The Web Audio API as a Standardized Interface Beyond Web Browsers

Benjamin Matuszewski, Otto Rottier

B. Matuszewski and O. Rottier, "The Web Audio API as a Standardized Interface Beyond Web Browsers"
J. Audio Eng. Soc., vol. 71, no. 11, pp. 790–801, (2023 November).
DOI: https://doi.org/10.17743/jaes.2022.0114

The Web Audio API as a Standardized Interface Beyond Web Browsers

BENJAMIN MATUSZEWSKI,[1] AND OTTO ROTTIER[2]
(benjamin.matuszewski@ircam.fr) (ottorottier@gmail.com)

[1] *STMS Ircam-CNRS-Sorbonne Université, Paris, France*
[2] *Software Engineer, Utrecht, The Netherlands*

In this paper, the authors present two related libraries, `web-audio-api-rs` and `node-web-audio-api`, that provide a solution for using the Web Audio API outside the Web browsers. The first project is a low-level implementation of the Web Audio API written in the Rust language, and the second provides bindings of the core Rust library for the Node.js platform. The authors' approach here is to consider Web standards and specifications as tools for defining standardized APIs across different environments and languages, which they believe could benefit the audio community in a more general manner. Although such a proposition presents some portability limitations due to the differences between languages, the authors think it nevertheless opens up new possibilities in sharing documentation, resources, and components across a wide range of environments, platforms, and users. The paper first describes the general design and implementation of the authors' libraries. Then, it presents some benchmarks of these libraries against state-of-the-art implementation from Web browsers, and the performance improvements that have been made over the last year. Finally, it discusses the current known limitations of these libraries and proposes some directions for future work. The two projects are open-source, reasonably feature-complete, and ready to use in production applications.

0 INTRODUCTION

The Web Audio API first proposed by Chris Rogers in 2011 has received increasing attention by developers, artists, and researchers in the last decade. The possibility of doing advanced audio processing and synthesis natively on the Web platform has unfolded a number of novel possibilities in different application domains such as music performance and creation, gaming, or online conferencing. With its release as a W3C Recommendation in 2021 [1], the Web Audio API has reached a point at which its growing community of users, amount of documentation, and tutorials make it an interesting option for someone willing to develop an audio application.

Despite this increasing interest and the large possible application domains, the present authors believe that the adoption of the Web Audio API is limited by the sandboxed and constrained environments that are Web browsers. Their hypothesis is that with its stable and standardized specification, the Web Audio API could provide an interesting solution for audio application outside Web browsers. This approach is in line with recent trends and pervasiveness of using Web technologies and standards outside the Web. For example, the Node.js[1] and Deno[2] efforts to implement standards such as JavaScript modules or the `fetch` API to improve interoperability with Web browsers, or the spread of the Electron[3] project that proposes a way to build native applications from Web-based technologies, can be seen as going in a similar direction.

In this paper, the authors present two related projects and libraries that go in such direction and make use of Web standards as a way to model APIs outside Web browsers. `web-audio-api-rs`[4] is a low-level implementation of the Web Audio API written in the Rust programming language [2, 3] and `node-web-audio-api` a library that provides JavaScript bindings to the Rust implementation for the Node.js platform.[5]

[1] https://nodejs.org/.
[2] https://deno.land/.
[3] https://www.electronjs.org/.
[4] https://github.com/orottier/web-audio-api-rs.
[5] https://github.com/ircam-ismm/node-web-audio-api.

The main objectives for these libraries is to reach full compliance with the specification and, on its Rust version, deviate from it only in specific, justified, and predictable cases. On the Rust side, the authors aim at providing an API that is both easy to use when coming from a JavaScript background with only a few adaptations to the Rust coding style and specificities and, inversely, can leverage on existing JavaScript documentation and tutorials for Rust users. For the Node.js bindings, the goal is to provide a drop-in replacement to run existing Web Audio code and libraries in a Node.js context without any modifications.

As such, the authors believe their proposal can unfold interesting possibilities in several application domains. For example, it could open novel perspectives in several artistic areas such as distributed music systems [4, 5] or digital musical instruments creation [6]. For start-ups and industries, it could provide a novel and standardized tool to support the path from prototyping to production (e.g., simple prototyping using JavaScript easily ported to Rust for performance and deployment) in different domains such as game development or sonification of embedded devices.

After a presentation of similar attempts in recent years (SEC. 1), the authors describe the general design and implementation of their libraries (SEC. 2). In SEC. 3, they present some benchmarks of their libraries against state-of-the-art implementation from Web browsers, as well as the performance improvements that have been made over the last year. Finally, in SEC. 4, they discuss the current known limitations of the libraries and propose some directions for future work.

1 RELATED WORKS

Several attempts have been made over the years to implement the Web Audio API as an autonomous library to be used outside Web browsers. For example, the LabSound project [7] proposes an open-source C++ library originally forked from the Webkit implementation. Although the library appears to be maintained, the authors acknowledge in the presentation of the project that "LabSound has deliberately deviated from the spec for performance or API usability reasons," a standpoint which is "expected to continue into the future as new functionality is added to the engine." Derived from this project, the node-audio library [8] proposes Node.js bindings built on top of LabSound. However, the project is presented as in an "extremely experimental state" and did not receive any update for 5 years.

The web-audio-engine [9] and web-audio-api [10] projects both propose an implementation of the Web Audio API written in pure JavaScript that could therefore be used within Node.js applications. However, these two libraries are obviously tied to the inherent limitations of the JavaScript language (e.g., single-thread, interpreted) and therefore cannot reach any serious performance comparison with low-level implementations. Furthermore, the first project has been archived by its author and no further support can be expected because the last published version of the second dates from 7 years ago.

Finally, the authors can cite the servo-media component [11], created in the context of the Servo project backed-up by Mozilla, and also implemented in Rust. Although much larger in its scope, the library contained some promising steps toward a low-level implementation of the Web Audio API. However, it can be observed from the examples that the public Rust API differs in many regards from the JavaScript API, preventing reuse of acquired knowledge or to easily port code from one language to the other. Additionally, because the abandonment of the Servo project by Mozilla in August 2020, the library does not appear to be actively developed or maintained.

In the growing Rust ecosystem, a number of audio libraries with different scopes and goals have been proposed. For example, the dasp project [12] offers a number of components that provide low-level abstractions for working with digital audio signals. Inside this suite, the dasp-graph component aims at creating modular and dynamic audio graphs. However, the component is very general purpose and does not provide higher-level building blocks as defined in the Web Audio API.

On the other side of the spectrum, the Rodio project [13] proposes a high-level, beginner-friendly audio playback library. The library provides a number of audio sources and filters, which can be glued together with mixers, delays, crossfades, etc. It lacks, however, a few important building blocks for more advanced audio applications such as allowing multiple input streams per node or automation events. Additionally, it renders the audio output frame by frame, which may be a performance issue in constrained environments.

Inside this general frame, the authors think their proposal has the potential of filling several gaps. First, from a general Web Audio API perspective, it provides a solution decoupled from Web browsers, potentially widening its application areas and community of users. Second, from a Rust perspective, it proposes an intermediary and extensible solution that is both tied to an industry standard and is not yet available in the ecosystem. Finally, from a JavaScript perspective, it provides a new platform (i.e., Node.js) to deploy existing Web Audio code.

2 DESIGN AND IMPLEMENTATION

In this section, the authors describe the main design and implementation aspects of their project considering both the Rust and JavaScript versions. The choice of the Rust language for the core library has been motivated by several reasons. Primarily, Rust is a low-level, memory-efficient language such as C or C++. Hence, Rust features memory efficient primitives (float, double, atomic, reference-counted, etc.) and allows one to choose between stack and heap allocation and to define custom memory allocators. As such, the language provides all the characteristics requested to build real-time audio engines with predictive timing and high performance.

Additionally, compared to C/C++, Rust introduces the concept of data ownership, which prevents the entire class of race conditions and concurrency issues that may arise in

multi-threaded execution. As such, it can guarantee that the concurrent execution of the control thread and render thread is free of data races. This analysis is performed at compile time, so there is no runtime cost of the memory safety rules. A drawback is that not all memory safe programs are allowed by the Rust compiler, forcing the developer to use sub-optimal solutions in specific cases (e.g., graph data structures). In these cases, the developer should benchmark the overhead and resort to `unsafe` operations when necessary. This has not been the case yet in the authors' library. Finally, the Rust ecosystem (e.g., compiler toolchain, dependency management, auto-generated documentation) is modern, easy-to-use, and beginner-friendly.

2.1 General Architecture

On multi-core processor systems, dynamic audio libraries typically split up work between a *control thread* and a *render thread*. This approach can be seen as a variation of the client/server architecture widely implemented in computer music–oriented languages and platforms [14–16]. The Web Audio API, constrained also by the specifics of the JavaScript language, makes no exception and requires the implementation of this pattern.[6]

In such an architecture, the render thread has the sole responsibility of rendering the audio graph and shipping the samples to the operating system (OS) so they can be played by the hardware. This thread has, therefore, very hard real-time constraints: if it is unable to compute the next block of audio samples within its time budget, around 2.9 ms for blocks of 128 samples (the default render size of the Web Audio API) at 44.1 kHz, underruns will occur, and audible artifacts will be produced. On the other hand, the control thread is user-facing and orchestrates all changes to the audio graph. It mainly allows users to change the topology of the audio graph (i.e., adding and removing nodes, changing the connections between them) and to access and update parameters of the nodes. The authors' implementation indeed follows this model and uses lock-free message passing for cross-thread communication.

Each node of the audio graph is therefore composed of two complementary objects (see Fig. 1) that are always created as pairs:

- **`AudioNode:`** User-facing object that implements the `AudioNode` interface from the W3C spec. The `AudioNode` does not perform any audio processing but allows the user to change the audio graph and rendering by sending messages to its related `AudioProcessor`.
- **`AudioProcessor:`** Object that is placed on the render thread to produce the actual audio samples. It cannot be directly manipulated by the user and relies on instructions received from its corresponding `AudioNode` to change its behavior.

[6]https://www.w3.org/TR/2021/REC-webaudio-20210617/#control-thread-and-rendering-thread.

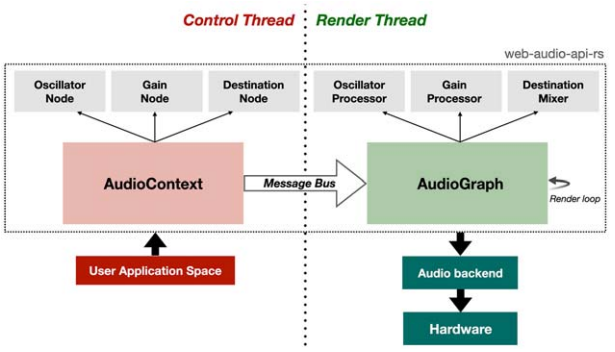

Fig. 1. General architecture of the library, each component living in the control thread (audio context and audio nodes) has an associated counterpart living in the render thread (audio graph and audio processors).

2.2 The JS Interface as the API Model in Rust

The authors' Rust implementation follows the API defined by the W3C specification as closely as possible considering the specificities of the language. Indeed, some differences are impossible to avoid in translating an API targeted at JavaScript to Rust. For example, the most obvious deviation is that the authors chose to conform to the Rust coding style standard, which enforces the use of `snake_case` for methods and `CamelCase` for `enum` variants.

Beyond this simple adaptation, an important set of differences lie in the implementation of object-oriented programming concepts that are very common in the specification but not supported by the language. First, Rust does not allow the authors to override property getters and setters. Therefore, instead of directly exposing attributes,[7] the authors offer two methods [i.e., `attribute()` and `set_attribute()`] as exemplified below:

```
// Specification IDL
interface AudioBufferSourceNode : AudioScheduledSourceNode {
    attribute AudioBuffer? buffer;
}

// Rust implementation
AudioBufferSourceNode::buffer() -> AudioBuffer
AudioBufferSourceNode::set_buffer(buffer: AudioBuffer)
```

Similarly, Rust does not provide any mechanism for method overloading. In such cases, rather than providing a single method with several `Option` parameters, the authors chose to expose several specialized methods. For example, `AudioScheduledSourceNode::start([time])` is derived as `start()` and `start_at(time: f64)`.

Finally, Rust is strongly designed toward composition and does not provide any inheritance mechanism. Therefore, to mimic inheritance, the authors use `Traits` for parent/extendable interfaces such as `AudioNode`, `AudioScheduledSourceNode`, or `BaseAudioContext` and composition of `struct` for dictionaries such as the `AudioNodeOptions` interface.

[7]Note that some attribute names such as `type` and `loop` are reserved keywords in Rust, and in these cases, the authors decided to append an underscore to the getter, e.g., `OscillatorNode::type_`.

The example code in Listing 1, which shows a scrubbing effect realized using a granular synthesis approach, illustrates what the authors think is a representative snippet of the resulting public Rust API. A given *File* is decoded into an `AudioBuffer` that is then consumed by several `AudioBufferSourceNodes` scheduled and configured to read the `AudioBuffer` back and forth at half speed.

```
use std::fs::File;
use std::{thread, time};
use web_audio_api::buffer::AudioBuffer;
use web_audio_api::context::{AudioContext, BaseAudioContext};
use web_audio_api::node::{AudioNode, AudioScheduledSourceNode};

fn trigger_grain(
  audio_context: &AudioContext,
  audio_buffer: &AudioBuffer,
  position: f64,
  duration: f64,
) {
  let start_time = audio_context.current_time();

  let env = audio_context.create_gain();
  env.connect(&audio_context.destination());
  env.gain()
    .set_value(0.)
    .set_value_at_time(0., start_time)
    .linear_ramp_to_value_at_time(1., start_time + duration / 2.)
    .linear_ramp_to_value_at_time(0., start_time + duration);

  let mut src = audio_context.create_buffer_source();
  src.set_buffer(audio_buffer.clone());
  src.connect(&env);
  src.start_at_with_offset(start_time, position);
  src.stop_at(start_time + duration);
}

fn main() {
  let audio_context = AudioContext::new(None);
  // grab an AudioBuffer from file
  let file = File::open("samples/sample.wav").unwrap();
  let audio_buffer = audio_context.decode_audio_data_sync(file).unwrap();
  // launch granular scrub process
  let period = 0.05;
  let duration = 0.2;
  let mut position = 0.;
  let mut incr = period / 2.;

  loop {
    trigger_grain(&audio_context, &audio_buffer, position, duration);

    if position + incr > audio_buffer.duration() - (duration * 2.)
      || position + incr < 0.
    {
        incr *= -1.;
    }
    position += incr;
    thread::sleep(time::Duration::from_millis((period * 1000.) as u64));
  }
}
```

Listing 1: Example code of a scrubbing effect realized with a granular synthesis approach written in Rust.

These differences prevent from dropping existing JavaScript source code into Rust and, thus, impede portability to some extent. However, the authors think the syntax and conceptual similarities can help developers to easily port their existing code to native in a very simple fashion. Although one could imagine creating some tool to automatically transpile JavaScript Web Audio code to Rust, providing such a tool would require substantial effort and is outside the scope of this project. The Node.js bindings described in SEC. 2.5 provide an intermediate solution to this particular issue.

2.3 AudioWorklet as the Internal Model

The public Rust API is de facto larger than the API defined by the specification given that the library must expose some of its internals to users to allow them to implement their own dedicated nodes. However, the design of the `AudioProcessor` interface is very much inspired by the `AudioWorkletProcessor` interface as defined in the specification [17]. Hence, `AudioProcessors` are stateful objects that will execute a callback of the following form on every render tick:

```
// Specification IDL
callback AudioWorkletProcessCallback = boolean (
    FrozenArray<FrozenArray<Float32Array>> inputs,
    FrozenArray<FrozenArray<Float32Array>> outputs,
    object parameters
);

// Rust implementation
fn process(
    &mut self,
    inputs: &[AudioRenderQuantum],
    outputs: &mut [AudioRenderQuantum],
    params: AudioParamValues,

    scope: &RenderScope,
) -> bool;
```

Here, `inputs[n][m]` and `outputs[n][m]` follow a planar layout where arrays of audio samples are stored in the `m`th channel of the `n`th input or output. The `parameters` object contains the computed values for each `AudioParam` of the `AudioProcessor` for this rendering quantum. The return value corresponds to the `tailTime` behavior, which allows the `AudioProcessor` to be dropped by the `AudioGraph` when 1) its related `AudioNode` has been dropped itself, 2) it has finished its rendering, and 3) it has no input connections left.

The additional `RenderScope` argument is modeled after the specification of the `AudioWorkletGlobalScope`. It contains the current time, sample frame, and sample rate, and in the future, it will allow processors to share data (such as wavetables or Fast Fourier Transform results) within the render thread.

The `AudioRenderQuantum` type used for `inputs` and `outputs` is a specialized container type that uses fixed-sized, reference-counted arrays with *copy on write* semantics. It allows for efficient implementation of up-mixing and down-mixing, fan-in/out of channels, and, in a general way, moving input and output buffers around without making copies or allocating memory.

2.4 Control and Render Threads

Method calls on the audio contexts, audio nodes, and audio params that occur in the control thread, typically spin off a control message to be handled by the render thread. These methods, which operate synchronously and do not block, are therefore safe to use on a UI thread. Additionally, the concept of *control thread* itself is more abstract in this implementation because all `AudioNodes` are implemented in a thread-safe way.

The communication between the control thread and render thread is established through a single asynchronous communication channel called the *control message queue*. The queue is a first-in first-out queue where items are ordered by time of insertion, the oldest message being at the front of the queue. This single message bus ensures that updates made to the audio graph are applied in the right order in the render thread.

Render thread–wise, the authors' implementation does not directly interface with the system audio backend (e.g., ALSA, ASIO) but delegates this functionality to cross-platform audio libraries. Currently, the default audio middleware is `cpal` [18], but the authors also offer experimental support for `cubeb` [19]—a backend used by various programs such as high-profile console emulators, but mostly developed by Mozilla for Firefox—behind a compiler flag. Both libraries support a large range of audio backends so

this library should work on a wide range of operating systems and hardware. The chosen audio backend will set up a real-time high priority OS thread to collect the audio samples. This is the thread in which all `AudioProcessors` are running. In a general manner, the entry point of an audio backend library is a callback that expects the authors to fill an array of float or integer interleaved audio samples (i.e., a `FnMut(&mut [f32|u16|i16])` in Rust idiom). Although the desired size of the output array can be specified (e.g., 128 samples per channel in the Web Audio API specification), in practice, this value is not always available or allowed by the underlying system. In such a case, the authors still render the audio graph piecewise in blocks of 128 samples, maintaining the accuracy and regularity of the computed audio time, but perform internal buffering to align the buffer sizes. For example, if the audio library callback data size is 192, two buffers of size 128 will be rendered on one over two backend calls.

In the render thread, the `AudioGraph` is responsible to store the `AudioProcessors` and render them in the right order by performing a topological sort of the nodes as defined in the specification.[8] This algorithm ensures that when node A has an outgoing connection to node B, the processor of A is called before B. Therefore, when A has rendered, its resulting samples are copied and mixed appropriately to serve as input for processor B. The ordering is cleared and recomputed each time a node is added to the graph or when the connections between them are altered. Additionally, during this topological sort, a cycle detection is performed to mute every node that is part of the cycle unless a `DelayNode` is present.

Following both the specification and the JavaScript behavior, an `AudioNode` can go out of scope (i.e., to `Drop` in Rust idiom) in the control thread, while its processing counterpart continues to run inside the render thread. In such a case, the renderer is released only when it has finished processing (governed by the `tailTime` behavior). To implement this *dynamic lifetime* of the audio nodes, the `AudioGraph`, therefore, keeps track of the processors that have finished their rendering and consequently cleans up branches of the audio graph that will no longer emit output. On the contrary, if the `AudioContext` itself is dropped in the control thread, the corresponding render thread is immediately halted, and all resources are released.

2.5 Node.js Bindings

On top of the core Rust implementation, the authors also provide JavaScript bindings to `web-audio-api-rs` for usage within the Node.js platform. The package is available on the *npm* registry under the `node-web-audio-api` package name. This step backward to JavaScript from the Rust implementation opens a perspective the authors propose to call *Isomorphic Web Audio*. With this library, it is indeed possible to write JavaScript Web Audio components that can run seamlessly in the browser or on the Node.js platform. As such, the authors believe this approach can be an interesting contribution in several areas such as distributed music systems using nano-computer without any screen (thus without Web browser) [20, 5] and more generally to the field of the Internet of Musical Things [4]. Additionally, the library opens the possibility for testing the whole implementation against the test suite from the web-platform-test project.[9]

The strategy used to build the library has been to generate most of the binding code directly from the Interface Description Language extracted from the specification.[10] This automatic approach has the additional benefit of testing and reinforcing the consistency of the Rust API. Furthermore, it allows to hide back the specificities of the Rust API described in SEC. 2.2 to expose an interface that is exactly similar to the one provided by Web browsers. As such, this new package potentially allows to reuse a whole set of existing higher-level libraries [21, 22] in a Node.js context and opens up possibilities of novel cross-environment workflows from prototyping to production.

At time of writing, while not completely up-to-date with the Rust implementation, the library already exposes all the nodes implemented on the Rust side and allowed to port an important subset of the existing Rust examples. Listing 2 shows a simple example of the usage of the library, i.e., an amplitude modulation synthesis with a periodic automation on the modulation frequency, running in a Node.js environment. It can be seen that, apart from the first `import` statement, which could be quite simply abstracted by a dedicated package, the script could perfectly run in a browser context without any further modifications.

```
import { AudioContext } from 'node-web-audio-api';

const audioContext = new AudioContext();

const modulated = audioContext.createGain();
modulated.connect(audioContext.destination);
modulated.gain.value = 0.5;

const carrier = audioContext.createOscillator();
carrier.connect(modulated);
carrier.frequency.value = 300;

const depth = audioContext.createGain();
depth.connect(modulated.gain);
depth.gain.value = 0.5;

const modulator = audioContext.createOscillator();
modulator.connect(depth);
modulator.frequency.value = 1.;

modulator.start();
carrier.start();

let flag = 1;

(function loop() {
  const freq = flag * 300;
  const when = audioContext.currentTime + 10;
  modulator.frequency.linearRampToValueAtTime(freq, when);

  flag = 1 - flag;

  setTimeout(loop, 10 * 1000);
}());
```

Listing 2: Example code of an amplitude modulation synthesis realized with the JavaScript bindings in a Node.js context.

[8] https://www.w3.org/TR/webaudio/#rendering-loop. Note that there are subtle differences in the present authors' implementation because they are storing edge information in a different way. The resulting sort order conforms to the specification though.

[9] https://github.com/web-platform-tests/wpt.

[10] https://webaudio.github.io/web-audio-api/#idl-index.

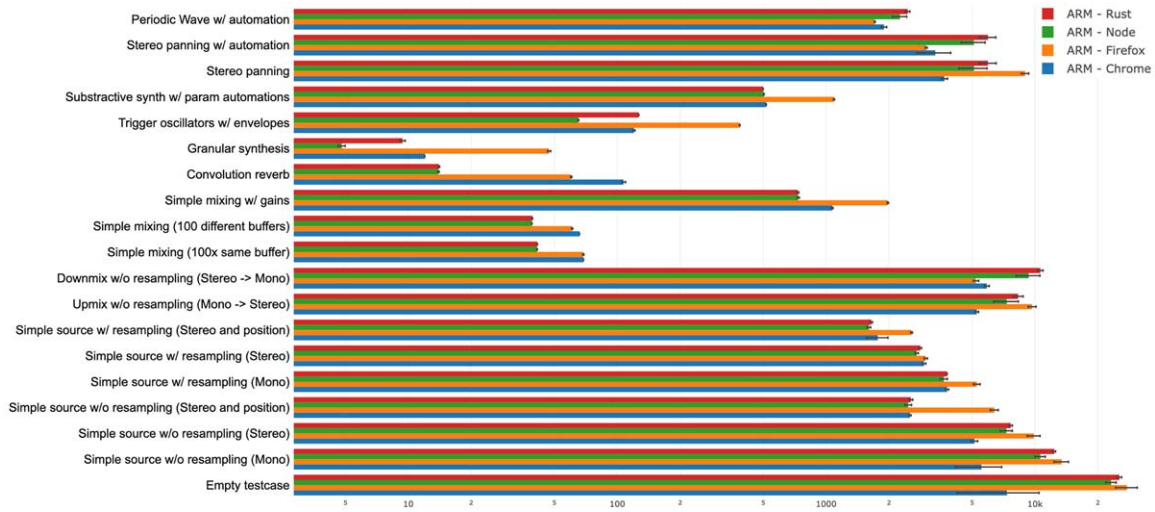

Fig. 2. Benchmarks run on a Macbook Pro 2020 (ARM) comparing the performances of the authors' library in its pure Rust version and with the Node.js bindings compared to the Firefox and Google Chrome Web browsers. Reported results are the speed-up compared to real time; therefore, higher values mean better performance. Note that the *x* axis is reported on a logarithmic scale.

3 PERFORMANCE

Audio processing systems typically face hard real-time constraints: when the production of new samples exceeds the time budget for a block (i.e., 128 samples for the Web Audio API), the rendering lags behind the actual playback, which produces discontinuities in the signal that are immediately perceived by listeners as "glitches." On the contrary, the more the overhead of the audio processing library is low, the more users can build complex audio graphs without running into issues. Performances of the render thread of any audio library are therefore a key aspect that must be carefully audited and optimized.

3.1 Benchmarks

To evaluate the current performances of this implementation (both in its raw Rust implementation and with the Node.js bindings) against state-of-the-art implementations from Web browsers, the authors chose to use the benchmarks developed by Adenot [23] to audit the performances of the Firefox Web browser. Although this specific test suite does not benchmark every single aspect of the Web Audio API and can therefore produce a biased picture, it still provides interesting highlights on key aspects of the implementation and allowed the authors to track their progress since the previously published results (cf. SEC. 3.2). The core principle of these benchmarks is to calculate an `AudioBuffer` (of generally 120 s) offline with different audio graphs and common synthesis methods. The time needed to calculate the buffer (i.e., between the start and end of the rendering) compared to the duration of the buffer therefore gives an estimation of the performances of the rendering against real time. The authors think this approach is particularly suited for benchmark comparisons between different languages because it cancels the JavaScript overhead in browsers and therefore only measures the audio processing time. As such, it allows to compare this implementation performance against the low-level C++ implementations from Web browsers.

The benchmarks have been run on two different computers: a MacBook Pro 2020 with an Apple M1 processor (ARM) and 8 GB of LPDDR4X-4,266 MHz RAM, and a MacBook Pro 2019 with a 2,3 GHz 8-Core Intel Core i9 processor and 16 GB of 2,400 MHz DDR4 RAM. For each tested platform, i.e., Google Chrome, Firefox, and the authors' Rust and Node.js libraries, the benchmarks were run five times with a dry compilation run for the Rust version and a full reload on a private browsing window between each run in the browsers. Reported results are the mean and standard deviation of these five executions and represent the speed-up compared to real time (i.e., *bufferDuration*/*processingTime*); hence, higher values mean better performance.

The ARM results presented in Fig. 2 (Intel results are reported in Fig. 4 in APPENDIX A.1[11]) show that, except in some specific cases, this implementation compares quite well against mature Web browsers' implementations: around 1.3 times slower than Google Chrome and 1.8 times slower than Firefox for the Rust version on the ARM computer if all the benches are taken into account.

A more detailed analysis of the results show that the authors are noticeably behind only on very specific cases which gives some hints on where their future efforts should concentrate: 1) parsing graphs composed of many nodes, which is clearly visible in the "Granular synthesis" case, and 2) convolution reverberation, which is known as a very hard and specialized problem that will require important work. If these specific cases are removed, the comparison gets far better: around 1.1 times faster than Google Chrome and 1.5 times slower than Firefox, which the authors consider very

[11]Note that for simplicity, the authors will only consider the ARM results in the current and following sections. Results from the Intel computer show a similar global picture.

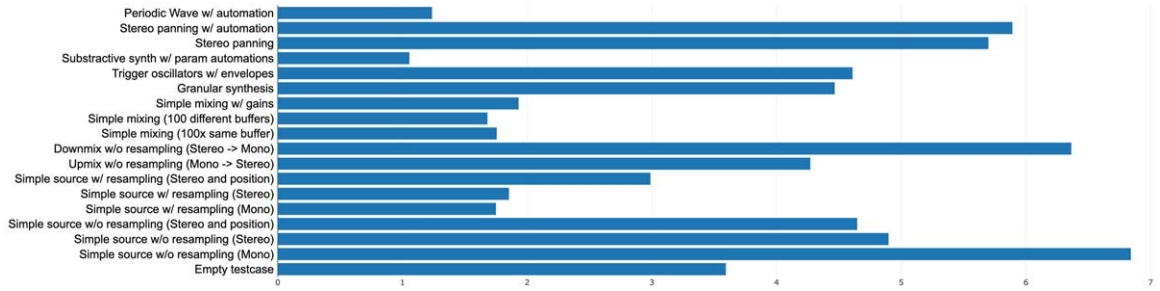

Fig. 3. Normalized performance improvements against the previously published version—Macbook Book Pro 2020 (ARM).

promising, considering these implementations have been optimized for years.

Another point worth mentioning is that, as expected, the performances of the Node.js version are almost in line with the Rust ones, which confirms initial assumptions for benchmarking, i.e., cancellation of the JavaScript overhead. However, an important discrepancy can be noticed between the authors' libraries in the two cases that involve graphs composed of many nodes, i.e., *Granular synthesis* and *Trigger oscillator with envelope*. This difference tends to highlight an issue in this implementation that will require further investigation.

3.2 Optimizations

Important work has been done in the last year to improve the performances of the library (see Fig. 3 for ARM results; Intel results are reported in Fig. 5 in APPENDIX A.1). For this purpose, the authors implemented an important set of optimizations beyond the application of common practices for audio processing such as avoiding heap allocation, mutexes, and thread locks, or simple compiler optimizations. In this section, the authors describe three points that they consider the most interesting and representative.

First, results from Fig. 3 show that the *Granular* and *Trigger oscillator with envelope* benches have been improved by a factor of around 3. This gain in performance is primarily due to improvements in the representation of the graph itself and improvements made to the algorithm for its parsing. As discussed above, although encouraging, results from Fig. 2 show that this critical point is not yet satisfactory and will require further work.

Second, important work has been done on the processing of `AudioParam`. The authors consider this point worth mentioning because they implemented a strategy that, although compliant with the specification regarding audio rate and control rate parameters behavior, can be described as a generalization over these concepts. Indeed, in the current implementation, *a-rate* params produce a value for each sample of a quantum only when an automation is occurring and behave as a *k-rate* param the rest of the time, producing only one single value for the entire render quantum. Such a strategy allowed the implementation of a number of simple optimizations in many nodes. For example, a gain node linked to a mute button can simply act as a pass-through or produce a silent buffer, while still behaving normally when transitioning from 1 to 0 or inversely.

Finally, because of the planar channel layout required by the specification and described in SEC. 2.3, the authors revised their processors to ensure channels are accessed and written one by one to avoid jumps between different sections of memory at the sample level. In some nodes, e.g., `AudioScheduledSourceNode` or `DelayNode`, such an approach required to implement an additional loop dedicated to compute and store intermediate results (e.g., absolute position at the sub-sample level), which may seem surprising but actually led to better performances.

An untapped source of performance improvements are fast-math operations. These operations break IEEE compliance for float calculations, such as reordering of instructions or assuming only finite numbers exist. However they can enable great performance improvements, such as better vectorization of float buffer computations, which are utilized a lot in audio processing. In Rust, these fast math flags can be applied granularly (at the function level) but are unfortunately not available in the stable compiler toolchain yet. In the near future, the authors will experiment with the nightly compiler toolchain to evaluate the possible performance improvements and when possible, will backport some of the improvements to work in the stable channel. This can be done, for example, by unrolling certain loops by hand and making them suitable for 4- or 8-lane f32 vector operations.

4 DISCUSSION AND FUTURE WORK

With the recent addition to their implementation of the features "Access to a different output device"[12] and the `AudioRenderCapacity`[13] interface, the authors are nearing feature completeness according to the specification. However, although preliminary work has been conducted to fully test their implementation against the Web Platform Test Suite,[14] the authors are conscious that a number of important points still needs to be tackled in order to provide a fully compliant and more efficient API.

[12]https://github.com/WebAudio/web-audio-api/issues/2400.
[13]https://github.com/WebAudio/web-audio-api/issues/2444.
[14]https://github.com/b-ma/node-web-audio-api-wpt.

4.1 Current Known Limitations

One important concept of the Web Audio API that is currently missing in the authors' implementation are asynchronous functions (i.e., methods that return a JavaScript `Promise`). The Rust equivalent of a function returning a `Promise` is an `async` function that landed in the core language in 2019 [24] and opened the possibility for libraries that perform input/output to offer asynchronous versions of their functionality. Although the authors have already drafted an asynchronous version of some important methods (i.e., `AudioContext.decodeAudioData`, `OfflineAudioContext.startRendering`, and `AudioContext.resume`), their implementation currently only exposes synchronous versions of the async functions in the specification. Although asynchronous APIs could be provided by spawning new threads to turn underlying synchronous APIs into asynchronous APIs, full asynchronous support is also dependent on downstream libraries (e.g., the media decoder) because they must support asynchronous execution before it can be used.

The Web Audio API is rather isolated from other Web standards, which makes it relatively simple to decouple it from the Web. However, it interfaces with few other Web standards that must be considered. Among them, the `MediaStream`[15] and `HTMLMediaElement`[16] interfaces are of great importance.

The `MediaStream` interface appears in the specification through the `MediaStreamAudioSourceNode` (input) and the `MediaStreamAudioDestinationNode` (output) interfaces and allows users to move audio data, respectively, into or out of the boundary of the audio context. These input/output possibilities are essential for building rich apps such as digital audio workstations, video conferencing, or online gaming. For now, the authors provide a minimal, spec-compliant implementation of the `MediaStream` that only allows audio tracks. Advanced features, such as solutions for clock drift of different media sources and sinks, or buffering and playback speed adjustments are not provided.

Likewise, the html `<media>` element, and its related `HTMLMediaElement` JavaScript representation, is an essential component of audio applications on the Web. It provides fetching of external resources, buffering, and different playback commands. The current implementation is very minimal because it only supports file-based media playback but, however, proposes a compliant interface:

```
let context = AudioContext::default();
let mut media = MediaElement::new("source.ogg").unwrap();
// other playback options are available
media.set_loop(true);
let src = context.create_media_element_source(&mut media);
src.connect(&context.destination());
media.play();
```

Besides these general and architectural issues, a few other limitations of this implementation at time of writing are worth mentioning. First, the `PannerNode` only supports mono input signal, multichannel input being downmixed to mono first. The `PannerNode` HRTF (head-related transfer function) database is relatively old, and the panning implementation does not perform any interpolation, which can produce audible artifacts for very fast-moving sound sources. Second, the `ConvolverNode` is limited to mono input as well. It implements a simple on-thread implementation of frequency-domain delay-lines, which is very inefficient and, therefore, only supports short impulse responses. Important work will need to be done to provide more efficient approaches such as hybrid direct form and multi-thread partitioned convolution [25, 26] and support larger impulse responses. Third, the `OscillatorNode` does not fully adhere to the specification: the built-in oscillators are not based on `PeriodicWave` and only apply rudimentary strategy to prevent aliasing (i.e., polyBLEP).

4.2 Extensibility

The question of extensibility is an important aspect that has been tackled in the specification with the legacy `ScriptProcessorNode` and more recently with the introduction of the `AudioWorklet` interface [17]. Such an interface is indeed required in JavaScript because of the particular nature of the language and of the necessity to be able to run arbitrary code in the high-priority audio thread. In a Rust context, this question is, however, posed differently because these limitations and constraints do not hold anymore.

Because the authors' implementation of audio processors closely resembles the signature of the `AudioWorkletProcessor` callback (see SEC. 2.4), they have decided so far to not implement the entirety of the `AudioWorklet` interface on the Rust side (e.g., `MessagePort`, etc.). Instead, the authors expose their `AudioProcessor` trait in the public Rust API library, and users are advised to use it in order to build custom audio nodes. However, regarding the Node.js version of the library, the question comes back in a similar way as within Web browsers. Therefore, in a future version of the library, the authors will consider implementing a compliant interface for `AudioWorklet`. Such addition would furthermore allow users to run existing Web Assembly [27] modules in different environments and languages, opening new perspectives for community-built audio modules.

The `ScriptProcessorNode`, which, contrary to the `AudioWorkletNode`, runs the audio callbacks in the control thread and, as such, is considered flawed by the specification editors, poses a different question. Indeed, the authors consider that in specific situations, the benefits of using this node outweigh the complexity of setting up an `AudioWorletNode` and handling communication between threads. More precisely, the authors consider this node interesting if not used to produce audio samples but only as a way to access the audio stream, which is not feasible using the `AnalyserNode`, to control other processes (e.g., to implement an envelope follower). Additionally, it provides a simple way to introduce audio processing in pedagogical situations in which setting up an `AudioWorkletNode` would also be overwhelming. Hence,

[15] https://www.w3.org/TR/mediacapture-streams/.

[16] https://www.w3.org/TR/2011/WD-html5-20110113/video.html#audio.

Table 1. Example of benchmarks results obtained with the Spotify test suite run on a Macbook Book Pro 2019 (Intel). Results are given in microseconds per second of processed audio data and node, so lower is faster.

Test	Chrome	Firefox	Node.js
GainAutomation-exp-a-rate	140	641	706
GainAutomation-linear-a-rate	67	182	314
Delay-default	79	819	580
Analyser	43	348	93

the authors have decided to implement this node in the near future despite the fact that it is considered deprecated in the specification.

4.3 Performances

To audit and further improve the performances of their library, the authors have conducted preliminary work to adapt the test suite proposed by Spotify[17] to their Node.js implementation.[18] Because this project focuses on different aspects of the API compared to the suite used in SEC. 3.1, first results tend to show a complementary picture from the results reported above. Table 1 presents a small subset of the results obtained that the authors think are representative of these differences. Analyzing these results and improving the implementation accordingly will therefore require important future work.

5 CONCLUSION

In this paper, the authors have presented two relates libraries: `web-audio-api-rs`, a novel implementation of the Web Audio API specification implemented in the Rust language, and `node-web-audio-api`, a library that provides Node.js bindings to the core Rust library. First, the general design and implementation of the libraries were described. The authors then presented the current performance of the two libraries, compared to the state-of-the-art implementations provided by Web browsers, and the improvements that have been obtained compared to the previously published results. Finally, the authors discussed the current known limitations of their implementations and future directions for the development of the project.

Although still a work in progress, the two libraries expose a stabilized API, implement an important part of the specification, and provide reasonable performances. The authors consider that their approach of considering Web standards as a tool for defining standardized APIs across environments and languages could provide an interesting contribution to the audio community in a more general manner. Indeed, one could imagine implementing bindings on top of the core Rust library in other languages such as Python or Go, allowing users to take advantage of the standardized nature of the API and extensive documentation to build audio applications in their language of choice.

As open-source projects, the presented libraries are currently mainly developed and maintained by the two authors of the paper, with outside contributions in the form of pull requests and reported issues. To consolidate the project further and help to share development efforts, one of the authors' next objectives is now to develop a broader community of users and contributors.

[17] https://github.com/spotify/web-audio-bench.

[18] https://github.com/b-ma/web-audio-bench.

6 ACKNOWLEDGMENT

The authors would like to thank all their contributors and especially Jerboas86 for their precious contributions to the project. The authors would also like to thank their colleagues at IRCAM for their ideas and support. The project has received support from the DOTS research project funded by the French National Research Agency (ANR-22-CE33-0013-01).

7 REFERENCES

[1] W3C, "Web Audio API Specification," *W3C Recommendation* (2021 Jun.). https://www.w3.org/TR/webaudio/.

[2] N. D. Matsakis and F. S. Klock, "The Rust Language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104 (2014 Dec.). https://doi.org/10.1145/2692956.2663188.

[3] O. Rottier and B. Matuszewski, "A Rust Implementation of the Web Audio API," in *Proceedings of the 7th Web Audio Conference* (Cannes, France), pp. 11 (2022 Jul.). http://doi.org/10.5281/zenodo.6767674.

[4] L. Turchet, C. Fischione, G. Essl, D. Keller, and M. Barthet, "Internet of Musical Things: Vision and Challenges," *IEEE Access*, vol. 6, pp. 61994–62017 (2018 Sep.). http://doi.org/10.1109/ACCESS.2018.2872625.

[5] B. Matuszewski, "A Web-Based Framework for Distributed Music System Research and Creation," *J. Audio Eng. Soc.*, vol. 68, no. 10, pp. 717–726 (2020 Oct.). http://doi.org/10.17743/jaes.2020.0015.

[6] I. Poupyrev, M. J. Lyons, S. Fels, and T. Blaine, "New Interfaces for Musical Expression," in *Proceedings of the Human Factors in Computing SystemsHuman Factors in Computing Systems*, pp. 491–492 (Seattle, WA) (2001 Mar.). http://doi.org/10.1145/634067.634348. N

[7] . Porcino and D. Diakopoulos, "Graph-Based Audio Engine," https://github.com/LabSound/LabSound (2023 Apr.).

[8] D. Ramirez, "Graph-Based Audio API for Node.js Based on LabSound and JUCE," https://github.com/ramirezd42/node-audio (2018 Apr.).

[9] mohayonao, "Pure JS Implementation of the Web Audio API," https://github.com/mohayonao/web-audio-engine (2018 Jan.).

[10] S. Piquemal, "Node.js Implementation of Web Audio API," https://github.com/audiojs/web-audio-api (2023 Feb.).

[11] Servo, "Media," https://github.com/servo/media (2023 Aug.).

[12] RustAudio, "The Fundamentals for Digital Audio Signal Processing," https://github.com/RustAudio/dasp (2022 Jul.).

[13] RustAudio, "Rust Playback Library," https://github.com/RustAudio/rodio (2023 Jul.).

[14] M. Puckette, "FTS: A Real-Time Monitor for Multiprocessor Music Synthesis," *Comput. Music J.*, vol. 15, no. 3, pp. 58–67 (1991 Autumn).

[15] F. Déchelle, R. Borghesi, M. De Cecco, et al., "jMax: An Environment for Real-Time Musical Applications," *Comput. Music J.*, vol. 23, no. 3, pp. 50–58 (1999 Sep.).

[16] J. McCartney, "Rethinking the Computer Music Language: SuperCollider," *Comput. Music J.*, vol. 26, no. 4, pp. 61–68 (2002 Dec.). http://doi.org/10.1162/014892602320991383.

[17] H. Choi, "AudioWorklet: The Future of Web Audio," in *Proceedings of the 44th International Computer Music Conference* (Daegu, South Korea), pp. 110 (2018 Aug.).

[18] RustAudio, "Cross-Platform Audio I/O Library in Pure Rust," https://github.com/rustaudio/cpal (2023 Aug.).

[19] Mozilla, "A Cross-Platform Audio Library in Rust," https://github.com/mozilla/cubeb-rs (2022 Oct.).

[20] B. Matuszewski and F. Bevilacqua, "Toward a Web of Audio Things," in *Proceedings of the 15th Sound and Music Computing Conference*, pp. 225–231 (Limassol, Cyprus) (2018 Jul.).

[21] N. Schnell, V. Saiz, K. Barkati, and S. Goldszmidt, "Of Time Engines and Masters: An API for Scheduling and Synchronizing the Generation and Playback of Event Sequences and Media Streams for the Web Audio API," in *Proceedings of the 1st Web Audio Conference*, paper 19 (Paris, France) (2015 Jan.).

[22] Y. Mann, "Interactive Music With Tone.js," in *Proceedings of the 1st Web Audio Conference*, paper 40 (Paris, France) (2015 Jan.).

[23] P. Adenot, "Benchmarks for the WebAudio API," https://github.com/padenot/webaudio-benchmark (2022 Feb.).

[24] The Rust Release Team, "Announcing Rust 1.39.0," *Rust Blog* (2019 Nov.). https://blog.rust-lang.org/2019/11/07/Rust-1.39.0.html.

[25] W. G. Gardner, "Efficient Convolution Without Input/Output Delay," presented at the *97th Convention of the Audio Engineering Society* (1994 Nov.), paper 3897. http://www.aes.org/e-lib/browse.cfm?elib=6335.

[26] E. Battenberg and R. Avižienis, "Implementing Real-Time Partitioned Convolution Algorithms on Conventional Operating Systems," in *Proceedings of the 14th International Conference on Digital Audio Effects*, pp. 313–320 (Paris, France) (2011 Sep.).

[27] A. Haas, A. Rossberg, D. L. Schuff, et al., "Bringing the Web Up to Speed With WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 185–200 (Barcelona, Spain) (2017 Jun.). http://doi.org/10.1145/3062341.3062363.

A.1 Performances on Macbook Pro 2019 (Intel)

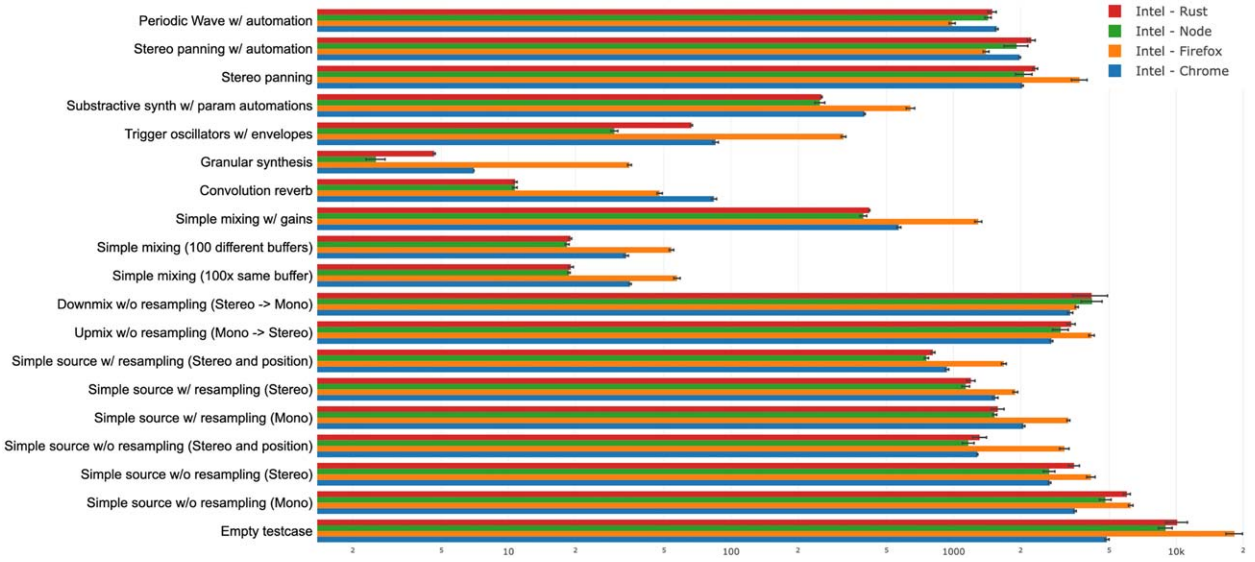

Fig. 4. Benchmarks run on a Macbook Pro 2019 (Intel) comparing the performances of the library in its pure Rust version and with the Node.js bindings compared to the Firefox and Google Chrome Web browsers. Reported results are the speed-up compared to real time; therefore, higher values mean better performance. Note that the x axis is reported on a logarithmic scale.

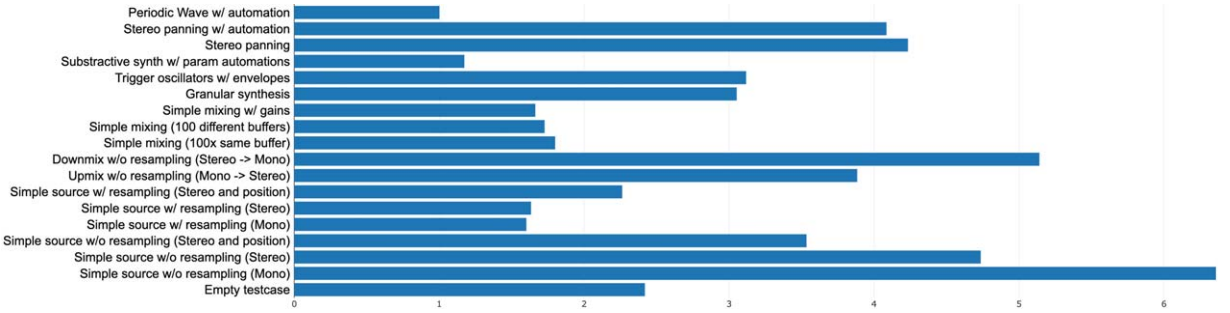

Fig. 5. Normalized performance improvements against the previously published version—Macbook Book Pro 2019 (Intel).

THE AUTHORS

Benjamin Matuszewski

Otto Rottier

Benjamin Matuszewski, Ph.D. in Aesthetics, Sciences and Technologies of the Arts, studied music and musicology before working several years as a developer in the media industry. Since 2014, he is a researcher and developer in the Sound Music Movement Interaction Team at IRCAM, where he conducts transdisciplinary research between engineering, music, design, and human-computer interaction on distributed and interactive music systems based on Web technologies. He also regularly collaborates on artistic projects.

•

Otto Rottier is a software engineer from Utrecht, The Netherlands, and is currently employed by the Dutch Government. In 2014, he finished his Master of Science in Theoretical Physics but has since left the field. He now applies his knowledge of wave forms and its mathematical properties in digital signal processing.