VST PLUG-IN MODULE PERFORMING WAVELET TRANSFORM IN REAL-TIME

Pavel Rajmic, Zdenek Prusa, and Robert Konczi [*]

SPLab, Department of Telecommunications
Brno University of Technology, Brno, Czech Republic

rajmic@feec.vutbr.cz, zdenek.prusa@phd.feec.vutbr.cz

ABSTRACT

The paper presents a variant of the segmentwise wavelet transform (blockwise DWT, online DWT or SegDWT) algorithm adapted to real-time audio processing. The implementation of the algorithm as a VST plugin is presented as well.

The main problem of segmentwise wavelet coefficient processing is the handling of the segment borders. The common border extension methods result in "false" coefficients, which in turn result in border distortion (block-end effects) after particular types of coefficient processing. In contrast, the SegDWT algorithm employs a segment extension technique to prevent this inconvenience and produce exactly the same coefficients as the wavelet transform of the whole signal would do.

In this paper we remove some of the shortcomings of the original SegDWT algorithm; for example the need for the "right" segment extension is canceled. The VST plugin module created is described from the viewpoints of both the user and the programmer; the latter can easily add their own method for processing the coefficients.

1. INTRODUCTION

In the past years, many algorithms appeared that were dedicated to "online" processing of signals by means of the wavelet transform. The necessity of producing transform coefficients in "real time" is natural in many applications, with the field of audio processing being the first of them. Algorithms that have been developed are usually tailored to specific types of application [1, 2, 3, 4, 5], and thus are not general enough and transferable.

We limit ourselves only to methods which produce the *exact* wavelet coefficients of the signal, i.e., as if the signal was known in advance. To mention the main drawbacks of the state-of-the-art methods, many of them are based on the assumption that the length of each segment is a *power of two*—this is in contradiction, for example, to the possibility of having segments of length $s = 96 \neq 2^n$ in ASIO (Audio Stream Input/Output) [6, 7]. The difference between 1024 and 2048 can be in some cases inadmissibly big. Fairly general algorithms can be come across [8, 9, 10], but most of them are quickly found to be limited to the *forward transform only*. And, several methods were designed to work with a *fixed wavelet filter*—this is mostly the case of image processing where, for example, the JPEG2000 file format utilizes the "CDF 9/7" biorthogonal wavelet for lossy image compression [11, 12].

As far as we know, the problem of computing exact transform coefficients when the signal comes in the form of consecutive segments has been first addressed in a general form in the Ph.D. thesis of the first author [13]. A version of the forward part (i.e., analysis) for audio signals was presented at DAFx 07 [14], and as such it was utilized in one of the Queen Mary Vamp Plugins [15], which are actually not of the VST-type.

Since that time several modifications have been published, for example those involving the lifting scheme in the real-time computation [16], and also a generalization to image processing, which allows for wavelet-type processing of arbitrarily sized image blocks in parallel [17]. And, of course, the inverse segmentwise wavelet transform is possible.

The highlighted generality of the SegDWT approach relies on the following facts:

- the wavelet filter(s) can be arbitrary with finite impulse response, including the biorthogonal ones,
- the transform depth can be arbitrary,
- the lengths of individual segments can be chosen arbitrarily (and they can even vary).

1.1. Alternatives

1.1.1. Usual windowing

At first glance, one can doubt if the task could be done by simpler means, via the usual signal "windowing" technique, as is related to the short-time Fourier transform or Gabor transform [18, 19, 20] and to signal analysis via reading the spectrogram. However, one finds out that if the generality were to be preserved, the problems must be solved such as

- it is not clear how to synchronize the wavelet coefficients which would come from the respective window time-shifts (this is due the downsampling, see below),
- if non-linear processing is introduced (denoising by means of thresholding, for example), this would certainly introduce errors.

The algorithm solving such problems would be of the same complexity as the one to be described in this paper. Article [4] copes with the first problem utilizing specific tricks but is limited to linear processing only.

1.1.2. Segments computed and processed separately

Another, even simpler approach would be to compute the wavelet coefficients from each input segment via DWT with no respect to the other segments. One can see this as a special case of windowing when rectangular window and no overlap is used. In DWT (see Section 2) the samples beyond the signal borders have to be

[*] This work was supported by project of the Czech Ministry of Education, Youth and Sports MSM0021630513, and by project FEKT-S-11-17 of Brno University of Technology.
We thank the reviewers for their comments.

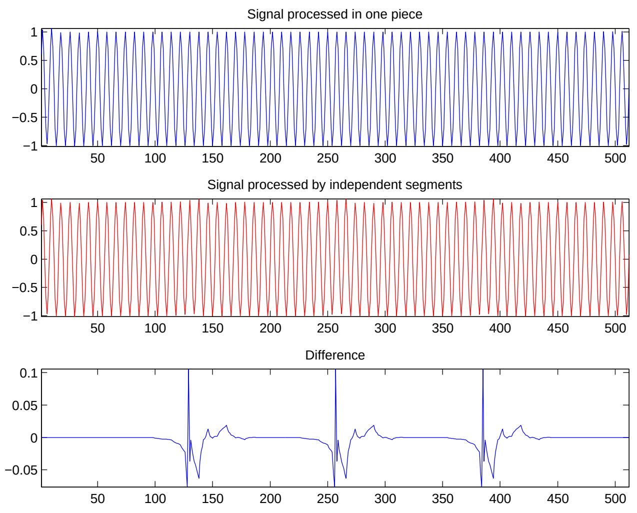


Figure 1: Artifacts appearing when individual segments are processed regardless of their neighbours. A sinusoid was processed non-linearly (hard thresholding) both at one time and segmentwise, respectively. The error of the results is depicted in the bottom graph. Segment length was $s = 128$, wavelet Daubechies 2 with $m = 4$ was used and the boundary treatment was simply zero-padding to ensure substantial differences.

extrapolated. For this reason, in this approach one introduces errors in the coefficients which are "located near the boundaries of the segments", see Fig. 1. This is called "block-end effects" or "border artifacts".

It is worth noting that the "wrong" coefficients might remain "undiscovered" when they are not subject to any processing. In other words, the "wrong" coefficients provide good recovery! It is simply because DWT is fully invertible. However, if one looks at the values of the coefficients, they are revealed immediately. Perfect recovery occurs even if "pointwise" linear processing is performed (i.e., each coefficient is modified linearly with no respect to the values of the others). But as one starts to use, for example, linear filtering in the wavelet domain (which combines the values of the coefficients, see an example in [4]) or even non-linear processing (e.g. denoising or quantization), substantial errors are introduced in the output signal. The greater the decomposition depth J, the larger the range of the border artifacts [21].

1.2. This paper

This paper presents an algorithm which is derived from the general one, and as such it does not suffer from any of the above drawbacks; it contains both the forward and inverse counterparts, and is particularly modified and optimized for real-time audio processing. The description of the implementation in the form of a VST (Virtual Studio Technology) [22] plug-in module is also given.

The concept of the developed system corresponds to the usual processing strategy: Signal decomposition (forward DWT) $\rightarrow$ Modification of the coefficients $\rightarrow$ Signal synthesis (inverse DWT).

Some of the properties of the general SegDWT algorithm will remain unused for the purposes of this paper. In particular, the segment length will be arbitrarily chosen but will remain fixed for all the segments. In addition, the VST host does not indicate that the "last" segment has to be processed, so there is no need to correctly finish the computation at the right border of the signal.

The paper is organized as follows: First, the "classical" dicrete wavelet transform is summarized in Section 2. The description is brief and serves as the basis for emphasizing the modifications made in the segmentwise algorithm, which is theoretically introduced in Section 3. Section 4 then discusses the main issues of the implementation as a VST plug-in module. Section 5 describes the user interface of the product and its functionality, while Section 6 serves as instructions for programmers who would like to either modify the algorithm or include their own code for processing the wavelet coefficients.

2. DISCRETE WAVELET TRANSFORM (DWT)

It was first discovered by Mallat [23] that the wavelet decomposition of signals is equivalent to the recursive filtering process. Mallat's algorithm will be recapitulated in this section.

In contrast to the usual wavelet processing practice, we do not assume the periodicity of the signal, and we do not suppose its whole length is a power of two. This has the following implications:

- We have to decide what kind of "boundary handling" is to be used during the computation; that is, how do we "guess" samples beyond the signal boundary; see [24, 25],
- we will automatically obtain slightly more coefficients than is the number of input samples [21].

One of the main operations in the DWT is downsampling (decimation). This means that in each decomposition step (see below), only every other sample is kept, the rest is neglected. One has to specify, however, if the even-indexed or the odd-indexed samples have to be kept. We assume even downsampling, i.e., samples 1, 3, 5 etc. are kept (the indexing assumed to begin with "0").

Algorithm 1 (DWT, classical)**.** *Let* $\mathbf{x}$ *be a discrete input signal, the two wavelet decomposition filters of length* m *are defined, highpass* $\mathbf{g}$ *and lowpass* $\mathbf{h}$, J *is a positive integer denoting the decomposition depth. The type of boundary treatment has been specified.*

```
1. Denote the input signal $\mathbf{x}$ by $\mathbf{a}^{(0)}$ and set $j = 0$.
2. A single decomposition step:
    (a) Extending the boundaries. Extend $\mathbf{a}^{(j)}$ from both the
        left and the right sides by $(m-1)$ samples, according
        to the specified type of boundary treatment.
    (b) Filtering. Convolve the extended signal with filter $\mathbf{g}$.
    (c) Cropping. Take the central part from the result, so
        that the remaining "tails" on both the left and the
        right sides have the same length of $m-1$ samples.
    (d) Decimation. Downsample the resultant vector.
   Denote the resulting vector by $\mathbf{d}^{(j+1)}$ and store it. Repeat
   items (b) to (d), now using filter $\mathbf{h}$, denoting and storing the
   result as $\mathbf{a}^{(j+1)}$.
3. Increase $j$ by one. If it now holds $j < J$, return to item 2,
   otherwise the algorithm ends.
```

The desired wavelet coefficients are stored in $J+1$ *vectors (of different length)* $\mathbf{a}^{(J)}, \mathbf{d}^{(J)}, \mathbf{d}^{(J-1)}, \ldots, \mathbf{d}^{(1)}$.

The inverse wavelet transform (IDWT) is performed in the reverse manner: first, the vectors of coefficients are upsampled, then filtered with a pair of filters, added up to one vector, which is finally cropped from both sides to obtain a result of the correct length. This is done J times until the time-domain signal is reached.

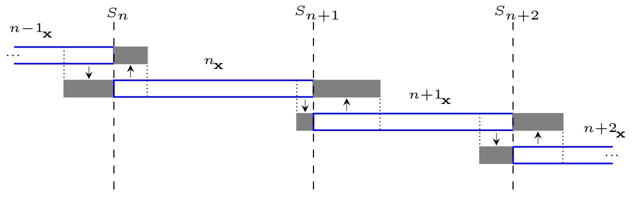


Figure 2: Segmentation of the signal $\mathbf{x}$ and the principle of border extensions. Note that the lengths of the extensions differ from segment to segment.

3. SEGMENTED WAVELET TRANSFORM (SEGDWT)

This section forms the theoretical core of the article. First, the most important principles of SegDWT are presented, and then the algorithm(s) are introduced in more detail. The statements below are proved/derived in [13].

Signal segments are given and they are to be processed consecutively. As can be seen in Fig. 2, the method extends each individual segment prior to further processing. Several signal samples taken from the $(n-1)$th segment (denoted ${}^{n-1}\mathbf{x}$) are appended to the nth segment, ${}^{n}\mathbf{x}$, from its left side; their number will be denoted $L(n)$. In a similar way, the number of samples appended from the right will be represented by $R(n)$. It can be shown that it must hold

$$R(n) + L(n+1) = r(J) \tag{1}$$

where

$$r(J) = (2^J - 1)(m - 1). \tag{2}$$

To put it in words, $r(J)$ is the necessary number of signal samples that must be "shared" between two consecutive segments. Naturally, $R(\cdot) \geq 0$ and $L(\cdot) \geq 0$. The purpose of the left extension is to provide enough samples from the preceding segment(s) to fully calculate the wavelet coefficients at the top-most level of decomposition. The purpose of the right extension is to align the end of each segment to be an integer multiple of 2^J, which results in the correct alignment of the resultant vectors of coefficients.

Although this denotation is simple, it is more effective to switch to a slightly more complicated one: Let S_n denote the index of the left-most sample within the nth segment (in the global point of view, prior to the extension). The very first signal sample is assumed to be located at the index $S_1 = 0$. The $r(J)$ samples can be split into the left extension $L(S_n)$ of the nth segment and the right extension $R(S_n)$ of the $(n-1)$th segment, $L(S_n) + R(S_n) = r(J)$. The following formulas for the extension lengths can be derived:

$$R(S_n) = 2^J \left\lceil \frac{S_n}{2^J} \right\rceil - S_n, \tag{3}$$

$$L(S_n) = r(J) - R(S_n). \tag{4}$$

Fig. 2 shows a situation where all segments are of equal length, which is typical in real-time audio processing. In such a case $S_{n+1} = S_n + s$ naturally holds, where s is the length of each segment. It is clear that in the just described way, the lengths of the extensions can vary from segment to segment (see again Fig. 2), and that the respective overlap lengths are thus *induced*, in contrast to the above-mentioned windowing, where the overlap is *fixed*.

Returning back to Eq. (3), if S_n is an integer multiple of 2^J, then $R(S_n)$ is clearly zero. In other words, choosing the segment borders in such a way would lead to the case when left-side extensions only are used; such a situation would be convenient because it simplifies both the algorithm and the implementation. However, as long as s is usually given by the user/host/system, the above requirement cannot be guaranteed to be fulfilled. In general, one can tackle this problem for the price of introducing an additional delay (equal to s samples) in the processing, which can be seen as waiting for loading samples from the "next" segment and then "resegmenting" to meet the above condition. Section 3.2 will cope with this problem in an alternative way to completely remove it.

3.1. Algorithms of Forward and Inverse SegDWT

The forward and inverse parts of the SegDWT algorithm are presented now. They are described separately, although they are designed to work in conjunction for real-time processing purposes. The forward part produces $J+1$ vectors of wavelet coefficients (one for the coarse coefficients and additional J vectors containing the details) from each segment, which naturally serve as the input of the inverse SegDWT. If these sets of vectors were appended together (level-by-level), the same coefficients would be obtained as if the classical (offline) DWT were performed on the whole signal.

It is worth mentioning that using these segmentwise, partial wavelet coefficients the *inverse* SegDWT algorithm does not reconstruct the processed segment fully. The reconstructed segment's first and last $r(J)$ samples form an overlap to the neighboring reconstructed segments and these must be added up; it is obviously caused by the convolution applied to finite-duration vectors; see Fig. 3. As a consequence, the input-output delay cannot be less than $r(J)$ samples.

Both the forward and inverse parts are described for an arbitrary intermediate segment, meaning that there are enough samples preceding and following the currently processed segment. The first (and the last) segment must be handled a little differently; we refer the reader to the sources cited above. The simple zero-padding used at the signal boundaries is natural in audio, where the sound is supposed to rise "from silence".

3.1.1. Forward SegDWT

The forward SegDWT is in principle similar to the well-known "overlap-save" method used in segmentwise convolution. The similarity lies in reusing the previous samples and discarding the unnecessary coefficients when the partial computation is finished. The resulting wavelet coefficients are then ready to be processed.

Algorithm 2 (Forward SegDWT)**.** *Given the depth J, a pair of wavelet filters $\mathbf{h}, \mathbf{g}$, where m denotes their length:*

1. Extending. *Load the nth segment (starting at index S_n) and determine its left $L(S_n)$ and right $R(S_{n+1})$ extensions using formulas* (2), (3) *and* (4) *Extend the segment properly.*
2. Transforming. *Calculate the wavelet coefficients up to depth J using the classical algorithm (Alg. 1), but* omit step 2a. *The cropping step 2c, however, applies and thus the segment is shorter by $m-1$ samples prior to the downsampling step in each iteration.*
3. Discarding. *Discard first $r(J-j)$ coefficients of each detail coefficient vector at level j; these coefficients were calculated redundantly—they are already a part of the previous segment's detail coefficient vectors.*
4. Storing. *Store the resultant coefficient vectors.*

3.1.2. *Inverse SegDWT*

The inverse SegDWT follows the "overlap-add" principle. The length of a reconstructed segment depends on the lengths of right extensions and it can be calculated by

$$s_{\text{rec}}(S_n) = s + R(S_{n+1}) - R(S_n) + r(J). \tag{5}$$

Algorithm 3 (Inverse SegDWT)**.**

1. Extending. *Append $r(J-j)$ zero coefficients to the beginning of each detail coefficient vector at level j.*
2. Inverse transforming. *Calculate the signal values using the classical inverse algorithm (in the original form). The reconstructed segment's length is $s_{\text{rec}}(S_n)$.*
3. Adding overlap. *Add the overlap from the previous segment to the first $r(J)$ samples of the current segment. Save the last $r(J)$ samples of the current reconstruction, which will serve as the overlap for the subsequent segment.*

A diagram of the complete forward-inverse "workflow" in a concrete case is depicted in Figure 3. The processing of the coefficients is not illustrated but could happen within the rectangular boxes, which contain the respective vectors of coefficients.

3.2. Modifications

As was stated above, for the wavelet transform (of the whole signal), the even type of up-/downsampling is considered. In practice this means discarding the very first sample after each single convolution has been performed. In our segmentwise approach, the left extensions have to be long enough to allow discarding the first sample at each level after the convolution, which clearly sounds like a waste of computational resources. Switching from the even to the odd type of up-/down-sampling, the necessary length of the left extension can be reduced by $2^J - 1$ samples. (This modification is made just internally and the even type of up-/down-sampling is mimicked globally.) Since the length of the right extension is not affected by such a change, this reduction cuts $r(J)$ down to

$$r_{\text{min}}(J) = (2^J - 1)(m - 2). \tag{6}$$

So the algorithm remains the same except that $r(J)$ is substituted by $r_{\text{min}}(J)$. This reduction reflects in the number of discarded coefficients after the forward transform, in the number of zero coefficients that are appended back prior to the inverse transform, and also in the lengths of overlaps during the reconstruction.

As entire segments are being loaded and as the possibly non-zero right extension violates causality, the processing has to be delayed by the length of the segment; but this additional delay might be sometimes unacceptable! There are two "workarounds" how to avoid it:

1. Restricting the segment lengths to achieve $(S_n \bmod 2^J) = 0$ and thus $R(S_n) = 0$. (This contradicts generality.)
2. Employing "negative" right extensions in the sense that the right border of the segment would be aligned with the *lesser* multiple of 2^J, and the samples that remain would be encompassed to the left extension of the following segment. I.e., (1) would still hold but $R(\cdot)$ could fall below zero.

A general solution to the problem requires revising the algorithm so that neither such a 2^J-alignment or right extension is needed and therefore the segment borders of the *input* and the *reconstructed* signal match in time. Then $L_{\text{noright}}(S_n)$ and $r_{\text{noright}}(J)$ coincide:

$$r_{\text{noright}}(J) = L_{\text{noright}}(S_n) = r_{\text{min}}(J) + (S_n \bmod 2^J), \tag{7}$$

and the number of redundant detail coefficients at level j that have to be discarded from the beginning of the respective coefficient vector after the forward transform is

$$r_{\text{min}}(J-j) + \left\lfloor \frac{(S_n \bmod 2^J)}{2^j} \right\rfloor. \tag{8}$$

3.3. Properties and limitations of SegDWT

The segment length must comply with

$$s \geq 2^J. \tag{9}$$

This relation could be theoretically withdrawn, but at the cost of making the algorithm more complicated.

The length of the right extension always satisfies

$$R(S_n) < 2^J. \tag{10}$$

The number of shared samples, $r(J)$, can be even greater than s, the segment length! This feature is newly included in the algorithm (and implementation), and although this makes the algorithm more complex, it allows for demanding combinations of parameters (long filters, short segments, deep level of decomposition).

4. IMPLEMENTATION

The implementation started from the template by J. Schimmel, which is accessible from URL [26]. The template is designed for creating VST plugin modules compatible with VST 2.4 specification. (Section 6 contains details for potential programmers who would like to modify the code/add their own wavelet processor.)

The VST plugin uses "SegDWT" library which was separately created the in C++ language. The library consists of the `SegDWT.h` and `SegDWT.lib` files, whose source codes are available at [27]. The library is processing single precision data types only. Both the forward and inverse transforms are implemented in the class `FloatSegDWT`. Wavelet filters and wavelet coefficient processor are injected into the class by means of the `FloatWfilter` and `IWaveletCoeffProcessor` type objects, respectively. The main public functions of the class are summarized in Listing 1.

The class instance can be created using two constructors. The object containing the (four) wavelet filters can be either supplied directly by a pointer or created in the constructor according to the enumeration data type `FloatWfilter::Type` value. Function `forwardOLS` takes the input array `in` and calculates wavelet coefficient arrays, which have to be allocated beforehand. Value `Sn` identifies the index of the first sample from the global indexing point of view. Function `inverseOLA` is complementary to `forwardOLS`. The function `process` initially calls `forwardOLS`, then processes the function of the `IWaveletCoeffProcessor` object and, lastly, the `inverseOLA`.

The storage of the "previous" samples and the samples of the overlap is handled internally. Routines for allocating memory for arrays of wavelet coefficients are included as well.

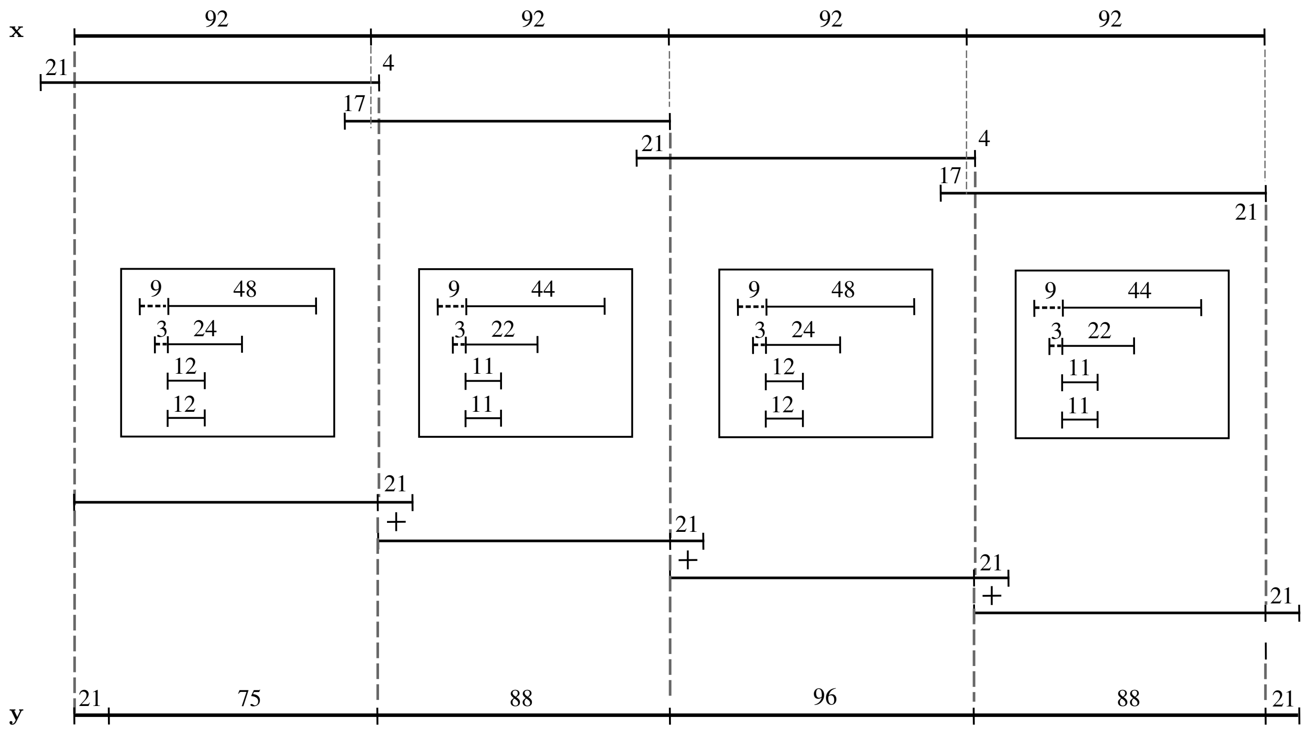


Figure 3: SegDWT algorithm(s) example. Input signal $\mathbf{x}$ is processed by segments of length $s = 92$. The length of the wavelet filters is $m = 4$ and the depth of decomposition is $J = 3$. This setup leads to $r(J) = 21$, which is divided between $L(S_n)$ and $R(S_n)$. Note that the reconstructed signal $\mathbf{y}$ is delayed by these $r(J)$ samples; the first $r(J)$ samples of the reconstructed signal can be viewed as the "reconstruction warmup" and should be set to zero. The values in the boxes represent wavelet coefficient vectors (from top to bottom, the detail coefficients vectors for $j = 1, 2, 3$ and one approximation coefficients vector for $j = 3$) belonging to the respective segments. The highlighted coefficients in levels $j = 1, 2$ are discarded according to step 3 in Alg. 2, and in the inversion they are appended back as zeros according to step 1 in Alg. 3.

Listing 1: Important functions from `FloatSegDWT` class.

```
class FloatSegDWT{
...
public:
FloatSegDWT(FloatWfilter::Type waveletType,
            int newJ=1);
FloatSegDWT(FloatWfilter* newWavelet,
            int newJ=1);

void forwardOLS(float* in,int inLen,
                float* out[],int outLen[],
                unsigned long Sn=0);

void inverseOLA(float* in[], int inLen[],
                float* out, int outLen,
                unsigned long Sn=0);

void process(float* in, float* out,
             int inLen,
             unsigned long Sn=0);

void setWaveletProcessor(
         IWaveletCoeffProcessor* procesor_);
...
};
```

4.1. Convolution and down/upsampling

The convolution and down/upsampling are realized in the time domain. The standard two-direction cyclic buffer is exploited and the convolution and downsampling are done together in a single step, for both the filters simultaneously, according to the formulas

$$a_{j+1}[n] = \sum_{k=0}^{m-1} a_j[2n - k + m - 1]h[k], \tag{11}$$

$$d_{j+1}[n] = \sum_{k=0}^{m-1} a_j[2n - k + m - 1]g[k], \tag{12}$$

for $n = 0, \dots, \lfloor \frac{S_n+s}{2^j} \rfloor - \lfloor \frac{S_n}{2^j} \rfloor + r(J-j)$. Here j stands for the currently processed depth of decomposition (increasing from zero to $J-1$), $a_j[\cdot]$ represents the individual approximation coefficients from vector $\mathbf{a}^{(j)}$, $d_j[\cdot]$ represents the detail coefficients from $\mathbf{d}^{(j)}$, $h[\cdot]$ and $g[\cdot]$ are the wavelet filters' samples. The formulas have to be modified a bit for the first and the last segments—they have to be treated slightly differently. However, the last segment cannot be identified properly in the VST live streaming audio setup.

The described process is equivalent to the "full" linear convolution followed by cropping $m-1$ samples from both sides, followed by the odd downsampling. This way, half the operations are saved.

In a similar manner, the upsampling and convolution in the inverse DWT are done together in a single step, for both the filters simultaneously, according to the formulas

$$a_j[n] = \sum_{k=0}^{\left\lfloor \frac{m-1+(n \bmod 2)}{2} \right\rfloor} a_{j+1}\left[\left\lfloor \frac{n}{2} \right\rfloor - k\right] \tilde{h}\left[2k + (n \bmod 2)\right] + \sum_{k=0}^{\left\lfloor \frac{m-1+(n \bmod 2)}{2} \right\rfloor} d_{j+1}\left[\left\lfloor \frac{n}{2} \right\rfloor - k\right] \tilde{g}\left[2k + (n \bmod 2)\right] \tag{13}$$

$n = 0, \ldots, \left\lfloor \frac{S_n+s}{2^j} \right\rfloor - \left\lfloor \frac{S_n}{2^j} \right\rfloor + r(J-j)$ and $j = J-1, \ldots, 0$. Here $\tilde{h}[\cdot]$ and $\tilde{g}[\cdot]$ are the samples of the reconstructing wavelet filters. Again, the number of operations is reduced in comparison to the equivalent calculation consisting of upsampling both a_{j+1} and d_{j+1}, followed by the linear convolution and the sum of the outcomes.

4.2. Fast convolution+downsampling via FFT is not faster

Although it may seem tempting to perform convolution and resampling directly in the frequency domain using FFT, so far our tests have shown that this approach brings only a negligible performance increase and just in some extreme situations. In the rest of cases, the FFT approach performs worse. Moreover, the frequency domain filtering and resampling bring, apart from segment size constrictions, complications with implementation, and require considerable revision of the SegDWT algorithm. The fact that the FFT approach does not perform so well in such situations is caused mainly by the short length of filters the wavelet filter bank comprises (i.e. $m \leq 20$) and by the relatively short segments, even after they have been extended $s_{\text{ext}} = r_{\text{noright}}(J) + s_{\text{buf}}$.

We compared our implementation of DWT analysis (forward transform only) in time domain with frequency domain implementation using FFTW [28] 3.3.1 default 32bit dll binary distribution using Intel C++ compiler 12.0.1 with \O3 optimization parameter. The tests were run 101 times and the median of the elapsed time was taken as the result, which is plotted in Fig. 4. Standard windows high resolution counter (`QueryPerformanceCounter`) was used as a timer. The FFTW plans were created beforehand. The testing machine was running Windows 7 Professional 64bit on Intel(R) Core(TM) i7 CPU 960 3.2GHz. We can conclude that the FFT implementation starts being beneficial for $J \geq 10$ and $m \geq 17$, since the segment length after (maximal) extension is dependent on J exponentially and on m linearly, and it will be $s_{\text{ext}} = 16368 + s_{\text{buf}}$.

5. USER'S GUIDE

The compiled VST plugin module is accessible through URL [27] in the ready-to-use form of a DLL file ($\sim$1.2 MB). It suffices to copy the file to the plugin directory of your VST host software before the host is run.

The graphical user interface (GUI) is a simple, minimal one and consists of two parts, see Fig. 5.

The left part of the plugin appears always the same. It allows the user to set the global gain *after* the signal synthesis — `Gain`, choosing the wavelet filter — `Wavelet`, the depth of decomposition — `Depth`, and the method of processing the wavelet coefficients — `Process`. Wavelet filter names and filters were adopted

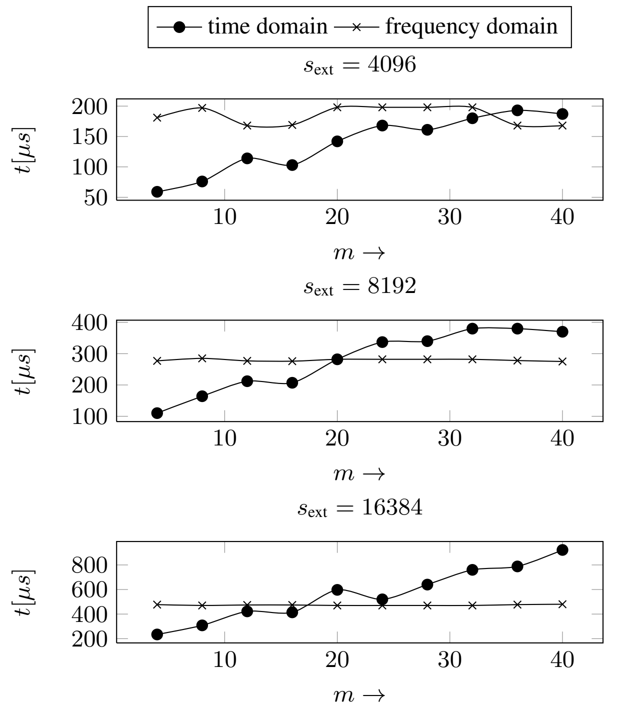


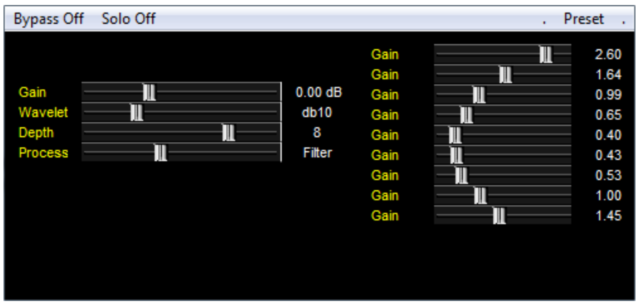
Figure 4: Comparison between time domain an frequency domain forward DWT implementations for different sequence lengths. Since the relative differences were not affected by the choice of the depth of decomposition, $J = 6$ was used.

from the Matlab Wavelet toolbox. The depth of decomposition J is limited by the size of the input buffer s_{buf} (which is controlled by the host application) such that $2^J \leq s_{\text{buf}}$, and at the same time, its maximum was set to $J = 10$.

The right part of the GUI depends on the selected `Process`. There are wavelet coefficient processors bundled with the plugin by default, however they serve mainly to "prove" the proposed algorithm. (Of course, if no modification was done to the coefficients, the output signal would be equal to the input signal up to numerical errors!) The controls at the right hand side allow setting parameters for the respective processors. The bundled processors are:

- *Default* — simply copies the wavelet coefficients and leaves them intact. This is incorporated to verify the perfect reconstruction.
- *Filter* — allows multiplication of wavelet coefficients by the specified values. Each decomposition level has its own value.
- *Hard Thr* — hard-thresholds each subband by a specified value, i.e., the coefficients in absolute values smaller than the threshold are set to zero.
- *Random* — each coefficient in each subband is randomly perturbed. The amout of the scattering is controlled by the specified parameters.

The number of sliders is $J + 1$ in all these cases, each of them linked to the respective decomposition depth. The depths go from the highest-frequency details to the approximation coefficients when taken from the top to the bottom.



Bypass Off Solo Off Preset

```
Gain     0.00 dB        Gain  2.60
Wavelet  db10           Gain  1.64
Depth    8              Gain  0.99
Process  Filter         Gain  0.65
                        Gain  0.40
                        Gain  0.43
                        Gain  0.53
                        Gain  1.00
                        Gain  1.45
```

Figure 5: VST plugin GUI when "Filter" is selected for processing the coefficients.

The delay of the output in comparison to the input is always equal to $r(J)$ regardless of the buffer size.

However, the limit of the CPU performance can be reached on some computers when a demanding combination of parameters is set. For example, $J = 10$, wavelet `db10` (Daubechies 10 with $m = 20$), which leads to $r(J) = 19456$ samples of the left extension which have to be processed in addition to the actual segment samples, whose minimal length is restricted to $s_{\mathrm{buf}} \geq 1024$.

We used two hosts for the (succesful) testing of the created plugin module. The first one was *DSOUND GT-Player (EDU) Express*, version 2.6 Feb 17 2006. This host is simple enough and great for debugging, etc. It is no longer supported, but it is downloadable from the archive [26]. The second host was *Cubase 4* (EDU), version 4.5.2 Build 274.

6. PROGRAMMER'S GUIDE

This section clarifies how to add your own real-time wavelet coefficient processor into the VST (2.4) plugin, to extend and adapt it to your specific needs.

The custom processor can be inserted into the plugin (or, more precisely, into the SegDWT library) by means of the Template pattern paradigm. To do this, create a class inherited from the interface called `IWaveletCoeffProcessor` and which implements all its virtual functions, see Listing 2. In the `setUserVariables` function in `vst_temp.cpp` file, dynamically create the instance of your processor, create the instance of the structure `ProcessorInfo` and fill the respective variables. Then append the structure object to the `processorList` vector.

Listing 2: Structure of `IWaveletCoeffProcessor` interface.

```
class IWaveletCoeffProcessor{
public:
        virtual void process(float** in,
                             float** out,
                             int* coefLens,
                             int J)=0;

        virtual void setParams(float* params,
                               int paramLen) = 0;

        virtual void getParams(float* params,
                               int paramLen) = 0;
};
```

Listing 3: Demonstration of accessing wavelet coefficients

```
void DefaultProcessor::process(float** in,
                               float** out,
                               int* coefLens,
                               int J){
  // temporary vaiables
  float* inSubband;
  float* outSubband;
  int jTemp = 0;
  int coefLen = 0;

  for(int j=1;j<=J;j++){
     /* initiation of temporary variables for
        j-level detail coefficients */
     jTemp = j-1;
     coefLen = coefLens[jTemp];
     inSubband = in[jTemp];
     outSubband = out[jTemp];

     // iteration over j-level detail coeff.
     for(int i =0;i<coefLen;i++){
     /****PLACE FOR j-th level i-th DETAIL
      ****COEFFICIENT PROCESSING*****/
       outSubband[i] = inSubband[i];
     /***************END************/
     }
  }
  /* initiation of temporary variables for
     J-level approximation coefficients */
  jTemp = J;
  coefLen = coefLens[J];
  inSubband = in[J];
  outSubband = out[J];

  for(int i =0;i<coefLen;i++){
    /****PLACE FOR i-th APPROXIMATION****
      ****COEFFICIENT PROCESSING*****/
    outSubband[i] = inSubband[i];
    /*********END*******************/
  }
}
```

After compiling and running the plugin in a host application, your processor should be accessible by means of the `Process` slider, at a position corresponding to its index in the `processorList` vector. To demonstrate how to access the individual wavelet coefficients, we display `DefaultProcessor process` function implementation in Listing 3.

7. CONCLUSION

The paper presents the theory and VST plugin realization of segmented wavelet transform allowing performing audio effects in the wavelet domain in real-time, with the property of having no border artifacts. This is achieved by means of the (modified) SegDWT algorithm. In contrast to the original version, the extension lengths are reduced and the necessity of the right segment extension is removed, while preserving the features of SegDWT. The segment length restriction is alleviated from $s \geq r(J)$ to $s \geq 2^J$, which also means that the dependency on filter length m is dropped.

Since the SegDWT algorithm is derived to operate with any filters (just the lengths of their impulse responses affect the algorithm), the usage of the algorithm is not limited to the discrete wavelet transform only. It is also utilizable for other transforms following the iterated filter bank structure with down-/up-sampling, e.g. wavelet packets, framelets [29].

We encourage the interested reader to create their custom wavelet coefficient processing routine and kindly provide us with the feedback.

8. REFERENCES

[1] D. Onchis and C. Marta, "Multiple 1D data parallel wavelet transform," *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pp. 178–181, 2005.

[2] D. Chaver, M. Prieto, L. Pinuel, and F. Tirado, "Parallel wavelet transform for large scale image processing," *Parallel and Distributed Processing Symposium, International*, p. 6, 2002.

[3] W. van der Laan, A. Jalba, and J. Roerdink, "Accelerating wavelet lifting on graphics hardware using CUDA," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 132–146, 2011.

[4] D. Darlington, L. Daudet, and M. Sandler, "Digital audio effects in the wavelet domain," in *Proc. of the 5th Int. Conf. on Digital Audio Effects (DAFX-02)*, Hamburg, 2002.

[5] R. Xia, K. Meng, F. Qian, and Z.-L. Wang, "Online wavelet denoising via a moving window," *Acta Automatica Sinica*, vol. 33, no. 9, pp. 897 – 901, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1874102907600421

[6] Steinberg Media Technologies GmbH. (2012) Audio stream input/output SDK. [Online]. Available: http://www.steinberg.net/en/company/developer.html

[7] M. Tippach. (2003–2012) Asio4all — Universal ASIO Driver For WDM Audio. [Online]. Available: http://www.asio4all.com

[8] B. Leslie and M. Sandler, "A wavelet packet algorithm for 1D data with no block end effects," in *Circuits and Systems, Proceedings of the 1999 IEEE International Symposium on*, vol. 3, jul 1999, pp. 423–426.

[9] J. Nealand, A. Bradley, and M. Lech, "Overlap-save convolution applied to wavelet analysis," *IEEE Signal Processing Letters*, vol. 10, no. 2, pp. 47–49, 2003.

[10] W. Jiang and A. Ortega, "Lifting factorization-based discrete wavelet transform architecture design," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 11, no. 5, pp. 651–657, 2001. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=920194

[11] M. Unser and T. Blu, "Mathematical properties of the JPEG2000 wavelet filters," *IEEE Transactions on Image Processing*, vol. 12, pp. 1080–1090, 2003.

[12] D. S. Taubman and M. W. Marcellin, *JPEG2000: Image compression fundamentals, standards, and practice*. USA: Kluwer Academic Publishers, 2002.

[13] P. Rajmic, "Exploitation of the wavelet transform and mathematical statistics for separation signals and noise, (in czech)," Ph.D. dissertation, Brno University of Technology, Brno, 2004. [Online]. Available: http://www.utko.feec.vutbr.cz/~rajmic/phd_thesis/rajmic-dizertacni_prace-revize_23_10_2007.pdf

[14] P. Rajmic and J. Vlach, "Real-time audio processing via segmented wavelet transform," in *Proceedings of the 10th international conference on digital audio effects DAFx10*, 2007. [Online]. Available: http://dafx.labri.fr/main/papers/p055.pdf

[15] T. Wilmering. QM vamp plugins: Discrete wavelet transform. Queen Mary University of London. [Online]. Available: http://vamp-plugins.org/plugin-doc/qm-vamp-plugins.html#qm-dwt

[16] Z. Prusa and P. Rajmic, "Segmented computation of wavelet transform via lifting schemes," in *34th International Conference on Telecommunications and Signal Processing*, Budapest, 2011, pp. 433–437.

[17] Z. Prusa, P. Rajmic, and J. Maly, "Segmentwise computation of 2D forward discrete wavelet transform," in *33rd International Conference on Telecommunications and Signal Processing*, Dunakiliti, 2010, pp. 1–4.

[18] K. Gröchenig, *Foundations of time-frequency analysis*. Birkhäuser, 2001.

[19] R. G. Lyons, *Understanding digital signal processing*, 2nd ed. New Jersey (USA): Prentice Hall PTR, 2004.

[20] P. L. Søndergaard, "An efficient algorithm for the discrete Gabor transform using full length windows," *submitted*, 2012.

[21] P. Rajmic and Z. Prusa, "Discrete wavelet transform: Detailed study of the algorithm," *in preparation*.

[22] Steinberg Media Technologies GmbH. (2012) VST audio plug-ins SDK. [Online]. Available: http://www.steinberg.net/en/company/developer.html

[23] S. G. Mallat, "A theory for multiresolution signal decomposition: The wavelet representation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, pp. 674–693, 1989.

[24] G. Strang and T. Nguyen, *Wavelets and Filter Banks*. Wellesley College, 1996.

[25] P. Rajmic and J. Maly, "Boundary effects in the wavelet transform of finite discrete signals," in *Proceedings of the conference TSP'2007*, Electrical Engineering Society. Brno: Brno University of Technology, 2007, pp. 81–84.

[26] J. Schimmel. (2009) Vst plug-in module template. [Online]. Available: http://www.utko.feec.vutbr.cz/~schimmel/DAP/download/VST_template.zip

[27] P. Rajmic. SegDWT web page. [Online]. Available: http://www.utko.feec.vutbr.cz/~rajmic/segwt

[28] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[29] I. Daubechies, B. Han, A. Ron, and Z. Shen, "Framelets: MRA-based constructions of wavelet frames," *Applied and Computational Harmonic Analysis*, vol. 14, no. 1, pp. 1–46, 2003.