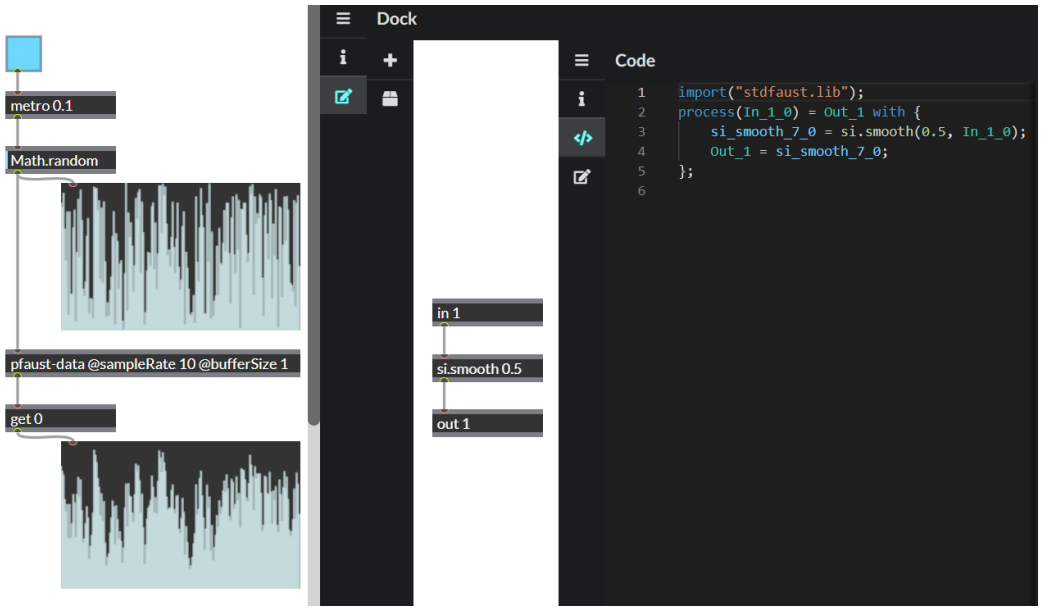

Figure 74 pfaust-data object example.

3.3.5 Faust FFT

Using the Web Audio API, web developers can easily get the real-time spectral data from an AnalyserNode that does Fast Fourier Transform (FFT) analysis on demand and returns magnitude per frequency bin via getByteFrequencyData or getFloatFrequencyData. However, we cannot use this method for serious spectral analysis and processing due to several reasons.

First, an FFT analysis usually outputs for each frequency bin a complex number value which infers both magnitude and phase information. But the AnalyserNode drops the phase information so that the returned frequency domain data is incomplete.

Second, the window function using by the FFT analysis is not controllable, it is hard-coded to a Blackman window.[102]

Third, for scenarios where users need to analyze successive windows such as with the Short-Time Fourier Transform (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 not able to reconstruct or modify input signal with given data and API.

In the Faust IDE, we used raw AudioWorklet output data and with a WebAssembly version of KissFFT[103] for STFT analysis and data visualization. The system is partially usable for spectral processing. For improving performance (and avoiding array copy between threads), we put the FFT module inside an AudioWorklet, implemented an IFFT API, and designed a flexible spectral processing framework around the FFT/IFFT API.

---

[102] https://webaudio.github.io/web-audio-api/#blackman-window

[103] https://github.com/j-funk/kissfft-js/

3.3.5.1 Case of the Spectral Processing System in Max

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. Max has a graph-based system that is dedicated to audio spectral processing: `pfft~`. It is widely used for musical project and has an interesting and viable design for a web-based environment to implement.

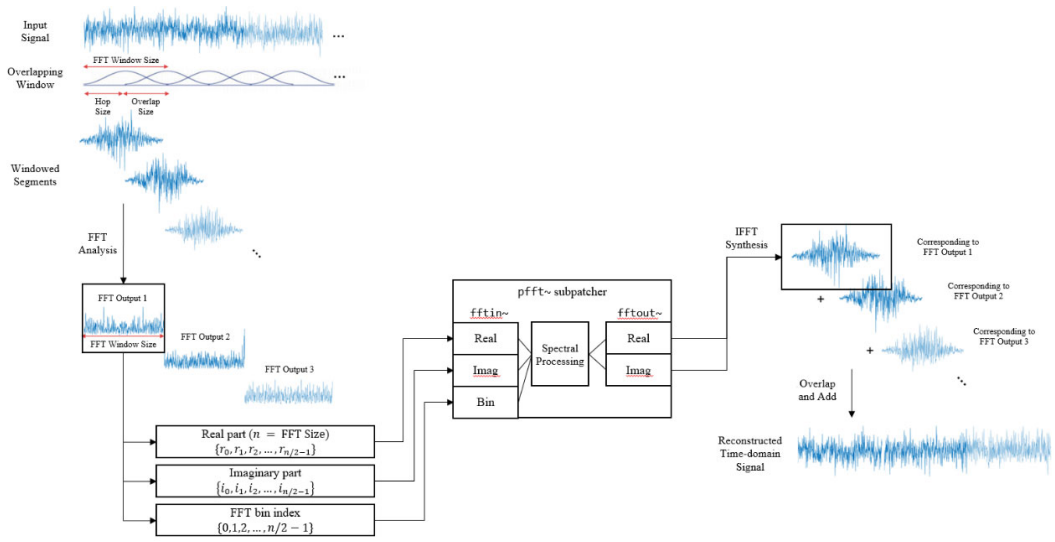

Figure 75 Max's pfft~ framework.

As Figure 75 shows, the `pfft~` Max 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. In this example, users get 512 complex numbers per FFT. 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 need 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.

The `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 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.

3.3.5.2 Spectral Processing Architecture on the Web

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* (Adenot, 2022) to avoid data moving, read/write pointers are stored for each buffer.

Based on a WebAssembly version of both FFTW and KissFFT libraries[104] 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.[105] In our version, 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:

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

The FFTed array has the same length as its input if n is power of 2. The array contains all the

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

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

necessary information about its input since $i_0 = i_{\frac{n}{2}} = 0$.[106] However, KissFFT outputs differently, which is:

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

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:

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

We also noticed that Max's pfft~ drops a pair of the last complex number $\left\{r_{\frac{n}{2}}, i_{\frac{n}{2}}\right\}$ 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.

For the implementation, we added in faustwasm library the proposed AudioWorklet. The processor presumes that the developer prepared a FFTUtils JavaScript class/object that has following static fields (Table 16):

Table 16 Static fields of a prepared FFTUtils class.

Field	Description
`windowFunctions`	Optional, an array of functions (possibly a static getter) that generates different window functions for FFT windowing. Each window function should have two parameters: current index and the window's total length. The function outputs the corresponding factor of the current index and will be repeatedly called to generate a whole window.
`getFFT`	An async function (or static method) that returns a class constructor that creates an InterfaceFFT class containing FFT/IFFT methods.
`fftToSignal`	A function (or static method) that converts (splits) from FFTed (spectral) signal to three arrays for Faust processor's input. It accepts in order 4 arrays: FFTed signal, real, imaginary, and index. It fills the arrays in place.
`signalToFFT`	A function (or static method) that converts (merges) from Faust processor's output to spectral data for IFFT. It accepts in order 3 arrays: real, imaginary, and signal for IFFT. It fills the arrays in place.
`signalToNoFFT`	A function (or static method) that converts (merges) from Faust processor's output to direct audio output. Its definition is the same as `signalToFFT`.

The `getFFT` function returns an InterfaceFFT class constructor. The InterfaceFFT includes

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

necessary methods for FFT/IFFT methods (Table 17):

Table 17 InterfaceFFT interface's methods.

Method	Description
`constructor`	The class constructor accepts one parameter indicating the FFT size. It will initialize the memory space for FFT/IFFT.
`forward`	The method performs a FFT on a given array. It accepts either a new Float32Array or a function that fills an internal Float32Array. The latter can be faster because it avoids an extra array copy. It returns the FFTed array.
`inverse`	The method performs a IFFT on a given array. It accepts either a new Float32Array or a function that fills an internal Float32Array. The latter can be faster because it avoids an extra array copy. It returns the IFFTed array.
`dispose`	The method frees the allocated memory space and disposes the class instance.

When the AudioWorklet processor received a whole frame of audio signal, it will do the following operations:

1) Apply the window function,
2) Use the `InterfaceFFT.forward` to perform an FFT,
3) Use the `FFTUtils.fftToSignal` to convert the FFTed signal array to Faust processor's inputs,
4) Call the Faust DSP,
5) Use the `FFTUtils.signalToFFT` or `FFTUtils.signalToNoFFT` to convert the Faust processor's outputs to the FFTed signal array.
6) Use the `InterfaceFFT.inverse` to perform an IFFT (skip this step if the `noIFFT` option is enabled),
7) Remove the windowing.

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.

For each audio output channel, we assume that the Faust DSP will output 2 channels containing real and imaginary values of the current frame. For example, a 2-in 2-out bypass Faust spectral processor can be written as (Figure 76):

- 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 = _, _, !, _, _;
```

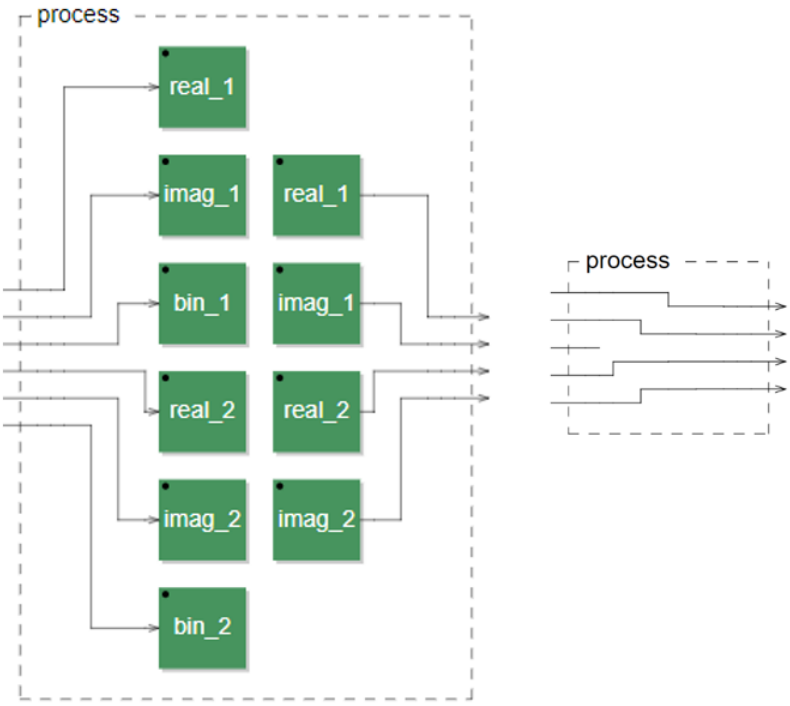

Figure 76 2-in 2-out bypass Faust spectral processor (left: verbose version, right: simplified version).

Figure 77 shows the architecture of the Faust spectral processor inside an AudioWorklet processor.

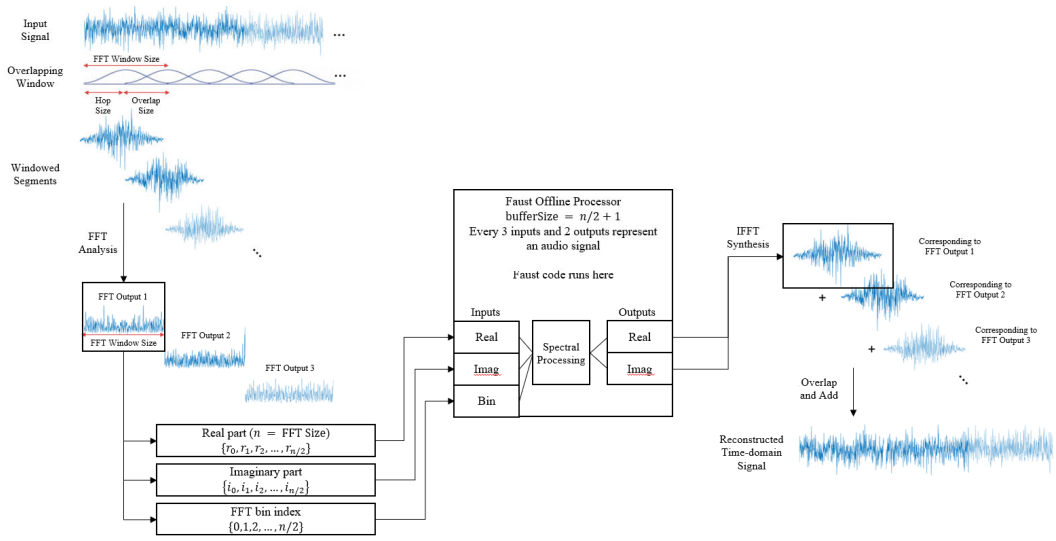

Figure 77 Web-based real-time spectral processing using Faust.

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 `nofft` 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[107] 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 (Figure 78).

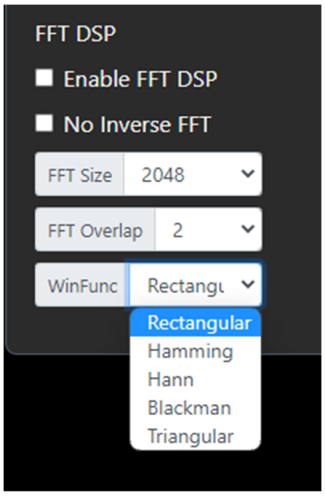

Figure 78 Window Functions of FFT processing in Faust IDE.

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 3.3.5.3.

3.3.5.3 Examples

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

To start with, a simple gain controller in a spectral processor will multiply both real and imaginary part by the gain factor (Figure 79):

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

[107] Available on https://faustide.shren.site/

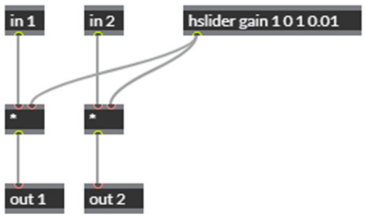

Figure 79 Simple spectral gain in Faust.

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 80):

```
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 {
    low = bin < cut_bin; // Check if the bin is lower than the cut_bin
    // If lower, use the parameter value, else 1
    gain_bin = (low == 0) * gain + low;
};
```

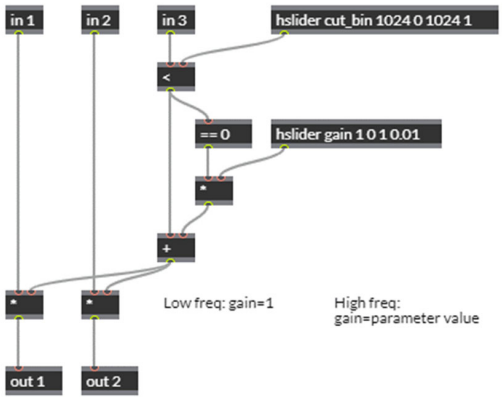

Figure 80 Simple FFT filter in Faust.

We can use the `fftSize` DSP parameter to get the actual FFT type information and make the cutoff parameter Hertz-based (Figure 81):

```
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);
cut_bin = cut / (ma.SR / fftSize); // FFT bin index of the cutoff frequency
```

```
// ...
```

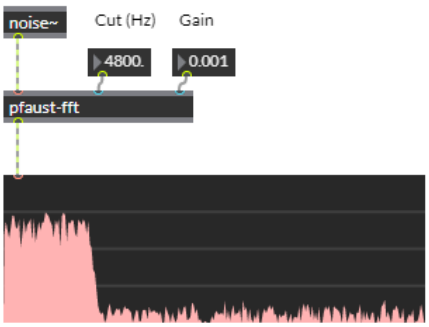

Figure 81 Spectral high-shelf filter running in JSPatcher.

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 82):

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

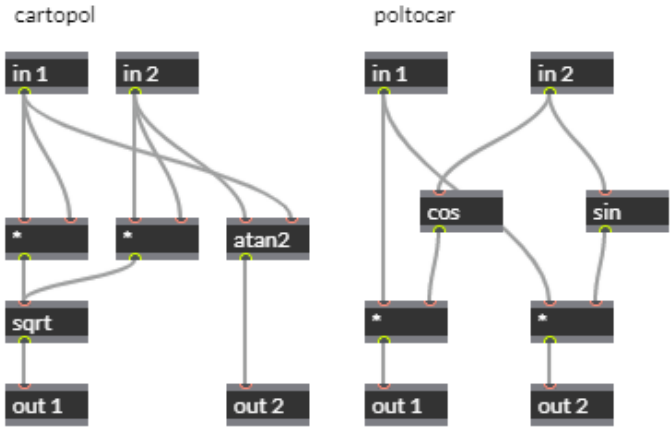

Figure 82 Cartesian-polar coordinates converter in Faust.

With the converter, cross synthesis and convolution effects can be easily implemented. Figure 83 shows how a spectral convolution looks like in a spectral Faust patcher, where cartesian-polar coordinates converters are implemented in subpatchers.

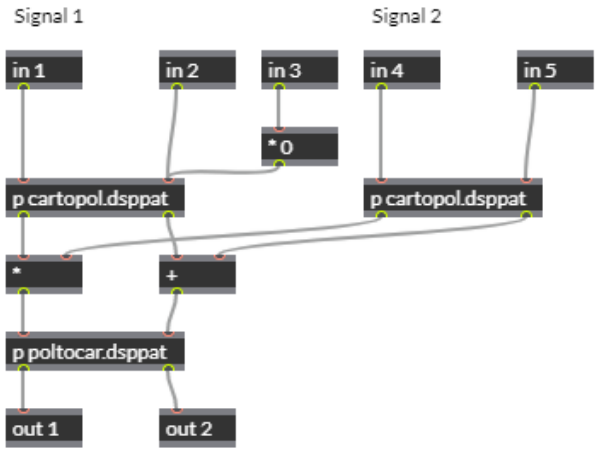

Figure 83 Spectral convolution in Faust patcher.

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 need 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 (Grey & Gordon, 1978) analyzer in Faust code, Figure 84 shows its runtime in JSPatcher.

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

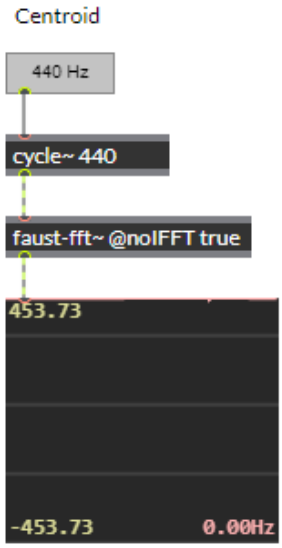

Figure 84 Spectral centroid analyzer written in Faust and running in JSPatcher.

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 85 shows how the denoiser, compiled in WAM2 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;
```

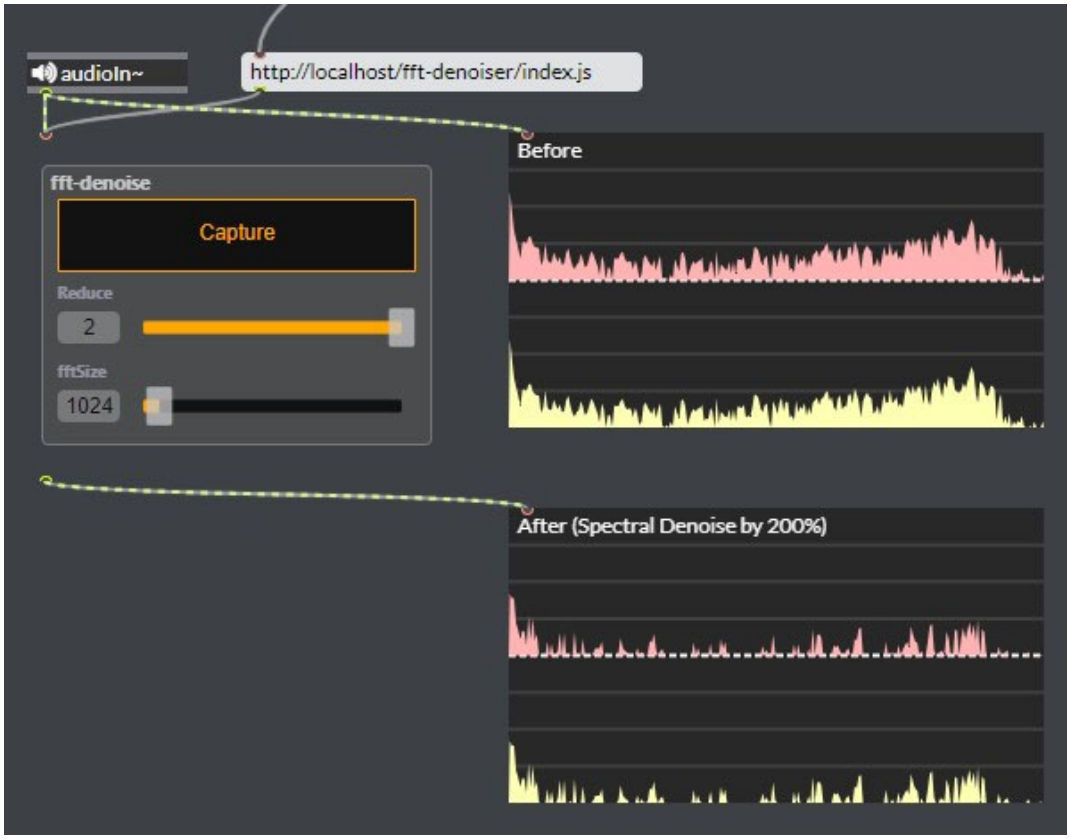

Figure 85 Spectral denoiser in WAM written in Faust and running in JSPatcher.

3.3.5.4 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 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 need 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. 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. It is hard to implement the same in Faust language.

Nevertheless, Faust spectral processor with it's the framework that we designed, *is one of the first musician-oriented spectral processing solutions available on the web platform.* Some examples are built for proof-of-concept purposes. A real-world use case will be presented in 4.4.

3.3.6 Summary of Contributions

We contributed various DSP tools based on Faust language that runs in JSPatcher:

- Programs and applications:
    - Some of the features are included in JSPatcher's main repository, see code written in following files:
        - Objects available in Faust patcher: https://github.com/Fr0stbyteR/jspatcher/blob/master/src/core/objects/Faust.tsx
        - Faust patcher analyzer: https://github.com/Fr0stbyteR/jspatcher/blob/master/src/core/patcher/FaustPatcherAnalyser.ts
        - Other Faust box objects: https://github.com/Fr0stbyteR/jspatcher/tree/master/src/core/objects/faust
    - Compiled DSP objects written in Faust for JSPatcher are available in an

independent GitHub repository: https://github.com/jspatcher/package-dsp
    - The experimental Faust FFT DSP is added in faustwasm's repository, see this file: https://github.com/grame-cncm/faustwasm/blob/master/src/FaustFFTAudioWorkletProcessor.ts
- Papers
    - The graph editor for Faust language was first presented and published in the International Functional Audio Stream (Faust) Conference 2020 (Ren, Pottier, et al., 2020);
    - The method for creating Faust-based JSPatcher objects was presented and published in the International Web Audio Conference 2022 (Ren, Letz, et al., 2022);
    - A paper on the Faust FFT DSP is submitted and accepted by the International Web Audio Conference 2024.

3.4 Computer-aided Composition on the Web

DSPs often provides parameters of which values can be changed in runtime to modify some internal state of the DSP, such as a gain controller's volume or a oscillators frequency. However, it is hard to create music only with these low-level parameters because music often has more complex structures in both time and frequency domain. Higher-level conceptions and visual representations need to be used to describe and generate music, especially for algorithmic composition.

To design such a CAC system for composers, abstractions of musical conceptions and the display of musical scores are two key features. OpenMusic, developed at IRCAM, shows the power of such a VPL in algorithmic composition with its composer-oriented design. The web environment, with its strong accessibility from device to device and flexibility from the UI perspective, is already a common platform for the low-code development system. It is likely to be highly suitable for a CAC system. High-quality digital musical scores can be dynamically rendered and displayed on a computer using the SVG (Scalable Vector Graphics) format. Libraries like VexFlow,[108] Guido,[109] Verovio,[110] or abcjs[111] can render music scores on the web. However, not all of them provide an API to interactively deal with the musical structure (model) behind the score. The challenge we are facing is a need for a JavaScript-compatible musical model system that can:

- Calculate and compose abstract music (like a MIDI file data structure),
- Play via WAM synthesizers,
- Be displayed as a musical score.

Therefore, we created a JavaScript library called Sol for the calculation of different musical concepts. It aims to deal with the musical model issue in the web-based CAC system. Then, a WebAssembly version of Guido[112] is used to render the score as it provides the AR (Abstract Representation) API to easily convert the musical model to the score. Finally, these features are

[108] https://www.vexflow.com/

[109] https://guido.grame.fr/

[110] https://www.verovio.org/

[111] https://www.abcjs.net/

[112] https://github.com/grame-cncm/guidolib

integrated as a package into JSPatcher.[113]

3.4.1 Modeling Musical Concepts

3.4.1.1 Pitch and Note

The difference between a pitch and a note can be ambiguous. In our library, "note" means different "pitch classes" in an octave. A note can be A, B, C, D, E, F and G with any number of sharp or flat accidentals.

Text strings can be used to construct a Note object. For example, "C###" "Db" or "Bx" are recognizable as "note C with 3 sharps" "note D with 1 flat" and "note B with double sharps." Internally, the note name and the accidentals will be preserved and can be used for further calculation. Integer numbers can also be used to construct a note as the offset in semitones from note C, in this case, the note name and the accidentals will be determined automatically.

Pitch is an extended Note object with additional octave information. It is then a more concrete concept as the frequency can be calculated. It is also possible to get a pitch from a frequency with an approximation of a semitone. The pitch's offset follows the $C4 = 60$ standard.

Both Note and Pitch object supports some kinds of mathematical calculations. Adding or subtracting a number from a Pitch will change by semitones its offset. The offset difference can also be calculated between two Pitches For example, the following code creates a C4 pitch, by adding 12 to get a C5 pitch, then by subtracting a C4 pitch to get 12.

```
const pitch = new Pitch("C4");
pitch.add(12).toString(); // => "C5"
// Now pitch is C5
pitch.sub(new Pitch("C4")); // => 12
```

The multiplication between a pitch and a number is supported to calculate an approximated pitch based on the current pitch as the fundamental frequency and a multiplication ratio. The division is also possible between the pitch and a number or another pitch to calculate a new pitch or the frequency ratio between them. For example, the following code creates a C3 pitch, by multiplying 4 (to its frequency) to get 2 octaves higher pitch C5, then by multiplying 1.5 to get a perfect fifth higher.

```
const pitch = new Pitch("C3");
pitch.mul(4).toString(); // => "C5"
// Now pitch is C5
pitch.mul(1.5).toString(); // => "G5"
```

3.4.1.2 Interval

Interval is the distance between two pitches, but it is more complex than just the number of semitones. With different note names and accidentals, the same distance, in equal temperament, can have different intervals such as diminished fourth and major third. Its calculation involves three properties: quality, degree and octave.

[113] https://github.com/jspatcher/package-cac