Real-Time Automatic Gain Control for Singing Voice Applications

Bachelor Thesis in the course Informatik

Author:

Nils Heine

Matriculation Number: 6703759

Signal Processing / Signalverarbeitung

Department of Computer Science, MIN Faculty

First Reviewer: Prof. Dr.-Ing. Timo Gerkmann

Second Reviewer: Dr.-Ing. Martin Krawczyk-Becker

Hamburg, October 19, 2017

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Motivation

When a sound engineer is editing a song he/she wants that 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 great 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. But the result 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. Obviously, this is a time consuming and monotonous task and therefore predestined to hand over to a machine.
Admittedly a professional studio singer can be able to reduce the unwanted dynamics as well but they will rarely succeed completely as a result of 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 can 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 singers. Consequently this problem will occur in many mixing sessions.

1.2 Objective

The idea 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 would save the engineer 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 a 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[1] '*algorithm to measure audio ... loudness*'. This algorithm is an important up-to-date 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 named "Vocal Rider"[2] was published by Waves[3], but the algorithm behind this plug-in is not published. The 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.
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] International Telecommunication Union, Recommendation ITU-R BS.1770-4 (10/2015)
[2] waves.com/plugins/vocal-rider
[3] Waves Audio Ltd. is company which is developing software audio effects for professional digital audio signal processing

Chapter 2

Basics

At the beginning of development ideas were tested in Python, later JUCE[1] framework based on C++ was used. The functionality of the plug-in was mainly tested in Logic Pro 9.

2.1 Python

Development was not started with a final blueprint for the plug-in. Especially at 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 program as for example in C++. Nevertheless, the plug-in was finally written in C++ (see below).
Furthermore, Python provides various packages which extend its scope by useful features. For example the matplotlib.pyplot[2] plotting framework enables to draw graphs of results. This was especially useful for testing on filter implementation (see chpt. 3.2) and subsequently comparing optimisation results later on (see chpt. 4). The numpy[3] package was essential for mathematical operations and the scipy[4] tools was very useful in terms of audio handling and optimization. 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 great predefined interface for audio plug-ins as well as the ability of fast processing due to the hardware-oriented C++ language. Speed of calculations can

[1] juce.com
[2] matplotlib.org/api/pyplot_api.html
[3] numpy.org
[4] scipy.org

be crucial for real-time audio processing.

2.2 JUCE framework and C++

JUCE is a cross-platform framework for audio applications based on C++. The main advantage for this study is that it contains the functions needed for compiling to a working VST[5] or AU[6] 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 parameters. 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 overwritten.

2.3 Test environment

The JUCE framework brings along two ways to run the plug-in. The fastest one is to build the plug-in as standalone which can be done directly from the IDE[7]. The plug-in is starting immediately and the main input and output channels could be chosen. This is perfect 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. But for this case JUCE has the Audio Plugin Host as solution. The Audio Plugin Host can host different plug-ins at the same time and visualises all inputs and outputs. It is enabling draw ing connections between those ports and the currently active audio interface of the operating computer. The advantage compared to a real DAW is that debugging output is provided through runtime.
Still it is reasonably 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

[5]Virtual Studio Technology plug-in architectur provided by Steinberg
[6]Audio Unit plug-in architectur provided by Apple
[7]integrated development environment

supported by JUCE. Due to the custom UI it was still possible to change calculation parameters at runtime.
For worthy tests in 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 a online library[8] of free educational use were obtained.
Before there was a plug-in 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.

[8] cambridge-mt.com/ms-mtk-newbies.htm

Chapter 3
Prototype

3.1 Overview

The basic prototype is working in five steps: filter, root mean square (RMS) calculation, gate, gain adaption, delay. Dependent on the current set of the buffer size the plug-in will receive buffer blocks of a known amount of samples. These buffer blocks will be processed stepwise using the method processBlock(). Inside this method each of the described work steps 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.
The processing chain starts with a low-cut and a high-shelf filter that will be applied sample wise on the incoming audio. The filters are a simplified mapping of the perception of sound pressure for human ears. 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[1] due to calculation time and real time capability.
After the filter procedure 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 will be determined in the calculation of the equivalent dB value. The plug-in is converting the linear audio samples into the logarithmic dB scale in order to display gain values and loudness goal (see 3.5/6) in dB in the user interface (UI) as it is the standard scale of DAWs. Thereby the executive sound engineer will intuitively know how to interpret and interact with the UI (see 5.8).
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 loudness goal (see 3.6 and 5.2). In this way the plug-in will not operate when it is fed with silence or irrelevant noise.
Next step after the gate is the gain adaption, i.e. the gated RMS value is compared to

[1] see ITU-R BS.1770-4

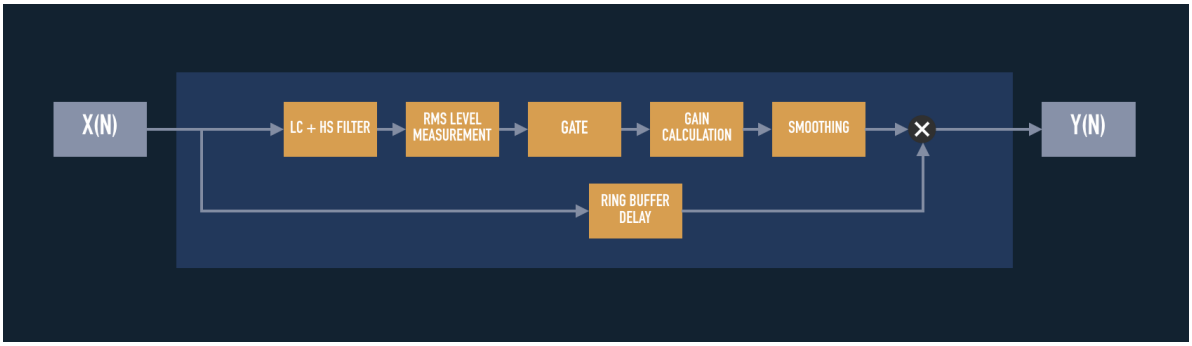

Figure 3.1: Prototype processing chain

the current loudness 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.
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.
During development most of the parameters described in the following sections were settable in a basic dummy UI. This was realised with the standard JUCE slider and button objects and used for tests and fast adjustments.

3.2 Filter

While gathering information on how to improve the plug-in the ITU-R BS.1770-4 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 human perception is of substantial interest for mixing a song, this was interesting for 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 the two filters. As described above, 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*'. 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. They are initialised at every plug-in startup in the JUCE method "prepareToPlay" with the current sample rate of the integrating DAW:

```
lowcut.setCoefficients(38.0, sampleRate, (1.0/2.0));
highshelf.setCoefficientsShelf(1681.0, sampleRate, 4.0);
```

The implementation is based on the biquad filter from 'DAFX: Digital Audio Effects'[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:

$b_0 = \frac{Q}{K^2Q+K+Q}$ $\quad b_1 = -\frac{2Q}{K^2Q+K+Q}$ $\quad b_2 = b_0$
$a_1 = \frac{2Q*(K^2-1)}{K^2Q+K+Q}$ $\quad a_2 = \frac{K^2Q-K+Q}{K^2Q+K+Q}$

Highshelf:

$b_0 = \frac{V_0+\sqrt{2V_0}K+K2}{1+\sqrt{2}K+K^2}$ $\quad b_1 = -\frac{2(K^2-V_0)}{1+\sqrt{2}K+K^2}$ $\quad b_2 = \frac{V_0-\sqrt{2V_0}K+K2}{1+\sqrt{2}K+K^2}$
$a_1 = \frac{2(K^2-1)}{1+\sqrt{2}K+K^2}$ $\quad a_2 = \frac{1-\sqrt{2}K+K^2}{1+\sqrt{2}K+K^2}$

Hence the loudness algorithm uses second order filters (with two delay memories) it works like this:

[2]DAFX: Digital Audio Effects, Second Edition. Edited by Udo Zölzer, 2011, John Wiley & Sons, Lid.

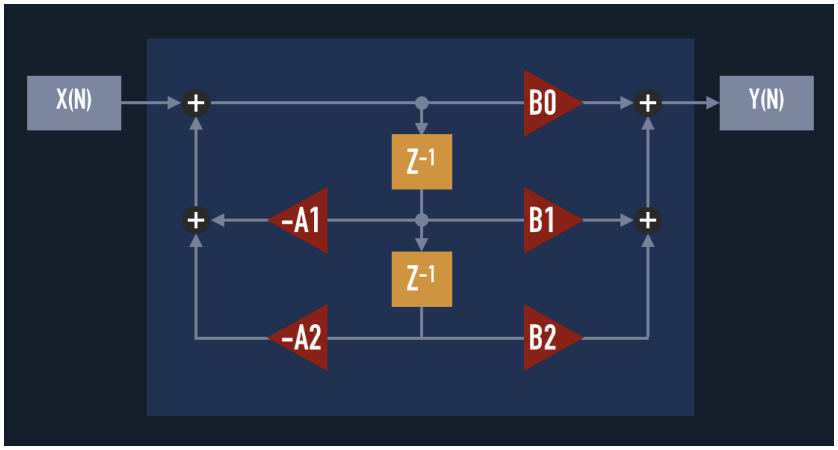

Figure 3.2: Second-order digital filter

It is the same as the implementation in the filter class:

```
double AutoVocalCtrlFilter::process(double sample)
{
    const double mid = sample - a1 * z_1 - a2 * z_2;
    const double out = b0 * mid + b1 * z_1 + b2 * z_2;

    z_2 = z_1;
    z_1 = mid;

    return out;
}
```

Before implementing in C++ the filter class was tested in Python. Therefore different signals with frequencies between 0 and 20000Hz were send through both filters and the resulting amplitudes were plotted in a graph via pyplot (see 2.1). The current algorithm results in a descent graph (Fig. 3.2). To test the C++ version of the filter the results of the same input with the previously tested Python implementation were compared. The 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:

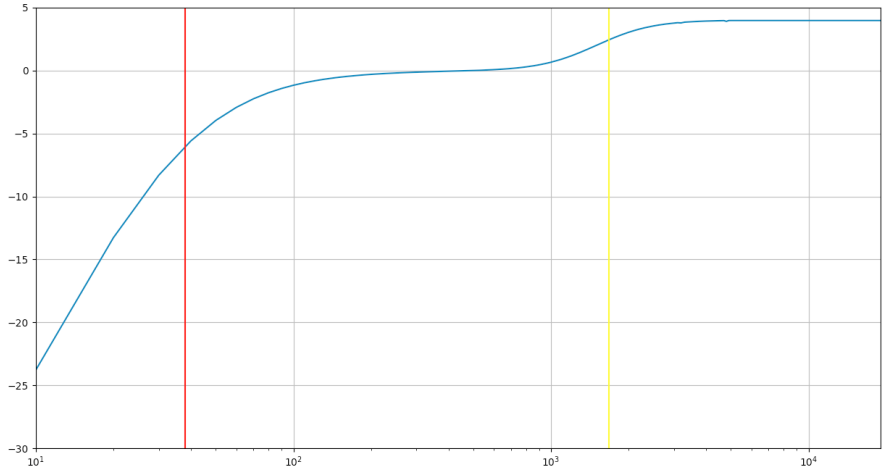

Figure 3.3: Plotted filter test

```
double updateFilterSample(double sample, AutoVocalCtrlFilter hs,
AutoVocalCtrlFilter lc)
{
    return hs.process(lc.process(sample));
}
```

The updateFilterSample method returned the filtered samples for further processing.

3.3 RMS

To measure the level of the audio signal at the current position the plug-in calculates a root mean square (RMS) value of the last 30 ms. This means it is calculating the root of the average from the squares of each sample in the set time window.
RMS calculation is used because it is part of the imitation of human perception and because it has the necessary time to calculate an average value as it does not need and does not want to react fast: 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*[3]'. This is also part of the ITU-R BS.1770-4 loudness standard. Additionally

[3]F. A. Everest, The Master Handbook of Acoustics, New York: McGraw-Hill, 2001.

the '*RMS is equal to the value of the direct current that would produce the same average power in a resistive load*[4]'.
The RMS implementation is based on 'Digital Audio Signal Processing' by Udo Zölzer[5] and can be performed in one line of code:

```
double updateRMS2(double sample, double last, double co)
{
    return (1. - co) * last + co * (sample * sample);
}
```

The filtered samples are used for this calculation and the result is written into the rms2 vector. The Co parameter 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 normal RMS procedure the final result is the square root of the value the plug-in results in. In this case the calculation can be skipped because it will be converted into the logarithmic dB scale in the next step and a square just changes the value of one coefficient in this process. The final RMS value will only appear as logarithmic dB but 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 3.6). They are determined with a formula by Zölzer (quell):

$$1.f - \exp(-2.2 * (1/currentSampleRate)/(ms/1000.))$$
$$= 1.f - \exp(-2.2 * (1/currentSampleRate)/s)$$

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

[4] wikipedia
[5] NOCH KORRIGIEREN

change will need the chosen amount of time to get from 10% of the final reaction to 90%. Illustrated in Fig. 3.4. The second part in the exponent $(1/currentSampleRate)/s)$ 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.

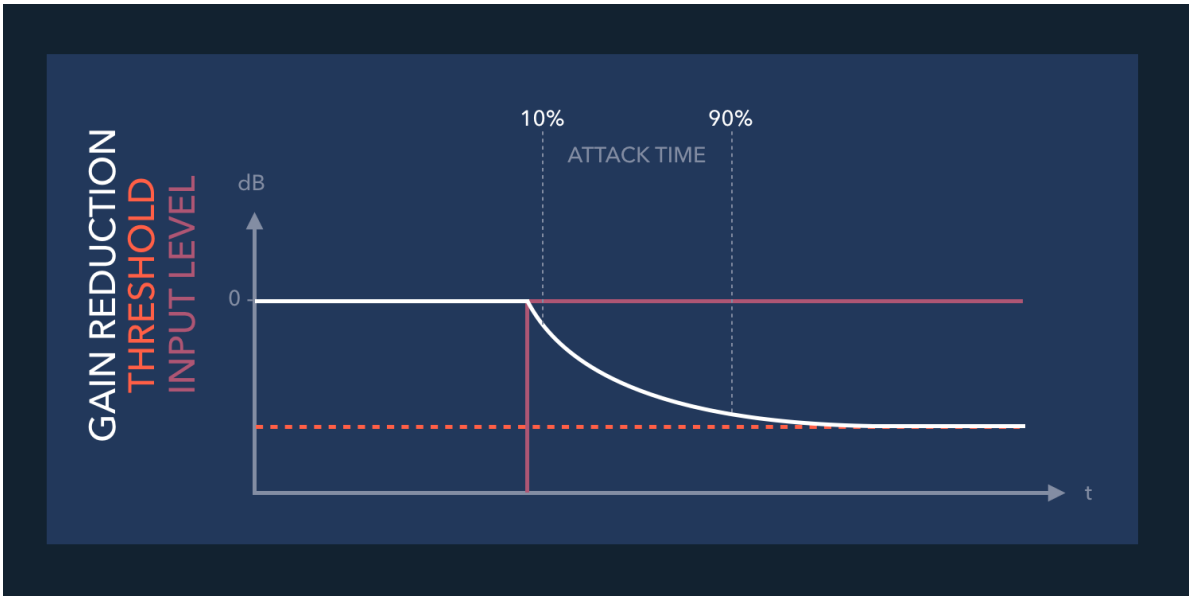

Figure 3.4: Attack time $t90 - t10$

3.5 Gate

The plug-in should operate while there are vocals 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 plug-in uses 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 loudnessGoal. As the threshold and the loudnessGoal are set in dB, the first step in the gate is to convert the transferred sample (rms2) into the logarithmic scale. It uses $10.0*$std::log10(rms2 $+1e-10$); instead of $20.0*$std::log10(rms $+1e-10$); because the rms2 is 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 loudnessGoal - gainRange. Thereby it is still possible to

use the whole gainRange for gain adaption and at the same time sort out the decay of the vocals. With this formula the threshold adjusts to the level of the vocals as the loudnessGoal is detected (see 5.2).
When RMS samples are not passing the gate and therefore being replaced by the loudnessGoal the plug-in adapts the gain to 0 (after a short period of time) because it has achieved its goal (see 3.6).

3.6 Gain

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

```
double updateGain(double sample, double lastGn)
{
    const double g = *loudnessGoal - sample;
    const double co = g < lastGn ? compressTCo:expandTCo;
    updateAutomation();
    return clipRange.clipValue((1 - co) * lastGn + co * g);
}
```

At first it computes the difference between the loudnessGoal and the current processed sample. The result is the gain factor that would be necessary to get it to the loudnessGoal (by multiplying in the linear number space).
Since the plug-in 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 is chosen by comparing the calculated gain factor g with the gain that the function had returned for the precious sample. If g is smaller than the last gain (lastGn) 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 compressTCo time coefficient. In contrast, when low level signals are amplified the ampTCo time coefficient is chosen. Different time coefficients for ampli-

fying 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 5.7).
When the gain is determined it will be converted back to the linear number space. Finally it just needs to be multiplied with the current sample.

3.7 Lookahead

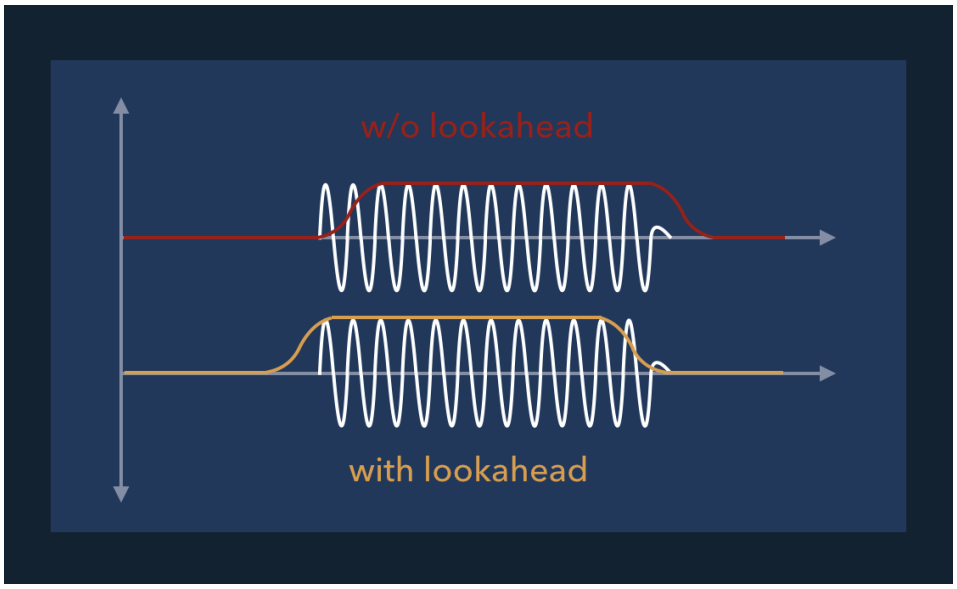

Figure 3.5: Gain adaption with and w/o lookahead

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 form 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 implemented which allows the plug-in to calculate the gain for the current samples while looking at future samples. This 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 buffer ahead of it which is pointing on the sample that will be multiplied with the determined gain. To make this possible 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.

```
...
gain[channel] = updateGain(updateGate(rms2[channel], newGate),
gain[channel]);
double g = pow(10, gain[channel]/20);
```

```
delayData[dpw] = channelData[sample];
...
const double o = delayData[dpr] * g;
...
channelData[sample] = o;
...
if (++dpr >= delayBufferLength)
        dpr = 0;
if (++dpw >= delayBufferLength)
        dpw = 0;
...
```

In order to avoid that the whole plug-ins 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. Therefore the lookahead time is chosen in adaption to the compress/amplification times and the tests with real vocal tracks (see 5.6).

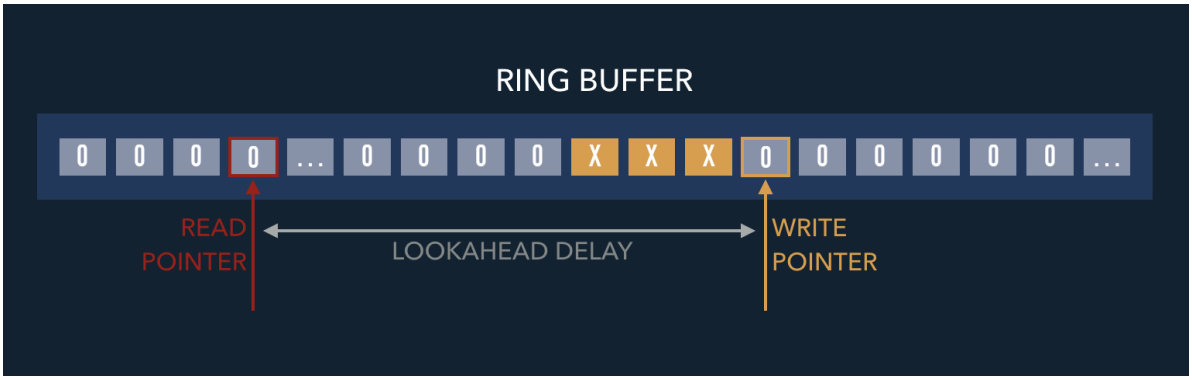

Figure 3.6: Initialized ring buffer with read and write pointers

Chapter 4

Comparison

After finishing the prototype of the plug-in comparison of the main functionality with the equivalent from WAVES was performed. It was to find out how likely the results can get with fitting parameters at the current version of the implemented gain adaption. Not only to see a important part was missing so far but also to find out how flexible the study plug-in is and to gather thoughts about how to set the parameters and constants later on.

4.1 How

It would not be very effective to try different picks for the parameters by hand and compare the outcome as there are at least a handful of parameters which can be adjusted to a huge amount of possible combinations. Therefore this task was given to the computer. Conveniently there is the scipy.optimize package for Python which deals with optimisation tasks for example the algorithmic minimisation of a problem according to the result of a self defined function.
To make use of this package the current code of the plug-in was primarily transferred from C++ back to Python were the previous algorithmic ideas were tested. After the Python duplicate was ready to use a function was written to compare the resulting audio files after processed with the "Vocal Rider" and the study plug-in. It 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.

4.2 Circumstances

In order to let the optimisation algorithm have the option of adjustment on different parts of the study plug-in, the loudnessGoal, RMS time, compress time, amplification time, gate and lookahead were declared as variables. At start a guessed values was set for each of the parameters and set them in a array which was altered through optimisation process and fed into the deviation function at every step of it.
For measuring the deviation for the current parameter array the function sums up the squared difference between both resulting audio files at every sample. The result from the "Vocal Rider" was therefore created in the DAW Pro Tools 11. The implementation from this study 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.

4.3 Approaches

The first tries unfortunately did not achieve a reasonable solution. Algorithmically the optimisation tests small variations in the parameter array and watches the outcome. If the initial guess is not well set, the differences are of very little amount in the comparison function and the scipy.optimize.minimize algorithm will not know were to continue its search. With a bad parameter guess the gain curves are dissimilar and small variations might find a local minima in diversion but not the wanted solution.
It follows that a good initial guess was needed. To obtain one the scipy.optimize.brute method was used which tries every possibility for every parameter in a predefined range with a predefined step size. As just a 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. This was a huge time saver and got to a result quite fast.
Now the reasonable initial guess was fed into scipy.optimize.minimize. With good initialisation this method returned useful results meaning parameter arrays with likely outcome to the WAVES plug-in. Just the graphical representation of the "Vocal Rider" gain adaption was not as expected for the reason of some gain values jumping out of the

line for single samples. After a bit of testing this could be explained by small rounding errors of the DAW at the export of audio files. While this was unproblematic at most times they 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 became obsolete as the algorithm was changed to use RMS averages for displaying the graphical evaluation.

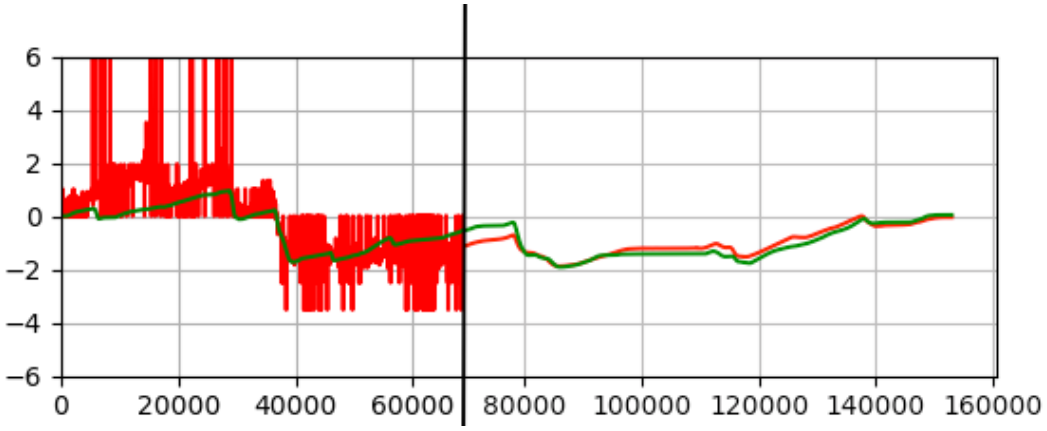

Figure 4.1: Optimisation result with (left) and without (right) rounding errrors

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*[1] 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.

4.4 Results

The differences between the default "Vocal Rider" component, the live version and the self-automated were negligible. The automated plug-in results in the same as the default

[1] "*TARGET sets the reference range for vocal mix positioning. [It] (. . .) will move the Rider Fader's '0' calibration.*", Vocal Rider User Guide, Waves, 2009

just with tiny jumps in the adapting gain. It does not adapt for every sample but still in pretty small time steps, probably derivable from the communication between DAW and plug-in. The Live version does no critical difference to the default as expected. It just adds a potentiometer to adjust the noise gate for live performances. Therefore in the further described test results it is always spoken of the default plug-in version.
The first thing discovered was that the WAVES plug-in does not use a lookahead and consequently the parameter optimisation always resulted in a lookahead of 0ms. This was to expect for the live component but not necessary for its standard use as todays DAWs can easily compensate the produced delay. As follows it was not seen as important to implement while the study plug-in it was done (reason explained in section 3.7).
The second thing resulting form the analysis was quite interesting. When the Waves plug-in does not detect a vocal signal, in contrast to the study plug-in it does not start to fade to 0dB gain immediately but after some ms of idle time. In assumption this feature wants the plug-in to ignore small gaps in the vocals were the singer is breathing or doing a rhythmic break. In this case a smoothed gain reset to 0dB is not necessary and can possibly be counterproductive as the gain adaption needs a bit of time. Further considerations about implementing something similar for the study plug-in concluded that it could work even better in combination with a small lookahead (see 5.4).
Despite the described obvious differences the optimisation algorithm was able to bring both plug-ins quite close in terms of the gain adaption. The "Vocal Rider" seems to need a short amount of time (about one second) to adjust its behaviour. This presumption resulted from different tests where both plug-ins were operating quite similar apart the beginning. As consequence the compare function was improved by ignoring the first second of the result.
In further testing at first the assumed was made that the Waves plug-in still alters its behaviour over time increasing with a bad initial target value (similar to loudnessGoal). This was due to the study plug-in just being able to imitate a section of the "Vocal Rider"s gain adaption for test files of longer duration instead of the fitting with the whole thing. At closer considerations of other causes for its behaviour some ideas were tested. Because the amplification of the "Vocal Rider" with slow attack time still results in greater gain alteration than the compression of it with very fast attack time it seemed as if the compression was attenuated while the amplification was not. The weakened compression reminded of standard Compressor behaviour. Therefore a 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.

Following to this 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 appeared to validate the suspicion and got both gain curves close together for the whole test files.

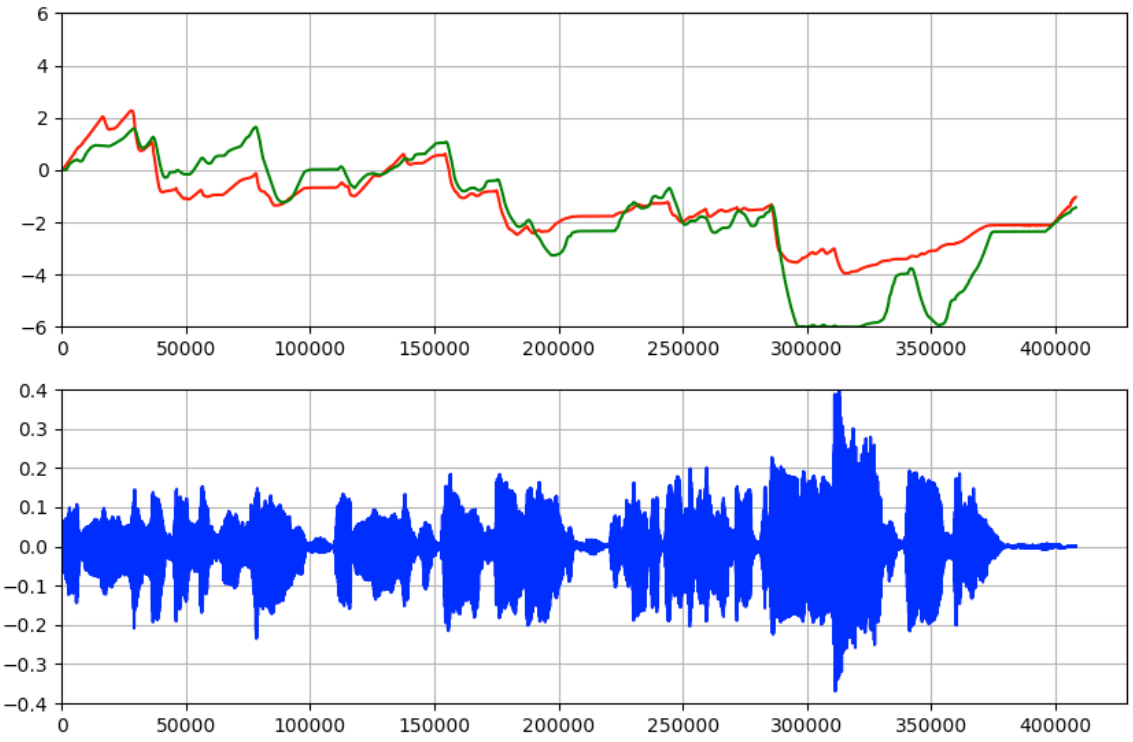

Figure 4.2: Gain Reduction comparison with interim optimisation result (red = 'Vocal Rider', green = study plug-in)

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

[2]proportion between increment of input and output signal above the threshold
[3]slope $= 1 - 1/\text{ratio}$

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

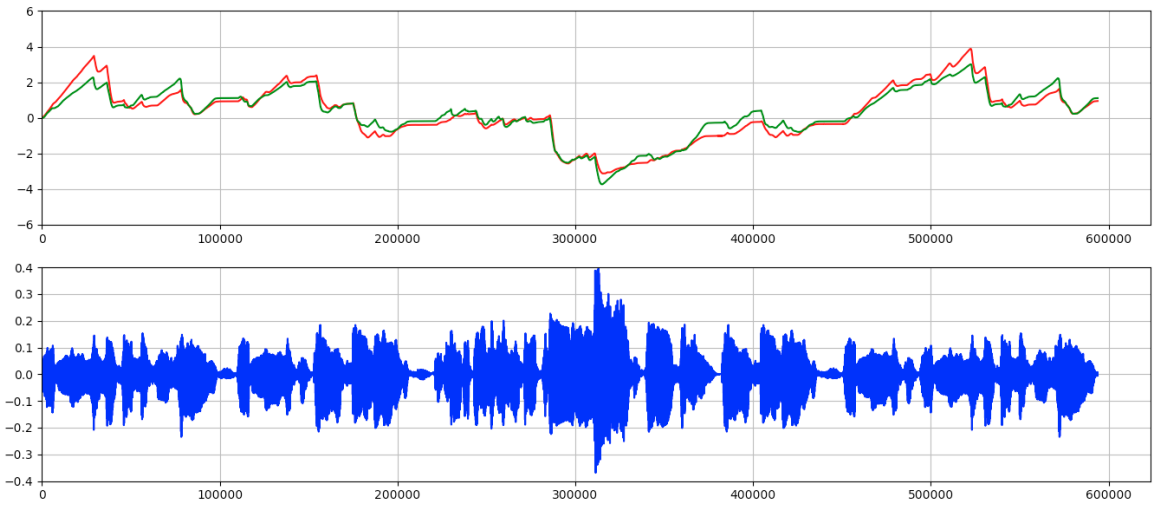

Figure 4.3: Gain Reduction comparison with final optimisation result (red = 'Vocal Rider', green = study plug-in)

Chapter 5

Improvements

5.1 Automation

With the assistance from the study plug-in a mixing engineer does not have to draw a gain automation him/herself anymore. Nevertheless can it be useful to have a automation written by the plug-in. On one hand 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 over and over again. 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 the total amount of necessary CPU power. Often single tracks are bounced in place[1] to avoid problems in performance. When the plug-in just reads a 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 by this feature. In principle automations are supported by the JUCE framework and it can be very simple to implement an standard automation process. The normal 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. With the study plug-in it was a special case which made it more difficult to realise. In case of the plug-in of this thesis 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.
Similar to the WAVES plug-in a button at the UI was implemented 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 done quite fast as it just bypasses the regu-

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

lar calculations and multiplies the gain value set by the DAW with the current sample. The communication between DAW and plug-in works very well on this part with the predefined parameter class from JUCE. The write mode produced 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 is more accurate as it needs to be for a good 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/expansion) or when the difference to the last drawn automation point became crucial. Unfortunately this did not work as well as expected. The output in the DAW would have been good but at some spots it made incorrect jumps. These jumps were of at most 2 samples but still unwanted. Due to several parties involved in this process the real cause of the jumps could not be figured out. Therefore other ideas had to be tested. Different timings and positions for the communication to the DAW were tested and additionally for the change of the parameter value. At first this made no difference in the result or even made it worse. Finally a new setting worked quite well.
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 but 12,5 instead of 41000 times a second or even more. Also very rational as it does not set new point when the gain is not adapting as this would be redundant. There are still jumps in the curve but this is no real problem any longer as they are very few and about at most 0.1dB gain difference. Additionally these jumps are now located at correct positions, there is just no smoothing which is not essential for such small amounts of adaption.

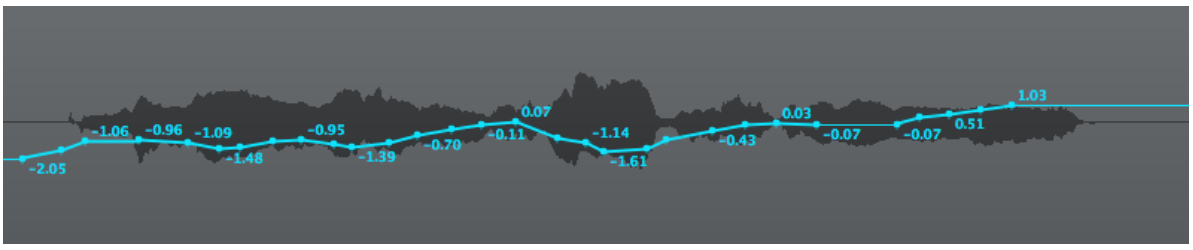

Figure 5.1: Automation written by the study plug-in into a DAW

5.2 Adapting Loudness Goal

The loudnessGoal parameter is not to understand 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 example when the loudnessGoal 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 half (only negative/positive gain adaption) for most of the time. Therefore it is important to set the loudnessGoal correct. In best practice the plug-in will compress the incoming signal as much as amplifying it.
To simplify the selection of a fitting loudnessGoal for a user, a input meter was implemented next to the loudnessGoal slider and the current gain adaption. In this way it is easier to see if the current setting makes sense. However it still remains a problem as the perfect loudness goal possibly alters through a song. So 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 loudnessGoal 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 negative gain values). The offset was finally subtracted from the current loudnessGoal and the sum of the gain values got reseted.
Problematic with this approach was that the loudnessGoal adaption was one second to late if the signal level changed. In addition it made some advantages of the plug-in needless 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.
It follows that only analysing 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 loudnessGoal. In conclusion a button was implemented accessible at the UI which switches the plug-in to loudnessGoal detection mode. During the time this mode 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 gainRange for this period of time to the maximum (+/- 10dB). This is done to find a most accurate average loudness rate and it has no negativ consequences as it just endures as long as the detection mode is active. If you

run the plug-in on detection mode through the full vocal track it calculates a decent loudnessGoal. To avoid the result being falsified by sections were no vocal signal is present, it only adapts on input with actual signal.
This is a task predestined to be done offline but due to the complexity of offline calculations in a plug-in operating in different DAWs the time was not enough to realise it in this thesis. For future work this would be a great opportunity to make use of the full ITU-R BS.1770-4 algorithm.
Although the loudnessGoal detection mode is probably the best current way to set the parameter, it is still allowed for a user to choose it him/herself due to own creative reasons.
When the loudnessGoal algorithmically adapts or is set to a new value on user terms the threshold of the plug-ins gate will change as well. Regardless of whether the average input is of small or great level the plug-in is therefore able to handle it with reasonable settings.

5.3 Side Chain

So far the plug-in works well on single audio tracks and reduces long term dynamics. This already lessens the work for a mixing engineer but the vocal level staying around the same amount over a whole musical piece is not necessary the wanted result. If the instrumental backtracks varies in its loudness it is desirable for the plug-in to do the same with its output gain. To do such a sing the plug-in requires additional information. Consequently a side chain input was added to its interface to realise this feature.
Side chains are not part of the standard I/O[2] layout of JUCE but supported since JUCE version 4.1. To realise another input, it was added in the BusesProperties() structure from the AudioProcessor class. As for the plug-in it is not necessary to have a side chain input, the supported bus layouts did not have to change. Therefore the embedding DAW is able to feed it with a signal but does not have to. For further testing in Logic Pro 9 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.
After implementing the new input the new information had to be used. To do this the plug-in reads the side chain buffer in its main processBlock() method. It needed infor-

[2] I/O = input/output

mation about the loudness of the backtrack (which should be fed into the side chain) to be able to determine a useful additional output gain. In process of detecting the loudness of the side chain input it is passing through the filter, RMS and gain calculations as the regular input signal does (see Fig. 5.2). This includes the transformation into logarithmic number space. Afterwards it is compared to the current loudnessGoal 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 that it does not interfere with the main plug-in calculations. Especially the loudnessGoal must not be changed through this process. Otherwise it would corrupt its results in terms of dynamic range and make the loudnessGoal detection useless.

While testing in real use cases it seemed 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 great 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 right version of vocal to backtrack relation but a great number of reasonable possibilities. Nevertheless for future improvements of the plug-in this part can certainly be simplified in its user interaction.

The side chain feature remains an usable addition to the main plug-in but becomes no part of it as it is not automatically profitable in every use case. That is why there is button at the UI to toggle the 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 dissimilar. Logic Pro 9 sends for example an empty signal containing only zeros. 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 which is toggled off by default, there is no urgent need of a silence detection always running in the plug-ins side chain processing chain.

A remaining question for the side chain implementation were how to set the average time coefficients for RMS calculations and gain adaption (see 5.6). On one hand it should not react on small changes for example when a musician in the instrumental is just accentuating certain beats, on the other hand it needs to react fast enough to calculate an appropriate gain for the first sung words of a new song part with a different loudness level. 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. Later with the implementation of the idle time (see 5.4) at gain adaption this behaviour could be avoided for the most part.

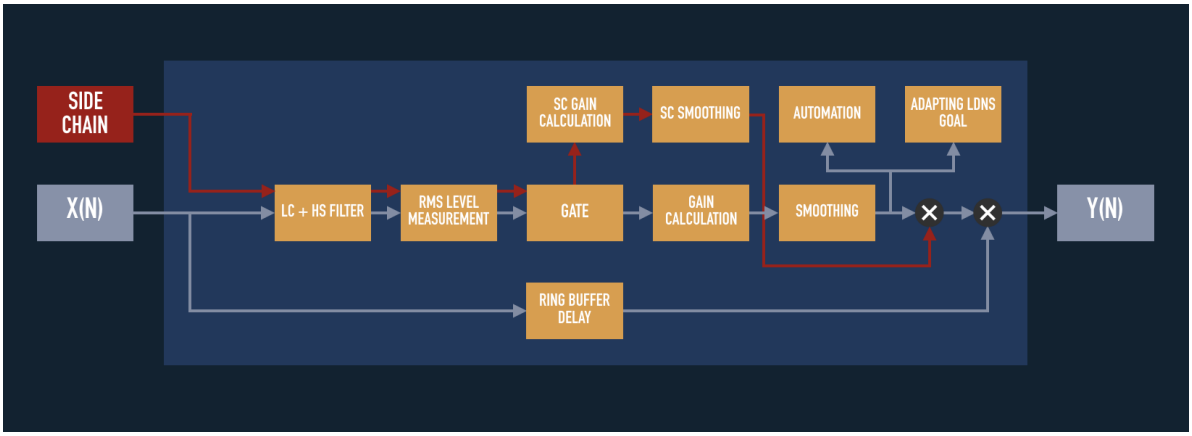

Figure 5.2: Improved processing chain

5.4 Idle Time

During the comparison of the study plug-in and the "Vocal Rider" it emerged the idea of implementing a idle time before fading to 0dB gain for the study plug-in. The effect of such a 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. Following there will be no problematic danger 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.

Every time the input was below the threshold, the current sample was replaced by the last one above for as long as the idle time was set. Due to the last sample above the threshold still being at proportional low level it did not work as planned. The subsequently gain adapting was consequently falling on 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 the loudnessGoal by the gate. Maybe it 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 for past samples and it seemed to complicated.
The current implementation is therefore based on the plan that the plug-in should detect at the gate if there are no vocals and then freeze the current gain at the following gain adaption. As the gain adaption is much slower due to compress/amplification time smoothing than the RMS averaging before the gate, it freezes the gain before it is able to adapt to the level change. For reasons of simplification everything was implemented in the updateGain method.

```
double AutoVocalCtrlAudioProcessor::updateGain(double sample,
double lastGn)
{
    if (sample != *loudnessGoal) {
        idleCount = 0;
    } else if (idleCount < maxIdleSamples) {
        idleCount++;
        return lastGn;
    }
...
}
```

It detects if the current level is below gate threshold by comparing the sample with the loudnessGoal. If the transferred RMS averaged sample is at the exact value of the loudnessGoal it can be assumed that the gate replaced its actual value. The rare case of a sample being randomly at exact the level of the loudnessGoal and therefore stop the gain adaption cant be eliminated in this simple implementation. Nevertheless this makes no a 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 the loudnessGoal 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 it was the next step to think about its advantages in terms of side chain integration. At this time there was still a problem with 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 with tests in a real mixing environment. The result with a reasonable set maximum was quite good as it severely reduced the gain drop during instrumental breaks as long as the side chain input level was adapted to the loudnessGoal. Consequently the UI was expanded with a pre gain slider for the side chain input which adds up before the comparison to the loudnessGoal. Controlling the input level of the side chain signal was possible before via a level controlled bus send but now the mixing engineer is able to do it all at one place while check the level meters for both signals right next to each other at the UI. As it was still difficult to adjust it correct for a whole song a self setting algorithm for the input gain was added to the already existing detection mode. As said before this would be significant better to do in an offline thread but works in real time for now.

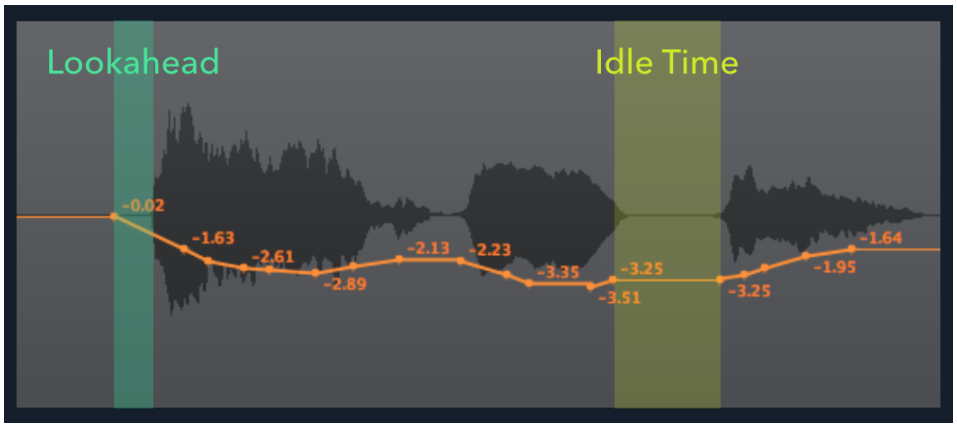

Figure 5.3: Idle time and lookahead marked at automation written by the study plug-in

5.5 Wet/Dry

Many plug-in effects for DAWs have an additional slider to adjust the wet/dry ratio of their outcome. In these terms 100In consideration of the use of such a slider in the

plug-in of this thesis, it seemed useful at first as it ables 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 gainRange. In addition that this is 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 intension of simplifying the UI to its foundation followed by the reduction of the settable parameters.

5.6 Parameter Reduction

As the plug-in slowly reaches its final state, it was necessary to transfer some settable parameters to fixed constants. In development it was useful to be able to change the important parameters while in runtime for example the compress and amplification times. Now it already emphasised which settings will result in the best outcome. Small changes could be slightly better depending on the current circumstances but they are not required for a good result. Additionally with wrong application these parameters could create a poor result. Therefore and with the likely possibilities of a user not knowing what each of these parameters does, it seemed to be advantageous to hide those as constants, 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 like already described, the gate. With the experience from testing the plug-in with many different settings in various circumstances the default choices for the parameters already got to reasonable values.
With a bad lookahead delay the gain adaption could happen 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. 5.1, 5.3).
The RMS averaging time shouldn't be to long as this adds up on the total reaction time of the plug-in. On the other hand it needs a big enough time window to calculate a useful average. At the beginning of development it was set to 60ms. The comparison

to the Waves “Vocal Rider” resulted in very small time windows (down to 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 would be greater then the compression time. Still the final settings were not yet set. So again, the comparison to the “Vocal Rider” served as an inspiration. Though with a 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 results from the comparison with the “Vocal Rider” were similar to a slow compressor and therefore still to fast to fit the idea of this thesis. As follows 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 is set after considerations 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 the loudnessGoal like described in 5.2 and is not settable in the UI anymore.

5.7 last CAP OF PLUG-IN

5.8 Design

Chapter 6

Evaluation of the Side Chain Feature

The plug-in got some improvements over the time. 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 done 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 demonstrate its advantages to the prototype in different circumstances.

6.1 Test conditions

Chapter 7

Results

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 Masterstudiengang 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