Real-Time Automatic Gain Control for Singing Voice Applications

Bachelor Thesis in the course Informatik

Author:

Nils Heine

Matriculation Number: 6703759

Abstract

The objective of this study was to develop a plug-in for digital audio workstations which reduces the long term dynamics of vocal signals to release mixing engineers of this task. The project started with the creation of a prototype which had the main functionality implemented. Subsequently, it was improved by adding additional features, i.e. being able to write a gain automation into a digital audio workstation or enable automatic parameters settings. In addition to these improvements a side chain feature was added extending the use of the plug-in by adjusting the vocal signal output level to the side chain input level (in anticipated use: the instrumental backtrack).

The prototype's outcome was compared to the commercially available plug-in "Vocal Rider"[1] which is a plug-in with similar functions. Some improvements were based on this comparison as well as it influenced the subsequently parameter setting.

Finally, the advantages of the side chain feature were evaluated by hearing tests with independent participants.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Motivation

When a sound engineer is editing a song he/she wants all recorded audio tracks to be perceptible in the final mix (apart from some exceptional cases). This is most important for audio tracks with notably significance for the musical piece. This thesis will deal with vocal tracks due to their meaning, main melody or recognition value of the song. The challenge for vocal tracks in the mix is the wide dynamic range that singers often use unlike for example a distorted electric guitar which mainly stays on the same loudness level and is therefore easy to mix with great presence. In almost every mix the vocals pass through an compressor to reduce their dynamic range. The result of this procedure is rarely satisfactory as compressors are working comparatively fast - too fast to compensate whole song parts or even some seconds of a different remaining vocal level, e.g., when a singer is changing his singing style or is singing quieter during an instrumental break which may not fit the mix. As a result it is a common procedure to automate an applied gain for every vocal track in the digital audio workstation (DAW) via sketching a gain curve by hand. This is a time consuming and monotonous task and therefore predestined to hand over to a machine.

Admittedly, professional studio singers could be able to reduce the unwanted dynamics as well but they will rarely succeed completely for the following reasons: As they are covering different singing styles from whispering to screaming it is hard to maintain in perfect proportion to the current backtrack. Additionally this could also effects the current recording setup which can be or has to change between takes (at least pre-gain changes will effect the resulting dynamics). Often a track is still changing its composition and volume level after the vocal recordings as the producer adds or removes certain tracks or effects to get a better result. And finally at the current time of home recordings, where it just needs a single computer to be able to do professional audio mixing, it is not to expect that the majority of vocal recordings are from professional studio

1

singers. Consequently the problem described will occur in many mixing sessions.

1.2 Objective

The idea of this study was to develop a DAW plug-in handling the described problem automatically: A plug-in that will sketch a gain curve for a vocal track in real time. This saves time during every mixing session and the outcome would be more accurate than doing the work manually. Furthermore, as an editing tool it should not add colouring to its processed signal, in this aspect differing from many compressors. This is important as the plug-in's main purpose is not a creative task but only to be a tool designed for relieving a mixing engineer of dull work.

As the recorded sound pressure level is not transferable to the perceived loudness, the goal was to adapt selected algorithmic features of ITU-R BS.1770-4[4]. This algorithm is a standard for adjusting audio files to a same humanly perceived loudness level. This will help the gained audio track to stand out in the mix at every part of the song.

A plug-in with similar purpose titled "Vocal Rider"[1] was published by Waves[1], but the algorithm behind this plug-in is not published. One aim of this study is to build a plug-in with features considered important and useful for the purpose described and to compare the resulting audio files in order to validate the algorithmic approach and extend the initial idea. In best case this will result in algorithmic progression for this so far rarely filled section of audio editing.

The main goal of this study is to enable the plug-in to affect long term dynamics on a single vocal track in real-time. In this case real-time means that the study plug-in should be able to work without any offline calculations. It is still allowed to have some samples of delay at the output, which can be compensated by a DAW. Therefore the plug-in could be used as a insert effect for a DAW track and would react on level changes at playback. As extended goal it is anticipated that the plug-in could additionally receive information about the backtrack via a side chain input and therefore be able to adjust the average loudness of the vocals in relation to the instrumental track.

---

[1]Waves Audio Ltd. is company which is developing software audio effects for professional digital audio signal processing

CHAPTER 2

Basics

2.1   Loudness

Including the human perception in the algorithm is a complex task as for example absolute threshold of hearing and the subjective perception of sound pressure are varying for different frequencies (see Fig. 2.1).
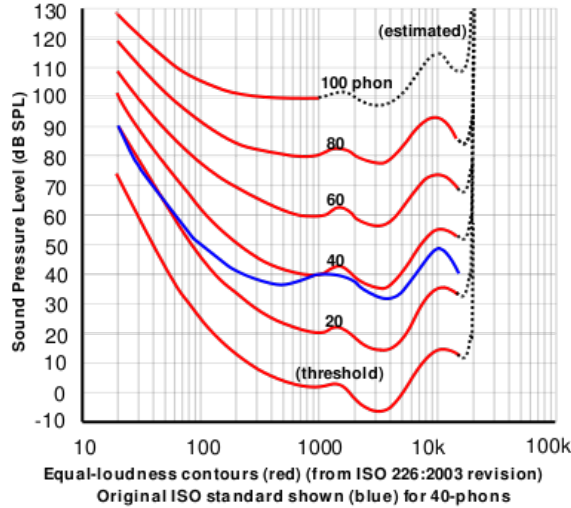
Figure 2.1: Loudness perception[2]

While gathering information on how to improve the plug-in in terms of human perception the ITU-R BS.1770-4[4] algorithm was evaluated. This algorithm classifies an audio file for its humanly perceived loudness. It is mainly used by television and music streaming services as the program loudness can be kept steady while switching content. As the perception is of substantial interest for mixing a song, this was investigated in this study. Due to a good documentation about how to implement the loudness algorithm a copy of it was build in Python and it was decided which elements to be adopted

in the plug-in. The first and second element were two filters. Their use is to mimic the human perception of sound pressure at different frequencies. The first filter is a low cut for the reason that human hearing is insensitive to low frequencies. The second filter is a high shelf and is '*used to account for the acoustic effects of the head*'[4]. This imitation of the human hearing is substantially simplified but cost-effective in terms of computation. The low cut filter has a cutoff frequency of 38 Hz, the high shelf of around 1681 Hz.
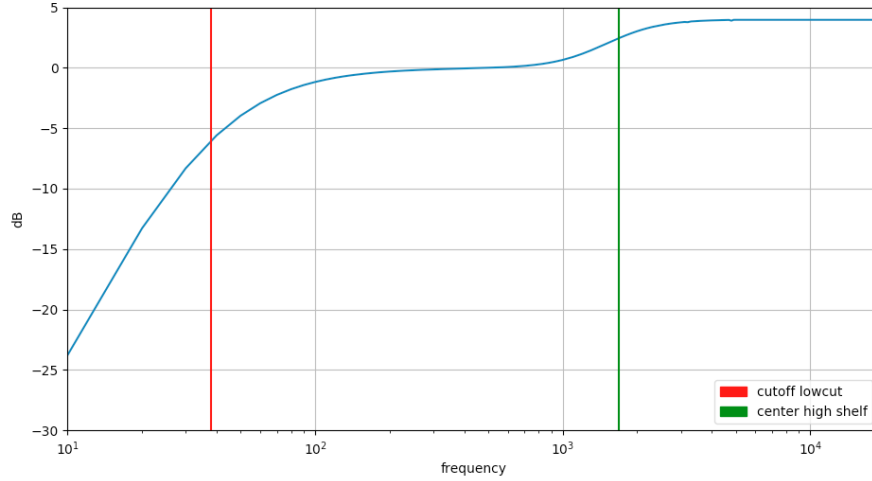
Figure 2.2: Plotted outcome of adopted filter implementation for the audible spectrum of frequencies

The filter section is followed by the determination of a root mean square (RMS) value for time windows of 400ms overlaping 75% (a value is saved every 100ms). This means it is calculating the root of the average from the squares of each sample in the set time window which equals the power of the signal.

RMS calculation is used because it is part of the imitation of human perception: a human will not interpret a small impulse of a handful of samples as loud as a audio signal at the same level of longer duration. For example '*a 3-msec pulse must have a level about 15 dB higher to sound as loud as a 0.5-sec (500-msec) pulse*'[5].

Subsequently, all RMS values below a absolute threshold of -70LKFS[1] are sorted out by a gate and a average value is determined. Another gate of -10LU below the average is applied on the rms values and a second averaging of measurements is performed. The resulting average is the loudness of the processed audio file.

---

[1]definition in [4]

2.2 Vocal Rider

'Vocal Rider' by Waves is a commercially available audio plug-in which realizes a automatic gain adaption for vocal signals.

Figure 2.3: UI of Waves 'Vocal Rider'[3]

The vocal signal's level is adjusted to the set *target* value (at top of Fig. 2.3). The *'target sets the reference range for vocal mix positioning. [It] (. . .) will move the Rider Fader's '0' calibration.'*[3]. Additionally a user can control the maximal gain adaption with the *'range'* slider and adjust the resulting level to the backtrack via a output gain. For results depending on the instrumental backtrack a side chain input can be used and adjusted with the *'music'* potentiometer.

As in this study the same basic idea is realized with a independent approach it is possible to adopt useful results from the comparison (see chapter 5) of both plug-ins and remain with potential advantages.

2.3   Optimization

For the comparison in chapter 5 optimization methods were used to find parameters
for the study plug-in to get to similar results as the 'Vocal Rider' does. Therefore the
scipy[6] methods scipy.optimize.brute and scipy.optimize.minimize were chosen.
Previously a function had to be written which returns the current problem as a number.
Thereby a smaller number equals a smaller problem.
The 'brute' optimization algorithm 'computes the function's value at each point of a
multidimensional grid of points, to find the global minimum of the function'[7]. The
according 'grid' is therefore previously defined for the current optimization problem.
This is done by initializing a set of user defined bounds and step sizes for each of the
parameters which is transferred to the function to be optimized.
The 'minimize' optimization algorithm per default 'uses the L-BFGS-B algorithm ...
for bound constrained minimization'[8]. This algorithm is able to compute faster and
more selective but needs a good inital parameter guess to get to a proper result.

CHAPTER 3

Prototype

3.1 Overview

The basic prototype of this study is working in five steps: filter, root mean square (RMS) calculation, gate, gain adaption, delay. The plug-in's algorithm will take blocks of a known amount of samples. These blocks will be processed one after another.

The processing chain of these blocks starts with a low-cut and a high-shelf filter that will be applied sample-wise on the incoming audio signal. The filters are a simplified mapping of the perception of sound pressure for human ears (see chapter 2.1). This is the first step to adjust the plug-in to loudness instead of sound pressure. However, the plug-in will not be running with the full loudness detection algorithm[4] due to calculation time and real time capability.

After filter processing every sample is passed to RMS calculation. The goal is to output the squared average of a previously specified period of time. The square root (final RMS result) will be determined in the calculation of the equivalent dB value (see Fig. 3.1). The algorithm is converting the linear audio samples into the logarithmic dB scale in order to display gain values and loudness goal $((L_{goal})$ see chapter 3.5/3.6) in dB in the user interface (UI) as this is the standard scale of DAWs. Thereby the executive sound engineer should intuitively know how to interpret and interact with the UI (see chapter 6).

When the dB conversion is finished, all samples pass through an initially specified gate. The gate will set all samples with lower dB value as its threshold to the current L_{goal} (see chapter 3.6, 4.2). In this way the plug-in will not operate when it is fed with silence or irrelevant noise.

Next step after the gate is gain adaption, i.e. the gated RMS value is compared to the current L_{goal}. Depending on the difference of both values it will result in an preliminary gain. The gain variations per sample are smoothed comparable to the RMS calculation. This leads to the final gain value.
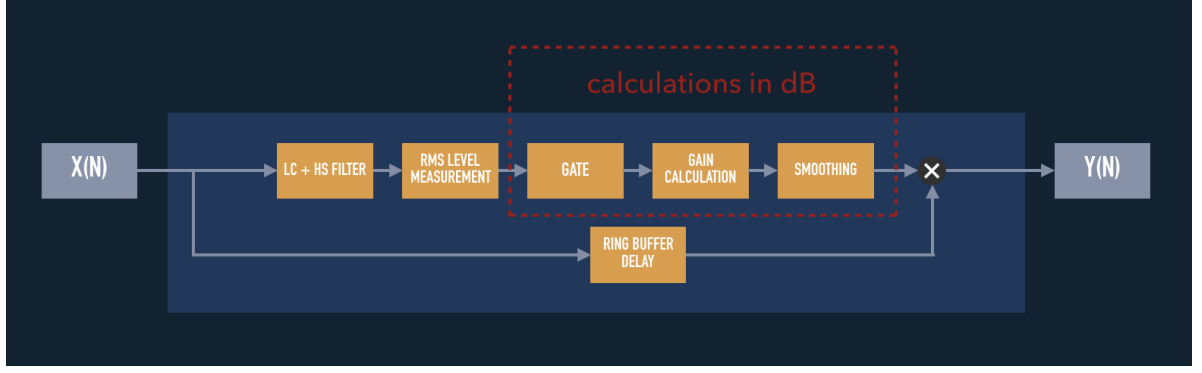
Figure 3.1: Prototype processing chain

Lastly, the new gain is multiplied with the current sample. Because the plug-in's behaviour is smoothed (intended to sound natural), it will not react instantly to the input. To compensate the reaction time it delays the input signal before multiplying the calculated gain. This delay is later offset by the DAW.

3.2 Filter

As to observe in Fig. 3.1 the processing chain starts with the two filters. They are adopted from the ITU loudness standard[4]. The implementation is based on the biquad filter from 'DAFX: Digital Audio Effects'[9].

$$x_h(n) = x(n) - a_1 x_h(n-1) - a_2 x_h(n-2) \tag{3.1}$$

$$y(n) = b_0 x_h(n) + b_1 x_h(n-1) + b_2 x_h(n-2) \tag{3.2}$$

The second order biquad filter architecture was chosen because it is a very flexible and simple solution with just two samples delay. The calculation of filter coefficients is adopted as follows:

f_c = cut-off frequency, f_s = sampling frequency (rate), $K = tan(\pi f_c / f_s)$,
Q = factor for height of the resonance, G = gain, $V_0 = 10^{G/20}$

Lowcut:

$$b_0 = \frac{Q}{K^2Q + K + Q} \tag{3.3}$$

$$b_1 = -\frac{2Q}{K^2Q + K + Q} \tag{3.4}$$

$$b_2 = b_0 \tag{3.5}$$

$$a_1 = \frac{2Q * (K^2 - 1)}{K^2Q + K + Q} \tag{3.6}$$

$$a_2 = \frac{K^2Q - K + Q}{K^2Q + K + Q} \tag{3.7}$$

Highshelf:

$$b_0 = \frac{V_0 + \sqrt{2V_0}K + K2}{1 + \sqrt{2}K + K^2} \tag{3.8}$$

$$b_1 = -\frac{2(K^2 - V_0)}{1 + \sqrt{2}K + K^2} \tag{3.9}$$

$$b_2 = \frac{V_0 - \sqrt{2V_0}K + K2}{1 + \sqrt{2}K + K^2} \tag{3.10}$$

$$a_1 = \frac{2(K^2 - 1)}{1 + \sqrt{2}K + K^2} \tag{3.11}$$

$$a_2 = \frac{1 - \sqrt{2}K + K^2}{1 + \sqrt{2}K + K^2} \tag{3.12}$$

Hence the loudness algorithm uses second order filters (with two delay memories) it works like illustrated in Fig. 3.2. It uses the direct form II as it needs half as much delay units per filter as direct form I does. The algorithmic implementation for the study plug-in adopts the processing visualized in Fig. 3.2 as well. This implementation is capable of many filter styles at different cutoff frequencies. For the study plug-in it is used for the low cut and the high shelf filter described above which are processed one after the other on the current audio sample.
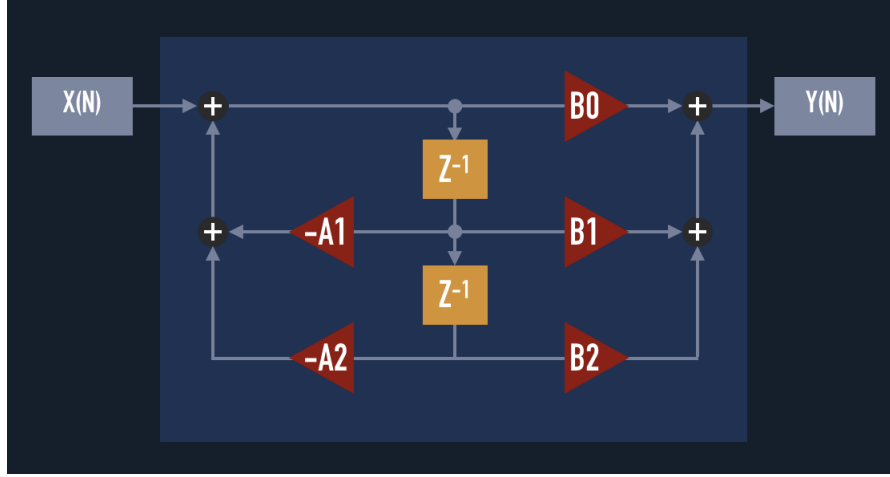
Figure 3.2: Second-order digital filter (direct form II)

After processing this section returns the filtered samples.

3.3   RMS

To measure the level of the audio signal at the current position the plug-in calculates an RMS value for the reasons described in chapter 2.1 and because it has the necessary time to calculate an average value as it does not need and does not want to react fast. This is also part of the ITU-R BS.1770-4[4] loudness standard. Additionally the '*RMS is equal to the value of the direct current that would produce the same average power in a resistive load*[10]'.

The algorithm calculates a RMS value of the last 30 ms.

The RMS implementation is based on 'Digital Audio Signal Processing' by Udo Zölzer[11] and can be performed in mainly one line of code:

$$rms^2 = (1 - c) \cdot rms^2 + c \cdot sampel^2 \tag{3.13}$$

The filtered samples are used for this calculation (3.13) and the result is transferred to the next section. 'c' is a time coefficient (see below) of small size. This parameter influences the weight on how much the current squared sample will inflict the quadratic mean. In the complete RMS procedure the final result is the square root of the value the described calculation 3.13 results in. In this case the final RMS determination is done

when it will be converted into the logarithmic dB scale in the next step, as a squared input just changes the value of one coefficient in this process. Therefore the final RMS value will appear as logarithmic dB in the next section and save a little time.

3.4   Time Coefficients

The time coefficients are used for RMS calculation and to realise the compress and amplification times (see chapter 3.6). They are determined with a formula by Zölzer[11]:

$$t_a = 1 - exp(-2.2 \cdot \frac{1}{f_s} / \frac{ms}{1000}) = 1 - exp(-2.2 \cdot \frac{1}{f_s} / s) \tag{3.14}$$

Zölzer had decided to use the exponential function because it draws a natural decay. He determined -2.2 for the first part in the exponent by solving an equation system to achieve an attack time $ta = t90 - t10$. This means that when a calculation similar to the RMS calculation defined above uses such an time coefficient, its reaction on a input change will need the chosen amount of time (s in seconds) to get from 10% of the final reaction to 90%. Illustrated in Fig. 3.3. The second part in the exponent $\frac{1}{f_s}/s$ calculates the proportion of one sample to the amount of seconds of the ta. This needs to be done because it will be used on every sample.
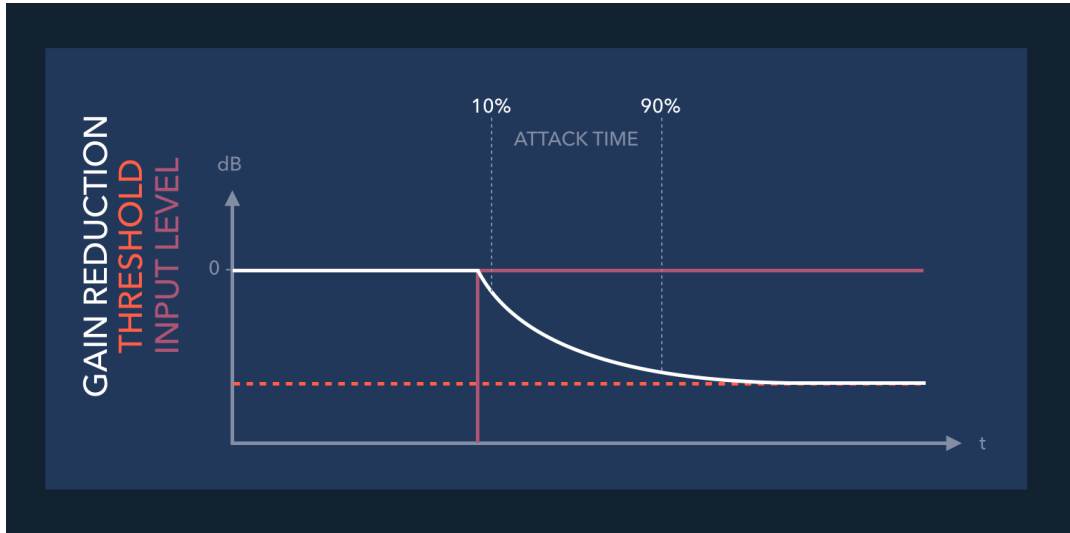
Figure 3.3: Attack time $t90 - t10$

11

3.5 Gate

The plug-in should operate while there are vocal signals and stop if there are none or just a soft decay. Otherwise unwanted effects would be produced by amplifying noise. Additionally, it would distort the applied gain for the actual vocals through strong gain increase at the gaps in-between.

To solve this problem the algorithm includes a gate. The gate checks for every sample if the sound pressure level is over a certain threshold. If not, it will be replaced by the current L_{goal}. As the threshold and L_{goal} are set in dB, the first step in the gate is to convert the transferred sample (rms^2) into the logarithmic scale (results in final RMS value).

$$10.0 \cdot log_{10}(rms^2 + 1e - 10) \tag{3.15}$$

$$20.0 \cdot log_{10}(rms + 1e - 10) \tag{3.16}$$

It uses calculation 3.15 instead of 3.16 as the rms^2 is transferred squared. It adds $+1e - 10$ to head off the undefined log10(0) case. After the conversion it gates the dB rms sample value at the dB threshold.

The threshold is defined as L_{goal} minus the currently allowed gain range[1]. Thereby it is still possible to use the whole gain range for gain adaption and at the same time ignore the decay of the vocals. With this formula the threshold adjusts to the level of the vocals as L_{goal} is detected (see chapter 4.2).

When RMS samples are not passing the gate and therefore being replaced by L_{goal} the algorithm adapts the gain to 0 (after a short period of time) because it has achieved its goal (see chapter 3.6).

---

[1]maximal absolute amount of adapted gain in dB

3.6 Gain

In the next step, the most crucial part is performed: the calculation of the final gain value for the current sample.

$$g = L_{goal} - sample \tag{3.17}$$

$$G_{new} = (1 - c_{gain}) \cdot G_{prev} + c_{gain} \cdot g \tag{3.18}$$

At first it computes the difference (g) between the loudness goal (L_{goal}) and the current processed sample. The result is the gain factor that would be necessary to get it to L_{goal} (by multiplying in the linear number space).

Since the algorithm is designed to react on loudness differences for longer duration than for example compressors or expanders do, the gain adaption is smoothed over a proportionate amount of time. The smoothing is attained similar to the RMS calculation but uses different time coefficients.

The fitting time coefficient (c_{gain}) is chosen by comparing the calculated gain factor g with the gain that the function had returned for the previous sample. If g is smaller than the last gain (G_{prev}) the gain for the current sample will be smaller, too. Therefore the dynamics of the input vocals will be compressed in relation to the last processed sample so it chooses the compression time coefficient. In contrast, when low level signals are amplified the amplification time coefficient is chosen. Different time coefficients for amplifying and compressing dynamics are useful as amplifying a signal can be risky, while compressing causes no trouble. For instance boosting a weak signal also boosts all the recorded unwanted noise. Additionally, digital dynamics are limited and as the signal is amplified it risks to clip at the 0dB cap and produce distortion.

After calculating the smoothed gain it gets clipped at the user chosen range up to +/- 10dB. On one hand side, this ables the user to adjust the maximum variation of dynamics on the other hand side it prevents the gain to increase up to problematically high values. This does not happen during normal use in an expected environment but can not be completely ruled out due to possible unknown software errors or an unknown environment. As an error in the adapted gain does not only affect further calculation results but also affects the mixing engineer who is listening to an amplified signal, it is of special importance to avert wrong values (see chapter 4.6).

When the gain is determined it will be converted back to the linear number space. Fi-

nally it just needs to be multiplied with the current sample.

3.7 Lookahead
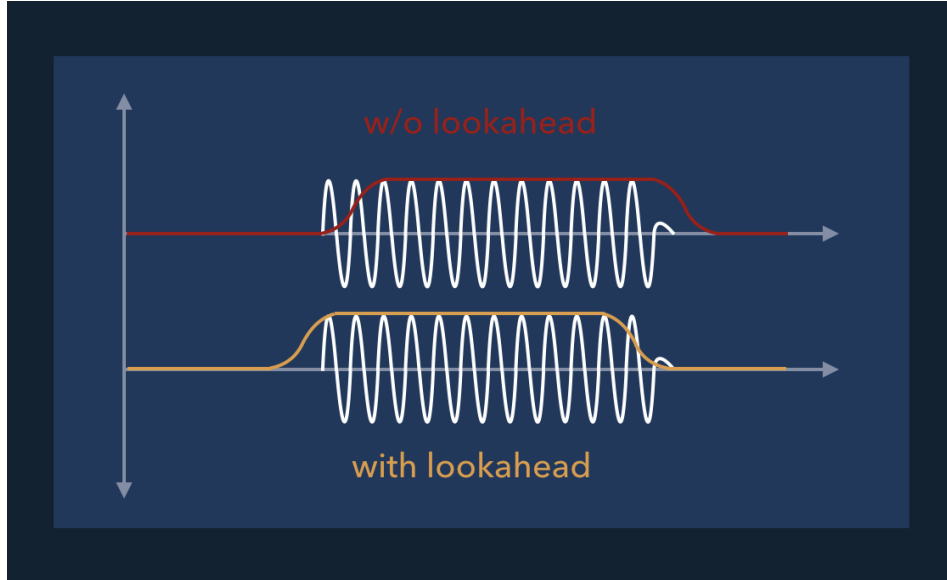
Figure 3.4: Gain adaption with and w/o lookahead

To achieve its goal the study plug-in is designed to react slow and therefore needs some time to adapt on a change in the average signal level. If it multiplies every sample with the gain value calculated from the same sample, the plug-in will be adjusted after the RMS has changed to the new average and the smoothed gain has slowly adapted. This will take some milliseconds and consequently the first part of the alternate signal is not perfectly gained. While in particular this part introduces a new section in a song, for example a chorus, it is desirable to have an adapted gain already at this point. In addition an early gain adaption is similar to the gain automation a mixing engineer would draw what the plug-in initially tries to displace. As he/she usually wants a smoothed transition between different gain values but also the adjusted gain already attained at the first sound of the following vocals, it results in a similar gain curve as the compensation of calculations in the plug-in would do.

To compensate the adaption time a simple lookahead feature was algorithmically implemented, which allows the plug-in to calculate the gain for the current samples while

looking at future samples. This is realized by a splitted input signal: One signal used to calculate the resulting gain adaption, the other signal is multiplied with the resulting gain but previously delayed. The second signal is transferred as output to the embedding DAW where a delay compensation is achieved.

Chapter 4

Improvements

4.1 Automation

Using the study plug-in, a mixing engineer will not have to draw gain automations manually. Nevertheless it can be useful to have an automation drawn by the plug-in. On one hand side it visualises the adapted gain curve better than the gain slider contained in the UI alone. On the other hand, it can save up calculation resources when the plug-in reads its own automation instead of processing the same vocal track in a repeated way. This is not necessary in simple mixing sessions but comes in handy when the mixed musical piece contains a great number of tracks with even more digital effects on each channel. Every digital effect increases CPU power needs. Often single tracks are bounced in place[1] to avoid problems in performance. When the plug-in just reads an automation it is no considerable part of this problem even if there are multiple vocal tracks using this effect.

For those reasons the study plug-in was extended with this feature supported by the JUCE framework (see chapter 6.3). The final solution sets a automation point every 80ms as long the gain changed about at least 0.1dB. Therefore the gain adaption is constantly drawn in the automation curve. It does not set a new point when the gain is not adapting as this would be redundant.

4.2 Adapting Loudness Goal

The L_{goal} parameter is not regarded as a gain controller. It is not designed to amplify the signal to a desired level. Instead, it should be adjusted to the actual loudness of the signal. In this way the plug-in can use its full range for changes on dynamics. For

---

[1]all effects are rendered and combined with the actual signal to a new audio file

example when L_{goal} is set far below the signal level, the plug-in will stay at its minimal allowed gain value (initalized with -6dB) as long as it gets the high-level input. Even if it is at one end of the user defined dynamic range it will still operate in one half (only negative/positive gain adaption) for most of the time. Therefore it is important to set L_{goal} correctly. In best practice the plug-in will compress the incoming signal as much as amplifying it.

A user is able to set L_{goal} according to the vocal level. However, it still remains a problem as the perfect L_{goal} possibly alters through a song. As a consequence it is difficult for a human to guess the average setting for the entire musical piece. Therefore some calculation-based solutions were tested.

The first approach was determining a new L_{goal} for every second. To achieve this the plug-in summed up all resulting gain values for each sample in this time period. Afterwards it calculated the average tendency (more positive or more negative gain values). The offset was finally subtracted from the current L_{goal} and the sum of the gain values got reset.

Problematic with this approach was that the L_{goal} adaption was one second too late if the signal level changed. In addition it diminished some advantages of the plug-in as it treated different parts of the vocals with different settings which results in some local benefits but leads to varying level through the whole track.

As a consequence analysing only parts of the vocal track would not lead to a profitable result. Furthermore, the plug-in should go through all critical parts of its input and therefore decide for the best suiting L_{goal}. In conclusion L_{goal}-detection was implemented. During the time this detection algorithm is active, the plug-in will not multiply the calculated gain with the signal but is still determining it, dissimilar to a regular bypass. In addition the plug-in re-adjusts its allowed gain range for this period of time to the maximum (+/- 10dB). This is performed to find the most accurate average loudness rate and it has no negative consequences as it endures as long as the detection is processing. If the plug-in runs the detection through the full vocal track it calculates a decent L_{goal}. To avoid the result being falsified by sections without vocal signal, it only adapts on input with actual signal.

This is a task predestined to be performed offline, but due to the complexity of offline calculations in a plug-in operating in different DAWs the resources to realise it were not available in this study. For future work it would be possible to make use of the full ITU-R BS.1770-4[4] algorithm for this task.

Although the L_{goal}-detection algorithm is probably the best current way to set the pa-

rameter, it is still allowed for a user to choose it manually for own creative reasons. When L_{goal} algorithmically adapts or is set to a new value chosen by the user the threshold of the plug-ins gate will change as well. Regardless of whether the average input is of low or high level the plug-in is thus able to handle it with reasonable settings.

4.3   Side Chain

So far the plug-in works well on single audio tracks and reduces long term dynamics. This substantially lessens the work for a mixing engineer but a average vocal level staying constant during a whole musical piece is not necessarily the wanted result. If the instrumental backtrack varies in loudness it is desirable for the plug-in to vary its output gain accordingly. To realise this variation the plug-in requires additional information. Therefore a side chain input was added to the plug-in's interface.

After implementing the new input this information had to be processed. This information about the loudness of the backtrack (to be fed into the side chain) is needed to enable determination of a useful additional output gain. In process of detecting the loudness of the side chain input the additional signal is passing through the filter, through RMS and gate calculations as the regular input signal (see Fig. 4.1). Passing includes transformation into logarithmic number space. Afterwards the signal is compared to the current L_{goal} and the calculated difference is smoothed. Now the smoothed gain is transferred back to linear number space and it is multiplied with the gain of the parallel determination from the main input.

It is important for the side chain processing not to interfere with the main input gain calculations of the plug-in. Especially L_{goal} must not be changed through this process. Otherwise it would corrupt its results in terms of dynamic range and make the L_{goal} detection useless.

While testing in real use cases it was found beneficial to add an additional output gain for the finally returned signal. This reduced the effort of matching vocal output level with the backtrack. It would be an advantage if the plug-in could set the output gain itself depending on the side chain input, but as for different music styles and creative decisions there is not one correct version of vocal-to-backtrack relation but a great number of reasonable options could be appropriate. Nevertheless during future improvements of the plug-in this part could be simplified in its user interaction.
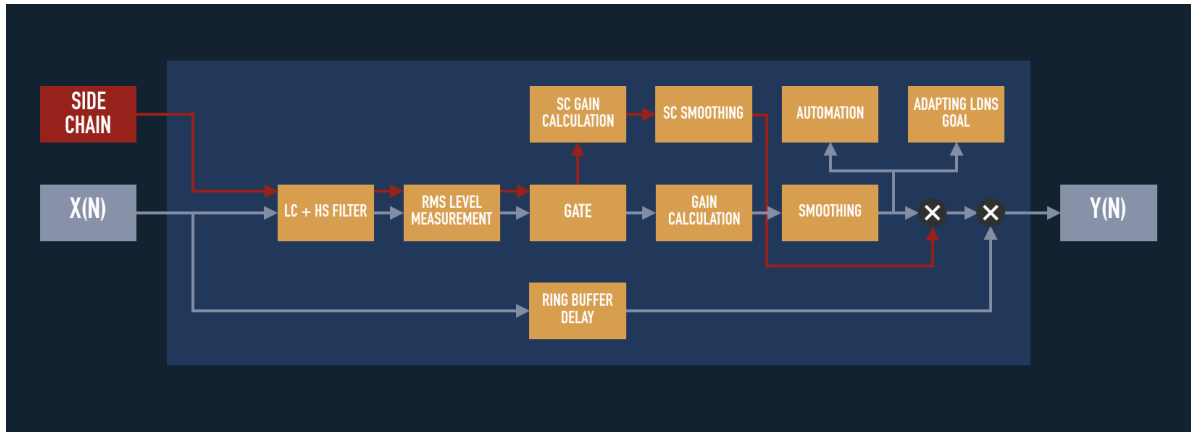
18

Figure 4.1: Improved processing chain

A remaining question for the side chain implementation is related to setting of the average time coefficients for RMS calculations and gain adaption (see chapter 6.1). On one hand side it should not react on small changes (for example when a musician is just accentuating certain beats in the backtrack), on the other hand side it needs to react fast enough to calculate an appropriate gain for the first words of a new song part with a different loudness level. It is not negligible that it operates with the lookahead already realised for the plug-ins main use.

Additionally, in the first approach it turned out to be a difficult problem for the side chain gain adaption to deal with instrumental breaks of longer duration (up to a full bar). While most of the instruments are pausing during those breaks the vocals should stay at the same level as it is often used to emphasize the remaining musicians. With the current implementation of side chain adaption instrumental breaks were causing a parallel slowly falling vocal level. After the implementation of the idle time (see below) at gain adaption this behaviour could be avoided for the most part.

4.4 Idle Time

'Idle time' describes the time the plug-in will wait with further gain adaption when the input level falls below a certain threshold. The effect of an idle time is that the plug-in will ignore short gaps between vocal signals. Therefore the plug-in does not change the adapted gain for example when a singer does a short rhythmic break in his/her vocals. So it will just use the actual vocals in the input for gain adaption similar to how a

mixing engineer would do it. Due to a reasonable short duration of the idle time it is still able to adapt to 0dB gain for parts without singing. Subsequently there will be no risk of amplifying unwanted noise. The short period after a vocal signal and before a part of silence were the plug-in is still idle will not remain much longer than the vocal decay and is even shortened by the amount of lookahead delay.

At the first approach in implementing this feature the function of the gate was expanded. Each time the input dropped below threshold, the current sample was replaced by the last one above threshold during the set amount of idle time. Due to the fact that the last sample above threshold is still being at comparably low level it did not work as expected. The subsequent gain adapting was consequently falling to even lower results than without the change at the gate because at idle time it got values near the lowest possible one which was just not set to L_{goal} by the gate. It is assumed that the error could be fixed by replacing the samples at idle time with a RMS value from a sample further ahead, but this would require a additional memory space for past samples and it seemed to be too complex for this study.

Therefore, the current implementation is based on the plan that the plug-in should detect previously to the gain adaption whether there is no vocal signal and then freeze the current gain. As the gain adaption is much slower due to compress/amplification time smoothing than the RMS averaging before the gate, it therefore freezes the gain value before it is able to adapt to the level change.

The algorithm detects whether the current level is below gate threshold by comparing the sample with L_{goal}. If the transferred RMS averaged sample is at the exact value of L_{goal} it can be assumed that the gate replaced its actual value. The rare case of a sample being randomly at exact the level of L_{goal} and therefore stop the gain adaption cannot be eliminated in this implementation. Nevertheless, this makes no noticeable difference as it just stops the slow gain adaption for one single sample (max $1/44100$ sec in expected use case). At the first sample differing from L_{goal} the idle timer is reset and the gain adaption continuous. When the maximum idle time is reached it has the same effect apart from resetting the idle time counter.

With the idle time working well in the main processing chain of the plug-in, its advantages in terms of side chain integration were evaluated. With the plug-in version developed so far there was still a problem related to instrumental breaks and their unwanted impact on the calculated gain correction. Therefore a implementation of idle time similar to the already existing one for the main input was used. At the final side chain calculations it was implemented and adjusted the length of the maximum idle time
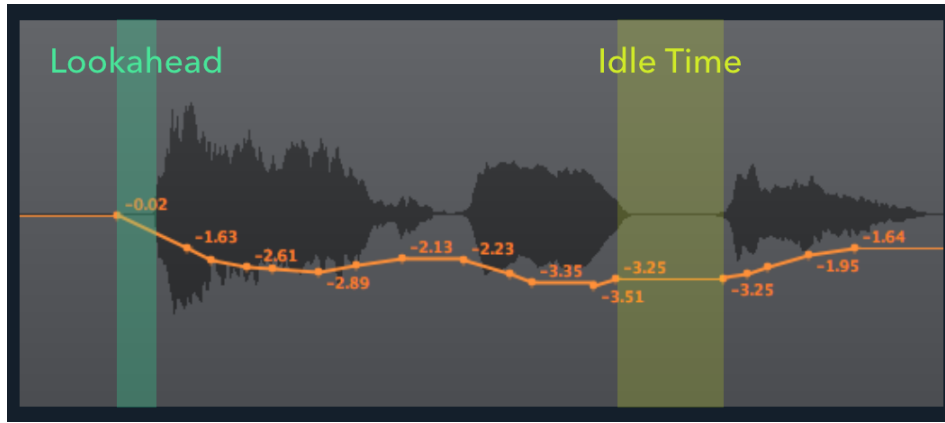
Figure 4.2: Idle time and lookahead marked at automation written by the study plug-in

with tests in a real mixing environment. The result with a reasonable set maximum was quite acceptable as it substantially reduced the gain drop during instrumental breaks as long as the side chain input level was adapted to L_{goal}. For better adjustment of side chain-to-L_{goal} relation, an additional input gain for the side chain was implemented. As it remains difficult to adjust it correctly for a whole song a self setting algorithm for the input gain was added similar to the already existing L_{goal} detection. As mentioned earlier this would be significantly better realised in an offline thread but also works in real time for now.

4.5 Wet/Dry

Many plug-in effects for DAWs have an additional slider to adjust the wet/dry ratio of their outcome. In these terms 100% "dry" is similar to the signal without the effect and 100% "wet" is a fully processed effect on the signal. This reduces the effort which would be necessary for sending the signal on a second track with the effect inserted to mix both tracks together in wanted proportions.

In consideration of the use of such a slider in the plug-in of this study, it seemed useful at first as it enables a user to set the intensity of dynamic compression. On the other hand this is already possible in a slightly different way with better outcome, by changing the current gain range. In addition that this feature has its main use in creative work, which is not the main objective of this plug-in, the final plug-in will not include a wet/dry slider for now. This decision was made with the intention to simplify the final

UI to its foundation followed by the reduction of the settable parameters (see chapter 6.1).

4.6   Final Gain Clipping

Due to the problems of a possible disproportional gain (see chapter 3.6) caused by an unexpected error, it needs to be checked before it is finally multiplied with the plug-in's input signal. This includes the gain of the plug-ins main functionality, the additional side chain gain and the user set output gain. It is not easy to set a good clip range[2] as with 3 different gain values multiplied it can result in problematically high amplification which can be on purpose in some cases. In todays DAWs usually clipping at 1.0 in linear number space is only happening at the master fader. As consequence it is unproblematic to mix a single audio track above this threshold as long as the final mix will happen below. Therefore it is not reasonable to clip the gain*input product for +/- 1.0 sample values.

As consequence not the final outcome could be clipped but the final multiplied gain which includes all three gain variables. The clip was set to operate in the linear number space as last instance before the final multiplication. It clips the final gain for the range of 0.0 to 6.0. The clipping range starts at 0.0 because negative gain in the linear number space does not make sense in this context as it would only invert the phase of the input signal in addition to the compression or amplification. 6.0 is the clip in terms of amplification which is similar to a approximately 15dB gain boost. This is lower than the highest value that can be achieved with all gain sliders on maximum setting but this is not expected to happen in reasonable use. With a clip at 6.0 the input signal could be amplified by 600%. If a signal needs stronger amplification in professional mixing it is presumable that the setting was done wrong or the recording is bad and should be replaced. If this could not be done or is not wanted it is necessary to edit the audio track before the start of the mixing session as the plug-in is not able to handle it alone. Since this case is still unlikely to occur in professional mixing the plug-in stays within the safety clipping range that will fit its intended use cases.

---

[2]maximal and minimal determined gain value

CHAPTER 5

Comparison

After finishing the prototype of the plug-in, comparison of the main functionality with the equivalent from Waves was performed to find out how likely the results can get with fitting parameters at the prototype version of the implemented gain adaption and also to evaluate the flexibility of the study plug-in and to gather thoughts about how to set own parameters and constants later on.

5.1   How

It is expected that it would not be effective to try different picks for the parameters manually and compare the outcomes as there are at least several parameters which can be adjusted to result in a lot of combinations. Therefore, this task is transferred to the computer program.

The scipy.optimize package for Python deals with optimisation tasks for example the algorithmic minimisation of a problem according to the result of a self defined function (see chapter 2.3).

Therefore a function to compare the resulting audio files after processed with the "Vocal Rider" and the study plug-in was developed. The resulting algorithm returns a value describing the deviation.

Through the optimisation process the parameter adjustments of the "Vocal Rider" differed between attempts but stayed constant in the process. The parameters of the study plug-in were changed continuously to achieve a preferably small deviation.

5.2 Circumstances

In order to let the optimisation algorithm amend different parts of the study plug-in, L_{goal}, RMS time, compress time, amplification time, gate threshold and lookahead delay are defined as variables. At start guessed values for each of the parameters are chosen and stored in an array which was altered through optimisation process and fed into the deviation function at every step of it.

For quantification of the deviation for the current parameter array the function sums up the squared difference between both resulting audio files for every sample. The result from the "Vocal Rider" was created in the DAW Pro Tools 11. The study implementation is called in the deviation function with the current parameter array.

When the optimisation search was done the outcome was fed into another function which displays the gain adaption from each plug-in in a collective graph. This gives a great overview of the result and remaining diversion of the implementations.

5.3 Approaches

Algorithmically the optimisation procedure tests small variations in the parameter array and watches the outcome. If the initial values were not set well, the differences calculated by the comparison function are small and the scipy.optimize.minimize algorithm did not know were to continue the search. With an unfavorable parameter set the gain curves are dissimilar and small variations might find a local minima in diversion but not the solution expected.

To obtain a good initial guess the scipy.optimize.brute method (see chapter2.3) was used. As just an approximation to the final solution was required large step sizes suiting the different ranges were set. Therefore the scipy.optimize.brute algorithm just had to deal with three to six possibilities per parameter.

The best initial values were fed into scipy.optimize.minimize. With good initialisation this method returned useful results meaning parameter arrays with likely outcome to the Waves plug-in. However the graphical representation of the 'Vocal Rider' gain adaption was not as expected: some gain values were jumping out of the line for single samples. After testing possible causes this could be explained by rounding errors of the DAW on export of audio files. While this error was unproblematic at most points in time,

it became visible with values near zero. Such small values could easily be doubled or increased even further by the errors which consequently affected the calculation of the local gain. At first this was solved this with a gate for small samples at gain calculation. Afterwards it got fixed anyhow as the algorithm was changed to use RMS averages for displaying the graphical evaluation.
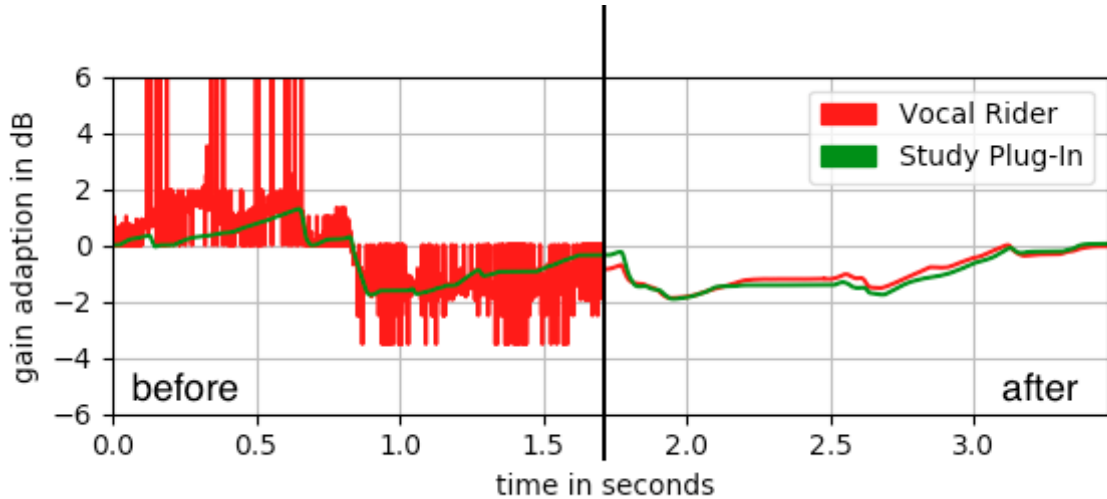
Figure 5.1: Optimisation result with (left) and w/o (right) rounding errrors

The behaviour of both plug-ins was compared by testing with optimised parameters on different vocal tracks in various length. The gain range of both plug-ins was set to +/-6dB. The *target* threshold of the "Vocal Rider" was adjusted to a reasonable value for all test files. The default and live component of the Vocal Rider were used. Additionally the plug-in from Waves wrote a gain automation and it read it through the export process for a further test file.

5.4   Results

The differences between the default "Vocal Rider" component, the live version and the self-automated were negligible. The automated plug-in provided in the same result as the default. It did not adapt for every sample but frequently after small consistent time steps, probably derivable from the communication between DAW and plug-in. The live

version created no critical difference to the default - as it could be expected. It just added a potentiometer to adjust the noise gate for live performances. Therefore, in tests further described results will always relate to the default plug-in version.

The first analysis result was that the Waves plug-in is not using a lookahead and consequently the parameter optimisation always resulted in a lookahead value of 0ms. This was to expect for the live component but not necessary for its standard use, as today's DAWs can easily compensate the produced delay. Obviously it was not considered as important by the authors in contrast to the plug-in design in this study as explained in section 3.7.

The second analysis result was that the Waves plug-in does not start fading to 0dB gain immedietly when it detects no vocal signal but only after several milliseconds of idle time, in contrast to the study plug-in's prototype. This feature was assumed to enables the plug-in to ignore small gaps in the vocals when the singer is breathing or doing a rhythmic break where a smoothed gain reset to 0dB is not necessary and can even be counterproductive (as the gain adaption needs a few milliseconds). Therefore it resulted in the implementation of a similar function for the study plug-in as it was assumed to work even better in combination with a small lookahead (see chapter 4.4).

Despite the described obvious conceptual differences the optimisation algorithm was able to bring both plug-ins to quite close results in terms of the gain adaption. The "Vocal Rider" seems to need a short time (about one second) to adjust its behaviour. This presumption resulted from different tests where both plug-ins were operating quite similar except for the first second. As a consequence the compare function was improved by ignoring the first second of the result.

Initially the study plug-in was able to imitate a section of the "Vocal Rider"s gain adaption for test files of longer duration, but not to imitate the gain adaption for the whole signal. Therefore causes for this behaviour were evaluated taken into account that the amplification of the "Vocal Rider" with slow attack time results in greater gain alteration than the compression of it with very fast attack time. It seemed as if compression was attenuated while amplification was not. The weakened compression reminded of standard compressor behaviour. Therefore an additional ratio parameter was implemented for the Python test version of the plug-in. This parameter reduces the value of the calculated gain adaption before it is smoothed and applied. Subsequently the new parameter was added to the parameter array for the optimisation algorithm. Simultaneously, the lookahead optimisation was removed as it always resulted in 0ms. The extended optimisation validated the assumption described above and got both gain
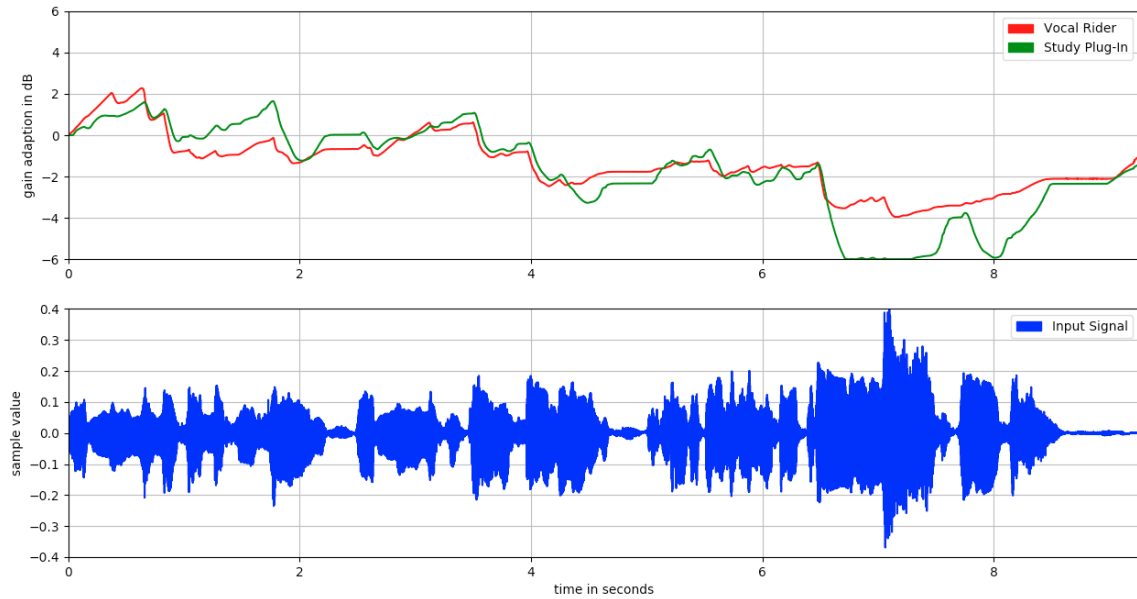
Figure 5.2: Gain Reduction comparison with interim optimisation result

curves close together for the whole test files (see Fig. 5.3).

In conclusion it can be stated that the "Vocal Rider" does gain adaption on the decreasing part quite similar to a compressor but without any colouring of the altered signal and a small ratio[1]. The study plug-in was able to get a similar result with a compress time of barely 100ms and a additional gain factor of 2/3 (1/3 slope[2]) before smoothing. In terms of amplification of the input the plug-in by Waves takes a comparatively long time period. It takes a attack time of about 2,4s for the study plug-in to imitate its performance. The gate threshold-to-L_{goal} relation of the study plug-in seems to fit the working method of the second test subject.

The comparison of the two plug-ins extended the main idea about some additional features and got some progress on the final set of the parameters to get to the main objective. Even though the results of the analysis of the "Vocal Rider" is not fitting the initial model of the study plug-in in all terms it still corroborated present work and leads to useful ideas.

---

[1]proportion between increment of input and output signal above the threshold

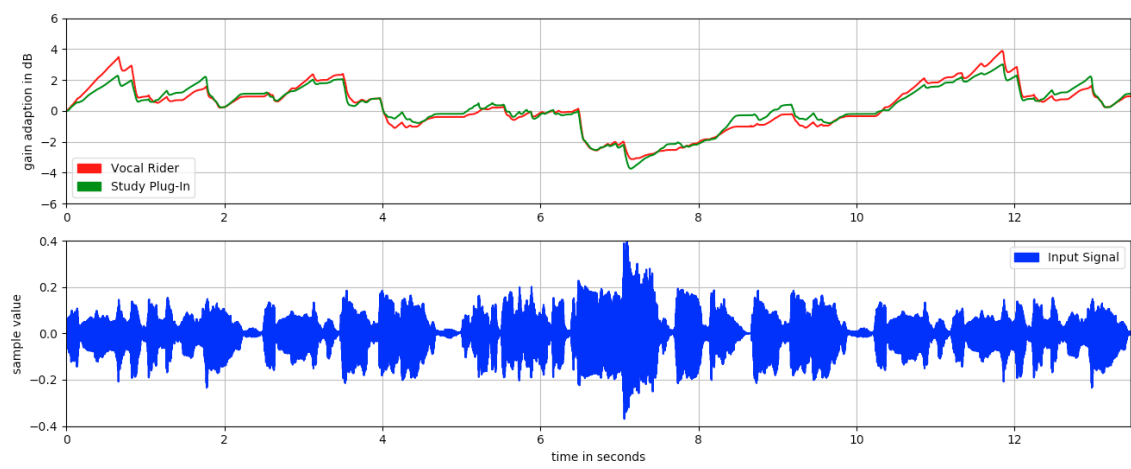[2]slope $= 1 - 1/\text{ratio}$

Figure 5.3: Gain Reduction comparison with final optimisation result

CHAPTER 6

Realization

6.1 Parameter Reduction

As the plug-in reached its final state and the comparison to the 'Vocal Rider' was completed, it was necessary to transfer some settable parameters to fixed constants. In the development phase it was useful to be able to change the important parameters while in runtime, for example the compress and amplification times. During the testing phase the best suiting parameter combinations were found. Small changes on the final set could be slightly better depending on the current circumstances but they are not required for a satisfactory result. Additionally, wrong application of the parameters settings could create a poor result. Therefore, considering the likely possibilities of a user not knowing what each of these parameters effects, it seemed to be advantageous to hide those parameter values, invisible at the UI.

This parameter reduction was applied to the lookahead delay, compress time, amplification time, RMS time, idle time, the side chain time constants and as already described, the gate. With the experience from testing the plug-in using a lot of different settings under various circumstances the default choices for the parameters already got to reasonable values.

With a inappropriate lookahead delay the gain adaption could happen to start to early or it has no noticeable effect. In the process of testing it commuted in at 60ms of delay as a good default value. For the parameter reduction it got set to 50ms as this resulted in the best outcome at additional test with special precision on the lookahead delay. A further test with a comparison of the drawn automation and the vocal signal waveform validated the new setting (see Fig. 4.2, 6.2).

The RMS averaging time should not be too long as this adds up on the total reaction time of the plug-in. On the other hand a minimum time window length is needed to calculate a useful average. In the beginning of development it was set to 60ms. The comparison to the Waves "Vocal Rider" resulted in very small time windows (down to

29

8ms) for the averaging at this plug-in, to imitate the outcome. As this seemed a bit fast in consideration of previous experiences, it got into further tests with altering RMS time in collaboration with changing compress and amplification times. As result the RMS time is finally set to 30ms.

With the initialisation of two different attack times for compression and expansion it was manifested that the expansion time should be greater than the compression time. Still the final settings were not yet set. So again, the comparison to the 'Vocal Rider' served as an inspiration. Setting an expansion time above two seconds it was not reacting fast enough to achieve the planned results, especially compared to the compression. As consequence the expansion time came down to 1500ms for the final build.

The compression time settings derived from the comparison with 'Vocal Rider' were similar to a slow compressor and therefore still to fast to fit the idea of this study. Thus, it had to be tested independently. This lastly resulted in a compression time of 600ms working out and, even though it was not planned, a slope of 1/3 at gain reduction. The reason for a additional slope were some vocal parts which got their level to low after compression. With a slope only at compression the gain adaption is balanced.

The idle time was set after focussing on the small gaps between phrases on vocal tracks. As it should be long enough to endure most of these gaps it is finally set to 500ms. A longer duration for the main functionality of the plug-in seemed unreasonable as it mainly extends the part after a vocal signal where the plug-in would just amplify noise. For the side chain input the plug-in takes a idle time set to two seconds to be able to ignore instrumental breaks. With a shorter setting it still results in decreasing vocal level while those breaks are happening. Furthermore it uses the same RMS time like the main input and smoothes the resulting gain similar slow to the amplification time with a time coefficient adjusted to 1600ms.

The gate just adapts with L_{goal} as described in 4.2 and is not settable in the UI.

6.2   Python

Development was not started with a final blueprint for the plug-in. Especially in the beginning several ideas on the basic algorithm, the gate or loudness detection were tested. In consequence the code had to be rearranged often. So Python came in handy as it focuses on code readability. In Python code there are fewer steps necessary to write the

same algorithm as for example in C++. Nevertheless, the plug-in was finally written in C++ (reasons see below).

Furthermore, Python provides various packages which extend its scope by useful features. For example the matplotlib.pyplot[12] plotting framework enables to draw graphs of results easily. This was especially useful for testing on comparison results (see chapter 5) and the filter implementation.

To test the filter class and cutoff frequencies, different signals with frequencies between 0 and 20000Hz passed through both filters and the resulting amplitudes were plotted in a graph via pyplot. The final algorithm results in a descent graph (Fig. 2.2). To test the C++ version of the filter subsequently the results of the same input with the previously tested Python implementation were compared.

The numpy[13] package was essential for mathematical operations and the scipy[6] tools were useful in terms of audio handling and optimization (see chapter 2.3). For this study Python version 3.6 was used.

Nevertheless Python was not the final choice for the plug-in as the C++ based JUCE framework offers a convenient predefined interface for audio plug-ins as well as the ability of fast processing due to the hardware-oriented C++ language. Speed of calculations could be crucial for real-time audio processing.

To make use of the scipy.optimize package at the comparison, the code of the plug-in's pototype was primarily transferred from C++ back to Python.

6.3   JUCE Framework And C++

JUCE represents a cross-platform framework for audio applications based on C++. Its main advantage is that it contains the functions needed for compiling to a VST[1] or AU[2] plug-in. Therefore, the main focus could stay on the algorithm of the plug-in during development. The JUCE audio plug-in template can be easily extended with a simple UI with sliders for the parameters of the algorithm. This is useful for testing the effects of individual parameter values. JUCE covers much of the communication with the DAW. Mostly this was fitting to the study plan and for this reason only a few parts in which the plug-in had special needs had to be replaced.

---

[1]Virtual Studio Technology plug-in architectur provided by Steinberg
[2]Audio Unit plug-in architectur provided by Apple

Input for a JUCE plug-in is transferred in separate buffer blocks. Dependent on the current set of the buffer size the plug-in will receive a buffer block of a known amount of samples. This buffer block will be processed using the method processBlock(). Inside the method each of the previously described work steps (see chapter 3, 4) will be performed directly or by calling a responsible method. Subsequently the processed samples will be written back into the current buffer and the next buffer will be handled.

To set correct coefficients for the filters at possibly varying sample rate f_s, the coefficients are initialised at every plug-in startup in the JUCE method "prepareToPlay" with the current f_s of the integrating DAW.

The lookahead delay is realised by a ring buffer with two pointers at different locations: One write pointer to write the current sample transferred from the DAW into the buffer which is also used to determine the gain adaption and one read pointer ahead of it which is pointing on the sample that will be multiplied with the determined gain (see Fig. 6.1). For correct realization, the gap between both pointers is as large as the set samples of the lookahead (converted from ms) and filled with zeros at the initialisation of the plug-in.

```
void  AutoVocalCtrlAudioProcessor :: updateDelay ()
{
    int  delayInSamples = msToSamples(* delayLength );
    delayReadPos = ( int )( delayWritePos − delayInSamples
    + delayBufferLength ) % delayBufferLength ;
    setLatencySamples ( delayInSamples );
}
```

When a pointer hits the end of the buffer it is set back to the start (see code example below) which leads to the imitation of a ring.

```
if (++dpr >= delayBufferLength )
        dpr = 0;
if (++dpw >= delayBufferLength )
        dpw = 0;
```

In order to avoid that the whole plug-in's output is delayed the setLatencySamples() (see first code example) method from JUCE was used to communicate the resulting

delay with the embedding DAW. Therefore a correct working DAW will send the signal earlier to the plug-in and the output remains at the correct position despite the lookahead.

For the development a fader in the UI was realised to adjust the lookahead at runtime, but in the final build the amount of samples is constant as the plug-in is designed to be as simple to work with as possible.
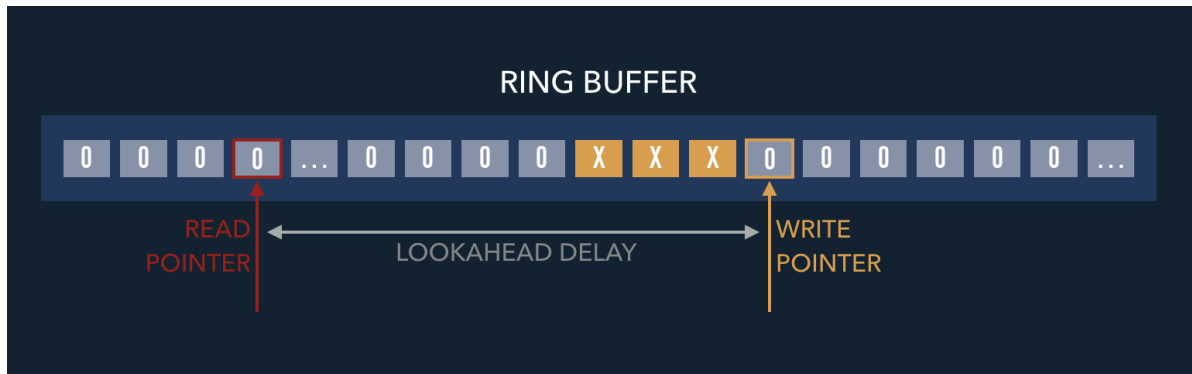

Figure 6.1: Initialized ring buffer with read and write pointers

In principle the implementation of automations is supported by the JUCE framework as well and it can be very simple to implement a standard automation process. The usual case would be a user interacting with the UI and the plug-in communicating these changes to the DAW on write mode. On automation read mode the changes are send back to the plug-in during playback and it does its normal processing chain with changing parameters with the user drawn timing. The study plug-in was a special case and thus, more difficult to realise the automation. In case of the study plug-in the write process of the automation should not be influenced by a user at intended use. Furthermore it should just receive and multiply the adapted gain from the drawn automation at read mode.

Like in the Waves plug-in a button was implemented at the UI to switch between read and write mode. Therefore the plug-in always knows if it needs to adapt the gain itself. The read mode implementation was performed quite fast as it just bypasses the regular calculations and multiplies the gain value set by the DAW with the current sample. The communication between DAW and plug-in works well on this part with the predefined parameter class from JUCE.

The write mode created more problems: The plug-in adapts the gain value at write

mode for every sample. When this would be communicated to the DAW like in normal automation process it has to draw at least 44100 adapted gain values each second. This easily overtaxes a DAW and would be more accurate as it needs to be for a acceptable result. The DAW expects just a few changes per second for the automation as its normal use case (user modifies parameter at UI) would generate. It follows that the drawn automation had to be simplified.

At the first attempts automation changes were only communicated when the gain adaption changes its direction (compression/amplification) or when the difference to the last drawn automation point became crucial. With this procedure the output in the DAW would have been acceptable but at some spots it made incorrect jumps. These jumps occured in duration of at most 2 samples but they were still unwanted. Due to involvement of several devices in this process the real cause of the jumps could not be figured out. Different timings and positions for the communication to the DAW and also change of the parameter values were tested until the final solution was found (see chapter 4.1). There are still jumps in the curve but this is acceptable as the jumps occur rarely and gain difference is about at most 0.1dB. Additionally these jumps are now located at correct positions without smoothing, which is not essential for such small amounts of adaption.

Figure 6.2: Automation written by the study plug-in into a DAW

To simplify the selection of a fitting L_{goal} for a user, a input meter was implemented close to the L_{goal} slider and the current gain adaption. Using this slider it is easier to check whether the current setting works appropriately.

Linked on the L_{goal} detection algorithm a button was implemented accessible at the UI which switches the plug-in to L_{goal} detection mode.

Side chaining as implemented in chapter 4.3 is not part of the standard I/O[3] layout of JUCE but is supported since JUCE version 4.1. To realise another input, it was added

---

[3]I/O = input/output

34

in the BusesProperties() structure from the AudioProcessor class. As for the plug-in it is not necessary to have a side chain input, supported bus layouts had not to be changed. Therefore the embedding DAW is able to feed it with a signal but does not need to do so. For further testing in Logic Pro 9[4] the plug-in had to be revalidated for the DAW, before it accepted the new I/O layout. This was not necessary for previous algorithmic changes.

The side chain feature is a addition to the main plug-in but becomes no constant part of it as it is not automatically profitable in every use case. For this reason a button was implemented at the UI to toggle side chain integration on and off. On further term this is useful as different DAWs will handle an undefined (when no input channel/bus is chosen) side chain input in a dissimilar way. For example Logic Pro 9 will send an empty signal containing only zero values. This would effect the outcome of the plug-in when side chain integration is active as it would interpret the signal as a quiet backtrack. With the side chain activation button toggled off by default, there is no urgent need of a silence detection always running in the plug-ins side chain processing chain.

As consequence of the idle time being added to the side chain feature the UI was expanded with a pre-gain slider for the side chain input which adds up before the comparison to L_{goal}. Controlling the input level of the side chain signal was possible before via a level-controlled bus[5] send but now the mixing engineer is able to do it all at one place while checking the level meters for both signals right next to each other at the UI.

6.3.1 Store Settings

At startup the plug-in regularly initialises with settings that may fit to typical standard cases but always have to be checked and adjusted properly for the current audio tracks. As a mixing engineer could interrupt his/her work, there is need to get its latest settings saved or else it would initialise with the unchanged startup setting. This problem already effects the UI at runtime if it is closed and opened again. Even if the parameters for processing stay correct in this case, the UI could show wrong values at sliders if not correctly refreshed.

To prevent this error the getStateInformation() and setStateInformation() which will be called by the DAWs are filled with the necessary save commands.

---

[4]Logic Pro 9 is DAW developed by Apple
[5]A Bus is a single path for multiple audio sources to be routed and processed.

```
void AutoVocalCtrlAudioProcessor::setStateInformation (const
void* data, int sizeInBytes)
{
    ScopedPointer<XmlElement> xmlState (getXmlFromBinary (data,
    sizeInBytes));
    if (xmlState != nullptr)
    {
        if (xmlState->hasTagName ("AutoVocalCtrl"))
        {
            *sc = (bool) xmlState
            ->getBoolAttribute ("sc", false);
            *read = (bool) xmlState
            ->getBoolAttribute ("read", false);
            ...
            *loudnessGoal = (float) xmlState
            ->getDoubleAttribute ("loudnessGoal", -20.0);
            *gainRange = (float) xmlState
            ->getDoubleAttribute ("gainRange", 6.0);
            ...
        }
    }
    ...
}
```

The plug-in creates a new xml file with the values of all settable parameters in the
getStateInformation() method and reads the xml file for parameter adjustment in the
setStateInformation() method. Finally the setStateInformation() method notices the
UI to refresh the according parameter sliders.

As DAWs are calling those methods at different timing but always at startup and before
the shutdown of the plug-in or the UI, all user-made made adjustments are saved and
regained at startup. Additional this procedure could save and regain presets for the
plug-in which can be used in other mixing session for example as orientation for the
parameter adjustment.

6.3.2 UI Design

The main focus of this study was the algorithm behind the plug-in. Nevertheless it was the intension that a user should be able to work with the plug-in without reading a manual. Therefore the JUCE dummy UI was refactored after parameter reduction.

The main problem of the previous UI was the lack of the possibility to differ between sliders for parameter adjustments and elements which were displaying inside information of the plug-in (see Fig. 6.3). Furthermore, the slider labels and value indicators were too small to display in complete length, while additionally confusing by their large number.
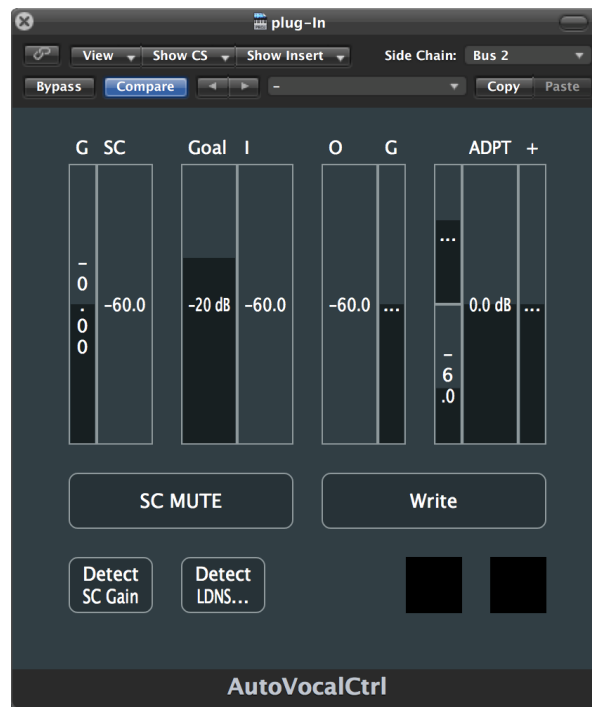
Figure 6.3: UI after parameter reduction

The main challenge in the UI re-design was highlighting of usable elements. To differentiate from only informational parts, all the changeable components are coloured red. The colour is chosen as it strongly diverges from its complementary colour green which is the plug-in's main colour (see Fig. 6.4). Additionally only the usable sliders and buttons got a coloured frame instead of the plain black outline of the pure metering components.

Figure 6.4: Final UI in Logic Pro 9

In the new design the gain range and loudness goal are set via red triangles ((1) in Fig. 6.4) pointing on the corresponding level/gain meter. This was done to let the user recognize the connection between these UI elements. L_{goal} should be set in considerati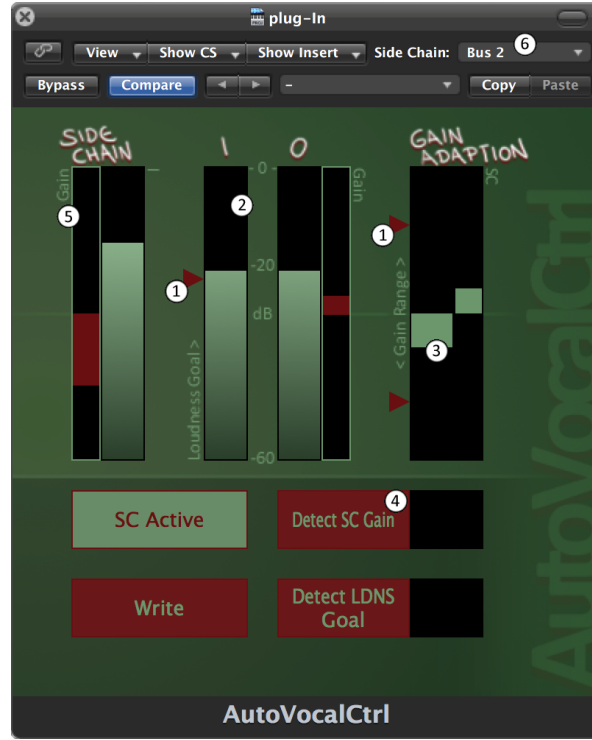on of the input level meter (2) right next to it and the gain range is directly modifying the maximal and minimal values of the connected gain reduction meter (3).

The three meters which are displaying the I/O levels (input, side chain input, output) are next to each other and visualising their value with a rising bar where a colour gradient is applied to. Therefore the green tone of the bar is getting brighter with increasing signal level to support a fast assessment of the user.

All UI elements excluding the buttons are divided in different sections: The I (input), O (output), side chain and gain adaption section, containing the related components. To indicate which task the single components fulfil, they have an additional label, on the side which is not adjoining to another UI element.

When the Side Chain feature is disabled, the corresponding gain detection button (4) and the side chain input gain slider (5) become disabled too, as they would have no effect. This can be recognized by the user as the red colour of these UI elements gets greyer and the slider loses its green frame which previously signalled a possible interac-

tion. The components which only display information are disabled for user interaction by default.

For most of the slider elements it is not necessary for the user to know the exact value but to get an approximation and therefore a feeling about what is happening. Clear slider values often seduce users to adjust sliders to pretty numbers (e.g. rounding to 5 or repeated digits) instead of only observing the change of the outcome. As consequence most of the slider values are removed in the new design. An orientation for the input and output meters is still located next to them but in a light green like the element labels in order to not distract from the main information.

The side chain input (6) is chosen in the embedding DAW and therefore not to set in the plug-in's UI.

6.4 Test Environment

The JUCE framework allows two ways to run the plug-in. The fastest way is to build the plug-in as standalone application which can be done directly from the IDE[6]. The plug-in is starting directly and the main input and output channels could be chosen. This is well suited for testing for small bug fixes or visual changes. The standalone procedure has its limitation in terms of e.g. a side chain input as it is not embedded in an surrounding DAW. For this case JUCE offers the Audio Plugin Host as solution. The Audio Plugin Host can host different plug-ins simultaneously and visualises all inputs and outputs. It is enabling connections between I/0 ports and the currently active audio interface of the operating computer. The advantage compared to a real DAW is that debugging output is provided during runtime.

It was necessary to test the plug-in in a real DAW for a realistic environment and to use all considered features, for instance writing an automation or comfortably feeding a real backtrack into the side chain input. Logic Pro 9 was used to run the study plug-in for the reason that it works with AU plug-ins which are per default supported by JUCE. Using the custom UI it was still possible to change calculation parameters at runtime. For tests inside the DAW professional recorded audio tracks are needed additionally. For testing side chain adaption it was necessary to work with full musical compositions and not only to remain on vocal pieces. Therefore fitting multitrack projects from an

---

[6]integrated development environment

online library[14] of free educational use were utilized (same projects as used in side chain evaluation, chapter 7).

Before all features came together, basic algorithmic tests were performed. Python was used because of its simplicity and helpful visual features. Therefore different approaches could be tested efficiently and visualised whether they have performed their task correctly.

CHAPTER 7

Evaluation of the Side Chain Feature

The plug-in was continuously improved during the testing period. One significant improvement was the side chain feature which will compensate for a whole further work step of the mixing engineer when it is functioning as intended. To proof the advantages of this feature a listening test was performed after the implementation. In the best case this could proof that there is no critical difference to a track volume automation additional to the main plug-in, drawn by a mixing engineer who knows about the level changes in the backtrack. At least it should verify the feature by demonstrating its advantages to the prototype in different circumstances.

7.1   Test Conditions

To receive results fitting the question about side chain advantages at the performed test, participants had to compare the backtrack-to-vocal level relation for audio files with the plug-in effecting the vocal gain with and without the side chain feature enabled. Additionally at every comparison there was a reference track where gain automations were drawn with oracle knowledge about the level change of the according backtrack. This track also had the plug-in inserted but without the side chain feature supporting. To establish the equivalence of the test scores an forth audio track with intensional divergent backtrack-to-vocal relation was added to perform as an anchor.

In the first 10 test sections 18 participants had to listen the reference track as aspired result and subsequently compare the backtrack-to-vocal level relation with four test items. The test items were mixes of the same song snippet but with varying gain adaption on the vocal tracks: the anchor with intensional divergent gain, a mix with the plug-in while the side chain feature was enabled, a mix altered by the plug-in without a side

chain input and an unmodified copy of the reference. While comparing these test items with the first audio file declared as reference the participants did not know about the differences in the creation of the other four. The task was to rank the test items in their similarity of backtrack-to-vocal level relation to the reference on a percentage scale.

As the main plug-in is designed to push comprehensibility and clarity of the vocals in the mix, the question remained if an additional gain from the side chain feature could influence this in an unfavorable way. Consequently four additional test sections appended the procedure where comprehensibility and clarity where rated by the participants.

For the realisation of a valid test a important thing to do was picking usable song parts for the comparison. Therefore three different multitrack projects (see chapter 6.4, [14]) were used as source for the audio snippets. As the benefit of the side chain feature is the adaption on backtrack level changes, at the beginning of all three songs the study plug-ins were initialised with settings to fit the local backtrack level. During this process L_{goal} and side chain input-gain were set by the detection algorithm of the study plug-in. In the middle of the musical piece the backtrack level was changed due to a level automation. This was done in addition to the regular level changes of the backtrack to intensify the differences of the varying plug-in setting to be tested. The reference vocals with oracle knowledge got the same automations as the backtrack and where supplementary levelled to fit the backtrack at all the different song parts. The study plug-in with the side chain feature enabled had to adapt to the backtrack level changes itself while the same plug-in operated on the third vocal variation without the additional side chain informations. The level of the anchor was set manually in order to not resemble the reference track.

To have a wide range of circumstances tested for the plug-ins three to four different snippets were taken from each of the songs. These snippets are containing song parts before and after the backtrack level automation just like clips from the exact part where the level change is happening. Additionally the test was extended by snippets of song parts with vocals during instrumental breaks as these circumstances were especially difficult to handle at side chain feature development.

While the backtrack-to-vocal level relations were tested with all different kinds of snippets, the comprehensibility and clarity comparison was made with clips where the plug-in with the disabled side chain feature was adjusted in its output gain according to the current backtrack level to have a fair comparison. A single test was deviating from this scheme and therefore evaluated independently.

The test was realised in a HTML5 JavaScript browser application build with beaqlejs-

0.2[15] using the MUSHRA test class. This framework already contained the necessary evaluation sliders and the functionality for the playback of the test items. To avoid confusion about comparison criteria the randomisation of test sections was disabled. Therefore the 10 backtrack-to-vocal level relation tests are followed by four comprehensibility and clarity tests. In order to receive independent results on each test the test items for evaluation are at randomised positions differing for each of the tests. During the tests the participants were listening to the audio clips on professional studio equipment in a quiet room for unadulterated results.

The results of test sessions were transferred into a table and subsequently evaluated.

7.2   Vocal Level

Due to the initialisation of the different set plug-ins for the test vocal tracks, the audio clips from the beginning of the songs have similar backtrack-to-vocal relations. Especially the oracle track and the plug-in without side chain have the same outcome at start. Despite the additional influence of the side chain algorithm the plug-in with the side chain feature enabled was ranked very close to the other two. The backtrack-to-vocal relation was judged not differing on average from the plug-in without side chain. As follows the side chain feature seems to have no unfavorable influence on the level outcome in comparison to a well gained (using the output gain to compensate the level gap between backtrack and vocals) plug-in without side chain.

At a backtrack level change the side chain feature has to adapt the level of the vocals and the plug-in without side chain will stay on the same output gain. As expected the vocal level results at backtrack level changes from the plug-in with side chain enabled were rated closer to the optimal reference. In average the side chain feature got the according test snippets rated 29% closer to 100% similarity with the reference as the plug-in without this feature. This average rating is very close to the rating of the reference copy. As consequence it is to conclude that the side chain feature leads to gain automations with similar results like a mixing engineer would draw. Additionally a advantage over the plug-in without a side chain input was observed.

The audio clips after backtrack level change, with the vocal level already adapted by the side chain feature, received ratings with even more advantage over the the plug-in without a side chain input. On average the side chain enabled plug-in was 35% closer

to the optimal result for each of those test files.

At instrumental breaks the ratings for the side chain adaption where about 5% behind the total average but still better than the plug-in without side chain.

In total the similarity to the reference in terms of backtrack-to-vocal relation was rated 62% for the plug-in without side chain. The side chain enabled plug-in got with an average value of 84% very close to the copy of the optimal reference which was rated with 87% similarity to itself on average.

7.3 Comprehensibility

Enhancing the comprehensibility and clarity of the vocals is an effect of the prototype plug-in in a professional mix as the vocals with lesser dynamics have fewer or no parts which are getting lost in a backtrack due to sectional lower loudness. Therefore it was important to test if the advanced comprehensibility and clarity of the vocals is decreased by the side chain feature which alters the outcome with an additional gain. As the relevant testing was done with audio snippets from the beginning of the song where the plug-in without side chain was initialised for the backtrack level, the question was if the additional side chain gain would have a bad effect or get similar results in comprehensibility and clarity.

In the according tests the participants had to rate comprehensibility and clarity also in comparison to a reference but with the exception that better clarity was rated with 100% too. As result of the tests both main testimonials where rated for 90% clarity in average. As conclusion the side chain feature does not seem to have a bad influence on clarity and comprehensibility as it got the same average test results as the plug-in without this feature.

7.4 Test Results

All in all the side chain feature proofed itself well in the accomplished testing. The clarity and comprehensibility of vocals does not seem to be effected in unfavorable manners by the enabled feature. In terms of backtrack level adaption it got the expected

advantage, over the plug-in without this feature, verified (see table 7.1) and even rated close to the optimal drawn level automation with oracle knowledge.

	comprehensibility	backtrack-to-vocal level relation		
	before level change	before level change	during level change	after level change
side chain advantage	0%	0%	29%	35%

Table 7.1: Side chain advantages for different song parts

In terms of the instrumental breaks the side chain feature could not continue its lead completely but still stays in advantage over the plug-in without these additional information. This is not the best result it could get but due to the initial difficulties it is still ranked as a success. While in the first versions of the side chain implementation it lowered the vocal level substantially during instrumental breaks, in the later versions there is no or only a very small level change observed under correct settings. Additionally, this is not an argument against using the side chain feature for those song parts, as in average it is still rated as advantage over the plug-in without the side chain feature, also by just considering instrumental breaks.

As the main backtrack level changes in the accomplished tests were artificially induced its results may not reflect for all possible circumstances. But as this supported the oracle knowledge of the reference track and made differences for the participants' ability to hear more clearly it seemed to be a good choice to do so. Additionally it is questioned whether every participant would be able to hear the differences if they had not been artificially induced differing enough.

The test results could be corroborated with additional participants but the 18 which had done the testing already revealed a plausible trend (see table 7.2).

	plug-in	plug-in + sc	plug-in + oracle gain (copy of reference)
similarity to reference on average	68%	86%	89%

Table 7.2: Similarity to reference of test candidates

Thereby it was observed that the side chain feature got near the optimal result and in any case validated its advantages over a plug-in without this feature. It was ranked better than its direct adversary in most of the cases (see Fig. 7.1).

As conclusion it seems to work properly and has legitimacy to be a part of the final product.
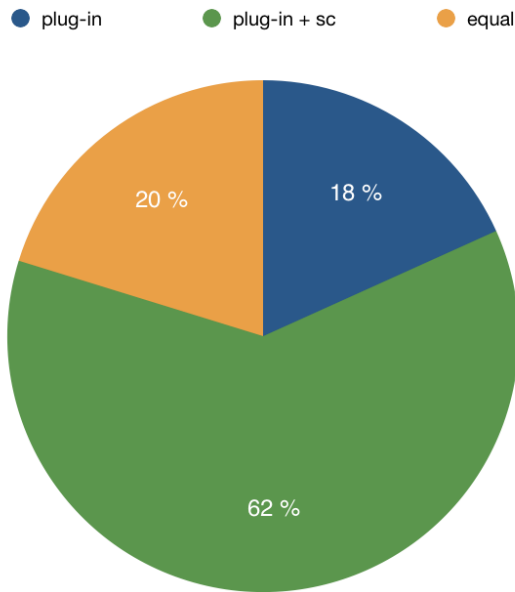

Figure 7.1: Plug-in setting with better rating (all tests)

CHAPTER 8

Future Improvements

The study plug-in meets the requirement of the objective but it could still be improved. For example, parameter detection is an optional feature which could be useful to set the plug-in correctly, especially at first interactions of a user. As described in chapter 4.2 and 4.3, it is only possible to calculate parameter values 'online', i.e. during playback of the song. Offline calculation is not implemented yet which would be significantly faster. As one purpose of the plug-in is to save time for the mixing engineer, it would be helpful to enhance the feature with offline calculations in future development. When offline calculation would be available it would be possible to measure the perceived signal level by using the loudness algorithm[4] which may further improve the plug-in's outcome.

The plug-in supports stereo input but processes the channels independently. For stereo recorded vocals, e.g. background choirs, it would be advantageous to have the option of linked calculations and equal gain adaption.

Another reasonable future improvement would be to extend the scope of application. Currently it is designed to work with vocal signals but other recordings may show similar problems e.g. recordings of the bass guitar. Therefore it would be useful to have the option to chose between different presets for the main constants of the plug-in's calculation, fitting the different signals.

Additional to possible algorithmic improvements the plug-in could be amended based on expanded testing for increased stability and correctness of the results.

CHAPTER 9

Summary

The objective of this study was to develop a plug-in for digital audio workstations which reduces long term dynamics of vocal signals in an automatic way to reduce manual work of mixing engineers.

The project started with the creation of a prototype which had the intended functionality implemented: A filter section for simplified imitation of human loudness perception introduces the algorithm of the prototype. Subsequently, a root mean square (RMS) average of the input signal is calculated. A following gate filters audio samples below an automatically adjusted threshold to disable the gain adaption for signal sections without vocals. After passing this gate the difference between loudness goal and current RMS average is determined. This difference is smoothed within a predefined time window and thus results in the final gain. Time window length is changing in dependency of a increasing or decreasing gain adaption (compression or amplification of signal). As last step, the determined gain is multiplied with the input signal.

The first version of the prototype was improved by adding additional features: Writing a gain automation into the DAW was made possible as well as holding the gain adaption in place for small breathing gaps within vocal signals. As option automatic parameter settings were implemented for loudness goal and side chain input-gain.

As further amendment to these improvements, a side chain feature was added extending the use of the plug-in by adjusting the vocal signal output level to the side chain input level (in anticipated use: the instrumental backtrack).

The prototype's outcome was compared to the commercially available plug-in "Vocal Rider"(Waves Audio Ltd., Knoxville, TN, USA) which is a plug-in with similar functions. For this purpose the study algorithm was implemented in Python and the parameters were algorithmically optimized for the smallest possible deviation between outcome of the two plug-ins. The result of the comparison lead to further improvements of the prototype function and of final parameter settings.

The final version of the plug-in was realized in the JUCE framework, which turned out

as a good choice: It provided basic but solid functionality to facilitate development of an audio plug-in, which otherwise would have been build up from scratch in a time-consuming manner.

Finally, the advantages of the side chain feature were evaluated by hearing tests with independent participants indicating that the prototype performed substantially better with enabled side chain feature.

In conclusion, a plug-in prototype for digital audio workstations facilitating automatic decreasement of long term dynamics for vocal signals in music production was developed. During the study the plug-in prototype was continuously improved and evaluated in terms of features, algorithms and parameter settings. Thus, the study has lead to a product ready to fulfil its intended task in daily music production. Nevertheless, options for further improvement were identified and described.

Bibliography

[1] waves.com. 'Vocal Rider', 2018. [Online]. Available:
    waves.com/plugins/vocal-rider [Accessed: 9-May-2018]. ii, 2

[2] en.wikipedia.org. 'Loudness', 2018. [Online]. Available:
    en.wikipedia.org/wiki/Loudness [Accessed: 9-May-2018]. v, 3

[3] *Vocal Rider User Guide.* Waves, 2009. v, 5

[4] *Recommendation ITU-R BS.1770-4 - Algorithms to measure audio programme
    loudness and true-peak audio level.* International Telecommunication Union, 2015.
    2, 3, 4, 7, 8, 10, 17, 47

[5] F. A. Everest. *The Master Handbook of Acoustics.* McGraw-Hill, 2001. 4

[6] scipy.org, 2018. [Online]. [Accessed: 9-May-2018]. 6, 31

[7] scipy.org. 'scipy.optimize.brute', 2014. [Online]. Available:
    docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.optimize.brute.html
    [Accessed: 9-May-2018]. 6

[8] scipy.org. 'scipy.optimize.minimize', 2018. [Online]. Available:
    docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html
    [Accessed: 9-May-2018]. 6

[9] Udo Zölzer, editor. *DAFX: Digital Audio Effects*, chapter 2 - Filters and delays.
    John Wiley & Sons Lid., second edition, 2011. 8

[10] en.wikipedia.org. 'Root mean square', 2018. [Online]. Available:
    en.wikipedia.org/wiki/Root_mean_square [Accessed: 9-May-2018]. 10

[11] Udo Zölzer. *Digital Audio Signal Processing.* John Wiley & Sons Lid., second
    edition, 2008. 10, 11

[12] matplotlib.org. 'pyplot', 2017. [Online]. Available:
    matplotlib.org/api/pyplot_api.html [Accessed: 9-May-2018]. 31

[13] numpy.org, 2018. [Online]. [Accessed: 9-May-2018]. 31

[14] cambridge mt.com. 'the 'Mixing Secrets' Free Multitrack Download Library', [Online]. Available: cambridge-mt.com/ms-mtk.html [Accessed: 9-May-2018]. 40, 42

[15] U. Zölzer S. Kraft. BeaqleJS: HTML5 and JavaScript based Framework for the Subjective Evaluation of Audio Quality. Karlsruhe, Germany, 2014. Linux Audio Conference. 43

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

_____   _____
Ort, Datum   Unterschrift

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel — insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen — benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

_____   _____
Ort, Datum   Unterschrift