SpadePort: An open-source implementation of SParse Audio DEclipper (SPADE) in the C Programming Language

Konstantinos Melessanakis

Bachelor's Thesis
Department of Computer Science
University of Crete

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
UNIVERSITY OF CRETE

Advisor: Prof. Yannis Stylianou
Supervisors: Dr George P. Kafentzis,
Dr Nick Kossifidis

December 5, 2024

Contents

List of Figures

Abstract

Audio processing, while extensively researched, still presents challenges that captivate researchers and professionals alike. Among these challenges, declipping stands out due to its intrinsic complexity and the plethora of potential solutions it elicits. One of the notable solutions proposed is the SPADE algorithm. Proposed in 2015, SPADE has garnered attention for its innovative approach which capitalizes on the inherently sparse nature of audio signals. As it represents the cutting-edge in declipping algorithms, there is significant interest in exploring its scalability and accessibility across various platforms.

This work serves to assess the viability of a C based open-source, cross-platform implementation of a novel variant of the synthesis version of SPADE named S-SPADE. We present the original and novel variants of SPADE and outline the existing MATLAB implementation performance characteristics. Our findings indicate not only the strengths and limitations of both implementations but also underscore the significant performance gains achieved by our novel implementation. Notably, our results indicate that SpadePort can achieve audio restoration at a pace that is up to 3.4 times faster than its MATLAB counterpart while retaining the same Signal-to-Noise improvements.

With implementation in hand, we aim to pave the way for new advancements in the world of audio declipping, emphasizing the ever-evolving nature of this dynamic field.

Chapter 1

Introduction

Since the start of recorded audio, the phenomenon of clipping has plagued audio engineers and audio enjoyers across the world. What is clipping one may ask. The simple explanation is the distortion that occurs in a recorded signal when the amplitude of the signal exceeds the maximum level that can be accurately represented and accommodated by the recording equipment. Clipping can occur in both ends of the pipeline, both in recording and playback. The issue we come to phase is the process of declipping; Taking a " distorted " signal and running it through a specific and structured process to recover or deduce the parts that were lost due to clipping.

Before moving more into declipping it is worth noting that this phenomenon is regarded as one of the most common types of signal degradation and in more detail, it is usually caused by an element in the signal path whose dynamic range is insufficient in relation to the dynamics of the signal that is being recorded or reproduced. If we may imagine a signal as a sum of valleys and peaks, clipping causes these aforementioned peaks of the signal to be cut and set to the maximum value of the dynamic range that the recording medium allows, in turn giving the result of a distorted, crackling or even popping sound. Many reasons may cause clipping but in the scope of this project, we are going to focus on the process of de-clipping. Declipping is the process of restoring the "lost" parts of the audio signal which we are going to call frames. Specifically, we are going to deal with cases of hard clipping where we split the signal into three sets. Reliable, High, and Low. If a frame exceeds a pre-determined threshold it is assigned to the High set and vice versa for the Low set. That results in a mask set that can appropriately describe the instances of hard clipping in an audio signal. The issue now is, how do we deduce the "lost" information that is pointed to from these clipping instances?

In the past, many proposals have been made to solve this problem. The original researchers of the proposed algorithm that we are called to implement name a few in the original work, however in the scope of this project we are going to focus on SPADE and its proposed variant S-SPADE new.

This proposed variant is accompanied by a MATLAB implementation which even though very thorough and performant suffers from the drawbacks of using MATLAB as a platform. So, we decided to implement this algorithm in an open and free fashion without any heavy and proprietary software for academic reasons and to experiment with declipping methods in the future.

1.1 Review of Related Work

The SPADE algorithm was originally developed by PANAMA and it is based on the expression [5] of declipping a linear inverse problem and the use of analysis and synthesis sparse regularization in the time-frequency domain. This approach was originally proposed by Kitic et al. in 2015 [3] and revisited in 2018 [8] where its performance in terms or restoration and convergence speed was tested and commented. However, in 2020 [9] the S-SPADE variant in question for our project was proposed that deems it the state of the art for the SPADE family of algorithms in terms of reconstruction quality and speed.

Other declipping algorithms are surveyed in the article in [6] with SPADE being a top performer among Social Sparsity with Empirical Wiener, Parabola-Weighted $\ell 1$ -minimization using Chambolle–Pock (analysis) and Parabola-Weighted $\ell 1$ -minimization using Douglas–Rachford (synthesis).

In terms of implementations all the algorithms surveyed in [6] are accompanied by a MATLAB toolbox [7] that contains implementations for said algorithms (including the base for our work porting the new S-SPADE variant). This toolbox is provided by Pavel Rajmic which is one of the original authors of the papers that initially proposed SPADE and its newest variants, so for the scope of our work we regard these implementations as "first-party" In addition the user andyr on Github [1] is providing an independent implementation of the analysis and synthesis variants of SPADE.

As far as we have surveyed there does not seem to be an open source implementation of the SPADE

algorithm available in a general use programming language. In addition, our current understanding is incomplete regarding any commercial implementations of the SPADE algorithm or well-documented algorithmic variations.

1.2 Motivation and Intent

Sound is integral to us human beings, we use it for communication, entertainment, expression, and in many other key ways to everyday life. Especially nowadays recorded sound is all around us. However, not all audio was recorded equally and since clipping is one of the most common reasons behind distortion we felt like an open and free implementation of a useful algorithm would be very helpful to many individuals.

One of these reasons is recovery, restoration, and preservation of audio from the past. Audio engineers these days have very sophisticated tools and recording equipment in their arsenal as well as having mediums that allow a high dynamic range to be recorded. However, before the advent of digital technology that was not always the case. With mediums of the past such as vinyl and tape, preserving the fidelity of an audio signal was much more demanding and required meticulous care due to the sensitive nature of these mediums. Also replicating a recording was not as easy as it is today.

Copying a tape or a record with no error was virtually impossible due to the analog nature of these mediums. Everyone knows the characteristic hiss of a tape or the cracks and pops of a not so well-preserved vinyl record. These are all cases of heavy clipping that deduct from the track and many times render it of much lower quality in regards to the original and in turn losing information/data.

With this project, we also wanted to provide a version of this algorithm that was free from any proprietary software and open to the world to experiment with and use. By going with the C programming language we can provide software that is open and able to be used without needing to own expensive planforms such as MATLAB. Also, by using a programming language that is so close to the silicon we can figure out new ways to optimize for performance and ways to escape the performance overhead imposed by MATLAB as well as support a multitude of potential platforms.

Chapter 2

The SPADE Algorithm and its Variants

SPADE was first proposed by Kitic et.al. in 2013 [2] and is described as a novel sparsity-based heuristic algorithm for non-convex regularization of the clipping inverse problem. The SParce Audio DEclipper is further derived into two variants. The synthesis and analysis variants serve to minimize the following NP-hard non-convex and non-smooth problems.

$$\min_{x,z} \|z\|_0 \ s.t.\ x \in \Gamma(y)\ and\ \|Ax - z\|_2 \leq \epsilon, \tag{2.1a}$$

$$\min_{x,z} \|z\|_0 \ s.t.\ x \in \Gamma(y)\ and\ \|x - Dz\|_2 \leq \epsilon \tag{2.1b}$$

As we mentioned in the introduction, SPADE starts by splitting the audio signal frames into Reliable, High and Low samples. Then to select only these samples from the specific set, the restriction operators of MR, MH, ML come into play. Given the information that for a specific variable ϑc the Mh restriction operator (or how we are going to call it for the scope of this project Mask) contains the samples of which the value is greater than ϑc and vice versa, the Mask Ml contains the samples that are bellow -ϑc. Finally, the Mask Mr contains all frames that are not clipped (are not included in either Mh, Ml masks). These masks can also be described as linear projectors or identity matrices with their specific columns removed that correspond to the High/Low and Reliable cases.

It is worth pointing out that for the scope of this SPADE implementation, the algorithm runs with a predetermined ϑc value for which any value above or below will be clamped to that specific value in turn describing the signal by the following formula.

$$y[n] = \begin{cases} x[n] & \text{for } |x[n]| < \theta_c \\ \theta_c \cdot sgn(x[n]) & \text{for } |x[n]| \geq \theta_c \end{cases} \tag{2.2}$$

With the given information we can describe Γ from equations 2.1 as the set of feasible solutions of time-domain signals.

$$\Gamma = \Gamma(y) = \{\hat{x} \mid M_R\hat{x} = M_R y, m_H\hat{x} \ \geq \theta_c, M_L\hat{x} \leq -\theta_c\} \tag{2.3}$$

Here $\hat{x}$ represents the restored signal and it is required to be a member of the set. The heuristic nature of the algorithm plays into the fact that finding the restored signal is a problem with endless possible solutions that satisfy the requirements for a successful restoration of the lost samples. Moving forward we are going to describe the analysis variant of the SPADE algorithm, A-SPADE, the synthesis variant S-SPADE, and take note of the changes to the original algorithm the authors are proposing giving us SPADE-new or SPADE-DoneProperly as they wish to call it. For both variants the problems in 2.1 can be recast as follows:

$$\arg\min_{x,z,k} \iota_\Gamma(x) + \iota_{\ell 0 \leq k}(z)\ s.t \begin{cases} \|Ax - z\|_2 \leq \epsilon, \\ \|x - Dz\|_2 \leq \epsilon, \end{cases} \tag{2.4}$$

Where $\iota_\Gamma(x)$ ensures that the reconstructed signal resides within the set Γ while $\iota_{\ell 0 \leq k}(z)$ is a concise representation for $\iota_{\{\tilde{z} \mid \|\tilde{z}\|_0 \leq k\}}(z)$ which imposes the sparsity on the coefficients.

The original authors chose to derive the problem into a set of three steps using the Alternating Direction Method of Multipliers which is further elaborated in [10]. ASPADE can in turn be represented in these three

ADMM steps:

$$x^{(i+1)} = \arg\min_{x} \left\| Ax - z^{(i)} + u^{(i)} \right\|_2^2 \; s.t \; x \in \Gamma, \tag{2.5a}$$

$$z^{(i+1)} = \arg\min_{z} \left\| Ax^{(i+1)} - z + u^{(i)} \right\|_2^2 \; s.t \; \|z\|_0 \leq k, \tag{2.5b}$$

$$u^{(i+1)} = u^{(i)} + Ax^{(i+1)} - z^{(i+1)} \tag{2.5c}$$

Which then translates to the following pseudo code.

Algorithm 1 A-SPADE from [4]

Require: $A, y, M_R, M_H, M_L, s, r, \epsilon$
1: $\hat{x}^{(0)} = y, u^{(0)} = 0, i = 0, k = s$
2: $\bar{z}^{(i+1)} = \mathcal{H}_k \left(A\hat{x}^{(i)} + u^{(i)} \right)$
3: $\hat{x}^{(i+1)} = \arg\min_x \left\| Ax - \bar{z}^{(i+1)} + u^{(i)} \right\|_2^2 \; s.t \; x \in \Gamma$
4: **if** $\left\| Ax^{(i+1)} - \bar{z}^{(i+1)} + u^{(i)} \right\|_2 \leq \epsilon$ **then**
5: terminate
6: **else**
7: $u^{(i+1)} = u^{(i)} + A\hat{x}^{(i+1)} - \bar{z}^{i+1}$
8: $i \leftarrow i + 1$
9: **if** $i \mod r = 0$ **then**
10: $k \leftarrow k + s$
11: **end if**
12: go to 2
13: **end if**
14: **return** $\hat{x} = \hat{x}^{(i+1)}$

In the above algorithm, it is poignant to define the operator $\mathcal{H}_k$ which performs hard thresholding to the complex coefficients that are retrieved by the DFT used by all SPADE versions. This operation sets all but the k largest magnitude values of its input array to zero. For all variants, the value of k is not predetermined but set at a low point and in each iteration, its periodically increased. This corresponds to the sparsity relaxation which means that after each iteration k gets larger and the estimated recovered signal z becomes less sparse. The hard thresholding step therefore serves as the sparsity relaxation step. The step and step rate are controlled by the $r > 0, s > 0$ parameters and finally, the parameter $\epsilon > 0$ which is the max iteration threshold.

Moving from the analysis variant we are in turn going to briefly outline the synthesis version of SPADE as well as the modifications proposed by the researchers of the original work. Taking the synthesis part of 7 we can alter it to the following:

$$\arg\min_{x,z,k} \iota_\Gamma(x) + \iota_{\ell 0 \leq k}(z) \; s.t \; Dz - x = 0 \tag{2.6}$$

This can then be transformed to the following ADMM steps:

$$z^{(i+1)} = \arg\min_{z} \left\| Dz - x^{(i)} + u^{(i)} \right\|_2^2 \; s.t \; \|z\|_0 \leq k \tag{2.7a}$$

$$x^{(i+1)} = \arg\min_{x} \left\| Dz^{(i+1)} - x + u^{(i)} \right\|_2^2 \; s.t \; x \in \Gamma \tag{2.7b}$$

$$u^{(i+1)} = u^{(i)} + Dz^{(i+1)} - x^{(i+1)} \tag{2.7c}$$

Which in turn translates into the following algorithm:

Algorithm 2 S-SPADE from [4]

Require: $D, y, M_R, M_H, M_L, s, r, \epsilon$
1: $\hat{z}^{(0)} = D^* y, u^{(0)} = 0, i = 0, k = s$
2: $\bar{z}^{(i+1)} = \mathcal{H}_k\left(\hat{z}^{(i)} + u^{(i)}\right)$
3: $\hat{z}^{(i+1)} = \arg\min_z \left\|z - \bar{z}^{(i+1)} + u^{(i)}\right\|_2^2 \; s.t \; Dz \in \Gamma$
4: **if** $\left\|\hat{z}^{(i+1)} - \bar{z}^{(i+1)}\right\|_2 \leq \epsilon$ **then**
5: terminate
6: **else**
7: $u^{(i+1)} = u^{(i)} + \hat{z}^{(i+1)} - \bar{z}^{i+1}$
8: $i \leftarrow i + 1$
9: **if** $i \mod r = 0$ **then**
10: $k \leftarrow k + s$
11: **end if**
12: go to 2
13: **end if**
14: **return** $\hat{x} = D\hat{z}^{(i+1)}$

The researchers of the current implementation however propose the following algorithm with changes that iterate on both analysis and synthesis variants of the SPADE algorithm:

Algorithm 3 S-SPADE$_{DP}$ proposed in [9]

Require: $D, y, M_R, M_H, M_L, s, r, \epsilon$
1: $\hat{x}^{(0)} = y, u^{(0)} = 0, i = 0, k = s$
2: $\bar{z}^{(i+1)} = \mathcal{H}_k\left(D^*(\hat{x}^{(i)} - u^{(i)})\right)$
3: $\hat{x}^{(i+1)} = \arg\min_x \left\|D\bar{z}^{(i+1)} - x + u^{(i)}\right\|_2^2 \; s.t \; x \in \Gamma$
4: **if** $\left\|D\bar{z}^{(i+1)} - \hat{x}^{(i+1)}\right\|_2 \leq \epsilon$ **then**
5: terminate
6: **else**
7: $u^{(i+1)} = u^{(i)} + D\hat{z}^{(i+1)} - \hat{x}^{i+1}$
8: $i \leftarrow i + 1$
9: **if** $i \mod r = 0$ **then**
10: $k \leftarrow k + s$
11: **end if**
12: go to 2
13: **end if**
14: **return** $\hat{x} = \hat{x}^{(i+1)}$

In the case of step 2.7b projection, they have chosen a significantly simpler element-wise mapping approach, akin to A-SPADE, as opposed to the original S-SPADE method, which required the utilization of a more intricate projection lemma. Also, the researchers note that the solution to the S-SPADEdp algorithm is achieved by the hard thresholding operation. However, due to the non-orthogonality of matrix D, the resulting vector serves as an approximate solution to 2.7a

2.1 The MATLAB S-SPADE Implementation

At this point in our work, we are going to go through the code provided by the original researchers that serves as a proof of concept implementation for their new SPADE algorithm. This implementation is designed to run on MATLAB and for what this current work is concerned, the code was tested with R2022a running on M1 Pro CPU. The main dependency of this implementation is the Large Time/Frequency Analysis Toolbox (LTFAT) for working with time-frequency analysis and synthesis.

Initially, the declipping code starts off by loading one of the sample audio files provided. The frames of the audio file are then normalized if the maximum absolute value of this sample array is less than 0.95. Every implementation [4] [10] uses an oversampled DFT for the signal transformation. A 1024 sample long Hann window is initialized with a window shift of `w/4`. After moving from line 44 the researchers use an LTFAT frame object that contains all necessary information for all the analysis and synthesis processes that are required in the main SPADE loop.

The synthesis and analysis processes are slightly modified to take into account the complex conjugates of each iteration chunk. The size of each chunk is controlled by the redundancy parameter set on line 45 which

directly influences the input array transformation initiated in frana. The `s` and `r` parameters are initialized to the value of 1, `s` is the increment of `k` that gives the sparsity relaxation to the hard thresholding part of SPADE described before and `r` is the number of iterations that need to be elapsed before incrementing the `k` parameter by `s`. Lastly, the max iteration number is calculated which is given by the function of the following:

```
ceil(floor(param.w*param.F.redundancy/2+1)*paramsolver.r/paramsolver.s)
```

The next step for the algorithm is to run through the audio sample array and split it into the three masks that we outlined previously (explanation part). We reiterate that these three masks are `Mh`, `Ml`, `Mr` for the High samples, Low samples, and Reliable ones. The clipping threshold that allows the audio samples to be split into these three masks is variable `theta` which is set at 0.1 initially but can be changed depending on the input audio. It is worth noting that SPADE does not take into account variable clipping thresholds but instead chooses to have a specific threshold that dictates the hard clipping discovery.

In the hard clipping method in addition to the clipping discovery, the signal is artificially hard-clipped to the theta value that is set to simulate a hard-clipped input signal. This of course does not fully match an observation one would make in a non-artificial clipping occurrence where the said instances could occur in various amplitudes in the total time of the signal.

Once the clipped signal array with all the required parameters is set, the next step is to go into the SPADE segmentation function which depending on the arguments provided can run the three different variants of the algorithm we outlined previously. For the scope of this work, we are going to focus on the `spade_new` variant. The last of the requirements before initiating the recovery loop is to set the required windows and to calculate the length of the final intermediate output. The length of the intermediate recovery array is a result of the following:

```
ceil(param.Ls/param.a)*param.a+(ceil(param.w/param.a)-1)*param.a
```

So as to allow it to be divisible by the window shift value `a` and a minimum amount of zeros that equal the window length `w`. The appended samples to the initial length of the audio signal will be zeroed out so as to avoid the periodization of nonzero samples.

Here, a Hann window is used based on the parameters that were set previously. The window is then peak normalized to give us the analysis window array. We also need the synthesis window and for that, the `gabdual` function is used which provides the canonical dual Hann window which is then multiplied by the window length. Both functions are part of the LTFAT toolbox.

Following the parameter setup and the helper array creation we come to the SPADE main outer loop. The input signal is split into N chunks, where N is the length of the extended recovery array divided by the window shift value. Then the index arrays need to be built which will facilitate the chunk retrieval during the outer loop of the algorithm. This takes the place of an `fftshift` function. For each outer iteration of said loop an index array is built that allows the selection of audio frames from the two input arrays. The one we are interested in is `data_block`, whose elements are taken from the `data_clipped` array that contain the hard-clipped audio samples. Each sample is multiplied with the corresponding analysis Hann window array element which in turn results in the main input array for the main SPADE algorithm loop.

Here lies the heart of the SPADE algorithm. `spade_new` starts off initializing all the variables with the parameters of the SPADE algorithm that we set up previously as well as initializing the thresholds and limits. It starts off executing the first DFT which is the `frana` immediate function we described previously. The input of this DFT is the current $\hat{x}$ which for the first iteration is equal to the input signal minus the `u` array which starts off as a zero-element array. Then internally the result of the DFT is normalized with the square root of the size of the transform and the resulting array is fed into the hard thresholding function. This operation is responsible for taking the array of coefficients that were produced from the DFT while taking into consideration the conjugate pairs, and then performing the following manipulation. It initially splits the array, divides the first coefficient by 2, and sorts the coefficients from high to low.

Then for input of `k` that was given to the function it restores k largest coefficients in place whereas the rest are zeroed out. Then for the final step of this function, it amplifies the first coefficient restoring the previous division, and computes the conjugates for the current array. The resulting array is the combination of the original array with the flipped conjugate array.

With this data now residing in the `zbar` variable an inverse DFT is run by the immediate function that resides in `frsyn` where the size of the inverse DFT is now restored to the chunk size again using the redundancy parameter. With its resulting array `Dz_bar`. We then calculate the L2 norm of the `Dz_bar` minus the `x_hat` where `x_hat` serves as the intermediate reconstructed sample array. The resulting value serves as the intermediate quality indicator of the reconstruction. A value that is saved and compared against the `e` parameter that was set during the initialization phase of the application. If that value is less or equal to `e`, SPADE terminates and the recovered chunk is `x_hat`. If not, a projection is performed which in technical terms means that for the recovered chunk $\hat{x}$, for each of the high hard clip frames indicated by the input mask `Mh`, each sample is set to

the largest value between the input clipped array and the sum of `Dz_bar` and `u`. The same happens for the low hard clip frames where $\hat{x}$ is set to the least of these two values. Lastly, for the reliable frames $\hat{x}$ is set to the respective clipped frame from the input.

Next is to update the `u` variable to the difference between the synthesized `Dz_bar` frames and the intermediate recovered sample array `x_hat`. Then sparsity is relaxed by incrementing parameter `k` by a step size of `s` which in turn in the next iteration will allow for more coefficients to be retained in the `hard_thresholding` function thus reducing sparsity. The loop runs until either a satisfactory recovery quality is reached or the main loop's max iterations are reached.

The result of `sspade_new` is then returned to the outer loop and the recovered sample array is rearranged using `ifftshift` and appended to the total recovered sample array by first multiplying each element with the synthesis window that was generated previously. By the end of the chunk loop `data_rec_fin` will contain the entirety of the frames that resulted from the synthesis SPADE operation.

The MATLAB implementation of the original work will then use the original samples for the audio signal to calculate the following Signal-to-Distortion Ratio (SDR) value:

$$\mathrm{SDR}(u, v) = 10 \log \frac{\|u\|_2^2}{\|u - v\|_2^2} \qquad [dB] \tag{2.8}$$

which will serve as a benchmark for the current work. All in all, the MATLAB implementation is an adequate representation of what is possible with the proposed algorithm.

However, such implementation could not be widely used due to some important factors. One of which is the necessity for a MATLAB license, something that may deter most people due to its cost. Also, MATLAB is a specialized tool, created for certain applications in science. It is not a general-use platform and does not allow for low level handling of data or optimizations, therefore limiting the control over the performance and expandability of the application. Further, the learning curve for MATLAB as a day-to-day tool is quite high even for a novice user given the setup that is needed prior to doing a run of the algorithm.

Chapter 3

SPADE-Port

For the open source implementation of the SPADE algorithm, we chose C. Everyone knows the advantages (and disadvantages) of the C ecosystem. It is a low level general purpose language that provides many avenues to performant and cross platform applications. Unfortunately, the C ecosystem is not so user friendly from the developer's point of view due to its lack of abstractions modern programming languages like Go, Rust, or Zig provide. Comparing the built in signal analysis features of MATLAB to the C ecosystem is not applicable. MATLAB provides many functions and ready to use features (with its toolboxes) that are baked into the platform (hence the cost). For the C implementation, we will have to resort to external libraries that would facilitate the same functionality.

Starting off, to be able to parse audio files into the application we went with libsndfile. Libsndfile, also known as the Sound File Library, is an open-source software library designed for reading and writing various audio file formats. Its cross platform nature, the multitude of audio formats supported, and easy sample data handling made it an easy choice for us. Using it we could direct the application to a WAV file and with a simple read procedure have the same array of audio samples that we were getting with MATLAB.

Secondly, we needed a way to integrate the LTFAT toolbox into our C application. Thankfully the team behind LTFAT also provides an open source backend library that is responsible for generating the analysis and synthesis windows needed. Unfortunately this library is not actively maintained and during the analysis phase of our work, we run into many issues with building the specific binaries to generate the libraries. One point of many issues is that libltfat was built around the linux ecosystem and our target architecture was Apple's new arm64 processors. We had to resort to editing our makefiles in order to allow the package to be compiled and to be used by our application.

Even with the libraries built, using said library also proved a challenge. Most of the functions are loosely documented so we had to resort to digging into the function definitions and example applications, in order to make sense of the way the authors designed this library to be used. Some tests later and after understanding the purpose of the functions that were needed, we were ready to integrate them into our implementation. Even though we only use the window generation and fftshift procedures its integration was really integral to having reliable and known good (from their MATLAB counterparts) helper functions.

Last but not least there's FFTW. The core of the SPADE algorithm are the double ffts performed in the main reconstruction loop. To take care of those we chose FFTW which is an open source library for computing discrete Fourier transforms and related operations. It is cross platform and the go-to choice for a wide range of scientific and engineering applications that involve digital signal processing. FFTW is known for its efficiency, speed, and performance due to it employing a sophisticated range of algorithms and optimizations to compute DFTs quickly and efficiently. It is also very accurate, adapting, and as we will see in the continuation very easy to use due to its very well-written documentation and straightforward operation. Also, it is worth noting that FFTW can also used by MATLAB when generating standalone applications as it serves as its DFT backend.

Our open source implementation is currently targeting macOS Sonoma 14.0 on arm64 and Ubuntu 22.04.3 LTS x86. Our implementation starts off the same way as the MATLAB one initializes. For each of the parameter sets, we will need we defined structs that will serve as containers for the needed parameters. We have the SPADE_FRAME struct which contains the frame type and the redundancy values. The SPADE_PARAMETERS that contains signal length, symmetric clipping threshold, the window length, shift and type as well as an instance of the SPADE_FRAME struct. Further, we have defined the PARAMSOLVER_PARAMETERS struct which contains the s and r parameters that are the k increment and the sparsity relaxation value, the epipsilon and maxit parameters that govern the stopping criteria and the maximum iterations allowed in the main reconstruction loop respectively.

One point of care that had to be taken across the whole implementation and especially during initializations is the main difference between MATLAB and the majority of programming languages which is 1-based indexing. This characteristic is seen often in languages that are mathematics and statistics oriented. When attempting

to port from a 1-based indexed to a 0-based index language especially when accuracy is of at most importance; index, minimum, maximum, and vector splitting and joining operations need extra analysis in order to fit the logic properly and make sure that edge cases and index operations are not misrepresented. An example in our implementation that we are going to see is the C implementation of the hard thresholding function where indexing and appending the intermittent arrays becomes a more complex operation than it's MATLAB counterpart.

Post initializations spadeport reads the requested file and builds an array of double precision floating point real numbers with the samples. Libsndfile when opening the sound file, produces a helpful container of metadata such as the count of frames, sample rate, channel number, etc. We are using this data in order to build an identical file once the spade algorithm has run. The next step at this point is to perform hard clipping on the input file. The input sample array as well as the output array that was allocated and the mask container are passed onto the function where the input array is iterated. Mimicking as the MATLAB implementation, for each of the input frames for each value that's higher or lower than tMin and tMax, the clipping mask data is updated with a 1 value if a hard clip was detected based on the threshold. Additionally for the result array which will be the clipped signal array if a high clip was detected the resulting value is tMax and vice versa for a low clip. The result of this function is going to be the input for the spade algorithm. Again, we reiterate that this is not a real-world scenario where there is a strict threshold for clipping.

Afterwards, we move into the spadeSegmentation function which will serve as our starting point for the SPADE algorithm. This function may take as input, both the clipped and the original sample array, as well as all the needed parameter containers. So far, we have tried to stay as close as possible to the original work with minimal alterations to the process of the original application, however, for the coming parts many of the technicalities will differ from the MATLAB implementation.

One of the main differences between the MATLAB application and ours is that for MATLAB everything is automatic. Memory allocation, array appending, finding maximum and minimum values in an array, initiating a dft, etc are all done without the programmer being in control of the inner workings of the code. That introduces a big divergence with our application where because we chose a low level language like C we have to manage everything ourselves. Especially for digital signal processing the process of arranging data, chunking, and running simple operations like we mentioned before can become complex especially when we're trying to produce a performant application. Even if we lose some of the creature comforts of a platform like MATLAB, we achieve higher throughput with a minimal memory footprint, both of which we will look into detail in the performance analysis of both implementations.

So, to start off the main spade recovery loop has to initialize of the arrays that are going to be needed. We tried to aggregate all needed arrays (which include output and intermediate arrays) so as to allow the compiler to be able to optimize the compiled executable to a higher degree. Initially the x_hat array is populated with the input signal chunk frames and for the first iteration the u array is zeroed out. The next step is to convert the sample data from a double precision floating point number to a complex taking into account the additional 0 complex values that are introduced from the essential doubling of the chunk data in line 72. Once the data is properly stored in arrays that can be used with FFTW we run the first forward DFT to compute the coefficients of given samples. One of the points of improvement that FFTW allows is the use of SIMD instructions. For this instruction set to be used, the array has to be 16-byte aligned in memory, something that malloc cannot guarantee. So, to use that feature we choose to allocate all arrays that will play a part in the FFTs with the provided FFTW functions, which will allow us to take advantage of the non negligible performance increase we can leverage for these complex operations.

Before moving forward with the data we obtain from the FFT, we need to normalize it first. This is built in by LTFAT in the MATLAB application, however, with ours's we need to iterate over the array and multiply it with the normalizing factor which is 1/N where N is the square root of the (segment) FFT's size we ran before. We want to comment that this process of inverse square root is a known point that can be enhanced and optimized for future improvements.

The next step is to feed the normalized coefficients array into the hard thresholding procedure. This procedure takes as input the aforementioned coefficients, the size of the transform, the sparsity relaxation parameter k as well as the output array zBar. As mentioned before, this function will set all but k coefficients to zero with k incrementing by step s in each iteration and therefore relaxing the sparsity aspect of the algorithm.

Practically in our code the function initially splits the array and halves the first coefficient. Then a sorted index array is generated. To do so the coefficients array needs to be sorted. Unlike the MATLAB operation, with our implementation, we are able to choose which algorithm to follow for the sorting process. We went with the POSIX quick sort which provides satisfactory performance and allows us to provide a custom comparator which we use in order to generate the sorted index array.

Continuing, the function for the intermediate zero array s replaces the k largest coefficients from the sorted original array and doubles the first dominant coefficient. To finish up the hard thresholding process, a flipped conjugate array is appended to the calculated resulting array and is in turn returned back to the main recovery loop.

The resulting array then is used as input for the inverse DFT operation done by the complex 2 real (c2r) plan of FFTW. For this operation the c2r plan is used because we are interested in the real part of the inverse DFT and this function gives us directly the results we are expecting. Afterwards, the DFT normalization is reversed in order to recover the computed recovered sample data which are now contained in DzBarFulll. This is done by dividing each element with the segment size and multiplying by the square root of said segment size. From this array, we extract the samples that correspond to the original chunk elements and then we compute the difference between that computed array and xhat which for the first iteration is equal to the input array y.

Next, we evaluate the l2 norm of the resulting array. The resulting value is deemed as the quality of the recovered chunk. If that value is less than the preset parameter epsilon the recovery stops and the return reconstructed sample array is xhat which is copied onto dataRec. If not, the elements of the DzBarInputDiff are projected onto the set of feasible solutions for the declipping problem in the time domain for the xhat array. That means that for each value of this array, if on a certain index, the sample was a reliable frame, its replaced with the corresponding clipped input array. If that frame was a high clip frame, its replaced with the max between the input clipped array and the sum of the u and DzBar array and vice versa for the low clip frames.

Finally, we have the update of the dual variable array u which is the result of the addition between its elements and the difference of the DzBar array and xhat where Dzbar contains the recovered frames for the current iteration and xhat contains the running set of reconstructed frames for this chunk. The last step in the process is to update the k parameter that informs the sparsity relaxation by k if the current iteration count is divided by r making essentially its step rate.

Given that the algorithm was successful and a sufficiently reconstructed set of frames was obtained with a total iteration count lower than the maxit parameter, the final set of frames is now residing in the dataRecBlock array. This array is then ifft shifted which rearranges a zero-frequency-shifted Fourier transform Y back to the original transform output that was achieved by the index generation performed before where the left and right half of the windows were swapped. Afterwards, each element of this final array is copied with the synthesis window array and its result is saved in dataRecFin. There lies the final reconstructed total set of frames for the audio file.

Before exiting, the application caries over the results to the export array whose contents are written to an audio file set up the same way as the input file and calculates the SDR ratio of the recovered signal against the hard clipped one allowing us to measure and tune the SPADE algorithm.

3.1 Issues and comments regarding the C open source implementation

Coming from a programmer's point of view the issues we faced during this project were numerous. The fact that we wanted to keep this project as a straightforward port limited by a lot the ways we could accomplish implementing the SPADE algorithm. One of such issues was the use of the same libraries if possible. We had to find a way to integrate libraries that made our application function as similar as possible to the work of the original authors.

Having taken that into consideration we ended up using libsndfile for the audio file i/o that enabled us to have code that first, was as simple as possible to understand from a third party with helpful documentation for future expandability and secondly to mimic the simple way that MATLAB handles files, which is a mostly "hands off" approach. That said for our implementation the support is limited to single channel PCM Signed 16-bit little-endian 16khz files.

Then came the point of choosing a library to handle the DFT transforms that were required in the main reconstruction loop of the SPADE algorithm. Due to the high number of libraries available nowadays (FFTW, kissFFT, MKL, vDSP) we had to be careful to choose one that would suit our needs in terms of performance, documentation, ease of integration, and platform compatibility and of course open to the public. However, when researching the way MATLAB does its FFTs we came to the realization that the basis for MATLAB's own FFT functions is FFTW.

That seemed the most appropriate way to get as close as possible to the accuracy provided by the work of the original implementation. Additionally, FFTW is renowned for its exceptional performance, and flexibility and is regarded as one of the fastest and most accurate open-source libraries in comparison to other proprietary or commercial ones with great documentation and functionality that would allow future tuning and expansion the current implementation.

The last of the needed libraries for our implementation was the one that was most heavily used by the original authors, LTFAT. The Large Time-Frequency Analysis Toolbox is an open-source toolbox for MATLAB that provides a comprehensive set of tools for time-frequency analysis and signal processing. In the original work, LTFAT was used heavily for its FFT wrappers and the analysis and synthesis window generation. On our side, we had to modify an unmaintained distribution of LTFAT's backend library in order to get access to the functions that would allow us to generate the same window arrays as the original work. The library however

had lacking, outdated documentation and platform compatibility issues. That sparked a lot of experimentation with the build system of the library as well as its functionality. In the end however, we managed to retain the original authors' vision and we managed to integrate the needed parts of LTFAT into our own application.

Moving away from the external requirements that arose for our implementation then came the point of moving inherent MATLAB functionality to a programming language that is not designed for mathematical or signal processing purposes but is more general purpose in nature. Many of the vector operations in MATLAB that are simple one-line commands need a more extended representation in C. Additionally the fact we mentioned before that is that MATLAB is a 1-indexed language whereas C is a 0-indexed one, complicated these vector operations even more. Extra care had to be taken to not skew the purpose of these operations. In addition, in C memory management is not done automatically, so to provide the same functionality at a similar speed we had to be wise in the way we were managing sample data and the FFT operations required by SPADE. Even if from a development standpoint this is an added point of care, we managed to minimize the footprint of the application and increase the speed at which we can achieve almost identical reconstruction.

Commenting on the entirety of our work, we could state that in its current form SpadePort results in reconstructions that are up to par with the original vision of the authors. However, we have managed to improve the speed of these reconstructions by 3.4 times, while simultaneously providing an application that can be built to run on any sufficiently powered system with a minimal memory footprint and with many avenues for future expansion due to its open source and cross-platform nature.

Chapter 4

Experiments

In this chapter, we will outline the methodology adopted to assess the performance and accuracy of our proposed SPADE implementation. Our experiments have been structured to provide insight into the comparative advantages of an open source implementation over the MATLAB one proposed by the original authors. The subsequent subsections delve into the specifics of our dataset, setup, and metrics.

Our evaluation employs the same dataset used by the original authors. This consists of five diverse samples that are PCM signed 16khz single channel files. They represent a broad spectrum of signals that range from spoken voice, instruments and music recordings with a duration between 5 and 9 seconds.

Both implementations were run on the same computer which is an Apple MacBook 14" with an arm64 Apple M1 Pro 8 core CPU. The operating system used was macOS Sonoma 14.0 and the software stack for the two implementations was the following. For the MATLAB implementation the version used was R2022a with version 2.5.0 of the LTFAT toolbox. For the C implementation the source code was compiled using Apple clang version 15.0.0 targeting the arm64 architecture.

The metric we used to deduce the quality of signal reconstruction was the SDR 2.8 delta between the clipped and the reconstructed signal. Throughout our research [6] this seems as one of the most prominent metrics when it comes to declipping algorithms. To evaluate the two implementations, we did five runs for each of the five audio samples. In the preprocessing hard clipping step, we used multiple clipping thresholds $\theta_c \in \{0.1, 0.3, 0.6, 0.9\}$ in order to gauge recovery performance in different clipping levels.

Upon closely analyzing the performance of the MATLAB implementation, it becomes evident that the performance of the algorithm is intricately tied to the chosen clipping threshold. Interestingly, the runtimes appear to converge towards a stable point when the clipping threshold is set at 0.3. The nature of the audio signal also plays a pivotal role in determining the speed of the algorithm's reconstruction process.

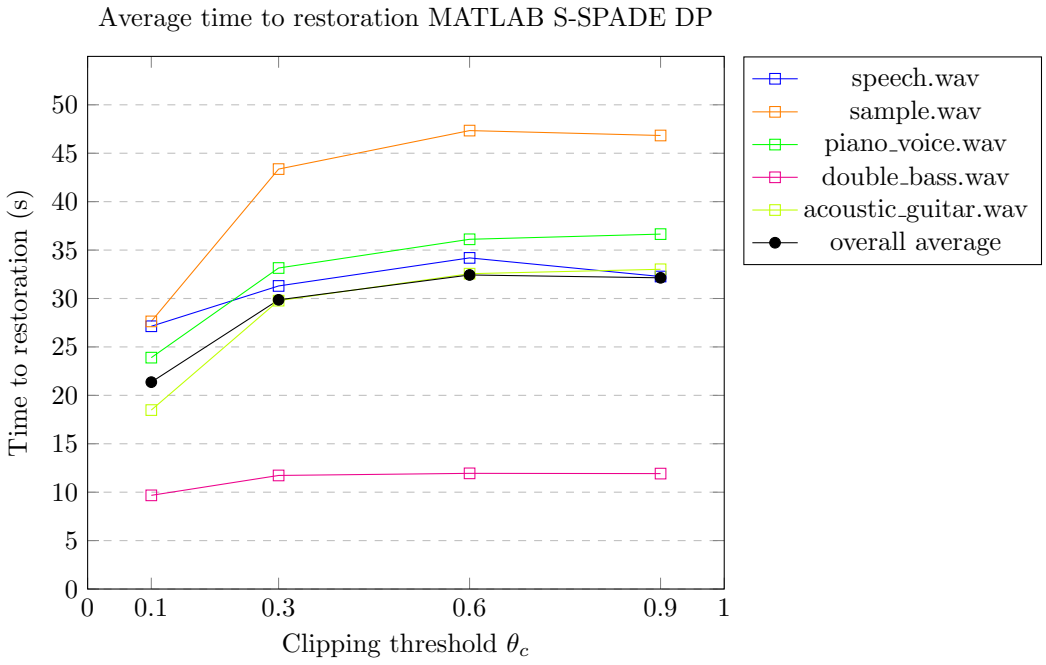


Figure 4.1: MATLAB S-SPADE performance

For instance, complex audio files like sample.wav, which incorporate musical elements, tend to require a longer duration for reconstruction. In contrast, simpler audio files, such as double_bass.wav, demonstrate a swifter and more consistent reconstruction speed across different clipping thresholds. Taking an average from our five test runs for each audio sample, the overall average runtime for the S-SPADE algorithm was found to be approximately 28.948 seconds. Given that the average duration of our tested audio samples was six seconds, this translates to an estimated average reconstruction time of about 4.8 seconds per second of audio.

Transitioning to our open-source SpadePort implementation, a considerably more homogenized performance profile emerges. Contrary to the MATLAB-based counterpart, the SpadePort implementation exhibits reduced sensitivity to variations in the clipping threshold. Notably, after stabilizing around the 0.3 clipping threshold, the runtime displays consistent and predictable behavior. Further, the nature of the audio sample seems to exert a substantially diminished influence on this implementation compared to its MATLAB analogue Employing a methodology consistent with the previous experiment with five iterations for each audio sample, the cumulative average runtime for SpadePort is approximately 8,516 seconds. This equates to an average processing time of roughly 1.4 seconds for every second of audio content.

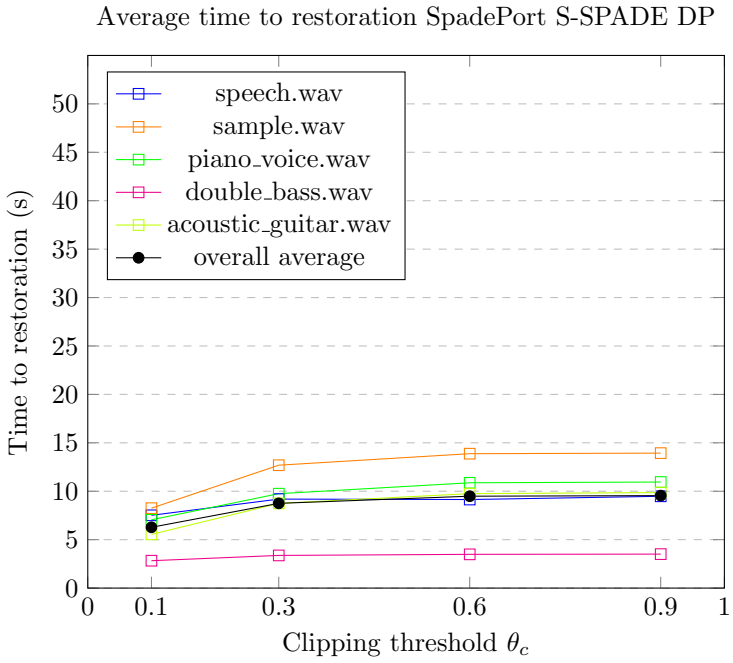
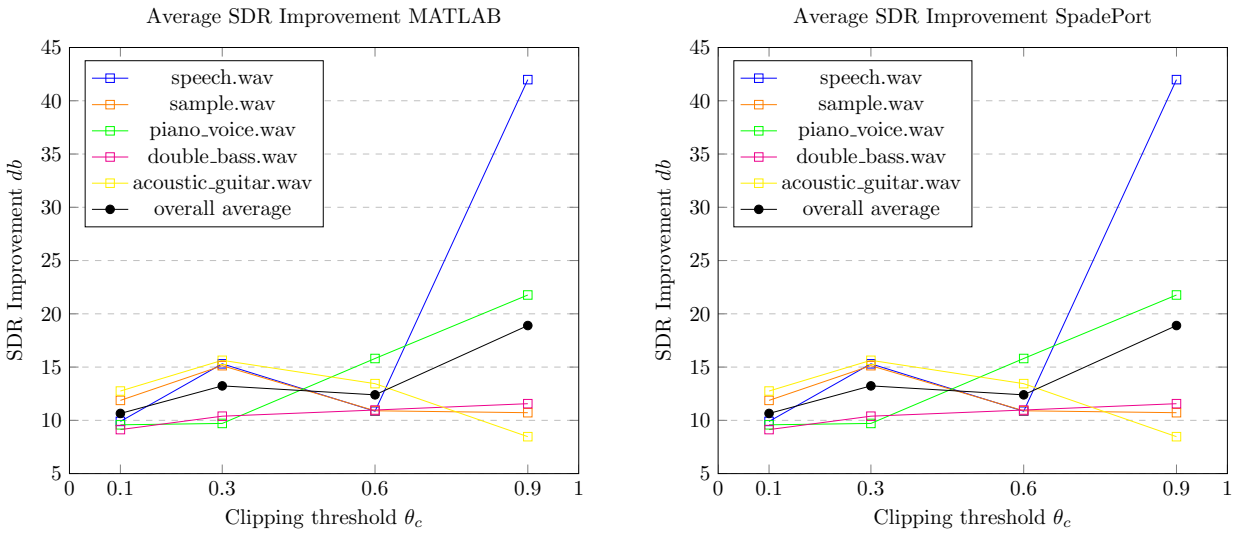


Figure 4.2: SpadePort S-SPADE performance

Our experiments indicated that the declipping process, resulted in an average of `13,981db` improvement in Signal-to-Distortion Ratio (SDR). This finding is consistent with both the MATLAB version and our own open-source solution demonstrating identical improvements to within 10^{-10}. The accompanying figures illustrate the average SDR gains for both implementations across the specified clipping range, θ_c, as previously stated.



(a) MATLAB S-SPADE SDR Improvement

(b) SpadePort S-SPADE SDR Improvement

To further demonstrate the efficacy of our implementation, we include visual representations in the form of waveform and spectrogram produced by both implementations. Utilizing a 512-point Hann window for the spectrograms, we generated these images with Audacity version 3.3.3 on macOS Sonoma 14.

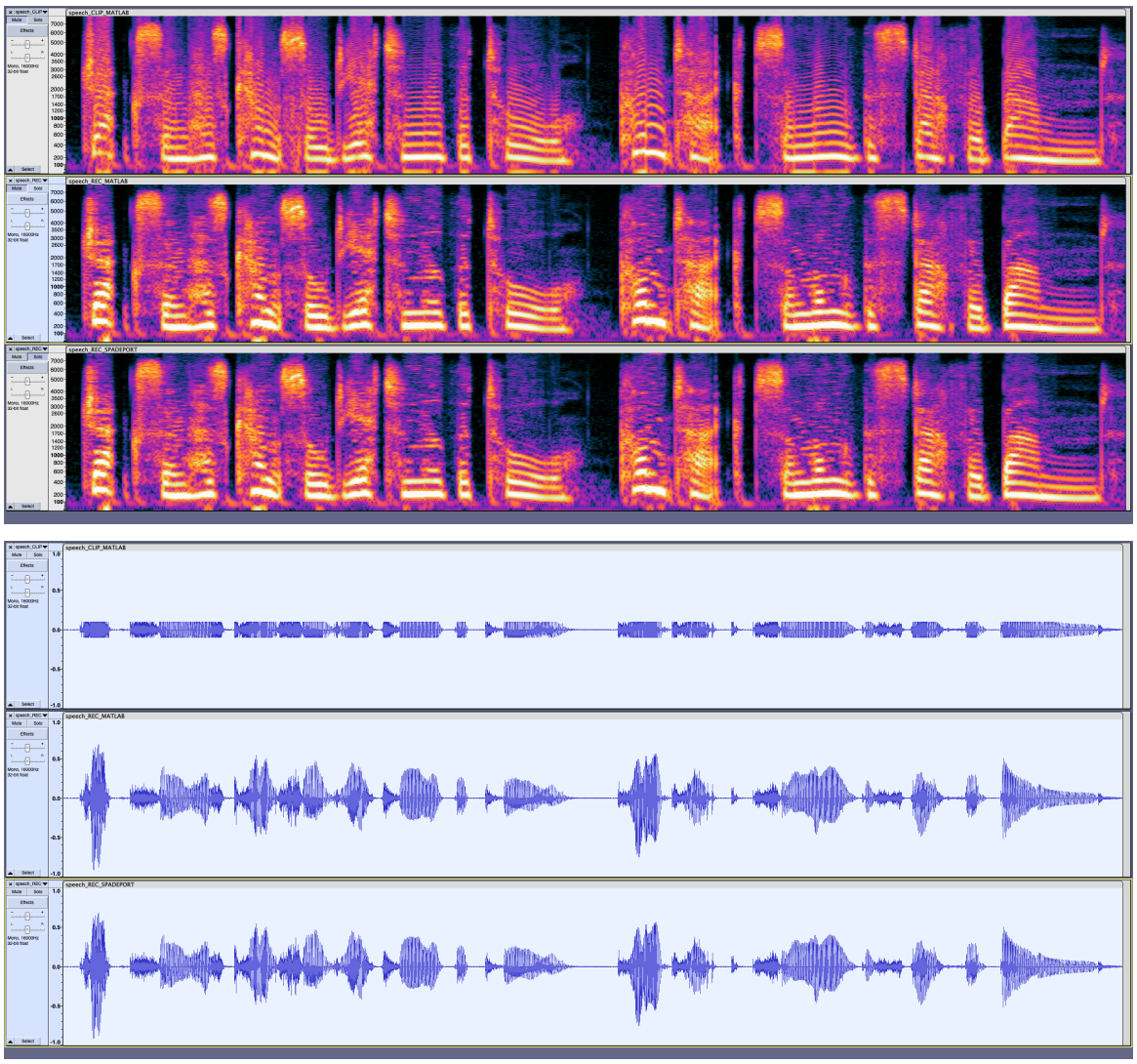


Figure 4.4: Specrogram, Waveform view for Clipped, MATLAB and SpadePort processed audio export files

To summarize, both the MATLAB-based approach and SpadePort offer valuable advantages. However, in terms of processing speed, SpadePort clearly outperforms its counterpart. It's noteworthy that SpadePort's performance is more consistent across different clipping levels and audio sample types, potentially making it a preferred choice for varied audio restoration projects. Equally important is the observation that both implementations yielded identical results in terms of SDR improvement between the clipped and reconstructed signal.

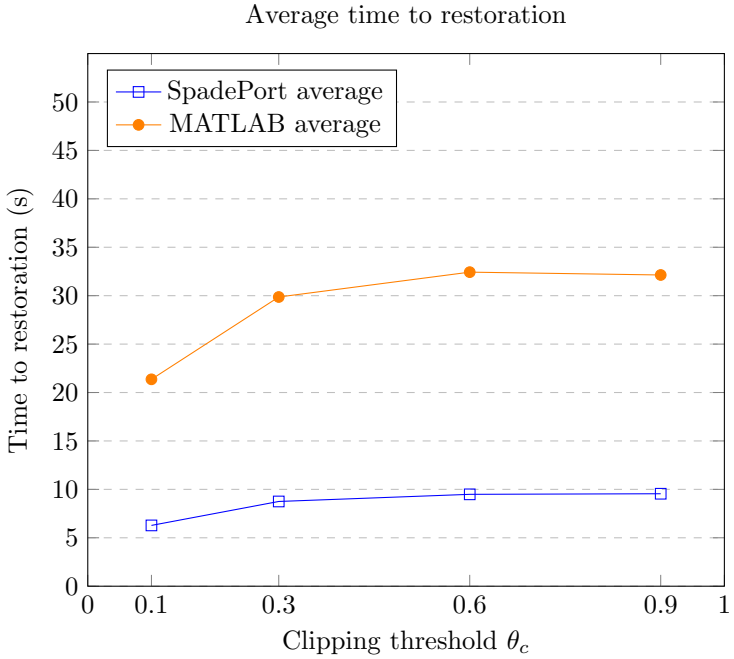


Figure 4.5: SpadePort & MATLAB S-SPADE restoration performance in the time domain

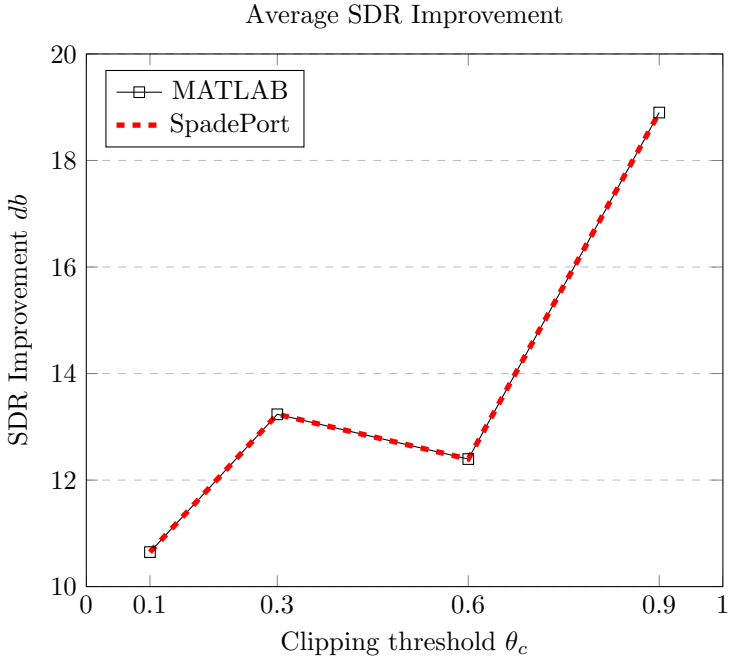


Figure 4.6: SpadePort & MATLAB S-SPADE restoration performance in terms of SDR improvement

While direct comparison of the output audio might reveal minor discrepancies, such differences are anticipated given the nature of the audio declipping problem and the fact that the DFT transform algorithms in these two platforms can yield different results on various environments. Neither method guarantees a flawless match with the original. Furthermore, the two implementations exhibit stark contrasts in resource utilization: the MATLAB approach, with its necessary toolboxes, required roughly 1.85GB of RAM, whereas SpadePort was

more resource-efficient, averaging a memory use of only 11.5MB. In total based on our experiments SpadePort's processing time was only 29.42% of what the MATLAB implementation took, rendering it approximately 3.4 times faster.

Chapter 5

Conclusions and Future Work

Within the dynamic and rapidly advancing field of audio declipping algorithms, this research evaluated the practicality and effectiveness of an open-source implementation of SPADE, a state of the art declipping algorithm. Our focus was directed at evaluating the performance metrics of a traditional MATLAB-based methodology in comparison with our proposed solution, a cross-platform, C-based implementation that is not only accessible to everyone but also adaptable to various operating systems.

In the course of our experiments, we discovered notable performance characteristics of the MATLAB SPADE implementation. Specifically for a diverse dataset of audio samples, an average of 4,8 seconds of processing time for every second of reconstructed audio was required. Beyond processing time, the MATLAB implementation's memory footprint was considerably higher, averaging around 1.8GB during its processing.

Delving deeper into its runtime behavior, the MATLAB implementation exhibited increased runtimes at clipping thresholds over 0.3. This suggests potential inefficiencies in the underlying algorithms, particularly those associated with sorting, Fast Fourier Transforms (FFTs), and vector operations. Such observations hint at opportunities for further optimization tailored to these specific signal processing tasks.

In stark contrast to the MATLAB approach, our open source C-based implementation showcased commendable performance attributes. Notably the exported audio files had identical improvement on their Signal-To-Data ratio between the reconstructed and the clipped signals. In terms of processing efficiency, our solution's required processing time resulted in an average of 1,4seconds for every second of clipped audio, rendering our implementation up to 3.4 times faster than the original proposed by the authors of the foundational paper.

Equally impressive was its minimal memory footprint, averaging a mere 11-12 MB, which underlines its potential suitability for embedded systems, or systems with memory constraints. Moreover, our implementation demonstrated a consistent performance that was largely detached by the clipping threshold values we tested, highlighting its efficiency across various declipping scenarios.

Considering avenues for further refinement of our C-based implementation, several prospective enhancements emerge. Foremost, there is significant potential in extending the file format support to the multitude of audio file formats available nowadays. Also, we could extend the algorithm to handle Stereo files. This improvement could leverage dual-vector FFTs, potentially executed in tandem. In scenarios where a recording undergoes damage to one of its channels, this approach enables one channel to inform the other, aiming for a reconstruction that closely mirrors the reliable audio.

From a performance perspective, the introduction of multithreading emerges as a possible augmentation due to the fact that every chuck that the SPADE algorithm is performed upon is independent. So the idea would be for large audio files, where the overhead required by the multithreading makes sense, one could split the audio signal in multiple segments where each one would be evaluated from an independent thread. This approach could prove nonefficient for chunk sets that could be easily reconstructed (silent or reliable parts of an audio signal). As a remedy we could introduce a dynamic load balancer where the framework could provision chunks to be reconstructed from an independent available thread.

Moreover due to the increased performance of our implementation, one could tune the algorithm to run fast enough to perform declipping to an almost real time stream where it could identify and remedy clipping instances on the fly making it invaluable to applications such as live audio streams. Lastly, we can't look over the possibilities modern machine learning techniques could introduce.

A possible enhancement to the algorithm would be to create a model that when trained robustly and cultivated from diverse audio signals, could inform the algorithm in real time and possibly result in satisfactory reconstruction with less processing overhead by using modern processors that can hardware accelerate such workflows.

In summation, our sincere hope is for the community to embrace our groundwork, to augment and refine it, and to collaboratively elevate it into a universally accessible and formidable tool.

Bibliography

[1] andryr. *spade-declipping*. https://github.com/andryr/spade-declipping. 2021.

[2] Clément Gaultier et al. ≪Sparsity-based audio declipping methods: overview, new algorithms, and large-scale evaluation≫. In: *CoRR* abs/2005.10228 (2020). arXiv: 2005.10228. URL: https://arxiv.org/abs/2005.10228.

[3] Srđan Kitić, Nancy Bertin, and Rémi Gribonval. ≪Sparsity and cosparsity for audio declipping: a flexible non-convex approach≫. In: *LVA/ICA 2015 - The 12th International Conference on Latent Variable Analysis and Signal Separation*. Liberec, Czech Republic, Aug. 2015, p. 8. URL: https://inria.hal.science/hal-01159700.

[4] Srđan Kitić, Nancy Bertin, and Rémi Gribonval. *Sparsity and cosparsity for audio declipping: a flexible non-convex approach*. 2015. arXiv: 1506.01830 [cs.SD].

[5] PANAMA. *SPADE the SParce Audio DEclipper*. https://spade.inria.fr, Last accessed on 2023-10-15. 2015.

[6] Pavel Záviška. *Audio Declipping*. https://rajmic.github.io/declipping2020/, Last accessed on 2023-10-15. 2020.

[7] Pavel Rajmic. *declipping2020_codes*. https://github.com/rajmic/declipping2020_codes. 2020.

[8] Pavel Zá viška et al. ≪Revisiting Synthesis Model in Sparse Audio Declipper≫. In: *Latent Variable Analysis and Signal Separation*. Springer International Publishing, 2018, pp. 429–445. DOI: 10.1007/978-3-319-93764-9_40. URL: https://doi.org/10.1007%2F978-3-319-93764-9_40.

[9] Pavel Zaviska et al. ≪A Proper Version of Synthesis-based Sparse Audio Declipper≫. In: *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019. DOI: 10.1109/icassp.2019.8682348. URL: https://doi.org/10.1109%2Ficassp.2019.8682348.

[10] Pavel Záviška, Ondřej Mokrý, and Pavel Rajmic. *S-SPADE Done Right: Detailed Study of the Sparse Audio Declipper Algorithms*. 2020. arXiv: 1809.09847 [math.OC].