Politecnico di Milano (2023), 1–9

POLITECNICO
MILANO 1863

HOMEWORK #2 REPORT

CTAG Dynamic Range Compressor

Nicolò Chillè, Enrico Dalla Mora, Rocco Scarano, and Federico Caroppo

Group Name: ***La Lobby*** **– Group ID: 3**

Abstract
The purpose of this document is to analyze and reverse engineer an existing JUCE plugin, specifically the dynamic compressor VST3 plugin mentioned in the title, CTAGDRC. We will focus on the special features of the plugin, such as the implementation of the LookAhead mode, the parameter automation with respect to Makeup and ballistics (i.e. attack and release), the compression algorithm and the general attributes of the project. The GUI management will also be covered.

Keywords: compressor, dynamic, feed-forward, lookahead, makeup, ballistics

The CTAG Dynamic Range Compressor is a VST3 plugin, built to be an easy-to-use, good-sounding tool, well suited for any kind of application. Given the articulation of the code in multiple DSP classes, reference to the folder tree in appendix for orientation. For the same reasons, when referring to the name of a class, said name will be highlighted in red, while methods will be highlighted in blue.

1. Topology

Dynamic Range Compression (DRC) is the process of **mapping the dynamic range of an audio signal to a smaller range**. Compressors of this class represent *nonlinear time-dependent systems with memory*; the gain reduction is applied smoothly and not instantaneously. The topology chosen by the author of the plugin is the **feedforward topology**, whose scheme is illustrated below:

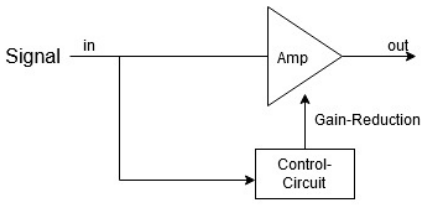

Figure 1. Feddforward topology of a compressor.

The gain reduction is computed by a *control circuit*, before being multiplied by the raw input signal.

2. Control Circuit Architecture

The control circuit is articulated as illustrated in the following schemes, included in the documentation of the plugin:

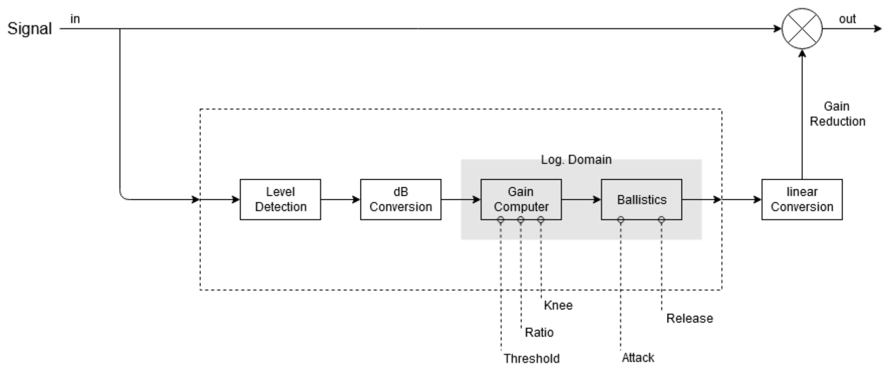

Figure 2. Basic compressor block diagram.

2.1 Level Detection

The copy of the signal entering the sidechain has its bipolar amplitude converted into a unipolar representation of level.

$$y_L(n) = |x_L(n)| \tag{1}$$

This operation happens in the class `Compressor`, which is responsible for the entire compression process. The `PluginProcessor` class simply feeds the input signal to its private `Compressor` attribute by calling its `process` method on the input buffer.

Code 1. Unipolar level conversion and peak sensing.

```
FloatVectorOperations::abs(rawSidechainSignal, buffer.getReadPointer(0),
    numSamples);
FloatVectorOperations::max(rawSidechainSignal, rawSidechainSignal, buffer.
    getReadPointer(1), numSamples);
```

The level of the signal is determined by its absolute value (*instantaneous signal level*), and this is known as **peak-sensing**[1]. The `max` function (`FloatVectorOperations` collection) selects the maximum value of each pair from the source vectors and copies it in the destination vector, i.e. selects the maximum value of amplitude between two channels.

1. An alternative is to use RMS-sensing and determine the signal level by its root-mean-square (RMS) value.

2.2 Domain conversion

The attenuation computation and the smoothing can be performed in linear or logarithmic domain. The CTAGDRC operates in the logarithmic domain, since usually logatithmic detectors outperform linear ones for a wide variety of signals. The conversion also happens in `Compressor.cpp`, by means of a call to a specific method, `applyCompressionToBuffer`, of the `GainComputer` class instance `gainComputer` (which is also a private attribute of the Compressor class).

Code 2. `applyCompressionToBuffer` callback.

```
gainComputer.applyCompressionToBuffer(rawSidechainSignal, numSamples);
```

`applyCompressionToBuffer` is not only responsible of the domain shift, but recursively call another `GainComputer` method, i.e. `applyCompression`:

Code 3. `applyCompressionToBuffer`.

```
1 void GainComputer::applyCompressionToBuffer(float* src, int numSamples){
2     for (int i = 0; i < numSamples; ++i)
3     {
4         const float level = std::max(abs(src[i]), 1e-6f);
5         float levelInDecibels = Decibels::gainToDecibels(level);
6         src[i] = applyCompression(levelInDecibels);
7     }
8 }
```

`applyCompression` is responsible for the **gain computing**, which will be discussed in the next section.

2.3 Gain Computing

As can be seen in line 6 of Code 3, `applyCompressionToBuffer` calls back to another method of the class `GainComputer`: `applyCompression`, i.e. the actual gain computing algorithm. The gain computing stage **calculates the amount of attenuation the input signal must undergo**, given its *current characteristics*. The input/output features of the compressor are defined by means of three parameters: **threshold**, **ratio** and **knee**. The compressor starts the attenuation, according to the ratio, once the level exceeds the threshold. The transition from uncompressed to compressed signal can be smoothed by means of a knee parameter (in the examined case, a so-called *soft knee*). The process is described by the following set of equations. W represents the width of the knee, R is the ratio and T represents the threshold value, while y and x stand respectively for the output and the input levels.

$$y = \begin{cases} x & (x-T) < -W/2 \\ x + (1/R - 1)(x - T + W/2)^2/(2W) & |(x-T)| \leq W/2 \\ T + (x-T)/R & (x-T) > W/2 \end{cases} \tag{2}$$

Obviously, $x - T$ is the overshoot, namely the exceeding signal level. The following implementation is the one employed by the `applyCompression` method of `gainComputer`:

Code 4. `applyCompressionToBuffer`.

```
float GainComputer::applyCompression(float& input) {
    const float overshoot = input - threshold;
    if (overshoot <= -kneeHalf)
        return 0.0f;
    if (overshoot > -kneeHalf && overshoot <= kneeHalf)
        return 0.5f * slope * ((overshoot + kneeHalf) * (overshoot +
            kneeHalf)) / knee;
    return slope * overshoot;
}
```

The computed level attenuation must now be smoothed. This passage, corresponding to the *Ballistics* block of the diagram in Figure 2, involves an **attack** and a **release** parameters, respectively defined as the time it takes for the compressor to take action once the signal exceeds the threshold and the time it takes for the compressor to recover once the signal falls under the threshold. This process is controlled via a so-called **smoothing filter** (digital one-pole filter):

$$y_L(n) = \alpha y_L(n-1) + (1-\alpha)x(n) \tag{3}$$

$$= \begin{cases} \alpha_A y_L(n-1) + (1-\alpha_A)x_L(n) x_L > y_L(n-1) \\ \alpha_R y_L(n-1) + (1-\alpha_R)x_L(n) x_L \leq y_L(n-1) \end{cases} \tag{4}$$

$$\alpha_A = e^{-1/\tau_A f_s}, \quad \alpha_R = e^{-1/\tau_R f_s} \tag{5}$$

Code-wise, when `Compressor` calls `applyBallistics` on `ballistics` (which is a private instance of `LevelDetector`), the method recursively calls the `ballistics` method `processPeakBranched` (i.e. *which implements the smooth peak detector*) on each sample of the buffer:

Code 5. `applyBallistics`.

```
void LevelDetector::applyBallistics(float* src, int numSamples){
    // Apply ballistics to src buffer
    for (int i = 0; i < numSamples; ++i)
        src[i] = processPeakBranched(src[i]);
}
```

Code 6. `processPeakBranched`.

```
float LevelDetector::processPeakBranched(const float& in){
    //Smooth branched peak detector
    if (in < state01)
        state01 = alphaAttack * state01 + (1 - alphaAttack) * in;
    else
        state01 = alphaRelease * state01 + (1 - alphaRelease) * in;
    return static_cast<float>(state01); //y_L
}
```

τ_A and τ_R, the smoothing filter design and its placement in the chain are all **design choices**, that thus affect the sound of the compressor. The author of the present implementation *placed the smooth branching peak detector behind the gain computer*: this implies that the detector does not act on the whole dynamic range of the input signal.

3. Automation

The presence, in the sidechain, of components specifically designed to automate some of the compressor's parameters is a peculiarity of the chosen plugin. The following illustration shows the location of said components, namely the **Crest Factor computer** and the **LookAhead processor**, in the chain.

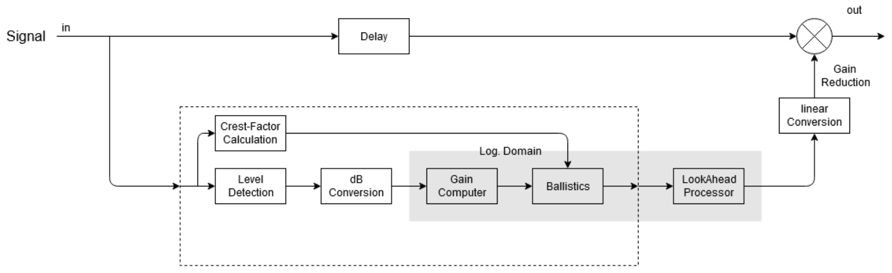

Figure 3. Compressor diagram with automation components.

3.1 *Crest factor*

The crest factor can be used to **automate the attack and release times** in an automated compressor model. It is defined as the ratio between the peak amplitude of a signal and its RMS (*root mean square*) value over a certain period of time, and it can be employed to get general information about the nature of the signal. ***A transient-rich signal has a consistently higher crest factor value with respect to a steady-state signal.*** The goal is thus to combine a peak detector and an RMS detector:

$$\begin{cases} y^2_{Peak} = \max\left[x^2(n),\ \ \alpha y^2_{Peak}(n-1) + (1-\alpha)\left|x^2(n)\right|\right] \\ y^2_{RMS} = \alpha y^2_{RMS}(n-1) + (1-\alpha)x^2(n) \end{cases} \tag{6}$$

$$y_C(n) = \frac{y_{Peak}(n)}{y_{RMS}(n)} \Longrightarrow \begin{cases} \tau_A(n) = 2\tau_{A_{max}}/y^2_C(n) \\ \tau_R(n) = 2\tau_{R_{max}}/y^2_C(n) - \tau_A(n) \end{cases} \tag{7}$$

From a code perspective, this is the sequential process:

1. **PluginProcessor:** `PluginProcessor` calls the `process` method of its private `Compressor` instance (named `compressor`).
2. **Compressor:** during the execution of `Compressor`'s `process` method, just before letting `gainComputer` process the attenuation, the `processCrestFactor` method of the `ballistics` private attribute (instance of `LevelDetector`) is called.
3. **LevelDetector:** if automation is enabled, the `processCrestFactor` method computes the Crest Factor by calling the `process` method on its private attribute `crestFactor` (instance of `CrestFactor`), which implements Equations 6 and 7. Then, two smoothing filters (simple moving average filters, instances of `SmoothingFilter`) are applied to the retrieved average values of attack and

release. At this point, the call returns to `ballistics` (`LevelDetector`), which sets the attack and release values.

The Crest Factor computation is implemented as follows:

Code 7. CrestFactor's `process`.

```
void CrestFactor::process(const float* src, const int numSamples){
    if (!peakState) peakState = src[0];
    if (!rmsState) rmsState = src[0];
    avgAttackTime = 0.0;
    avgReleaseTime = 0.0;
    for (int i = 0; i < numSamples; ++i){
        const double s = static_cast<double>(src[i]) * static_cast<double>(
             src[i]);
        peakState = jmax(s, a1 * peakState + b1 * s);
        rmsState = a1 * rmsState + b1 * s;
        const double c = peakState / rmsState;
        cFactor = c > 0.0 ? c : 0.0;
        if (cFactor > 0.0){
            attackTimeInSeconds = 2 * (maxAttackTime / cFactor);
            releaseTimeInSeconds = 2 * (maxReleaseTime / cFactor) -
                 attackTimeInSeconds;
            avgAttackTime += attackTimeInSeconds;
            avgReleaseTime += releaseTimeInSeconds;
        }
    }
    avgAttackTime /= numSamples;
    avgReleaseTime /= numSamples;
}
```

3.2 LookAhead Mode

LookAhead mode allows the compressor to **anticipate incoming peaks** and **effectively operate on very fast transients**, that may otherwise pass through even a really quick attack time. This relieves the compressor of complete reliance on its attack time allowing it to adapt to abrupt variations in the signal amplitude. The most basic implementation of this feature simply relies on delaying the input signal: the attenuations computed by the `GainComputer` will affect the signal before the actual hit of the transient. Usually, the delay will be of the order of *a few milliseconds*, and thus the output signal won't be perceived as being out of time. However, phase issues may arise between the output signal and other signals in the mix. The LookAhead implementation is organized in the following steps:

3.2.1 Gain reduction computation

Standard compressing stage: create a side-chain signal (max over all channels), process it detecting the peaks and attenuate them via the usual algorithm.

3.2.2 Delay of both signals

Both signals must be delayed of the same amount in order to avoid the application of gains before or after the transients happen. Usually, a delay around 5 ms is a good compromise between latency and distortion suppression[2].

2. The delay value can be rounded to samples, in order to avoid the need for a fractional delay.

3.2.3 Gain reduction smoothing

The objective is to smooth (**fade-in**) the aggressive gain reduction values to prevent possible distortions. The applied delay guarantees a small time window to apply the gain-ramp (i.e. the fade-in). To do so, it is necessary to examine the gain-reduction values and add a fade-in for each peak that's setting in too fast. The easiest way is and work our way back to the first sample starting from the most recent one of the block (i.e. the last one). This process can be interpreted as the application of a **time-reverse filter with a low-pass**[3]. This must be done "manually", that is *sample by sample*. Let's take a look at the class `LookAhead`, and in particular to the method `processSamples`, which implements the actual algorithm:

Code 8. `processSamples` and `fadeIn`.

```cpp
void LookAhead::processSamples() {
    int index = writePosition - 1;
    if (index < 0) index += bufferSize;
    int b1, b2;

    // ‘numLastPushed‘ many samples are to be processed in the first run
    getProcessBlockSize(numLastPushed, index, b1, b2);
    float nextValue = 0.0f;
    float slope = 0.0f;

    // First run
    fadeIn(index, b1, slope, nextValue);

    // Second run
    if (b2 > 0){
        index = bufferSize - 1; //start from the last sample
        fadeIn(index, b2, slope, nextValue);
    }

    // ‘delayInSamples‘ many samples are to be processed in the next block
    if (index < 0) index = bufferSize - 1;
    getProcessBlockSize(delayInSamples, index, b1, b2);
    bool procMinimumFound = false;

    // The samples already underwent processing: if one of the samples is
         below the ramp value, it's the new minimum, and has been faded-in
         already. Thus, the break ends the process.
    fadeInWithBreak(index, b1, slope, nextValue, procMinimumFound);
    if (!procMinimumFound && b2 > 0){
        index = bufferSize - 1;
        fadeInWithBreak(index, b2, slope, nextValue, procMinimumFound);
    }
}

inline void LookAhead::fadeIn(int& index, int range, float& slope, float&
     nextValue) {
    for (int i = 0; i < range; ++i){
        const float sample = buffer[index];
        if (sample > nextValue){
            buffer[index] = nextValue;
            nextValue += slope;
        } else {
            slope = -sample / static_cast<float>(delayInSamples);
            nextValue = sample + slope;}
        --index;
    }
}
```

3. Note that a regular low-pass would reduce the gain of the peaks.

```
46 inline void LookAhead::fadeInWithBreak(int& index, int range, float& slope,
       float& nextValue, bool& procMinimumFound)
47 {
48     for (int i = 0; i < range; ++i)
49     {
50         const float sample = buffer[index];
51         if (sample > nextValue)
52         {
53             buffer[index] = nextValue;
54             nextValue += slope;
55         }
56         else
57         {
58             procMinimumFound = true;
59             break;
60         }
61         --index;
62     }
63 }
```

Note that the samples are gain-reduction values (in *dB*): thus, it is necessary to look for negative peaks (local minima). Once one is found, the **slope** (`slope`) can be computed, and consequently the **next value of the fade-in** (`nextValue`, `fadeIn`) can be determined. When a value below the set threshold (i.e. the current fade-in value) is found in the buffer, this means that a new minimum has been found. Though it might not be as deep as the previous one, it comes in earlier (recall that *the process is time-reversed*). Therefore, the fade-in slope gets updated. Note that, in correspondence of `line 20`, all the gain-reduction values have been computed, though it is possible that the first sample was assigned such a high gain-reduction that now it needs a fade-in itself. Thus, the gain ramp must be applied even further in the past[4].

4. This is the actual reason the LookAhead was implemented.

Appendix 1. Folder Structure

Along with the standard JUCE classes `PluginProcessor` and `PluginEditor`, the CTAGDRC comes with a number of DSP (digital sound processing) algorithms, each implemented via its own class. Here, a folder tree of the source codes is reported as a reference:

```
CTAGDRC
└── Source
    ├── dsp
    │   ├── include
    │   │   ├── Compressor.h
    │   │   ├── CrestFactor.h
    │   │   └── ...
    │   ├── Compressor.cpp
    │   ├── CrestFactor.cpp
    │   ├── DelayLine.cpp
    │   ├── EnvelopeFollower.cpp
    │   ├── GainComputer.cpp
    │   ├── LevelDetector.cpp
    │   ├── LevelEnvelopeFollower.cpp
    │   ├── LookAhead.cpp
    │   └── SmoothingFilter.cpp
    ├── gui
    │   ├── include
    │   ├── Meter.cpp
    │   ├── MeterNeedle.cpp
    │   └── ...
    ├── PluginEditor.cpp
    ├── PluginEditor.h
    ├── PluginProcessor.cpp
    └── PluginProcessor.h
```