Wolfgang Hrauda

Virtual binaural acoustics in VLC player: HRTFs and efficient rendering

Project thesis
Audio engineering project

Graz, University of Music and Performing Arts
Institute for Electronic Music and Acoustics
Head: Dipl.-Ing. Dr.techn. Alois Sontacchi

Acoustics Research Institute
(Austrian Academy of Sciences, Vienna)

Supervisor: Dr. Piotr Majdak

Graz, September 2015
Abstract

VLC player is a common software for playback of various video and audio formats. However, it currently lacks the capability of adequately presenting spatial audio signals via headphones. Therefore, the Acoustics Research Institute recently started development of the module “SOFAlizer”. It uses head-related transfer functions (HRTFs) to simulate a virtual loudspeaker setup (e.g. 5.1 surround) which signals are presented via headphones. The HRTFs are loaded from files using the spatially oriented format for acoustics (SOFA), an AES standard for storing 3D audio data.

In this project, the existing „proof-of-concept“ of SOFAlizer will be further developed into a fully functional module for VLC player. This involves addressing signal processing issues such as handling of HRTFs and audio signals with different sampling rates, as well as the efficient implementation of the convolution. Furthermore, the source code will be revised, improved, and commented.

Kurzfassung

VLC-Player ist ein weit verbreitetes Programm zur Wiedergabe diverser Video- und Audioformate. Allerdings fehlt dem VLC-Player bis dato die Funktionalität räumliche Audiosignale über Kopfhörer adäquat zu spatialisieren. Daher wurde vor kurzem am Institut für Schallforschung mit der Entwicklung eines Moduls mit der Bezeichnung “SOFAlizer” begonnen. SOFAlizer simuliert die vom Audioformat vorgesehene Lautsprecheranordnung (wie z.B. 5.1 Surround) über Kopfhörer unter der Benutzung von HRTFs (head-related transfer functions). Diese werden aus SOFA-Dateien (spatially oriented format for acoustics), dem AES Standard für räumlich-orienierte Audiodaten, geladen.

In dieser Projektarbeiten wird SOFAlizer, derzeit als „proof-of-concept“ vorhanden, zu einem funktionsfähigen Modul für den VLC-Player weiterentwickelt. Dabei werden signalverarbeitungstechnische Fragen, insbesondere die Behandlung von HRTFs und Audiosignalen mit unterschiedlichen Abtastraten, sowie die effiziente Implementierung der Faltung, behandelt. Außerdem soll der Quellcode überarbeitet, verbessert und kommentiert werden.
Table of Contents

1 Motivation ............................................................................................................. 3

2 Virtual auditory display using head-related transfer functions ............... 5
  2.1 Human sound source localization .................................................................. 5
  2.2 Binaural reproduction of sound sources .......................................................... 6
  2.3 Numerical representation of head-related transfer functions (HRTFs) in Spatially oriented format for acoustics (SOFA) ........................................... 8

3 Digital signal processing for binaural rendering ......................................... 11
  3.1 Bandlimited resampling .................................................................................. 11
  3.2 Time domain convolution .............................................................................. 12
  3.3 Fast convolution ............................................................................................ 13

4 Development for VLC Player ........................................................................... 15
  4.1 The program concept .................................................................................... 15
  4.2 Audio filter modules ..................................................................................... 15
  4.3 Version control and submitting patches .......................................................... 16
  4.4 Development tools ....................................................................................... 18

5 SOFAlizer: Binaural rendering in VLC player .............................................. 21
  5.1 Integration within VLC Player ....................................................................... 21
  5.2 Cross-compilation issues .............................................................................. 23
  5.3 Program structure and flow .......................................................................... 24
  5.4 Source code improvements .......................................................................... 26
  5.5 Resampling of filter responses ...................................................................... 27
  5.6 Audio processing: Implementation of time domain convolution ............. 31
  5.7 Audio processing: Implementation of Fast Convolution ......................... 34

6 Conclusion and outlook .................................................................................... 39
1 Motivation

The advantage of multi-channel audio playback is that the listener can be “enveloped” by the soundscape, possibly even virtually placing them in a certain environment such as a concert hall or any situation from everyday life. The film industry, in particular, has taken advantage of this and at the time of this writing, it is very common that commercially available DVDs include a 5.1 surround audio track.

On the other hand, a proper high-quality surround playback system is expensive due to the number of loudspeakers involved and is also more demanding regarding the consumed space. Compromises in loudspeaker quality to lower the price and in the setup also compromise the perceived effectiveness of the presentation. In addition, a lot of people nowadays like to listen to music or watch films via headphones for various reasons including portability and low disturbance to fellow human beings.

Therefore, creating an artificial auditory scene by simulating a multi-channel playback situation on headphones is a logical objective. The only way to precisely simulate such a setup is by filtering each channel's signal with head-related impulse responses (HRIRs) corresponding to the position that the loudspeaker of that channel would have in a multi-channel setup. HRIRs here correctly recreate the source-direction-dependent reflections and diffractions at head, torso, and pinna of a human listener. The result of such a computation can be presented to a listener via headphones and creates the impression of listening to a real multi-channel playback system, because the simulated signals that arrive at the ear drum are theoretically identical to the real case. This technique is referred to as binaural rendering.

VLC player is a powerful and commonly used open-source media player that is compatible to a wide range of video and audio formats. However, at the time of writing, it lacks the capability of rendering multi-channel audio data for presentation via headphones as described above.

The Acoustics Research Institute (ARI) in Vienna, Austria, has a lot experience with measuring and modeling head-related transfer functions (HRTFs)\(^1\), as well as – more recently – the initiation and development of a standardized common storage format for HRTFs called SOFA. SOFA has recently become the AES standard for 3D audio data and thus has the potential to serve as a basis for a common binaural rendering plugin. VLC player proved to be an adequate environment, which is why development of such a plugin called SOFAlizer was started in summer of 2013. The target is that the plugin becomes an official part of VLC player.

In the future, the realism of the experience might be further improved by using head-tracking systems. Such systems constantly update the rendering parameters in a way that the created virtual auditory scene is perceived as stable despite head movements. Of course, this requires constant updating of the required impulse responses, which needs to be taken into account when designing the handling of impulse responses in the program flow of SOFAlizer.

---

\(^1\) HRTFs are the frequency domain representation to HRIRs and can be derived by Fourier transforming HRIRs. They are thus almost equivalent.
The aim of this project is to improve the early proof-of-the-concept version in many aspects including implementation of a more efficient rendering algorithm (fast convolution), solving the problem of handling HRTFs with different sampling rates, debugging and enhancing the source code (partly based on comments from the official VLC player developers), as well as verbose documentation of the source code to allow for future professional software development.
2 Virtual auditory display using head-related transfer functions

The goal of SOFAizer is to create a virtual auditory display, i.e. virtual sound sources around the user, for listening via headphones. Therefore, the very basics of how humans localize sound sources and the role of head-related transfer functions within that process are introduced. Any details of this complex topic are beyond the scope of this writing, though. It is then shown how HRTFs can be used to virtually reproduce sound sources for binaural listening via headphones. To this end, the representation of HRTFs in the SOFA format are discussed.

2.1 Human sound source localization

As discussed by Zhon and Xie [ZHO14], four types of cues help human beings to identify the direction of a sound source:

- Interaural time difference (ITD) is the difference in time-of-arrival of a sound between the left and the right ear. These localization cues are mainly used for frequencies up to approximately 1- 1.6 kHz. Above approx. 1.5 kHz, wavelengths are shorter than the distance between our ears leading to phase ambiguities our brain cannot resolve, although the ITD of a signal’s envelope might still be detected (see [MAC02]).

- Interaural level difference (ILD) is the difference in sound pressure level of a sound between the left and the right ear. These localization cues are mainly relevant for frequencies above 1.5 kHz, as below that frequency, sound waves are bent around our head.

- Spectral cues are the result of frequency- and direction-dependent filtering of the incident sound waves due to reflections, scattering and shadowing at the head, torso and pinna. For all source directions, where ITD and ILD are nearly identical (so-called “cone of confusion”, see Figure 1), these cues help to dissolve this ambiguity.

- Small head movements also contribute to dissolve the mentioned ambiguity, because they introduce small changes to ITD and ILD.

The first two types of cues are relatively simple to analyze, describe mathematically and simulate. They allow for localization in the horizontal plane (excluding front-back distinction), whereas the spectral cues are used for front-back distinction and vertical localization. They are unique to each person, because of the differences in anthropometry between human beings. This leads to localization fails, if e.g. our pinna shape is altered, although our brain is actually capable of re-learning that (see [HOF98]).

Head-related transfer functions (HRTFs) describe the acoustical transmission from a point source at a certain source direction in the free-field to one ear:

2 In fact, the ILD is only constant on the cone of confusion, if the head is assumed to be a sphere.
HRTF(\(\varphi, \theta, r, f\)) = \frac{P_{L,R}(\varphi, \theta, r, f)}{P_0(f)} \tag{1}

with \(P_{L,R}\) being the sound pressure level (frequency domain) at the left or right ear (usually at the entrance of the ear canal) and \(P_0\) being the sound pressure level (also frequency domain) at the center position of the head, without the head present. \(\varphi\) is the azimuth angle of the source position from the listener’s perspective and \(\theta\) the elevation angle. \(r\) is the distance (“radius”) between source and listener. A binaural pair of HRTF represents ITD, ILD and spectral cues for a given source direction. [ZAA10] explains that the distance between source and listener does not significantly affect HRTFs for distances above 1.3 meters. However, when considering sources in the near-field, the distance between source and listener must be taken into account, as well.

![Figure 1: Cone of confusion (green, from [MAJ14a])](image)

Generally, the perception of distance to a source is also part of sound localization. To sum it up, briefly: In a room it depends on the ratio between direct signal and reflections from room boundaries (see [BRO99]). Whereas in a free-field situation, loudness and timbre of a perceived sound in conjunction with our experience about what that sound “usually sounds like”, gives an estimate to the distance. Distance perception within the near-field of a sound source is a complex matter, as interaural differences and spectral cues change depending on the distance (see above and [MOO99] for further reading).
2.2 Binaural reproduction of sound sources

In reality, when we are listening to the sound of any source in a free-field-like environment, the sonic waves that occur at the entrances of our ear canals are filtered at our pinna, head and torso and also exhibit an ITD and ILD, all depending on the source direction. As HRTFs include all of this information, the sonic waves at our ear canal entries can be derived by filtering the source signal with the HRTF corresponding to the source position. If this is done simultaneously for the left and right ear and the resulting so-called binaural signals are presented to a listener via headphones, this virtual scenario theoretically does not differ from the natural scenario. Thus, the listener's perception of the synthesized binaural sound source is identical to the perception of the same source in "real world". This includes perceived source direction, distance and externalization. Externalization means that listeners localize a sound outside their head, as opposed to inside their head, which is usually the case when listening via headphones. This is particularly crucial for convincing virtual auditory scene generation.

Practically, it is not that simple, as during the acquisition of HRTFs various errors are introduced, such as:

- noise, distortion, limited frequency range of equipment
- positioning variations of microphones and subject
- non-ideal measurement rooms.

During playback the headphones also introduce additional distortion and possible positioning mismatch. As described in [HRA13], p. 14ff., these errors can partly be compensated for, e.g. by spectral subtraction of the equipment transfer function.

Apart from that, theoretically perfect reproduction of a sound source requires the use of the listener's personal HRTFs (often called "individualized HRTFs"), because even small differences in anthropometry significantly deteriorate localization performance (see above). Measuring high-quality individualized HRTFs is possible, if adequate equipment, measurement room and technical know-how are available. Acquisition of individualized HRTFs for masses of mainstream consumers is currently still beyond practicability, though. However, research to simplify the HRTF acquisition is done with promising results (e.g. [ZIE14]).

In the meanwhile, HRTFs of artificial mannikins with averaged anthropometry or adaption processes to fit a general set of HRTFs to an individual (see e.g. [TAM12]) can be used with quite satisfying results for the perceived source direction. Particularly in terms of externalization, though, it is crucial to use individualized HRTFs, so far.

Room information in the sound presentation can also help with the externalization and can also benefit the perceived naturalism. It could either be generated artificially by adding early reflections or even applying some more complex reverberation algorithm. An other possibility is to measure HRTFs in a listening room (as opposed to an anechoic chamber) which allows to also capture the natural room impulse response resulting in so-called binaural room impulse...
Depending on the application, head-tracking could be used to rotate and elevate the auditory scene depending on the listener’s head movement.

### 2.3 Numerical representation of head-related transfer functions (HRTFs) in Spatially oriented format for acoustics (SOFA)

After measurement and post-processing (see [HRA13], p. 14ff.), HRTFs are usually stored using some form of digital numerical format. As [HRA13] shows, the actual storage formats differed a lot between various institutions and research laboratories making HRTF exchange and usage difficult. Therefore, a standardized storage format “Spatially Oriented Format for Acoustics” (SOFA) has been initiated in 2012. The Audio Engineering Society has adapted it as the basis for a new standard format for three-dimensional audio data AES 69-2015 ([AES15]).

SOFA provides storage capabilities for any kind of spatially oriented audio data, but provides so-called conventions to more precisely specify storage of common scenarios, such as SimpleFreeFieldHRIR for head-related impulse responses (refer to the project website [SOF14a] for more information). Majdak and Ziegelwanger describe how HRTFs can be stored using the SimpleFreeFieldHRIR convention in [MAJ14b].

SOFA uses netCDF, a data form for “self-describing, machine-independent” and “array-oriented scientific data” ([NET15a]), as a numerical container. A Matlab/Octave API is provided on the project website [SOF14a] for read/write access and manipulation of SOFA files, but for usage in C-related programming environments (like in SOFAlizer), data have to be read manually using the officially provided libraries from unidata (see [NET15b]).

SOFA files comprise of global metadata variables valid for the whole file such as the name of the used convention or the data type used (e.g. finite impulse responses (FIR), infinite impulse response coefficients (IIRBiquad)). The actual audio data is of the specified type and is stored in a three-dimensional matrix variable. Other variables describe the geometry of the measurement setup, such as the positions of the loudspeakers relative to the listener.

All variables containing actual numbers rather than just descriptions must conform to pre-defined dimensions. For example, one source position must always consist of three coordinates and there might be one source position valid for all measurements or one source position for each measurement. In SOFA notation this means that the dimension of the variable SourcePosition must be [IC] or [M C]. The most important dimensions in this context are shown in Table 1.

---

3 In this context, it is worth noting that although HRIR refers to impulse responses in the time domain and HRTF refers to their corresponding frequency domain representations, they are simply related by the Fourier transform. Therefore, the terms are almost equivalent even if not exactly identical.


<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Singleton dimension, defines a scalar value</td>
</tr>
<tr>
<td>M</td>
<td>Number of measurements</td>
</tr>
<tr>
<td>R</td>
<td>Number of receivers (e.g. the two ears)</td>
</tr>
<tr>
<td>E</td>
<td>Number of emitters (e.g. number of loudspeakers in a complex measurement setup)</td>
</tr>
<tr>
<td>N</td>
<td>Number of data samples describing one measurement (e.g. length of one impulse response)</td>
</tr>
<tr>
<td>C</td>
<td>Coordinate dimension. Always three.</td>
</tr>
</tbody>
</table>

Table 1: SOFA dimensions names (adapted from [SOF14c])

As SOFAHizer exclusively works with the *SimpleFreeFieldHRIR* convention, it follows the descriptions and definitions in [SOF14b]. The most relevant variables used in SOFAHizer are:

- **SourcePosition**: The different sound source positions in an HRTF are modeled as a change of SourcePosition, so its dimension should be \([M C]\). The coordinate system type is arbitrary; currently “spherical” (azimuth angle, elevation angle, radius) or cartesian coordinates are supported.

- **Data.IR**: The actual impulse responses are contained in the matrix variable Data.IR which has the dimension \([M R N]\).

- **Data.Delay**: Is an additional broadband delay “before” the actual impulse responses and could be available as \([I C]\) or \([M C]\).

- **Data.SamplingRate**: Is a scalar value (dimension \([I]\)) which defines the sampling rate of the impulse responses contained in Data.IR and the unit of the delay values in Data.Delay. The unit of Data.SamplingRate is set to Hertz in *SimpleFreeFieldHRIR*.

Within SOFAHizer, the custom struct variable of type `nc_sofa_s` holds all the information of one SOFA file. Thus, this struct allows for a complete numerical representation of an entire set of head-related impulse responses in the C-based audio filter module SOFAHizer for VLC player. Table 2 shows the members of the struct and their size. The short description also shows how each of the struct members is related to a certain variable or dimension in SOFA.
<table>
<thead>
<tr>
<th>Type</th>
<th>member name</th>
<th>Size</th>
<th>Description / relation to SOFA file</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>i_ncid</td>
<td>sizeof(int)</td>
<td>NetCDF ID of the opened SOFA file</td>
</tr>
<tr>
<td>int</td>
<td>i_n_samples</td>
<td>sizeof(int)</td>
<td>Number of samples in one impulse response. Corresponds to dimension N in SOFA file.</td>
</tr>
<tr>
<td>int</td>
<td>i_m_dim</td>
<td>sizeof(int)</td>
<td>Number of Measurements positions. Corresponds to dimension M in SOFA file.</td>
</tr>
<tr>
<td>float*</td>
<td>p_sp_a</td>
<td>sizeof(float)*M</td>
<td>Azimuth angles of all measurement positions for each receiver (i.e. the two ears). Read from SourcePosition in SOFA file.</td>
</tr>
<tr>
<td>float*</td>
<td>p_sp_e</td>
<td>sizeof(float)*M</td>
<td>Elevation angles of all measurement positions for each receiver (i.e. the two ears). Read from SourcePosition in SOFA file.</td>
</tr>
<tr>
<td>float*</td>
<td>p_sp_r</td>
<td>sizeof(float)*M</td>
<td>Radii of all measurement positions for each receiver (i.e. the two ears). Read from SourcePosition in SOFA file.</td>
</tr>
<tr>
<td>float*</td>
<td>p_data_ir</td>
<td>sizeof(float)<em>2</em>M*N</td>
<td>Impulse responses for left and right ear at each measurement position.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[IR_pos1L_s1 IR_pos1L_s2 ... IR_pos1L_sN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IR_pos1R_s2_pos1R_s1 ... pos1R_sN IR_pos2L_s1 ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IR_pos2R_s1 ... IR_posM_R_sN]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Read from Data.IR in SOFA file.</td>
</tr>
</tbody>
</table>

Table 2: Members of struct nc_sofa_s in SOFAlizer. Their size in memory and their relation to variables and dimensions in a SOFA file are given.
3 Digital signal processing for binaural rendering

3.1 Bandlimited resampling

Digital audio signals are usually sampled at a constant discretization time interval, the reciprocal of which is referred to as the sampling frequency or sampling rate. In case of binaural rendering, if an audio signal stream has a different sampling rate then a head-related impulse response (HRIR) set, one of the signals need to be converted to a different sampling rate which is referred to as resampling. As a new discretization time step is used when resampling a signal, it is required to find the values of the signal at an arbitrary time in-between two samples. One approach is to use linear or higher-order interpolation, however, for high quality audio-related purposes a more precise approximation is necessary (see [SMI15], p. 2 of pdf version).

The resampling library used in this project is based upon bandlimited interpolation of discrete-time signals as described in [SMI15].

Assuming a continuous-time signal $x(t)$ is sampled with a time interval $T_s$ (with a corresponding sampling frequency $f_s = 1 / T_s$), it needs to be bandlimited to $f_s/2$ according to the Shannon Theorem. If the Shannon Theorem is met, the signal can be uniquely reconstructed despite the sampling process. This can give rise to the following considerations:

- The process of sampling can be interpreted as multiplication of the time domain signal with a pulse train, which has unity values at integer multiples of $T_s$ and is zero, otherwise.
- The bandwidth restriction can be interpreted as a multiplication with a rectangular window in frequency domain, which corresponds to convolution with a sinc function in time domain.\(^4\)
- Thus, the pulse (corresponding to one sample value) convolved with a sinc function $h_s$ results in this sample's contribution to the reconstructed time domain signal as shown in Figure 2.

\(^4\) sinc(x) = sin(x) / x
The entire reconstructed time domain signal is thus found by adding shifted sinc functions weighted with the sample values at each time instant $n$ as shown in Figure 3.

The bandlimited interpolation algorithm is based on a different interpretation of the above figure and equation:

The signal value $x(t)$ at an arbitrary time $x$ can be found by assuming the peak of one sinc function to be at time $x$. Then, the contribution of the discrete signal sample at time instant $n$ is found by multiplying its value with the value of the sinc function at the time instant $n$. Again, adding the contributions of all signal samples results in the reconstructed signal.

In practice, this allows to obtain the signal value $x(t)$ at any desired time $x$ during the resampling process. However, the infinite sum over the filter coefficients needs to be truncated. And secondly, the filter coefficients are also discrete, which means that interpolation might be necessary to find the desired coefficients.

When using a sinc look-up table with sufficient oversampling and accuracy of filter coefficients, though, it is possible to assure that the error will not be determined by these quantization errors.
effects, but rather by the design of the lowpass filter itself as shown in [SMI15].

The resampling library *libsamplerate*, which is used in this project, provides look-up tables of different sizes which allow the user to choose the optimal trade-off between quality and required computational power.

### 3.2 Time domain convolution

From linear time-invariant (LTI) system theory it is known that the output signal $y[n]$ of a system can be found by convolution of the input sequence $x[n]$ with the impulse response $h[n]$ of the system (from [JOC03]):

$$y[n] = \sum_{k=-\infty}^{k=+\infty} h[k] x[n-k]$$

A causal, discrete impulse response consisting of $N$ samples $h_0, h_1, h_2$ etc. can be written in vectorial form as $h = [h_0 \ h_1 \ h_2 \ ... \ h_{N-1}]^T$. With a tap-input vector $x[n] = [x[n] \ x[n-1] \ x[n-2] \ ... \ x[n-N+1]]^T$ containing the last $N$ input samples, the above convolution sum can be simplified to a vector multiplication:

$$y[n] = h^T \cdot x[n]$$

From a programming point of view, this equation can be implemented using a loop to multiply and add the corresponding input samples and filter coefficients. The output signal at time instant $n$ can thus be obtained provided the last $N$ input samples are known. However, the problem is that the computational complexity is proportional to a square function: $O(N^2)$.

When considering binaural rendering of a common 5.1 surround signal, 5 input channels must be convolved with the corresponding impulse responses to both the left and the right ear. In case of the SOFAlizer audio filter module for VLC player, this means that 10 convolutions need to be computed in real-time. A modern laptop can achieve this for impulse responses up to a length of a few hundred samples, but at that point it is running at full CPU consumption just for audio playback, which is not desirable from a usability point of view. Therefore, an alternative concept for computation of the convolution needs to be used.
3.3 Fast convolution

The fast convolution method is based on the principle that linear filtering can either be achieved by convolution of input signal and impulse response in the time domain (see previous section) or equivalently by multiplication of the spectra of these two signals (frequency domain; see [SMI11]):

\[
Y[k] = H[k] \cdot X[k]
\]

with \(Y[k] = \text{FFT}(y[n])\), \(H[k] = \text{FFT}(h[n])\), \(X[k] = \text{FFT}(x[n])\) \(5\)

with \(X[k]\) being the spectrum of the input signal, \(H[k]\) being the transfer function of a system (in case of binaural rendering this is an HRTF) and \(Y[k]\) being the spectrum of the desired output signal; \(k\) is the discrete frequency variable. The desired output signal \(y[n]\) in the time domain is obtained by inverse-transforming \(Y[k]\):

\[
y[n] = \text{IFFT}(Y[k])
\]

The key point is that the fast fourier transform (FFT) can be used to transform the signals forth and back between time and frequency domain in a computationally very efficient way. If

\(N\) ... filter length,
\(M\) ... number of input samples to be processed,

the convolution length \(L\) should be at least \(L = N + M - 1\) in order to avoid circular convolution. As FFTs are most efficient when operating with vector lengths that are powers of 2, both signals \(X[k]\) and \(H[k]\) are usually zero-padded to the next power of 2 of length \(L\).

Such usage of the FFT allows to reduce the computational complexity to be proportional to \(O(1.5 L \log_2(L))\).

From an implementation point of view, the real-time input signal \(x[k]\) is split in segments of length \(M\), zero-padded and then convolved with the zero-padded impulse response taking the more efficient route via the frequency domain. Figure 4 exemplarily shows three such segments \(x_F[n]\), \(x_{F+1}[n]\) and \(x_{F+2}[n]\). Due to the decay of the impulse response (IR), the contribution of one input segment to the output \(y[n]\) overlaps to the next segment. Therefore, this overlapping part of the output signal has to be saved in a buffer and then added to the output signal of the next segment.
Figure 4: Overlap-and-add of three input signal segments $x_F[n]$, $x_{F+1}[n]$ and $x_{F+2}[n]$ convolved with an impulse response $h[n]$ (adapted from [SMI11]).
4 Development for VLC Player

4.1 The program concept

VLC player is developed by VideoLAN [VID15], a non-profit organization dedicated to open-source multimedia software. The team is international, though centered in France, and mainly consists of volunteers contributing to the project.

VLC player is a free, open-source and portable media player, which is widely used. Therefore it is a well-suited platform for development of a real-time binaural rendering tool using SOFA files.

From a programming point of view (see [VLC13]), the perhaps most distinguishing feature of VLC player is its modularity. Demuxers, decoders, audio and video filters, output and even the user interface are modules. One advantage of the using modules is, that it is possible to write different versions of a certain module for different operating system. This way, portability between operating systems can be achieved in a clear and practical way.

The modules are managed by the so-called libVLCcore, which also handles threads and synchronization of audio and video at the output. Decoding and playing are done by separate threads and are thus asynchronous. Audio and video decoding are also separated, but the samples are time-stamped to allow for properly synchronized presentation at the right time.

When starting playback of an audio file, libVLCcore dynamically loads the required filters and creates a “filter pipeline” consisting of the modules necessary for successful playback. As a simple example, if a file has an audio sampling rate of 44.1 kHz but the output is required to be 48 kHz, a resampling module (a so-called audio filter module) is loaded and inserted to the audio processing chain. Likewise, other modules such as equalizer and compressor are inserted, if the user enables them.

Conceptually, SOFAlizer, the binaural rendering plugin presented in this thesis, is thus realized as an audio filter module. Like with other effects such as the equalizer, it can be inserted to the processing chain by enabling it in the audio effect settings.

4.2 Audio filter modules

The generic structure of audio filter modules follows the generic structure of general modules. Each module includes a description of itself and a definition of its parameters in the module descriptor section which begins with the command vlc_module_begin().

Apart from that, modules require an Open function which is called when VLC tries to load the module. It is passed a pointer of type vlc_object_t to the audio module which can be cast to type filter_t. filter_sys_t is a member of filter_t and will be used as a complex struct data type for p_sys, where private data to be used within the module is stored.

Set up of the filter parameters and memory allocation for the private data is also usually done in
Open. If Open fails, the module cannot be loaded and any allocated memory needs to be freed to avoid leakage.

When unloading a module, the function Close is called, which is expected to free all memory and close any I/O devices etc. Close is only called when a module has already been successfully loaded, so it is not used for error handling if Open fails.

Audio filter modules have another important function called DoWork, which is the audio processing routine. It is passed a pointer to the filter structure of type filter_t containing the private filter data in p_sys and a pointer to the input buffer consisting of the buffer itself and some settings (such as buffer length).

All audio processing is done in DoWork. The results of the computation (i.e. the output samples) are written to the output buffer also consisting of the buffer itself and the same settings. DoWork returns a pointer to this output buffer.

The source code of a module should be located in the right subdirectory, for example, all audio filter modules are located in modules/audio_filter (from the VLC base directory’s point of view). The corresponding Makefile.am in that folder needs to be edited to tell the compiler that the module exists and which source files are needed for it.

4.3 Version control and submitting patches

VLC player is developed and improved by numerous people and it is thus very important to have a good version control system which handles all the contributions and suggestions from different people. The main tool used in this context is Git, an advanced version control system.

On the git sub-page of the VLC developers corner [VLC13], a good overview of how to use Git with VLC player is given, whereas [CHA09] features an in-depth introduction to Git itself. The aim of this section is thus to provide an insight to the workflow and the most important commands in conjunction with development of an audio filter module like SOFAlizer, rather than giving an exhaustive introduction to Git.

After installing Git, the VLC source code can be obtained by cloning it to a local folder with the following command in the terminal:

```bash
git clone git://git.videolan.org/vlc.git
```

This downloads the official VLC source code from the github repository including the entire history of changes. If only the latest version is required, adding --depth x to the command downloads just the last x revisions of VLC.

After that, VLC player needs to be compiled for the target operating system. Detailed instructions are given in the developer's wiki. Once compilation works, the basic workflow in

---

5 https://wiki.videolan.org/Git/, last visited on 18th June, 2015
6 https://wiki.videolan.org/UnixCompile/
terms of version control is that you

• edit some of the code or add new files,
• then make a “commit”, which is like a snapshot of the current state of the project,
• produce a patch which includes all the differences between the previous and the latest commit and,
• finally, post this patch to the vlc-devel mailing list for developers, where it is reviewed and commented.
• If the VLC developers decide that your contribution should be included in the official version of VLC player, they add it as a new commit to the official github repository.

Before committing any changes, it is important to explicitly add any new files you have created to the repository, otherwise they will show up as “untracked”, but will not be included to the commit. For example, the main source file of SOFAlizer could be added as follows:

```bash
git add modules/audio_filter/sofalizer/sofalizer.c
```

For details on committing, see [CHA09], but if you simply want to commit all changes that you made, you can use

```bash
git commit -a
```

Other important commands are:

```bash
git log ... list the history of commits in the current repository within the command line
gitk ... list the history of commits in the current repository in a graphical interface
git show --color commit-number ... shows all changes in a commit
```

The `commit-number` is a checksum that is listed, e.g. with `git log`. The above `git show` command is very handy to check for unwanted white space in the source code, as it will be highlighted in red.

If you wish to modify the last commit, this can be done with

```bash
git commit -a --amend
```

where `-a` again simply adds all recent changes to the commit.

Once the commit is ready to be posted, empty the `patches` folder, create a patch from the latest commit and then send it to an arbitrary e-mail address. The example given here shows the devel list address, but it could be good practice to first send the patch to your own address to check whether everything is right:

```bash
rm -Rf patches
git format-patch -o patches origin
git send-email --to vlc-devel@videolan.org patches
```

A module with a graphical user interface (GUI) is preferred to be sent in two separate patches

---

7 Please note that before first using the mailing functionality, `git-send-email` needs to be set up to correctly use your personal email account (see [https://wiki.videolan.org/Git/#Mail_Setup](https://wiki.videolan.org/Git/#Mail_Setup), last visited on 21st May, 2015).
that still belong together, though. The idea is to have two commits, where one includes all changes related to the GUI and one includes the main functionality. This might require to split the last commit into two, which can be achieved, as follows:

-  
  - git rebase HEAD
  - git add -p ... allows you to go through all edited chunks and decide whether or not to use them in the current commit
  - git add ... if you want to add an entirely new file
  - git commit -s ... creates a commit from the changes you included

Start with the changes that shall become PATCH 1 and then repeat the procedure starting with `git add -p` to include the changes for PATCH 2 (usually the GUI-related changes).

If for any reason the last 2 commits need to be changed (number is arbitrary, of course), execute a `rebase` command and then follow the displayed instructions:

-  
  - git rebase -i HEAD~2

Once the two commits are ready to be posted, the above patch and mailing commands change to:

```
  git format-patch -o patches HEAD~2
  git send-email --compose --no-chain-reply-to --to mail@address.com patches
```

Finally, the SOFAZizer source code can be obtained in similar fashion from the official SOFA github repository:

```
https://github.com/sofacoustics/ (last visited on 21st May, 2015)
```

### 4.4 Development tools

A couple of development tools were used to alleviate the debugging process during the development phase. These shall be briefly introduced, here, starting with gdb and valgrind; then going on to some useful code snippets for printing to file (and possible visualization with other software) and measuring execution time.

**gdb**

As introduced in [GDB09], gdb is a “GNU debugger”, that helps to find programming errors and memory leaks by giving insight to what the program is doing at a certain point of execution.

It is started from the command line. Once it is running, the executable to be tested needs to be loaded (e.g., assuming we already are in the `bin` folder of the VLC player source tree):

```
  file ./vlc-static
```

Then it can be run within gdb:

```
  run
```
gdb allows to set breakpoints at certain points in the code and get more verbose information if any errors, such as segmentation faults, occur. This includes the line the crash occurred at and the possibility to display a “backtrace”, which shows the function stack at the time of the crash.

The gdb commands that were found to be most useful during the development of SOFAlizer are:

```
kill ... terminates a crashed application
break file1.c:6 ... breakpoint at line 6 of Files file1.c
break my_func ... breakpoint at function my_func(...)
info breakpoints ... shows information about all breakpoints
delete x ... delete breakpoint x
continue ... continues the program after a breakpoint halt
step ... only executes the next line after a breakpoint halt
next ... similar to step, but does not go through sub-routines line-by-line
    (generally, simply hitting Enter repeats the previous command)
print my_var ... prints the value of the variable my_var (print/x for hexadecimal output)
backtrace ... traces function stack that lead to a crash (e.g. segmentation fault)
where ... same as backtrace, but to be used when the program is still running
```

**valgrind**

As explained in [ALL11], valgrind is very helpful for tracking memory leaks or investigating invalid memory access.

Similar as with gdb, VLC player needs to be started with valgrind, but in this case, a single terminal command is sufficient:

```
valgrind --tool=memcheck ./vlc-static
```

The following options can be added to get more verbose information:

```
--leak-check=yes or --leak-check=full and additionally --show-leak-kinds=all
```

This can give very helpful in-depth information about the memory operations of the software-under-test. However, the disadvantage is that in case of a large piece of software like VLC player, the command line output is too long to find anything.

Therefore, it has proven advantageous to redirect the output to a log file, which allows to search for certain terms (e.g. “sofalizer”) with a simple text editor. Thus, the following command can be used:

```
valgrind --tool=memcheck --leak-check=yes --show-leak-kinds=all ./vlc-static > log-file-name 2>&1
```

Both valgrind and gdb slow down the software-under-test; in the case of valgrind this is very noticeable.
Useful C commands
Apart from these tools, conventional printf commands often help to gain better understanding of what goes on in the program without the need to set breakpoints and print variables manually. In some cases, in particular when dealing with signals, plots of these signals can help a lot to check for or find errors. For this purpose, values (e.g. an impulse response before and after resampling) can be printed to files which can then be read and visualized by, e.g., Matlab. For example, in case of the resampling implementation this helped to verify and debug an assumed error (see Figure 6 for an example of a possible graphical output).

From a programming point of view, the stdio.h header needs to be included and file pointers are used to open a new file to which the program can print:

```c
#include <stdio.h>
...
FILE fp_pre_resample; /* file pointer */
fp_pre_resample = fopen( "pre_resample", "w" ); /* open file */
...
for( int j = 0; j < p_sys->sofa[i].i_n_samples; j++ ) /* print to file */
    fprintf( fp_pre_resample, "%f\n", ( p_sys->sofa[i].p_data_ir + j ) );
...
fclose( fp_pre_resample ); /* close file */
```

Sometimes, execution time of a certain part of the code needs to be evaluated, e.g. when comparing different implementations of a certain functionality or different quality settings of the resampling algorithm.

Similar to before, an own header needs to be included. The timer is started and then stopped after execution of the code section of interest.

```c
#include <time.h>
...
clock_t start, stop;
double f_t = 0.0;
start = clock();
/* code to be evaluated */
...
stop = clock();
f_t = ( double ) ( stop - start ) / CLOCKS_PER_SEC;
printf( "\nrun time: %f\n", f_t );
```
5 SOFA\textit{izer}: Binaural rendering in VLC player

5.1 Integration within VLC Player

SOFA\textit{izer} is a \textit{module} of type \textit{audio filter} in VLC player. It relies on the netcdf library to load SOFA files which contain the head-related impulse responses (HRIRs) used for binaural rendering. In order to add a new audio filter module and enable the use of a library, some configuration files need to be edited:

- \texttt{configure.ac} in the root directory of the VLC player sources
  - add \texttt{sofalizer} to the following list:
    \begin{verbatim}
    AC_CHECK_LIB(m,cos, [VLC_ADD_LIBS([adjust ... oldmovie glspectrum sofalizer],[-lm])
    \end{verbatim}
  - enable netcdf package:
    \begin{verbatim}
    PKG_ENABLE_MODULES_VLC([SOFALIZER], [], [netcdf >= 4.1.1], [netcdf support for sofalizer module], [auto])
    \end{verbatim}

- \texttt{Makefile.am} in modules/audio_filter:
  - add entry for SOFA\textit{izer} and assign all source files related to SOFA\textit{izer} to it:
    \begin{verbatim}
    libsofalizer_plugin_la_SOURCES = \ sofalizer/sofalizer.c \ sofalizer/resampling/samplerate.c ...
    libsofalizer_plugin_la_LIBADD = $(LIBM)
    \end{verbatim}
  - add libsofalizer_plugin.la to audio_filter\_\texttt{LTLIBRARIES}

Of course, netcdf must be installed to provide read-access to the SOFA files (as long as it is not provided as part of VLC player) and all necessary source files must be placed in the SOFA\textit{izer} source directory \texttt{modules/audio_filter/sofalizer}. During compilation, netcdf must be enabled by adding \texttt{--enable-sofalizer} to the \texttt{./configure} command.

For faster user interaction, a graphical user interface (GUI) has been added to SOFA\textit{izer}. When a parameter is changed on the GUI, the corresponding callback function in the SOFA\textit{izer} source code is called and the internal processing is updated to match the new settings. The actual interface is created by adding it to the source code in the \texttt{qt4} directory of VLC player. For each parameter, a link between the callback variable in the audio filter module and the actual GUI controls in the audio effect settings need to exist.

- In \texttt{modules/gui/qt4/dialogs}, a new tab for SOFA\textit{izer} is added to the audio effect settings:
  \begin{verbatim}
  Sofalizer *sofalizer = new Sofalizer( p_intf, audioTab );
  audioTab->addTab( sofalizer, qtr("SOFA\textit{izer}" ) );
  \end{verbatim}

\footnote{The SOFA\textit{izer} source code files plus the edited configuration files can be obtained from \url{https://github.com/sofacoustics/SOFAlizer} (last visit 27\textsuperscript{th} May, 2015).}
CONNECT( writeChangesBox, toggled(bool), sofalizer, setSaveToConfig(bool) );

• add one slider for each callback variable (e.g. sofalizer-select) to the UI in modules/gui/qt4/components/extended_panels.cpp:

```cpp
Sofalizer::Sofalizer( intf_thread_t *p_intf, QWidget *parent )
    : AudioFilterControlWidget( p_intf, parent, "sofalizer" )
{
    i_smallfont = -1;
    const FilterSliderData::slider_data_t a[6] =
    {
        { "sofalizer-select", qtr("Select File"), "", 1.0f, 3.0f, 1.0f,
            1.0f, 1.0f },
        { "sofalizer-gain", qtr("Gain"), qtr("dB"), -20.0f, 40.0f, 0.0f,
            1.0f, 1.0f },
        ...
    };
    for( int i=0; i<6 ;i++ ) controls.append( a[i] );
    build();
```

• add a declaration of the previous to the corresponding header file extended_panels.hpp

The graphical user interface (GUI) of SOFAlizer shows up as a tab in the audio effect settings of Figure 5: Graphical user interface (GUI) of SOFAlizer (left) and advanced preferences with all user-controllable parameters of SOFAlizer (right).

The graphical user interface (GUI) of SOFAlizer shows up as a tab in the audio effect settings of...
VLC player. More verbose settings can be made in the advanced preferences panel as shown in Figure 5. The GUI consists of a enable/disable checkbox and a couple of sliders:

- **Select File:** Choose between three different SOFA files and thus HRIR sets to be used for binaural rendering. The impulse responses and all related data are already loaded at filter start-up. This allows for instant switching between the files during playback without any interruption. The paths of the SOFA files can be specified in the advanced preferences.

- **Gain:** Global gain applied during the rendering process. This feature can be used to compensate for possible volume abnormalities in HRIR sets.

- **Switch:** If Switch has a value of 0, the position of the virtual loudspeakers depend on the input file format and the settings of the Rotation and Elevation sliders. With the integer values 1 – 4, it is possible to place all virtual loudspeakers at four different positions defined in the advanced preferences (parameter Elevation/Azimuth Position 1 – 4).

- **Rotation:** Rotates the virtual loudspeakers by applying the specified offset to the azimuth angle of their position (if Switch is 0).

- **Elevation:** Elevates the virtual loudspeakers by applying the specified offset to the elevation angle of their position (if Switch is 0).

- **Radius:** Allows to choose the distance between listener and virtual loudspeakers. It requires SOFA files which contain HRIR measurements at various distances from the subject. If the specified distance is not available, the closest available source position is chosen.

In addition, the advanced preferences offer settings for the audio processing algorithm (time domain or frequency domain convolution) and the quality of HRIR resampling, which both need to be set before runtime (in the current implementation).

### 5.2 Cross-compilation issues

A Windows 32 bit compatible version of VLC player can be cross-compiled on Unix following the instructions on VLC developers corner. The original proof-of-the-concept version of SOFAlizer successfully cross-compiled with VLC 2.1.0 on Ubuntu 13.04, however, does not compile after updating the operating system to Ubuntu 14.04. With the development version VLC 3.0.0, cross-compiling failed regardless of the operating system version. A lot of effort was put into successful cross-compilation and some of the issues could be resolved. However, the main issue, seemingly a confusion between the graphical interface qt version 4 and 5, remained. Via the VLC forum, the VLC team confirmed that, once a Linux version of SOFAlizer is accepted by the them, they would take care of a successful Windows build, anyway. For this reason it was

---


decided that resolving cross-compilation is not a major objective as it seemed to require a lot of time and effort.

Instead, it was decided to develop SOFAizer for Unix and then occasionally copy the new source files to a virtual machine running Ubuntu 13.04 and cross-compile for Windows with VLC 2.1.0. This way, the functionality of SOFAizer can still be sufficiently tested on Windows.

The integration of the new source files to the "old" project works as expected. However, it is important not to replace any configuration or GUI-related files, but only those parts of these files that affect SOFAizer.

Another little drawback is that in the file float_cast.h of the resampling library, a problem with asm assembler commands occur when cross-compiling. Thus, the #ELIF section in the code belonging to WIN32 (lines 150 - 184) needs to be commented out as a workaround, which prompts the fallback solution using standard C casts instead of the assembler commands.

Finally, it is necessary to copy the pre-compiled netcdf library for Windows Visual Studio to the contrib/i686-w64-mingw32 folder before compilation. After compilation and before execution, some dll files need to be copied to the folder of the Windows executable (or at least make sure the system finds them at a location specified in the PATH variable). 11

5.3 Program structure and flow

The basic structure of SOFAizer follows the general audio filter module structure described in section 4.2 Audio filter modules. The first section of the source file mainly comprises of:

- copyright information
- header file includes
- definition of some macros (e.g. N_SOFAR for number of SOFA files to be loaded simultaneously for instant comparison)
- definition of the structs (contains information about the state of the filter, the rendering and stores the impulse response data)
- static variables and macros are defined for menu help texts
- function declarations where necessary
- module descriptor (including GU variables)

The rest of the source file contains all functions. The main functions Open, DoWork and Close are called at filter start-up, during playback and when closing the filter, respectively. GUI callbacks are called whenever a slider on the GUI is changed by the user.

The following gives a structured overview of what these functions do and the most important

11 The following files need to be copied: hdf5.dll, hdf5_hl.dll, libcurl.dll, msvcp100.dll, msvcr100.dll, netcdf.dll, szip.dll, zlib.dll, zlib1.dll
sub-routines they call:

- **Open**: Is called when loading SOFAlib, initializes the module and loads relevant data.
  - variable initializations, get user settings from GUI and advanced preferences
  - **LoadSofa** is called three times in a loop (each call loads data from one SOFA file)
    - opens SOFA file
    - gets dimension names and lengths from SOFA file
    - checks file and convention type (i.e. if file follows SimpleFreeFieldHRIR convention)
    - loads source positions, impulses responses and sampling rate of the data
    - loads broadband delay (including handling of different dimensions)
  - resample impulse responses (if IR sample rate does not match audio stream rate)
  - set up rendering parameters (such as ring buffer length)
  - allocate memory for currently used set of impulse responses, ring buffers, and current virtual loudspeaker positions
  - **CompensateVolume**: volume normalization
  - **GetSpeakerPos** gets virtual loudspeaker positions from input file format
  - **LoadData**: Loads currently needed set of impulse responses.
    - variable initializations and temporary memory allocation (for IRs)
    - **FindM** finds impulses responses closest to the required source positions (which are either based on the input file format or on the user settings if the Switch slider on the GUI is used to place all virtual loudspeakers at a certain position)
    - loads impulses responses first to temporary memory then to nc_sofa_s struct
  - define DoWork as the audio processing routine
  - add callback functions for the GUI variables (which are called whenever the corresponding variable is changed on the GUI)

- **GUI callbacks**: Called if a slider on the GUI is changed.
  - store new value in p_sys struct
  - **LoadData**, i.e. reload IRs or HRTFs based on the new GUI settings

- **DoWork**: Is called for each input segment during playback. Gets input buffer (played back audio samples) and returns output buffer (processed output signal).
  - prepare and allocate output buffer
In case of frequency domain processing, if FFT length has changed:

- retrieve new FFT length
- allocate configuration memory for KISS FFT with new settings
- `LoadData`: Loads impulse responses similar to above, but also zero-pads them to the new FFT length and then transforms them to frequency domain via FFT.

- prepare thread data for left and right channel
- computation of the convolution in sub-routines:
  - In case of time domain processing: `sofalizer_Convolute` (see section 5.6 Audio processing: Implementation of time domain convolution) is called in two threads for parallel computation of left and right output channel (this almost double performance).
  - In case of frequency domain processing: Call `sofalizer_FastConvolution` (see section 5.7 Audio processing: Implementation of Fast Convolution).

- count clipped output samples

- **Close**: Assures that SOFAfile closes neatly.
  - delete callbacks
  - free allocated memory via `FreeAllSofa` and `FreeFilter` (see next paragraph)

With C programming, it is particularly important to also take care of correct error handling. Whenever memory is allocated or a function, which might fail, is called, any memory that had been allocated up to that point needs to be freed. There are three functions which summarize free operations and thus allowing for more compact error handling:

- `CloseSofa`: Closes one given SOFA file and frees its associated memory.
- `FreeAllSofa`: Free all memory allocated for storage of information from SOFA files.
- `FreeFilter`: Free all memory allocated in `p_sys` struct.

---

12 Note that this includes the first time `DoWork` is called, because before that, FFT length is initialized to zero. Calculating the first transformation of IRs during Open unfortunately is impossible, as the input buffer length (which also determines the FFT length) cannot be retrieved at that point (see https://forum.videolan.org/viewtopic.php?f=32&t=124373&p=417808, last visited on 10th June, 2015).
5.4 Source code improvements

During the work on this project, the source code has been constantly improved in many aspects. Part of these improvements were based on the comments and suggestions from various VLC developers in the mailing list vlc-devel, where several revisions of the code have been posted.

A significant step was the verbose documentation of the entire code in form of comments within the code itself. Before, comments were quite sparse, whereas now, the documentation facilitates future development and will also help future developers of the project when getting familiar with the code.

Most of the pointer variables had to be renamed to follow the VLC code conventions due to an initial misunderstanding of these conventions.

Along that line, some error paths were fixed and simplified. The dimension check of Data.Delay was also improved in terms of logics and readability.

Some scenarios, where SOFAlizer would cause VLC player to crash, were investigated and resolved. For example, when loading long impulse responses (in that case approx. 32.000 samples) a segmentation fault would occur. Finally, it turned out that the temporary arrays in which the impulse responses were stored during loading (in LoadData) could not handle such large amount of data. Using dynamic heap variables instead, immediately solved the problem, once it was located.

Another segmentation fault issue when using SOFA file 3 was analyzed and turned out to be a case of a “zero-based index” vs. “one-based index” mess-up.

In addition, the handling of file load errors was improved. Up to then, SOFAlizer would only load if all three SOFA files could successfully be loaded, otherwise the module would be reported as corrupt. In the current implementation, SOFAlizer loads if at least one SOFA file can be loaded successfully. File load errors are still reported as debug messages. In case that a file could not be loaded but is selected in the GUI, the audio output is muted by setting the internal variable p_sys->b_mute to true, causing the output buffer to be filled with zeros.

Small performance optimizations included using the function clz (count leading zeros in an integer value) to determine the buffer and FFT lengths as a power of two. Where possible, floating point versions of functions (as opposed to double precision, which basically wastes CPU power) were used (e.g. fabs instead of abs).

Struct members were reordered so as not to destroy efficient packing in memory (see [RAY15]).

A lot of minor improvements and corrections were made such as an update on which strings need to be translated for internationalization, fixed priority of threads for convolution, correct usage of the functions var_Inherit vs. var_CreateGet, etc.

The following section describes how the problem of matching the sampling rate of impulse responses and audio stream was addressed and solved.
5.5 Resampling of filter responses

In the initial “proof-of-the-concept” version of SOFAlizer, the sampling rate of the audio processing was determined by the sampling rate of SOFA file 1 defined in the advanced preferences. Thus, the requested sampling rate of the input audio stream of SOFAlizer was set to this value during the initialization. If the original sampling rate of the audio stream differed from the rate of SOFA file 1, VLC player set up a resampling module before SOFAlizer, in order to match the sampling rate of the audio stream to the requested rate. If the sampling rate of SOFA file 2 or 3 differed from the rate of SOFA file 1, they had to be discarded and set to be invalid and unusable during the initialization of SOFAlizer.

In addition, resampling the audio stream crashed VLC when playing back 16 bit wave files.\textsuperscript{13} A bug report was filed,\textsuperscript{14} as suggested by Jean-Baptiste Kempf (main developer of VLC player) in the developer's forum. However, it would only have been worked on upon providing a simpler test case. It was decided not to further pursue this, because even after resolving this issue, SOFA files with different sampling rates could not have been loaded at the same time.\textsuperscript{15}

Instead, the more beneficial approach of matching the sampling rate of the impulse responses to the rate of the audio stream was chosen. It exhibited the following advantages:

- no constant resampling of audio stream necessary, which saves performance
- loading SOFA files with different sampling rate at the same time is no longer a problem
- no need to resolve the above mentioned crash

This approach requires to resample each set of impulse responses that has a sampling rate different from the audio stream at the initialization stage. In order to achieve the resampling, the first idea was to use one of the libraries already included in VLC player. Including them in the SOFAlizer source code worked well and compiled without difficulties. However, at runtime, functions defined in those libraries were said to be undefined, which could not be resolved. When asked for help in the forum, Jean-Baptiste Kempf advised to re-implement those functions, anyway, to be less dependent. Therefore, it was decided to use the Secret Rabbit Code (SRC) resampling library (see [SRC11]). It is also included in VLC player, but the source code is available under the GNU General Public License (GPL), which allowed to include the needed files directly in the SOFAlizer source directory.

Another question to deal with is when to resample the impulse responses:

- One option is to resample all responses of all loaded SOFA files during filter initialization. This has the disadvantage that it might take some time until file playback can be started (depending on number and length of impulses responses, and on quality and efficiency of

\textsuperscript{13} See corresponding thread in VLC forum: https://forum.videolan.org/viewtopic.php?f=32&t=113578&p=385666#p385666, last visited on 18th June, 2015

\textsuperscript{14} https://trac.videolan.org/vlc/ticket/12814, last visited on 18th June, 2015

\textsuperscript{15} Theoretically, the audio stream sampling rate could be changed during playback as a workaround to achieve this. However, this seems not to be possible, anyway.
the resampling algorithm). The advantage is that head rotation can easily be simulated by rotating the virtual sources, which requires different sets of impulse responses to be loaded at each change of the rotation (e.g. when using SOFAlizer in conjunction with head-tracking).

- The second option is to only resample the currently needed set of impulse responses during playback (one IR per channel and ear). Of course, this saves a lot of computational resources, but one the other hand, it is totally impracticable for quickly changing head rotations.

It was decided to go for the first option and resample the impulse responses during filter initialization. If required, the waiting time before playback can be shortened by lowering the resampling quality of the SRC algorithm. A drop-down menu in the advanced preferences of VLC player allows the user to change this parameter for SOFAlizer and choose between the following settings:

- zero-order hold (very fast)
- linear (very fast)
- Sinc interpolation (three different qualities available)

SRC provides different application programming interfaces (APIs) for sample rate conversion of a simple block of samples, chunk-wise conversion for real-time applications and even functionality for time-varying output rate (see [SRC11]). For the application within SOFAlizer, the simple API consisting of the single function src_simple is sufficient. It takes a block of samples and converts its sample rate by a given conversion ratio and with a given conversion quality (see above).

The resampling is performed in a loop within the Open function, which loads all data from the three SOFA files to the _p_sys filter struct (function LoadSofa). For each file, its sampling rate is compared to the rate of the audio stream and if they do not match, the impulse response data of the file are resampled.

At first, the parameters of the sampling rate conversion such as the conversion ratio are determined. The new number of samples in one impulse response is given as

\[
i_{\text{n samples new}} = \text{ceil} \left( \frac{\text{double}}{\text{p_sys->sofa[i].i_n_samples}} \times d_{\text{ratio}} \right)
\]

which is the next integer of the original sample rate multiplied with the conversion ratio. These parameters and the pointers to input/output samples memory are stored in a struct of type SRC_DATA which is then passed to src_simple along with the user-defined resampling quality:

\[
i_{\text{err}} = \text{src_simple( } &\text{src, i_resampling_type, 1 )};
\]

src_simple is called in a for loop which consecutively converts all impulse responses of the currently processed SOFA file. The multi-channel mode of src_simple (specifying the number of channels to be converted in the last function argument) cannot be used here; it expects the samples of the channels to be interleaved, which is not the case with SOFA files.
After conversion, the memory for the impulse responses in `p_sys` of that file is reallocated to the new required number of samples. Then, the converted samples are copied from a temporary output buffer memory to the memory in `p_sys`. It is also necessary to update the broadband delay that can be separately specified in samples for each impulse response. A simple multiplication with the conversion ratio assures that the delay in seconds stays the same as before with the new sampling rate. As this also involves rounding to integer sample values, an even more accurate solution would be to add the delay to the IRs before resampling.

For debugging purposes, impulse responses were visualized by writing their values to files that can then be loaded into Matlab (see section 4.4 Development tools). Figure 6 exemplarily shows a comparison of an impulse response from the ARI database before and after sampling ratio conversion from 48.0 kHz to 44.1 kHz.

The upper plot shows the result of sampling rate conversion by means of linear interpolation, which is a computationally simple and cheap method. It reveals the inaccuracy of this method, which is particularly noticeable in the first couple of main peaks. The interpolation not only results in wrong magnitude values but even accounts for a slight time shift.

The middle and lower plots both show the results of sampling rate conversion using sinc interpolation (see section 3.1 Bandlimited resampling) with different accuracies of filter coefficients. They both get very close to the original signal and the difference between the two variants is hardly observable in the plots.

When using the linear interpolation performance (upper plot) as a reference, the fast sinc interpolation (middle plot) is roughly seven times slower, whereas the best quality sinc interpolation (lower plot) is about 40 – 60 times slower. This clearly points to the fast sinc interpolation as the winner in terms of a compromise between accuracy and efficiency. However, for rather short impulse responses (256 samples) one entire ARI HRTF set (1550 measurement points, each for both ears) could be converted in about 0.4 seconds on a modern laptop, even with the highest quality setting, which is still quite acceptable. If even faster conversion is required, parallelizing the resampling using threads might be investigated in the future.
5.6 Audio processing: Implementation of time domain convolution

At the time of this writing, SOFAlizer can use both conventional time domain convolution and frequency domain convolution by means of the Fast Fourier Transform (FFT) to perform the filtering of the input signals with the head-related impulse responses. A drop-down menu in the advanced preferences of VLC player allows the user to choose between these two settings.

In both cases, the processing is prepared and handled in the DoWork function. The output buffer memory is allocated and the settings are copied from the input buffer (e.g. number of samples in one buffer). However, different functions are then called for the actual computation of the convolution, depending on the chosen processing type.

In the case of the time domain convolution, the function sofalizer_Convolute is called twice: Once in one thread for the left and once in one thread for the right channel. All required pointers and...
variables are condensed into one struct of type \texttt{t\_thread\_data} for the left channel and one struct for the right channel. These structs are then passed to \texttt{sofalizer\_Convolute}. This means that the functions itself works in a way that it creates the output samples for either the left or the right channel, depending on the data passed to it. Although creating and destroying two threads per audio frame might seem unconventional, it actually nearly doubled the performance of SOFAlizer.

After computation of the output samples, \texttt{DoWork} adds the number of clipped samples in both channels, which are counted in the convolution computation functions, and issues an error message if any samples are above $\pm 1.0$.

The most important variables in the convolution functions – both time- and frequency domain – are:

\begin{verbatim}
float *p_src ... pointer to audio input buffer memory
float *p_dest ... pointer to audio output buffer memory
float *p_temp_ir ... pointer to memory block of impulse responses
float *p_delay ... pointer to broadband delays for each IR to be convolved
int i_input_nb ... number of input channels (including LFE channel)
int i_n_samples ... number of samples in one impulse response
float *p_ringbuffer[i_input_nb] ... starting addresses of ring buffer for each input channel

int i_buffer_length ... length of ring buffer for one input channel
int i_write ... current write position in ring buffer
\end{verbatim}

The ring buffer is used to store the past input samples for computation of the convolution. The length of one ring buffer is therefore determined from the longest impulse response plus its delay. For sake of performance, the next power of two is then chosen as the actual buffer length.

When reading from or writing to the ring buffers, a modulo operator is often needed to keep the read/write index within the bounds of the buffer. As the ring buffer length always is a power of two, the modulo operation can be replaced by a logical AND with the same number minus one, which is computationally much simpler.\textsuperscript{16} Therefore, a helper variable \texttt{i\_modulo} is introduced as \texttt{i\_buffer\_length – 1} and the expression \texttt{x \& i\_modulo} can be used to obtain \texttt{x} modulo the buffer length.

The input buffer contains the samples of all input channels in interleaved form. As described below, they are consecutively written to the ring buffers corresponding to each channels:

\textsuperscript{16} See \url{http://stackoverflow.com/questions/6670715/mod-of-power-2-on-bitwise-operators} and \url{http://stackoverflow.com/questions/3072665/bitwise-and-in-place-of-modulus-operator} for more detailed explanations (both links visited on 11\textsuperscript{th} February, 2015).
The output buffer contains the samples of the left and right channel in similarly interleaved form:

![Layout of the samples in the output buffer](image)

**Figure 8: Layout of the samples in the output buffer**

The basic implementation approach for the convolution is to

- go through all samples of all channels in the current input buffer,
  ```c
  for ( int i = t_data->p_in_buf->i_nb_samples ; i-- ; )
  ```
- write them to their respective ring buffers and
  ```c
  for ( int l = 0 ; l < i_input_nb ; l++ )
      *( p_ringbuffer[l] + i_write ) = *( p_src++ );
  ```
- then compute the output sample-by-sample for each channel by
  - taking the past N input samples\(^{17}\) from the ring buffer,
  - multiplying them with their corresponding N samples from the impulse response and
  - adding the multiplication results up.
  ```c
  for ( int j = i_n_samples ; j-- ; )
      *( p_dest ) +=
          *( p_ringbuffer[l] + ( ( i_read++ ) & i_modulo ) ) *
          *( p_temp_ir++ );
  ```

In the multiply-and-add step, the variable `i_read` is used for indexing the read operation from the ring buffer and is constantly increased by one as the loop advances through all samples of the impulse response. Its initial value before the loop is defined as follows:

\[
i_read = ( i_write - *( p_delay + l ) - ( i_n_samples - 1 ) + i_buffer_length )
\& i_modulo;
\]

It is based on the current writing position `i_write`, which points to the most recent sample.

\(^{17}\) N being the length of one impulse response.
However, as the impulse responses are time-reversed in the memory, the first convolution multiplication requires to get the input from \( i_{n\_samples} \) time steps ago. In addition, the broadband delay of the \( l \)-th channel \( *(p\_delay + l) \) is applied by subtracting this value from the read index. \( i_{buffer\_length} \) is added but only serves to stay away from negative numbers, as the result is taken \( mod i_{buffer\_length} \), anyway. Due to the sufficient length of the ring buffer, the required input samples are always available, independently of the length of the input buffer. In a last step, the LFE channel is added to both left and right output signals.

### 5.7 Audio processing: Implementation of Fast Convolution

The time domain convolution is very intensive on the CPU due to the large number of operations (see section 3.2 Time domain convolution). In fact, longer impulse responses (e.g. 2048 samples) cannot be rendered in real-time without annoying dropouts. Computing the convolution as a multiplication in frequency domain is usually much more efficient, provided that the Fast Fourier Transformation (FFT) is used to transform the signals to the frequency domain (see section 3.3 Fast convolution). The first step in the implementation of the fast convolution was hence to assure the availability of a function that calculates the FFT of a given data vector.

The first approach was to include existing functions from other parts of VLC player. Both the WMA decoder and a spectrum visualization module have their own FFT functions. However, direct access to these files was not possible for the same reasons as with the resampling functions (see section 5.5 Resampling of filter responses). It was decided to simply copy all necessary source files from these folders to the SOFAlizer folder, which allowed to successfully include, access and use the FFT functions.

The FFT from the visualizer immediately worked, but had a couple of restrictions:

- By default, it only uses 16 bit processing (type \texttt{short int} for the sound samples).
- By default, it only returns a power spectrum, not the complete complex result.
- The length of the FFT is hard-coded via a \texttt{DEFINE} command; so it cannot be changed at runtime.

While the first two points could be changed relatively easy, the third point would have required serious re-writing of the code. This option was dropped in order to be able to follow possible future updates to the used FFT function.

The FFT from the WMA decoder was better suited regarding its design. One minor drawback, however, was that integer variables are used for the sound samples, internally. The main obstacle, though, was that the FFT function itself expected the incoming data to be pre-processed in a certain way. This pre-processing was duplicated from the WMA decoder implementation to the SOFAlizer code, but correct results were still only achieved with special, sparse input sequences.
As a successful solution of these issues would have increased the programming complexity too much, it was decided to go for another approach, namely, to use the so-called KISS FFT library, which has a compatible license and worked out-of-the-box.

Other possible solutions, such as the common and very powerful FFTW library, were dropped due to their complexity and thus number of source files involved. They would have needed to be included as proper additional libraries in VLC player, which would have made integration of SOFAlizer even harder (than it already is, given that netcdf is used and needs to be established and accepted as an additional library).

The second issue was to determine the required length of the FFT and – linked to that – the question when to transform the impulse responses the frequency domain: In order to exploit the performance benefits of the fast convolution, one input buffer block should be processed at once. To avoid circular convolution, the length of one FFT is thus given as the number of input buffer samples plus the length of the longest impulse response (also taking the broadband delay into account). The problem is that the input buffer block size cannot be retrieved in the Open function of SOFAlizer, because, according to Rémis-Denis Courmont (an official VLC developer), it might change during playback. There also is no maximum buffer size, except physical memory limitations, as he explained in the VLC forum. Hence, the IRs have to be transformed right at the beginning of playback (in the first call to DoWork, when the input buffer size is first known), because it is necessary that the IRs are zero-padded to the correct FFT length before the transformation. And, theoretically, they have to be transformed again, if the input buffer size changes.

Transforming all IRs of all SOFA files at once has the advantage, that IR sets can be instantly changed (e.g. due to a GUI operation by the user or due to a head-tracking system). However, the transformation to the frequency domain took so long that the next call to DoWork happened, before it was finished. Therefore, we decided to only transform the currently needed IRs (based on the current GUI settings) in the LoadData function. Later on in the future, when using SOFAlizer with a head tracking system, this might put a lot of strain on the CPU, as the required IRs constantly change. However, this could be solved by internally storing the transformed IRs and then only transforming those that had previously not been transformed.

To sum the program flow up, DoWork checks if the input buffer size been changed (or has never been retrieved, so far), and in that case obtains the FFT length as the next power of two of the required convolution length (which is number of input buffer length plus IR length). Then, the configuration for FFT and IFFT with the obtained length using the KISS library is set up and stored in the p_sys struct for future use:

```
p_sys->p_fft_cfg = kiss_fft_alloc( i_n_fft, 0, NULL, NULL );
p_sys->p_ifft_cfg = kiss_fft_alloc( i_n_fft, 1, NULL, NULL );
```


LoadData is called, which – if the program is in frequency domain processing mode – loads the required impulse responses for each channel based on the current GUI settings in a loop, zero pads them to the convolution length and transforms them to frequency domain ($i_{\text{offset}}$ being an index for accessing different channels):

```c
kiss_fft( p_sys->p_fft_cfg, p_fft_in_l, p_data_hrtf_l + i_{\text{offset}} );
```

Note that in case that the FFT length does not change, but the GUI settings change, LoadData is also called to get the new required impulse responses.

After successfully setting up the FFT with a new length, the basic routine in DoWork continues (similar to the one described above in section 5.6 Audio processing: Implementation of time domain convolution), but then calls the function sofalizer_FastConvolution. The used variables are similar to the time domain processing case, as well.

One difference is that the ring buffer memory is smaller for the fast convolution, because it serves a different purpose: As the convolution of an input block of length $M$ with a filter response of length $N$ yields an output signal of length $M + N – 1$, it is necessary to save the additional $N – 1$ output samples (herein referred to as “overflow”) and add them to next $N – 1$ samples of the output signal. The ring buffer length is the same as before, but only two channels (left and right output channel) are required. With this appropriate length and careful read/write operations, even the overflow of impulses responses that are longer than one input/output block are correctly added to the subsequent output signal blocks.

The first step of the the fast convolution computation is to initialize the output buffer with the samples from the overflow buffer, where $i_{\text{n_read}}$ is the number of samples to be read (either the length of the overflow or of the output buffer) and $i_{\text{write}}$ is a constantly increased index to the overflow ring buffer (only shown for the left channel here and in the following code snippets):

```c
for( int j = 0; j < i_{\text{n_read}}; j++ )
* ( p_dest + 2 * j ) = * ( p_ringbuffer_l + i_{\text{write}} );
```

Then, the samples read from the overflow buffer are re-set to zero for future use:

```c
* ( p_ringbuffer_l + i_{\text{write}} ) = 0.0;
```

The rest of the output buffer is also initialized to zero.

For each input channel, the input signal is written to the real part of a zero-initialized (to achieve zero-padding) temporary FFT variable, sample-by-sample,

```c
for( int i = 0; i < i_{\text{n_conv}}; i++ )
  for( int j = 0; j < i_{\text{output_buffer_length}}; j++ )
    p_fft_in[j].r = * ( p_src + i + j * i_{\text{input_nb}} );
```

and then transformed to the frequency domain similar to above using the KISS FFT function.

Now that both signals (input signal and IRs) are available in frequency domain, the complex multiplication is carried out sample-by-sample following this scheme:
\[ Y = X \cdot H = (X_r + jX_i)(H_r + jH_i) = X_rH_r - X_iH_i + j(X_rH_i + X_iH_r) \]  

(7)

\( X \) ... complex spectrum of the input signal block  
\( H \) ... complex spectrum of the HRTF  
\( Y \) ... complex spectrum of the desired output signal block  

with subscript \( r \) and \( i \) for real and imaginary parts of the complex spectra.

For example, the real part of the left output channel is calculated as follows:

\[
p_{\text{fft\_in\_l[j].r}} = /* \text{left output channel (real): */} \]
\[
( p_{\text{fft\_out[j].r}} \times ( p_{\text{hrtf\_l + i\_offset + j}} )->r - p_{\text{fft\_out[j].i}} \times ( p_{\text{hrtf\_l + i\_offset + j}} )->i );
\]

The resulting left and right spectra are transformed back to time domain via IFFT. Their real parts are used to fill the output buffer:

\[
\text{for( int j = 0; j < i\_output\_buffer\_length; j++ )} 
\text{*( p\_dest + 2 \times j ) += p_{\text{fft\_out\_l[j].r}} / (float) \_i\_n\_fft;}
\]

The rest of the non-zero part of the output signal is saved in the overflow buffer:

\[
\text{for( int j = 0; j < \_i\_n\_samples - 1; j++ )} 
\text{i\_write\_pos = ( i\_write + j ) \& \_i\_modulo;}
\text{*( p\_ringbuffer\_l + i\_write\_pos ) += p_{\text{fft\_out\_l[i\_output\_buffer\_length + j].r}} / (float) \_i\_n\_fft;}
\]

Similar to the time domain processing case, the LFE channel is added afterwards and the number of clipping samples in both output channels is counted and stored in the respective variables.

Measuring the code execution time, for impulse responses of 256 samples, exhibited that fast convolution was twice as fast as time domain convolution. For longer impulse responses the performance gain even increases as the computational complexity of time domain convolution is quadratic.

The usually problematic delay with fast convolution due to block-wise processing is no problem, in this case. VLC player anyway uses a buffer and processes audio data block-wise. The synchronization between audio and video is taken care of by the libVLCcore.

If head-tracking is to be used, the reaction time of the entire system to a the user's head movements will be of great importance to maintain a realistic effect. This depends on the speed of the interfacing with the head-tracking hardware and also on how quickly and how often SOFAlizer can response to head position by updating the used impulse response set. For example, it might be advisable to store HRTFs after transformation in an internal memory so as to avoid the need of constant re-transformation of the same impulse response data to the frequency domain. Such considerations were still out of the scope of this project, at the time of writing, but might become important in the future and were thus kept in mind.

One problem with the current implementation occurs when changing parameters on the GUI, e.g. the rotation slider. In this case, one segment of input samples is convolved with one set of impulse responses, whereas the next segment of input samples is convolved with the new set of
impulse responses based on the new GUI settings. However, the filtered signal from the previous segment decays and is added to the filtered output signal of the next segment (as shown in Figure 4). Thus, signals that were filtered with two different impulse responses are added which results in artifacts that are audible as slight clicks. Despite their low volume compared to usual music or film signals, they are still clearly audible, particularly when thinking about the possible future use of head-tracking systems, where constant change of impulse responses is required. Cross-fading the input samples of adjacent input segments would probably solve this issue. However, this requires shorter windows, to store the input samples from the previous input segments for usage in the next call of DoWork, and to correctly manage the overflow between the segments despite overlapping windows.

As the fast convolution correctly works for constant GUI parameters and its optimization for variable GUI parameters would be quite a lot of programming effort, this further step is out of the scope of this project.
6 Conclusion and outlook

In this project, SOFAlizer has been substantially improved in terms of various aspects, most prominently computational efficiency of the rendering, flexibility regarding different sampling rates, resolving of bugs, code quality and documentation.

As explained in section 5.7 Audio processing: Implementation of Fast Convolution, the efficiency of the binaural rendering in SOFAlizer has almost been doubled for head-related transfer functions (HRTFs) with a common length of 256 samples with even more performance gain for longer impulse responses. The performance could be further optimized by transforming two real-valued signals at the same time (as real and imaginary part of the FFT input signal) and then decomposing them to obtain the two desired output signals. In addition, parallel computation using threads (as in the time domain convolution case) needs to be investigated for further efficiency improvements. The mentioned issue with slight clicks when changing GUI parameters needs to be addressed in the future.

Further performance gain could be achieved by using the subband representation of HRTFs as suggested by Marelli et al. [MAR15]. It is based on a processing chain that involves a synthesis filter bank, a transfer matrix and a synthesis filter bank. The improved efficiency is a result of sparse approximation of the transfer matrix whilst maintaining the relevant HRTF properties.

The question of how to handle impulse responses with different sampling rates and, in particular, with sampling rates different than the rate of the audio stream was solved by incorporating the source code from the “Secret Rabbit Code” library and using its simple resampling functionality. The resampling quality can be adjusted in the advanced preferences which allows the user to tailor the trade-off speed vs. quality to the circumstances.

A couple of bugs were solved and numerous small improvements, simplifications and corrections to the source code were made as described in section 5.4 Source code improvements. The source code has been extensively documented in form of comments within the code, which should greatly facilitate the work of future developers. More verbose documentation is found in this project thesis.

Establishing a constant routine of thoroughly addressing the feedback given in the VLC developers’ mailing list gave rise to many of these improvements. As a consequence, SOFAlizer should be close to becoming an official part of VLC player. However, progress in this area was slow as answers were usually coming rather slow and in many cases only after additional inquiry.

To sum it up, the next step in making SOFAlizer an efficient and reliable audio filter module for VLC player has been taken and a solid basis for future professional software development has been laid.

The next step would be to actually integrate SOFAlizer to the official VLC repository, which is in control of the VLC developers, however. Apart from that, the suggestions for further efficiency optimization given herein should be addressed. The above mentioned issue with slight artifacts
when changing GUI parameters during frequency domain processing should be resolved. After that, interaction with a head-tracking system feeding its data to VLC player can be approached.
References


[MAJ14b] Piotr Majdak, Harald Ziegelwanger, Efficient Representation of Head-Related Transfer Functions using Spatially Oriented Format for Acoustics, 2014
Hrauda: Virtual binaural acoustics in VLC player: HRTFs and efficient rendering


[SOF14c] Piotr Majdak, Markus Noisternig, SOFA (Spatially Oriented Format for Acoustics) specifications version 0.6, available online: http://sourceforge.net/projects/sofacoustics/files/SOFA%20specs%200.6.pdf/download (last visited: 5th March 2015), 2014


CHA09: Scott Chacon, Pro Git, available online: https://github.s3.amazonaws.com/media/progit.en.pdf (last visited: 3rd March 2015), 2009


