hold function can be used to maintain the output value while calculating the next result. The code below shows an example of a spectral centroid [6] analyzer in FAUST code, Figure 9 shows its runtime in JSPatcher.

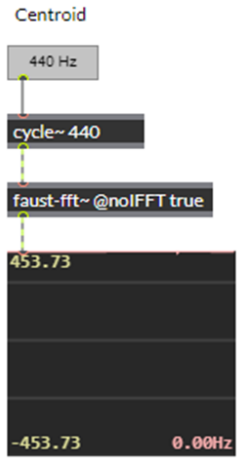

Figure 9: Spectral centroid analyzer written in Faust and running in JSPatcher.

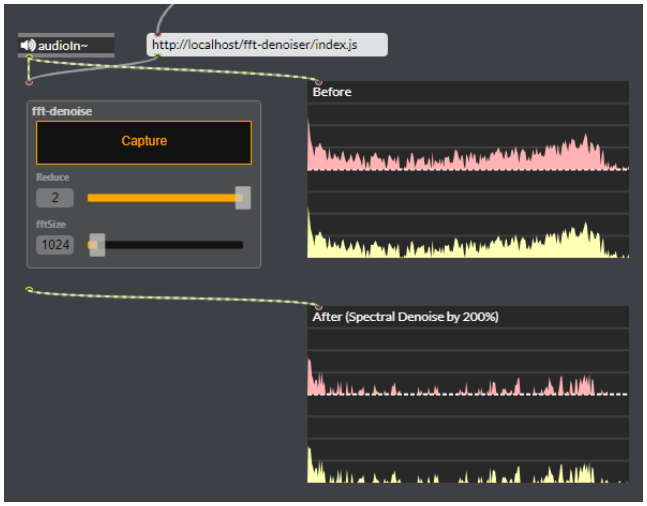

Figure 10: Spectral Denoiser in WAM written in Faust and running in JSPatcher.

```
import("stdfaust.lib");
// global variable
fftSize = hslider("fftSize", 1024, 2, 16384, 1);
// Bins from 0Hz to Nyquist frequency
bufferSize = fftSize / 2 + 1;
freqPerBin = ma.SR / fftSize;
fftprocess(r, i, bin) = out, out with {
    mag = r * r + i * i : sqrt;
    // reset for each frame
    divident = mag : *(bin) : + ~ *(bin > 0);
    // sum of the magnitude of this frame
    divisor = mag : + ~ *(bin > 0);
    out = divident / divisor * freqPerBin : ba.sAndH(bin
          == bufferSize - 1);
};
process = fftprocess;
```

4.5 Spectral Denoiser

Using FAUST's recursive operation to create a loop, we can make a spectral denoiser that "freezes" a reference spectral frame and process the future input frames, reducing the magnitude of each bin by the frozen frame. It memorizes a background noise print when the user clicks the button and removes it from the input audio. Figure 10 shows how the denoiser, compiled in WebAudioModules 2.0 (WAM) [3] format and loaded from a local URI, reduces noise from the microphone input using two spectroscopes.

```
import("stdfaust.lib");
// global variable
fftSize = hslider("fftSize", 1024, 2, 16384, 1);
// Bins from 0Hz to Nyquist freq
bufferSize = fftSize / 2 + 1;
// cartesian to polar
cartopol(x, y) = x * x + y * y : sqrt, atan2(y, x);
// polar to cartesian
poltocar(r, theta) = r * cos(theta), r * sin(theta);

// Button to freeze the current frame
freezeBtn = checkbox("Capture");
// Denoise amount
reduceSld = hslider("Reduce", 0, 0, 2, 0.01);

freeze(rIn, iIn, bin) = out with {
    // If the Capture button is on, put the current
         frame bins in a loop
    freezeSignal(sig, frz) = orig + freezed with {
        orig = sig * (1 - frz);
        freezed = orig : @(bufferSize) : + ~ (*(frz) : @
             (bufferSize - 1)) * frz;
    };
    out = freezeSignal(rIn, freezeBtn), freezeSignal(iIn
         , freezeBtn);
};

fftproc(rIn, iIn, bin) = out with {
    // Get the current bin and the frozen bin's
         magnitude and phase
    pol = cartopol(rIn, iIn);
    mag = pol : _, !;
    phase = pol : !, _;
    pol_frozen = freeze(rIn, iIn, bin) : cartopol;
    mag_frozen = pol_frozen : _, !;
    phase_frozen = pol_frozen : !, _;

    // Reduce the magnitude and keep the phase
    out = poltocar(mag * (1 - freezeBtn) + (mag -
         mag_frozen * reduceSld) * freezeBtn : max(0),
         phase);
};

process = fftproc;
```

5 Discussion

The approach that we adopted for spectral processing in the AudioWorklet context is not the only viable one. In fact, using pure JavaScript code for spectral manipulation can be more straightforward in some cases. However, there is no existing JavaScript framework for real-time spectral processing, and it is complicated to write every spectral processor from a native AudioWorklet class. The audio buffering and the FFT module usage in this work can be further generalized for such a JavaScript framework for audio developers. Since our work aims to provide intuitive tools for artists and non-professional audio developers, we use a VPL JSPatcher and a domain-specific languag (DSL) FAUST that are closer to the music domain and are likely more acceptable for these people.

Most of the audio DSLs are optimized for audio buffer manipulations that are calculations based on vectors (1-demensional arrays). It is an interesting challenge to find a way to process spectra on these DSLs, Max's `pfft~` approach — considering real/imaginary values as 2 audio channel and providing additional bin indexes — has been used for years in various music applications which proves its ability for spectral processor algorithm implementation. This is the main reason that we adopt the same approach. However, it still has some limitations. For example, as the spectrum is "flattened" to a stream of complex numbers that are given one after another, and the algorithm needs to output the spectrum in the same order/format, we cannot use the information of a higher bin index to process data of a lower bin in a same frame, unless we introduce a 1-frame delay and process the last frame when it has been completely streamed. In other words, with this approach, if some algorithms, typically pitch-shifters, need the information about the whole spectrum before being able to process it, they can only output the processed spectrum in the next frame which adds some latency.

Compared to `pfft~`, FAUST spectral processor is still in an experimental status and may needs further development for certain features. First, `pfft~` contains a regular Max patcher that can mix audio-rate and control-rate data. For some values such as analysis result, it can be more efficient to store them as a control-rate variable rather than an audio-rate signal, while every variable in a FAUST DSP is an audio-rate signal. Second, larger memory space is available via `buffer~` object in `pfft~`, making it more flexible for complex data processing and analysis. For example, some spectral processors can interpolate between different preloaded spectra which requires features to load files into buffers. It is hard to implement the same in the web-based FAUST environment.

Nevertheless, the current design of the FAUST spectral processor make it one of the first musician-oriented spectral processing solutions entirely available on the web platform. Some examples are built for proof-of-concept purposes. We will continue to improve them, implement new spectral processing algorithms, and find new use cases. This work also encourages us to extend the spectral processing feature to other FAUST-targeted languages and platforms.

6 References

[1] P. Adenot. A wait-free single-producer single-consumer ring buffer for the Web, June 2022.

[2] M. Borins. From Faust to Web Audio: Compiling Faust to JavaScript using Emscripten. In *Linux Audio Conference (LAC-14)*, Karlsruhe, Germany, May 2014.

[3] M. Buffa, S. Ren, O. Campbell, T. Burns, S. Yi, J. Kleimola, and O. Larkin. Web Audio Modules 2.0: An Open Web Audio Plugin Standard. In *Companion Proceedings of the Web Conference*, Lyon, France, Apr. 2022. Association for Computing Machinery.

[4] A. Correya, J. Marcos-Fernández, L. Joglar-Ongay, P. Alonso-Jiménez, X. Serra, and D. Bogdanov. Audio and Music Analysis on the Web using Essentia.js. *Transactions of the International Society for Music Information Retrieval*, 4(1):167–181, Nov. 2021.

[5] A. A. Correya, D. Bogdanov, L. Joglar-Ongay, and X. Serra. Essentia.js: A JavaScript Library for Music and Audio Analysis on the Web. In *International Society for Music Information Retrieval Conference*, pages 605–612, Montréal, Canada, 2020.

[6] J. M. Grey and J. W. Gordon. Perceptual effects of spectral modifications on musical timbres. *The Journal of the Acoustical Society of America*, 63(5):1493–1500, May 1978. Publisher: Acoustical Society of America.

[7] Y. Orlarey, D. Fober, and S. Letz. FAUST : an Efficient Functional Approach to DSP Programming. In E. D. France, editor, *New Computational Paradigms for Computer Music*, pages 65–96. Paris, France, Jan. 2009.

[8] H. Rawlinson, N. Segal, and J. Fiala. Meyda: an audio feature extraction library for the Web Audio API. In S. Goldszmidt, N. Schnell, V. Saiz, and B. Matuszewski, editors, *Proceedings of the International Web Audio Conference*, WAC '15, Paris, France, Jan. 2015. IRCAM. ISSN: 2663-5844.

[9] S. Ren, S. Letz, Y. Orlarey, D. Fober, R. Michon, M. Buffa, L. Pottier, and Y. Yu. Modernized Toolchains to Create JSPatcher Objects and WebAudioModules from Faust Code. In *Proceedings of the International Web Audio Conference*, Cannes, France, July 2022. Université Côte d'Azur.

[10] S. Ren, S. Letz, Y. Orlarey, R. Michon, D. Fober, E. Aamari, M. Buffa, and J. Lebrun. FAUST online IDE: dynamically compile and publish FAUST code as WebAudio Plugins. In A. Xambó, S. R. Martín, and G. Roma, editors, *Proceedings of the International Web Audio Conference*, WAC '19, pages 71–76, Trondheim, Norway, Dec. 2019. NTNU. ISSN: 2663-5844.

[11] S. Ren, L. Pottier, M. Buffa, and Y. Yu. JSPatcher, a Visual Programming Environment for Building High-Performance Web Audio Applications. *Journal of the Audio Engineering Society*, 70(11):938 EP – 950, Nov. 2022.