Realtime signaal synchronisatie met acoustic fingerprinting

Ward Van Assche

Promotoren: dr. Marleen Denert, Joren Six
Begeleider: prof. Helga Naessens

Masterproef ingediend tot het behalen van de academische graad van
Master of Science in de industriële wetenschappen: informatica

Vakgroep Informatietechnologie
Voorzitter: prof. dr. ir. Daniël De Zutter

Vakgroep Kunst-, Muziek- en Theaterwetenschappen
Voorzitter: prof. dr. Francis Maes

Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2015-2016

UNIVERSITEIT
GENT

Faculteit Ingenieurswetenschappen en Architectuur
Vakgroep Informatietechnologie
Voorzitter: Prof. Dr. Ir. Daniël De Zutter

Realtime signaal synchronisatie met acoustic fingerprinting

door

Ward Van Assche

Promotoren: Dr. Marleen Denert, Joren Six
Scriptiebegeleider: Prof. Helga Naessens

Masterproef ingediend tot het behalen van de academische graad van
Master of Science in de industriële wetenschappen: informatica

Academiejaar 2015–2016

Voorwoord

Zonder hulp van buitenaf zou ik er nooit in geslaagd zijn om mijn masterproef tot een goed einde te brengen. Daarom wil ik verschillende mensen bedanken die een grote rol gespeeld in één of meerdere fases van dit eindwerk.

Eerst en vooral wil ik mijn externe promotor, Joren Six, bedanken voor het vertrouwen en de ondersteuning die hij mij tijdens het uitwerken van deze masterproef heeft gegeven. De kennis en inzicht die ik van hem heb meegekregen op vlak van digitale audio (en alles wat ermee te maken heeft) neem ik de rest van mijn loopbaan mee. Ik vond het ontzettend leerrijk om mijn interesse in muziek en geluid te kunnen combineren met mijn opleiding informatica.

Verder wil ik ook mijn interne promotor, Marleen Denert, bedanken voor het opvolgen en nalezen van mijn thesis. Haar opbouwende kritiek was van onschatbare waarde.

Ten slotte wil ik ook mijn ouders, zusje en vrienden bedanken die mij altijd gesteund hebben tijdens mijn opleiding tot industrieel ingenieur.

En uiteraard wil ik ook u, de lezer, bedanken voor de interesse in mijn onderzoek. Ik wens u veel leesplezier.

Ward Van Assche, juni 2016

Toelating tot bruikleen

Realtime signaal synchronisatie met acoustic fingerprinting

door

Ward Van Assche

Masterproef ingediend tot het behalen van de academische graad van
Master of Science in de industriële wetenschappen: informatica

Academiejaar 2015–2016

Promotoren: Dr. Marleen Denert, Joren Six
Scriptiebegeleider: Prof. Helga Naessens

Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Vakgroep Informatietechnologie
Voorzitter: Prof. Dr. Ir. Daniël De Zutter

Samenvatting

De meeste experimenten die aan het IPEM worden uitgevoerd maken gebruik van verschillende soorten sensoren (accelerometers, druksensoren,...). Een veelvoorkomend probleem is de de synchronisatie van de data van elke sensor. Bij het huidige synchronisatiesysteem wordt elke sensor verbonden met een microfoon die het omgevingsgeluid opneemt. Met technieken zoals acoustic fingerprinting en het berekenen van de kruiscovariantie kan de latency tussen de audiosignalen zeer nauwkeurig bepaald worden. Met deze latency kan vervolgens de sensordata gesynchroniseerd worden. Het huidige systeem kan enkel als naverwerking uitgevoerd worden. Een realtime en meer gebruiksvriendelijk systeem is erg gewenst. In dit onderzoek is er nagegaan of dit mogelijk is. Om de huidige synchronisatiealgoritmen in realtime bruikbaar te maken waren aanpassingen en optimalisaties nodig. Dit onderzoek heeft ook geleid tot enkele Max/MSP modules. Met behulp van deze modules is het mogelijk om het volledige synchronisatieproces in realtime uit te voeren in zonder het schrijven van één lijn code.

Trefwoorden

synchronisatie, sensoren, audio, geluid, realtime, signalen, streams, acoustic fingerprinting, kruiscovariantie, digitale signaalverwerking

Real-time signal synchronization with acoustic fingerprinting

Ward Van Assche

Supervisor(s): Joren Six, Marleen Denert

Abstract— **Many experiments use sensors such as accelerometers and pressure sensors. A common problem after these experiments is the synchronization of data of each sensor. The current synchronization system requires each sensor to be connected to a microphone recording the sound of the environment. With efficient audio-to-audio alignment techniques the latency can be detected very accurately. Because the detection is a post-processing step a real-time and more user-friendly solution is desirable. This paper explores the possibilities to do this. To use the current synchronization algorithms in real-time they had to be changed and optimized in different ways. The new system resulted in a Max/MSP module which makes it possible to run the synchronization in real-time without writing a single line of code.**

Keywords— **Signal Synchronization, Audio Alignment, Real-Time, Acoustic Fingerprinting, Cross-covariance, Digital Signal Processing**

I. Introduction

EXPERIMENTS in various research areas use sensors and video cameras to capture the environment. Before sensor data and video recordings can be analyzed properly they have to be synchronized accurately in time. This is no easy task because of various reasons. The first problem is that the sensor datastreams can be heterogeneous: the sample rate can vary and the resulted data can be different (video data or numerical samples). Another difficulty is to cope with the fact that some data sources are sampled unreliably. This can lead to dropped samples and drift which cause unexpected latency changes.

Most techniques require a post-processing step: the data has to be synchronized manually or by software *after* the experiment. This is impractical and time-consuming. A technique which can avoid this is desirable.

The most straightforward way to perform synchronization is by adding markers. This technique is described in [1]. How a marker is placed depends on the type of stream. A short sound can add a marker in an audio stream, a bright flash can do this in a video stream. The latency can be found by calculating the difference between marker positions. The usability of this method is limited because of its poor scalability. The synchronization of a large number of streams can be very challenging. Dropped samples and drift can only be detected when the markers are repeated each time interval, which is impractical. Actual synchronization using this technique is only possible as a post-processing operation.

In [2] a method is described which uses a clock signal to synchronize streams in real-time. However this method avoids the post-processing step it does not fit the requirements: each device (sensor, video camera) should accept a clock-signal as input. Because these devices (especially cameras) are very expensive this method is not feasible for the stated problem.

In [3] the stated problem is approached in an entirely different way. The described method does not try to synchronize the streams directly. Instead, the (recorded) environment sound is embedded in each stream. This ploy reduces the initial problem to audio-to-audio alignment. Because the recorded environment sound is almost the same for each stream the problem is much easier to tackle. The audio-to-audio algorithms described in the article perform very well. The latency can be detected with a precision less than 1ms. However the article only describes a post-processing approach, the used algorithms look very interesting.

The next part of this paper will describe a method to synchronize streams in real-time using the algorithms mentioned in the previous article. The word "real-time" can be ambiguous in a signal-processing context. Therefore it's important to specify some requirements for a real-time system. Because the latency-detecting algorithms need some amount of audio in order to determine the latency it's impossible to immediately output the synchronized signals. In this paper the real-time restriction refers to the fact that the synchronization algorithms are executed while the sensors are collecting data. The post-processing step should be avoided.

II. Determining latency

The method described in [3] uses two latency detecting algorithms which are complementary. By combining them, latency can be detected accurately.

A. Acoustic fingerprinting

Acoustic fingerprinting is a fast and robust technique for comparing audio fragments. A method using acoustic fingerprinting is described in [4]. The method uses fingerprints based on spectral peaks. Each fingerprint contains condensed information based on typical audio properties. This technique allows finding similar audio fragments ignoring noise and other disturbing background sounds.

However the initial application was identifying an audio recording using a huge database containing a myriad of fingerprints it's also possible to compare the fingerprints of recordings mutually. The latency can be detected by calculating the offset between the fingerprints. The precision varies around $16ms$ to $32ms$ depending on the used parameters.

B. Cross-covariance

The cross-covariance (also referred to as cross-correlation) is a calculation which measures the degree of similarity between two time sequences. Because an audio signal is a time sequence this calculation can be used for determining the latency. When the cross-covariance value is calculated for each possible shift

between two audio fragments the shift with the highest result determines the latency.

This method can determine the latency to the nearest sample. The precision in milliseconds depends on the sample rate: when the sample rate is $8000Hz$ the maximum precision is $1/8000Hz = 0.125ms$.

A disadvantage to this method is its performance. The time complexity of the algorithm is $O(n^2)$ where n is the number of samples in each signal. Finding the latency between two audio fragments of $10s$ at a sample rate of $8000Hz$ would asymptotically result in $6.4 \cdot 10^9$ computations, which is impracticable on a regular computer.

C. Refining the results

The two previously described algorithms are very complementary. Acoustic fingerprinting allows finding the latency very fast and robustly. The cross-covariance algorithm can detect the latency between two tiny pieces of audio very accurately. These advantages can be easily combined.

By determining the raw latency with acoustic fingerprinting, the number of samples used in the cross-covariance calculation can be limited. By cutting the raw latency from the corresponding audio fragment, the new latency is reduced. When the acoustic fingerprinting algorithm uses a precision of $32ms$, the cross covariance should be calculated on two audio fragments of at least 256 samples (when the sample rate is $8000Hz$). This asymptotically results in 65 536 computations, which is much more feasible than without the acoustic fingerprinting step.

D. Optimizations

Many optimization are applied to the latency-detecting algorithms. The most important one is the repeated execution of the cross-covariance algorithm. The cross-covariance algorithm is very sensitive to noise and other undesirable sounds. Since the algorithm calculates the similarity of the waveforms these sounds can be harmful for the final result.

Because the cross-covariance algorithm is executed on a very tiny piece of audio it can be executed multiple times on a slightly different location. The most frequent latency is the final result of the algorithm. However the results are much more accurate after this optimization, the influence on the performance is limited. This because the time complexity function remains the same.

III. Buffering streams

In order to avoid the post-processing step the real-time streams have to be buffered. The algorithms are executed on the consecutive buffers of each stream. The buffer size (t) will be expressed in seconds of audio it can contain. It determines the maximum latency which can be detected. A minimum length of $10s$ is desirable to make the algorithms perform well. In order to detect latency changes as fast as possible a step size (s) has to be chosen (also expressed in seconds). The step size determines the interval between the moments a new buffer for each stream is created. When the step size is smaller than the buffer size the overlap between two consecutive buffers is equal to $t - s$.

Both the buffer size and step size influence the time between a latency change and its detection. Since the latency can only be detected when more than half of the buffer contains the modified

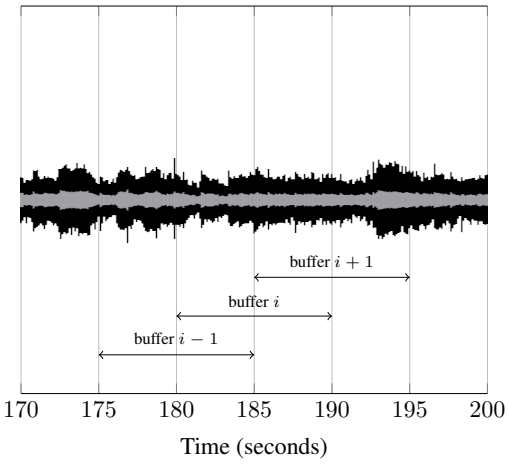


Fig. 1. Visualization of a buffer containing $10s$ of audio with a step size of $5s$.

latency, the best case detection speed is equal to $t/2$. By applying a smaller step size the worst case scenario can be improved: it is equal to $t/2 + s$. Figure 1 shows a buffer where $t = 10$ and $s = 5$.

IV. Latency filters

The avoidance of the post-processing step makes it harder to manually fix potential errors. When using audio recordings of poor quality, it's recommended to use a moving median latency filter. By using this filter the consecutive latencies are pushed in a queue. Each time a new latency is determined the median is of the values in the queue are is returned. Depending on the queue size, some peaks in the consecutive latencies are flattened.

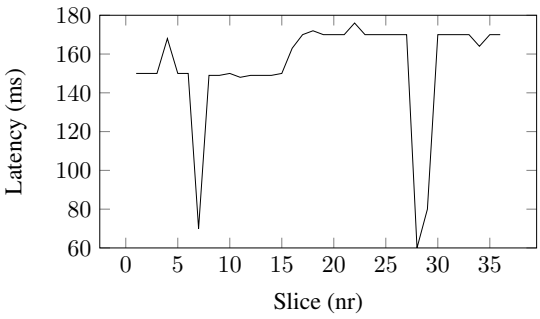


Fig. 2. The sequence of latencies without filtering.

Figure 2 shows a sequence of latencies without filtering, figure 3 shows the same sequence after filtering with a moving-median

V. Synchronization

The actual synchronization is performed by adding silence to each stream. The latencies of each stream are converted to the amount of silence which has to be added to each stream. It's a good practice to synchronize the streams by adding whitespace. However this can also be done by dropping samples the extra loss of data is undesirable.

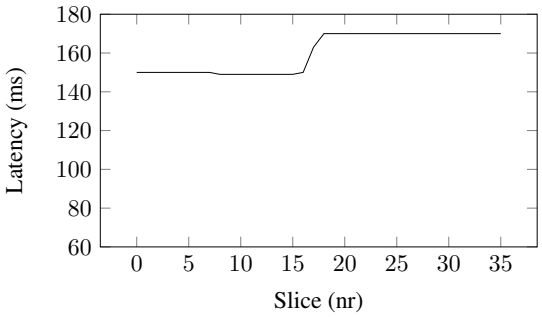


Fig. 3. The sequence of latencies after filtering.

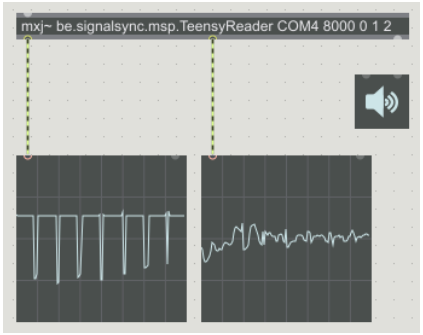


Fig. 5. The TeensyReader module in Max/MSP.

VI. Results

This research resulted in two Max/MSP modules[1]. One module is able to read sensor datastreams accompanied by an audiostream from a *Teensy microcontroller* into the Max/MSP environment. The second module is designed to perform the actual synchronization.

A. Using a Teensy

A Teensy is a microcontroller which can be used to attach several sensors to a microphone with a negligible latency between the streams. This property is very useful for the stated problem. Because the actual synchronization is performed in Max/MSP it's required to read the analog Teensy pins into Max/MSP which isn't natively supported. To support this, a new module had to be written: the *TeensyReader*. The module requires several parameters: port name, sample rate of the Teensy, index of the first analog pin (A3 $\rightarrow$ 3), index of the audio pin (starting at 0 for the first analog pin) and the number of analog pins to read.

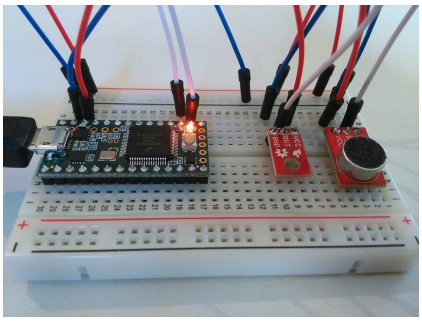


Fig. 4. A Teensy microcontroller.

Figure 4 shows a picture of a Teensy connected to a microphone and an infrared sensor.

Figure 5 shows the TeensyReader module reading signals from the infrared sensor and microphone. The signals are displayed on two scopes.

[1]Max/MSP is software and a visual programming language which allows processing audio and video signals. This can be done by connecting existing modules each having a specific task. Max/MSP allows writing own modules in Java or C++.

B. The Sync module

The *Sync* module performs the actual synchronization. It uses the software library which is responsible for buffering the streams and calculating the latency. The module uses the latency to add silence to each stream and output the synchronized streams.

The module requires one parameter which is a character representation of the stream structure. The parameter is a comma separated string where each part consists of one 'a' character and any number of 'd' characters. An audio stream which should be used for synchronization (using the audio-to-audio alignment algorithms) is represented by 'a', an attached data stream by 'd'. The character order determines the order of inlets and outlets of the module.

Figure 6 shows the Sync module created with parameter `dad,addd`. The synchronized outlets are sent to the *sfrecord* module which writes the synchronized streams to a file.

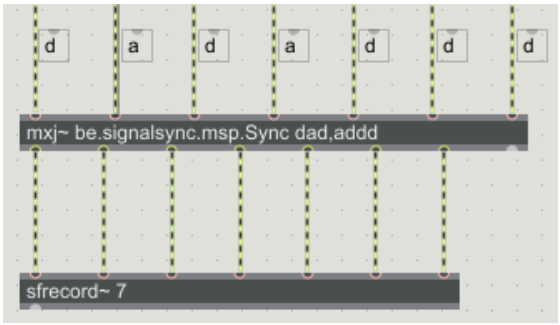


Fig. 6. The Sync module in Max/MSP. The data patch cable is labeled with `d`, an audio patch cable with `a`.

VII. Conclusion

Several tests proved that the synchronization works very well without the post-processing step. Even audio recorded from poor quality can be synchronized easily avoiding the post-processing step.

References

[1] David Bannach, Oliver Amft, and Paul Lukowicz, "Automatic event-based synchronization of multimodal data streams from wearable and ambient sensors," in *Smart sensing and context*, pp. 135–148. Springer, 2009.

[2] Javier Jaimovich and Benjamin Knapp, "Synchronization of multimodal recordings for musical performance research.," in *NIME*, 2010, pp. 372–374.
[3] Joren Six and Marc Leman, "Synchronizing Multimodal Recordings Using Audio-To-Audio Alignment," *Journal of Multimodal User Interfaces*, vol. 9, no. 3, pp. 223–229, 2015.
[4] Avery Li-Chun Wang, "An industrial-strength audio search algorithm," in *ISMIR 2003, 4th Symposium Conference on Music Information Retrieval*, 2003, pp. 7–13.
[5] Joren Six, Olmo Cornelis, and Marc Leman, "TarsosDSP, a Real-Time Audio Processing Framework in Java," in *Proceedings of the 53rd AES Conference (AES 53rd)*. 2014, The Audio Engineering Society.

Inhoudsopgave

Gebruikte afkortingen

IPEM	Instituut voor Psychoakoestiek en Elektronische Muziek
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
SFT	Short Time Fourier Transform
ECG	Elektrocardiogram
DTW	Dynamic timewarping
USB	Universal Serial Bus
ADC	Analog-to-digital converter
PCM	Pulse-code modulation
UML	Unified Modeling Language

Hoofdstuk 1

Introductie

1.1 Probleemschets

Het probleem dat in deze masterproef zal worden onderzocht doet zich heel specifiek voor bij verschillende experimenten die aan het IPEM worden uitgevoerd. Dit is de onderzoeksinstelling van het departement musicologie aan Universiteit Gent. De focus van het IPEM ligt vooral op onderzoek naar de interactie van muziek op fysieke aspecten van de mens zoals dansen, sporten en fysieke revalidatie. [3]

Om de relatie tussen muziek en beweging te onderzoeken worden er tal van experimenten uitgevoerd. Deze experimenten maken gebruik van allerhande sensoren om bepaalde gebeurtenissen om te zetten in analyseerbare data.

Bij een klassiek experiment wordt onderzocht wat de invloed is van muziek op de lichamelijke activiteit van een persoon. Alle bewegingen worden geregistreerd met een videocamera en een accelerometer.

Hierbij moeten drie datastreams worden geanalyseerd: de videobeelden, de data van de accelerometer en de afgespeelde audio. Een uitdaging hierbij is de synchronisatie van deze verschillende datastreams. Om een goede analyse mogelijk te maken is het zeer gewenst dat men exact weet (tot op de milliseconde nauwkeurig) wanneer een bepaalde gebeurtenis in een datastream zich heeft voorgedaan, zodat men deze gebeurtenis kan vergelijken met

de gebeurtenissen in de andere datastreams. Door de verschillen in samplefrequentie en door de latencies die zich in elke opname kunnen voordoen is dit zeker geen sinecure. [22]

Bij het IPEM maakt men gebruik van een systeem waarbij audio opnames het synchronisatieproces vereenvoudigen. Het principe werkt als volgt: men zorgt ervoor dat elke datastream gekoppeld is aan een perfect gesynchroniseerde audiostream, afkomstig van een opname van het omgevingsgeluid. In het voorgaande experiment is dit eenvoudig te verwezenlijken. Bij de videobeelden kan automatisch een audiospoor mee worden opgenomen. De accelerometer kan geplaatst worden op een microcontroller vergezeld van een kleine microfoon. Aangezien beide componenten zo dicht op de hardware geplaatst zijn is de latency tussen beide datastromen te verwaarlozen.[1] De afgespeelde audio is uiteraard al een perfecte weergave van het omgevingsgeluid. Figuur 1.1 toont een screenshot van het programma waarmee de synchronisatie op dit moment wordt uitgevoerd.

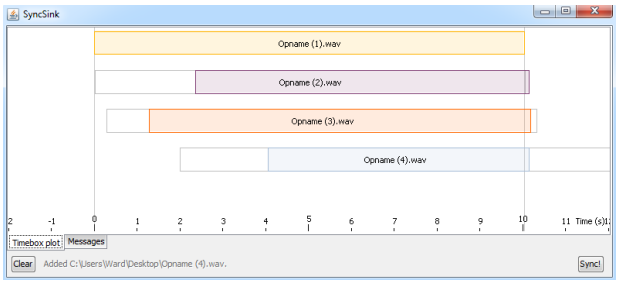
Na het uitvoeren van het experiment beschikt men dus over de gegevens van drie datastreams, waarbij er aan elke datastream een quasi perfect synchrone opname van het omgevingsgeluid is gekoppeld. Aangezien het experiment in één ruimte is uitgevoerd zijn de verschillende opnames van het omgevingsgeluid zeer gelijkend. Het probleem van de synchronisatie van de verschillende datastromen kan bijgevolg gereduceerd worden tot het synchroniseren van de verschillende audiostromen.

Door de typisch eigenschappen van geluid is het niet zo moeilijk om verschillende audiostreams te synchroniseren. Bij het IPEM heeft men een systeem ontwikkeld dat hiertoe in staat is.

Dit systeem heeft in de praktijk echter heel wat beperkingen. De grootste beperking is dat het synchronisatieproces pas kan worden uitgevoerd wanneer het experiment is afgelopen. Deze verwerking kan ook enkel handmatig uitgevoerd worden. De opgenomen audio- en databestanden moeten worden verzameld op een computer waarna vervolgens de latency tussen de audiostreams bepaald kan worden. Met behulp van deze latency kunnen de datastreams worden gesynchroniseerd.

[1] De latency van de audioverwerking op een *Axoloti* microcontroller is vastgesteld op $0.333ms$. Meer informatie: `http://www.axoloti.com/more-info/latency/`

Voor de musicologen die deze experimenten uitvoeren is deze werkwijze veel te omslachtig. Daarom is een eenvoudiger realtime systeem om de synchronisatie uit te voeren zeer gewenst.



Figuur 1.1: De gebruikersinterface van SyncSink: een programma waarin de opgenomen fragmenten gesleept kunnen worden na afloop van het experiment. Vervolgens wordt de latency berekend. Meer informatie is te vinden in artikel [22].

Een ander probleem is dat de resultaten van het kruiscovariantie algoritme soms afwijkingen vertonen die moeilijk te verklaren zijn. De oorzaak hiervan zal worden onderzocht. Ook is het kruiscovariantie algoritme in vergelijking met het acoustic fingerprinting algoritme véél gevoeliger voor storingen en ruis, veroorzaakt door slechte opnames. Aangezien de opnameapparatuur (zeker op microcontrollers) bij de uit te voeren experimenten vaak van slechte kwaliteit is, is het belangrijk om de algoritmes voldoende robuust te maken zodat ze hier mee om kunnen.

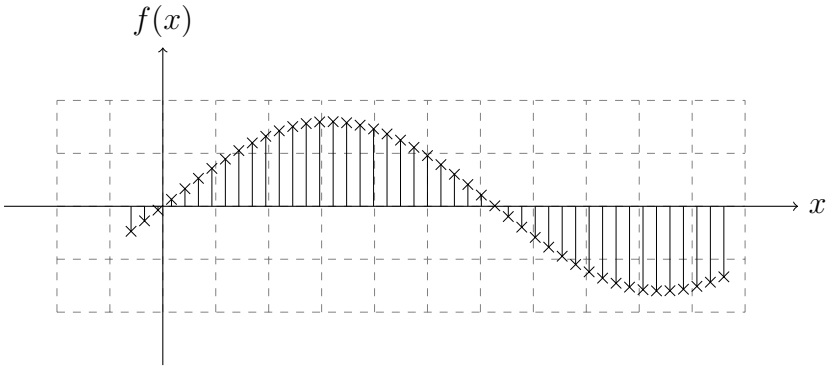
1.2 Digitale audio

Het vervolg van deze scriptie onderstelt dat de lezer een basiskennis heeft inzake digitale audio. In deze inleiding worden de belangrijkste zaken hieromtrent uitgelegd.

Om geluidsgolven digitaal te kunnen verwerken moeten ze worden geconverteerd naar reeksen van discrete waarden. Deze omzetting gebeurt met een ADC: een analog-to-digital converter. De meeste ADC's maken gebruik van de PCM (pulse-code modulation) voorstelling van audio. Bij PCM wordt het analoge signaal op regelmatige tijdstippen gesampled en omgezet in discrete waarden. Figuur 1.2 toont schematisch hoe dit in zijn werk gaat. PCM audio heeft verschillende parameters die een invloed hebben op de uiteindelijke kwaliteit

van de audio. De belangrijkste parameters zijn de samplefrequentie (*sampling rate*) en bitdiepte (*bit depth*).

Figuur 1.2: Samplen van een analoog audiosignaal in de vorm van een sinusgolf. Met toestemming overgenomen van artikel [20].



Samplefrequentie

De samplefrequentie bepaalt het aantal samples per seconde en wordt uitgedrukt in Hertz (Hz). Bij het bepalen van de samplefrequentie is het van belang om rekening te houden met het *bemonsteringstheorema van Nyquist-Shannon*. Deze stelling zegt dat de samplefrequentie minstens dubbel zo hoog moet zijn dan de hoogste frequentie van de te converteren audio. Bij het samplen aan een lagere frequentie treedt informatieverlies op. Deze stelling wordt in detail besproken in het originele artikel [16] van Nyquist.

Het menselijk oor is in staat om geluiden te detecteren tussen $20Hz$ en $20kHz$. Om informatieverlies bij het samplen van geluiden binnen dit bereik te voorkomen is het dus vereist om een minimale samplefrequentie te hanteren van $2 \times 20kHz$. De standaard samplefrequentie voor muziek is net iets hoger: $44.1kHz$.

De frequentie van de menselijke stem varieert tussen $30Hz$ en $3000Hz$. De minimale samplefrequentie voor het digitaliseren van een stemopname is dus $6kHz$. In de praktijk wordt meestal een minimum gehanteerd van $8kHz$.

Bitdiepte

De bitdiepte is het aantal bits waarmee elke gesamplede waarde wordt voorgesteld. Meestal wordt er gebruik gemaakt van 16 bit *signed integers*.

De bitdiepte bepaalt het dynamische bereik van audio. Dit is de verhouding tussen het stilst en luidst mogelijk weer te geven volume. Deze verhouding, uitgedrukt in decibel, kan worden berekend met volgende formule:

$$DR = 20 \cdot \log_{10}\left(\frac{2^Q}{1}\right) = (6.02 \cdot Q)dB \tag{1.1}$$

In deze formule staat DR voor het dynamische bereik en Q voor de bitdiepte. Volgens deze formule heeft 16 bit audio een theoretisch dynamische bereik van ongeveer 96 dB. De werkelijke waarde kan hier echter van afwijken door filters die zijn ingebouwd in audiosystemen.

Bovenstaande informatie is gebaseerd op artikel [20], introductievideo [6] en boek [13].

Weergave in software

In computerprogramma's waarin digitale audio verwerkt wordt, wordt elke sample voorgesteld als een getal. Afhankelijk van de implementatie kan een 16 bit sample op verschillende manieren worden verwerkt. Enkele mogelijke voorstellingen: signed integer (-32768 tot 32767), unsigned integer (0 tot 65536) of floating point (-1 tot 1). In deze thesis en de bijhorende software wordt de floating point notatie gehanteerd.

1.3 Evaluatiecriteria

Het te ontwikkelen systeem moet voldoen aan heel wat vereisten. In deze sectie zullen de vereisten eenduidig geformuleerd en besproken worden.

Realtime synchronisatie

Een cruciale vereiste is dat de toepassing in *realtime* moet kunnen werken. Concreet wil dit zeggen dat de gesynchroniseerde data na het experiment onmiddellijk beschikbaar moet zijn.

Een écht realtime systeem bouwen is in de praktijk niet mogelijk. De algoritmes vereisen een bepaalde hoeveelheid aan data voordat de latency berekend kan worden. Daarom moeten de streams gebufferd worden, wat er toe leidt dat het systeem niet meer realtime is in de enge zin van het woord. Om een realtime systeem zo goed mogelijk te benaderen wordt een beperking opgelegd: een buffer met als maximumgrootte de hoeveelheid data verzameld in tien seconden. De buitengaande gesynchroniseerde streams hebben dus 10 seconden vertraging ten opzichte van de realtime streams.

Detecteren van gedropte samples

De beperkte resources van een microcontroller kan voor problemen zorgen bij het verwerken van streams. Zo kan het gebeuren dat er gegevens van streams verloren gaan. In het vakjargon worden dit ook wel *gedropte samples* genoemd. Bij de synchronisatie leidt dit probleem tot een plotse verhoging van de latency. Hoewel het onmogelijk is om de gedropte samples te reconstrueren is het wel gewenst dat de gewijzigde latency gedetecteerd wordt en dat hiermee wordt rekening gehouden bij de verdere verwerking. De snelheid waarmee dit probleem gedetecteerd kan worden hangt eveneens af van de manier waarop er gebufferd wordt. Een detectiesnelheid van 10 seconden is aanvaardbaar voor deze toepassing.

Detecteren van drift

Elke stream heeft een eigen samplefrequentie. Het is belangrijk dat de samplefrequentie gekend is om de gegevens correct en precies te kunnen verwerken. Het kan echter voorvallen dat de samplefrequentie bij de verwerking op microcontrollers toch niet zo nauwkeurig bepaald is. Een stream waarbij de samplefrequentie $1Hz$ afwijkt van de theoretische waarde zal na 60 seconden een latency hebben opgebouwd van 60 samples. Bij een samplefre-

quentie van $8000Hz$ komt dit overeen met $7.5ms$.[2] Dit probleem mag zeker niet worden verwaarloosd.

1.4 Bestaande methoden

Er bestaan verschillende methoden om datastreams te synchroniseren. Welke methode te verkiezen is hangt volledig af van de toepassing.

In deze sectie komen de belangrijkste methoden aan bod en zullen ze worden getoetst aan de probleemstelling en de evaluatiecriteria.

1.4.1 Event-gebaseerde synchronisatie

Deze methode wordt beschreven in [7, 22] en is een eenvoudige, vrij intuïtieve methode om synchronisatie van verschillende datastreams uit te voeren. De synchronisatie gebeurt aan de hand van markeringen die in de verschillende streams worden aangebracht. In audiostreams kan een kort en krachtig geluid een markering plaatsen. Een lichtflits kan dit realiseren in videostreams. De latency wordt bepaald door het verschil te berekenen tussen de tijdspositie van de markeringen in de streams. De synchronisatie kan vervolgens zowel manueel als softwarematig worden uitgevoerd.

Deze methode kent heel wat beperkingen. Zo vormt bij de synchronisatie van een groot aantal streams de schaalbaarheid een probleem. Ook wanneer er in een stream samples gedropt worden of er drift ontstaat, leidt dit tot foutieve synchronisatie. De methode kan deze twee problemen niet detecteren tot er opnieuw markeringen worden aangebracht en de streams gesynchroniseerd worden. Verder laten ook niet alle sensoren toe om markeringen aan te brengen: zo is de synchronisatie van een ECG onmogelijk met deze methode.

Het handmatig synchroniseren met behulp van deze methode blijkt derhalve in een realtime situatie niet mogelijk. Wanneer de synchronisatie echter door software wordt uitgevoerd is

[2]Berekening: $60/8000Hz = 0.0075s = 7.5ms$

deze methode wel in realtime bruikbaar. In dat geval moet er per tijdsinterval een markering worden aangebracht om de problemen veroorzaakt door drift en gedropte samples te overbruggen.

1.4.2 Synchronisatie met een kloksignaal

Artikel [14] beschrijft een methode waarbij door een kloksignaal realtime streams van verschillende soorten toestellen worden gesynchroniseerd. Hiervoor gebruikt men standaard audio en video synchronisatieprotocollen. Elk toestel kan gebruik maken van verschillende samplefrequenties en communicatieprotocollen.

De methode gebruikt een *master time code* signaal dat verstuurd wordt naar elk toestel. Dit laat het realtime analyseren van elke stream toe. Bij deze analyse kan vervolgens meteen de samplefrequentie en latency bepaald worden.

Een groot nadeel van dit systeem is dat elk toestel een kloksignaal als input moet kunnen toelaten en verwerken. In het geval van de verwerking van videobeelden kan deze methode enkel gebruikt worden met zeer dure videocamera's waarbij de sluitertijd gecontroleerd kan worden. Bij goedkopere camera's (zoals's webcams) moet men op zoek gaan naar alternatieven. [22]

1.4.3 Dynamic timewarping

Dynamic timewarping (DTW) is een techniek die gebruikt wordt voor het detecteren van gelijkenissen tussen twee tijdreeksen[3]. Aangezien een gedigitaliseerde audiostream een tijdreeks is kan deze techniek worden aangewend om de latency te bepalen tussen gelijkaardige opnames van het omgevingsgeluid. In de probleemschets (1.1) is er uitgelegd hoe datastreams met behulp van het omgevingsgeluid gesynchroniseerd kunnen worden.

DTW is een algoritme dat op zoek gaat naar de meest optimale *mapping* tussen twee tijdreeksen. Hierbij wordt gebruik gemaakt van een padkost. De padkost wordt bepaald

[3]Een tijdreeks is een sequentie van opeenvolgende datapunten over een continu tijdsinterval, waarbij de datapunten elk baar na telkens hetzelfde interval opvolgen.

door de manier waarop de tijdreeksen niet-lineair worden kromgetrokken ten opzichte van de tijdas[19]. De minimale kost kan in kwadratische tijd berekend worden door gebruik te maken van dynamisch programmeren [11]. DTW is een veelgebruikte techniek in domeinen zoals spraakherkenning, bio-informatica, data-mining, etc [18].

Aangezien DTW het toelaat om tijdreeksen krom te trekken is het gewenst dat zowel het verleden als de toekomst van de streams voor het algoritme toegankelijk is. Een uitbreiding op dit algoritme beschreven in [11] laat toe één tijdreeks in realtime te streamen mits de andere stream op voorhand is gekend. Toch houdt deze uitbreiding geen oplossing in voor het gestelde probleem. Alle streams komen immers in realtime toe en de latency tussen de streams moet zo snel mogelijk achterhaald worden.

Het bufferen van de binnenkomende streams en vervolgens het DTW algoritme uit te voeren op de buffers leek een mogelijke manier om dit probleem te omzeilen.

Of het algoritme na deze aanpassing voldoet aan onze vereisten diende een klein experiment uit te wijzen. De resultaten hiervan zijn te vinden in appendix A.

Het experiment toonde evenwel aan dat DTW niet bruikbaar is voor de realtime stream synchronisatie. De resultaten bleken niet nauwkeurig genoeg, zeker niet wanneer ook de performantie van het algoritme in beschouwing werd genomen.

1.4.4 Acoustic fingerprinting

Acoustic fingerprinting is een techniek die in staat is om gelijkenissen te vinden tussen verschillende audiofragmenten. Hierbij is het eveneens mogelijk om de latency tussen de audiofragmenten te bepalen. Net zoals bij DTW kan dit algoritme gebruikt worden om datastreams te synchroniseren met behulp van het omgevingsgeluid.

De techniek van acoustic fingerprinting extraheert en vergelijkt fingerprints van audiofragmenten. Een acoustic fingerprint bevat gecondenseerde informatie gebaseerd op typische eigenschappen van het audiofragment. De kracht van dit algoritme schuilt in haar snelheid en robuustheid. Het is immers uitzonderlijk bestand tegen achtergrondgeluiden en ruis. Door deze eigenschappen is het algoritme in staat om in enkele seconden een database met

miljoenen fingerprints van audiofragmenten te doorzoeken. De bekendste toepassing van acoustic fingerprinting is de identificatie van liedjes op basis van een korte opname[4].

Het is onder meer deze techniek die het IPEM gebruikt om de opgenomen audiostreams van experimenten te synchroniseren. In tegenstelling tot *Shazam* wordt er niet op zoek gegaan naar matches in een database maar worden ze gezocht tussen de opgenomen audiofragmenten. Het uitgangspunt is immers dat er tussen de opnames gelijkenissen moeten gevonden kunnen worden.

Door haar snelheid en robuustheid lijkt dit algoritme te voldoen aan de vereisten om datastreams realtime te kunnen synchroniseren. Het is wel noodzakelijk dat de streams gebufferd worden alvorens het algoritme kan starten. Drift en gedropte samples kunnen gedetecteerd worden door het algoritme iteratief op korte gebufferde fragmenten uit te voeren. Na elke iteratie kan een eventuele wijziging worden opgemerkt. Zie 2.1.1 voor een meer gedetailleerde bespreking van dit algoritme.

1.4.5 Kruiscovariantie

De laatste methode is net zoals de twee vorige methodes in staat om de latency tussen audiofragmenten te bepalen. Deze methode kan daarom ook aangewend worden om datastreams met behulp van opnames van het omgevingsgeluid te synchroniseren.

Kruiscovariantie (ook wel kruiscorrelatie genoemd) berekent de gelijkenis tussen twee audiofragmenten sample per sample en kent een getal toe aan de mate waarin de fragmenten overeenkomen. Door deze berekening voor elke verschuiving uit te voeren kan de latency tussen de fragmenten bepaald worden.

Deze methode is eveneens toepasbaar op realtime streams door gebruik te maken van buffering. Het iteratief uitvoeren van het algoritme op de opeenvolgende buffers zorgt ervoor dat gedropte samples en drift gedetecteerd kunnen worden.

In sectie 2.1.2 wordt dit algoritme verder in detail behandeld.

[4]Het grootste voorbeeld hiervan is de smartphone app Shazam. Deze app is de eerste toepassing dat gebruik maakte van dit algoritme.

1.5 Doel van deze masterproef

Dit onderzoek wil drie zaken bereiken:

Selectie en optimalisatie van algoritmes

Er wordt op zoek gegaan naar de algoritmes waarmee het probleem kan worden opgelost. Verder dienen de algoritmes en bijhorende parameters te worden geoptimaliseerd om in deze toepassing zo efficiënt mogelijk te presteren. Indien mogelijk zal er worden geprobeerd om de algoritmes waarvan er bij het IPEM al een implementatie beschikbaar is te hergebruiken.

Het beoogde doel is dat de algoritmes in staat zijn om audio opgenomen met een basic microfoon op een microcontroller te synchroniseren met een nauwkeurigheid van minstens één milliseconde.

Ontwerp en implementatie van een softwarebibliotheek

Het tweede doel van het onderzoek betreft het schrijven van een softwarebibliotheek. Deze bibliotheek zal gebruik maken van de geoptimaliseerde algoritmes om de latency tussen de verschillende audiostromen te bepalen. Deze bibliotheek moet vanuit andere software kunnen worden opgeroepen.

Ontwerp en implementatie van een gebruiksvriendelijke interface

Uiteindelijk is het de bedoeling dat dit onderzoek resulteert in een gebruiksvriendelijke applicatie die toegankelijk is voor onderzoekers/musicologen zonder uitgebreide informatica kennis. De software moet in staat zijn om van verschillende datastreams (vergezeld van een audiostream) te synchroniseren en het gesynchroniseerde resultaat weg te schrijven naar een persistent medium.

Hoofdstuk 2

Methode

2.1 Algoritmen

In sectie 1.4 van deze scriptie zijn de voornaamste methoden waarmee datastreams gesynchroniseerd kunnen worden beknopt besproken. Hoewel de meeste algoritmen niet voldeden aan de vereisten bleken er twee toch zeer geschikt voor snelle en nauwkeurige synchronisatie van realtime streams. In dit gedeelte zullen deze methoden in detail worden behandeld. Ook wordt er onderzocht in welke mate het mogelijk is om deze algoritmes te combineren tot één systeem.

2.1.1 Acoustic fingerprinting

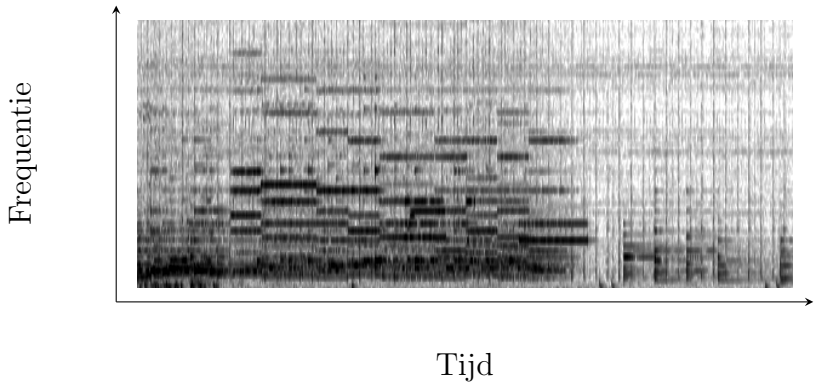
Bij het acoustic fingerprinting algoritme worden fingerprints geëxtraheerd uit audiofragmenten. Het zoek naar gelijkenissen gebeurt door de fingerprints met elkaar te vergelijken.

Features

Een cruciale stap bij de ontwikkeling van een acoustic fingerprinting systeem is het bepalen van een betrouwbare *feature* om de fingerprints op te baseren. Een feature is een kenmerk waarmee het mogelijk is om audiofragmenten van elkaar te onderscheiden. Mogelijke fea-

tures zijn bijvoorbeeld *onsets*[1] of frequentie. Een andere zeer goed bruikbare feature zijn de *spectrale pieken* in het tijd-frequentie spectrum van de geluidsfragmenten. Deze feature is compact op te slaan en bevat veel informatie over het opgenomen audiofragment. Hierdoor wordt de kans kleiner dat fingerprints gematcht worden zonder dat ze daadwerkelijk gebaseerd zijn op hetzelfde geluid.

Figuur 2.1: Spectrogram van *Talk Talk - New Grass*. De donkere vlekken zijn pieken zijn frequentie-intervallen die aan een relatief hoge energie voorkomen.



Werking

Een acoustic fingerprinting systeem gebaseerd op de extractie van spectrale pieken gaat in verschillende stappen te werk:

Eerst wordt het tijdsignaal (de typische golfvorm) van elk geluidsfragment omgezet tot een verzameling functies in het frequentiedomein. Deze omzetting gebeurt met het *Fast Transformation* algoritme (FFT). Het tijdsignaal wordt in kleine stukjes onderverdeeld (standaardgrootte: 512 samples, zie 3.2.2). Elk stukje audio wordt opgeslagen in een buffer waarop vervolgens het FFT algoritme op wordt uitgevoerd[2]. De opeenvolgende buffers worden genummerd met een *buffer index*. De inhoud van de buffer kan gezien worden als een signaal in het tijddomein. Het resultaat van het FFT algoritme is de fouriergetrans-

[1] Een onset is een markering in de tijd die het begin van een piek aanduidt. In artikel [9] wordt de betekenis en detectie van onsets uitgebreid beproken.

[2] Het uitvoeren van een FFT op zo'n klein stukje audio wordt ook wel de *Short Time Fourier Transformation* of SFT genoemd

formeerde van dit signaal: een eindige reeks van frequentie-intervallen. Elke frequentie-interval is genummerd met een *bin index*. De verzameling van de fourier getransformeerden van elke buffer stelt het audiofragment voor in het tijd-frequentie domein.

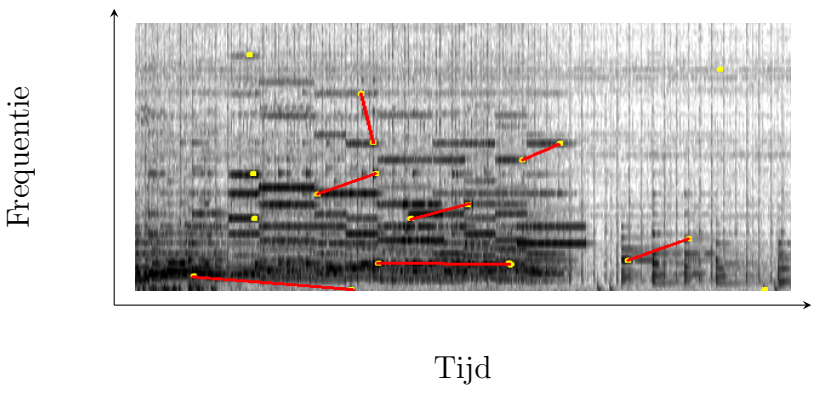
De grafische voorstelling hiervan wordt het spectrogram genoemd. Een spectrogram is het duidelijkst wanneer op de x-as de tijd en op de y-as de frequentie wordt weergegeven. De intensiteit waarmee een bepaalde frequentie voorkomt kan worden aangeduid door gebruik te maken van verschillende kleuren of contrasten. Figuur 2.1 toont een spectrogram waarbij frequentie met een hoge intensiteit donkerder zijn weergegeven.

In artikel [17] wordt het FFT algoritme uitgebreid besproken.

Na het omzetten van de te vergelijken geluidsfragmenten naar hun tijd-frequentie representatie kan er naar kandidaat-pieken worden gezocht. Dit zijn lokale maxima waarbij de hoeveelheid energie waarmee de frequentie voorkomt hoger is dan bij zijn buren [21]. In het spectrogram kan elk donker vlekje gezien worden als een kandidaat-piek.

Wanneer deze stap is afgerond kunnen de fingerprints bepaald worden. Een fingerprint is een de verbinding tussen twee spectrale pieken. Welke kandidaat-pieken gebruikt zullen worden in fingerprints hangt af van de implementatie van het algoritme en de ingestelde parameters. Enkele parameters die hier invloed op hebben zullen in sectie 3.2.2 van deze scriptie besproken worden. Figuur 2.2 toont een spectrogram waarop enkele kandidaat-pieken en fingerprints zijn aangeduid.

Figuur 2.2: De kandidaat-pieken (gele stipjes) en fingerprints (rode lijnen) van een fragment uit *Talk Talk - New Grass*.

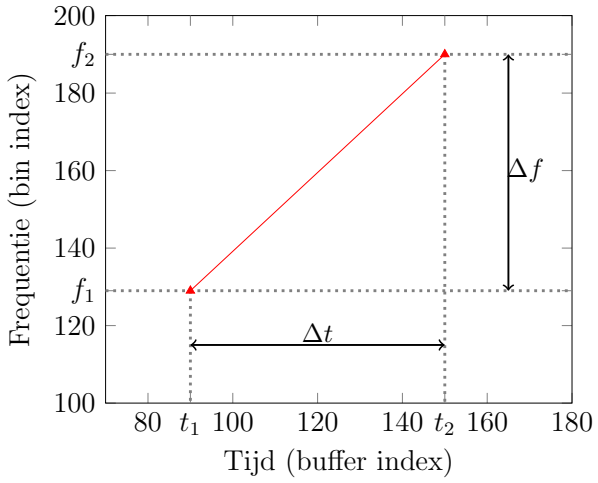


Na het bepalen van de fingerprints worden ze opgeslagen in een datastructuur waarin snel naar matches kan worden gezocht. Om dit mogelijk te maken is het noodzakelijk dat er van de fingerprints enkele typerende getallen bepaald worden.

- $f1$ en $f2$: de frequentie van de spectrale pieken van de fingerprint.
- $t1$ en $t2$: de tijd van de spectrale pieken van de fingerprint.
- Δf: het verschil van de frequenties van beide spectrale pieken van de fingerprint.
- Δt: het verschil van de tijd van beide spectrale pieken van de fingerprint.

Figuur 2.3 toont een schematische voorstelling van een fingerprint waarop deze getallen zijn aangeduid.

Figuur 2.3: De anatomie van een fingerprint in het tijd-frequentie domein. De rode lijn stelt de fingerprint voor tussen twee (niet afgebeelde) spectrale pieken. De typische parameters van de fingerprint zijn aangeduid op de assen. Met toestemming overgenomen uit artikel [22].



Bij het zoeken naar matches kan er gesteund worden op enkele typische eigenschappen van fingerprints:

Twee overeenkomende fingerprints uit twee geluidsfragmenten zullen dezelfde frequenties ($f1$ en $f2$) hebben. Bijgevolg is ook het verschil in frequentie (Δf) gelijk.

De tijd van de spectrale pieken ($t1$ en $t2$) komt meestal niet overeen. Bij een toepassing zoals Shazam is het bijvoorbeeld geen vereiste om een opname te maken vanaf het begin van een liedje. Het moment van de opname mag volledig willekeurig worden gekozen. Bij

het synchroniseren van streams wordt gezocht naar het verschil tussen de begintijden ($t1$ van elke fingerprint) van de overeenkomstige fingerprints van de audiofragmenten. Dit tijdverschil (Δt) is wel gelijk bij elk paar overeenkomende fingerprints.

Uit voorgaande eigenschappen kan geconcludeerd worden dat fingerprints uit twee audiofragmenten matchen wanneer $f1$, Δf en Δt gelijk zijn. Om deze parameters snel met elkaar kunnen te vergelijken wordt er een berekening uitgevoerd die deze parameters omzet in één enkel getal. Dit getal wordt de hash van de fingerprint genoemd. Samen met deze hash wordt ook $t1$ en een identificatie van het geluidsfragment bijgehouden.

Artikel [21] geeft meer informatie over de omzetting van deze drie getallen tot een hash.

Een fingerprint kan bijgevolg gezien worden als verzameling gegevens met de volgende structuur: $(id; t1; hash(f1; \Delta f; \Delta t))$. Het zoeken naar fingerprints met overeenkomstige hashwaarden is mogelijk in $O(1)$ door gebruik te maken van een hashtabel. De precieze werking hiervan valt buiten de scope van deze scriptie.

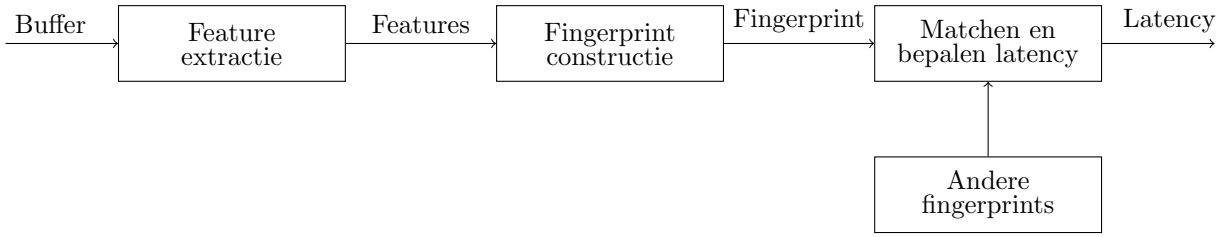
Om te bepalen of twee audiofragmenten wel degelijk overeenkomen wordt er gezocht naar alle fingerprints met een overeenkomende hashwaarde. Van elk paar overeenkomende fingerprints wordt het verschil tussen $t1$ berekend. Dit verschil wordt de *offset* genoemd. Het vinden van een groot aantal matches met dezelfde offset wijst op een sterke gelijkenis tussen de audiofragmenten. De precieze waarde van "een groot aantal" wordt bepaald door een parameter van het algoritme.

Bepalen van de latency

Acoustic fingerprinting kan gebruikt worden om streams te synchroniseren door de ze eerst te bufferen. Wanneer een buffer volledig is opgevuld kan deze net zoals een kort audiofragment worden verwerkt door het algoritme. De latency tussen streams wordt bepaald door de offset die in vorig paragraaf werd beschreven: het verschil tussen de $t1$ waarden stelt namelijk de verschuiving tussen de geluidsfragmenten voor.

Figuur 2.4 toont schematisch alle stappen die moeten worden doorlopen om met behulp van acoustic fingerprinting de latency tussen audiofragmenten te bepalen.

Figuur 2.4: Schematische voorstelling van synchronisatie met behulp van een acoustic fingerprinting systeem.



Een uitgebreidere beschrijving is te vinden in artikel [25]. De methode die in het artikel en deze scriptie besproken werd is beperkt tot het vergelijken van audiofragmenten die in tijd noch toonhoogte gewijzigd zijn. Aan het IPEM is een aangepaste methode ontwikkeld die dit wel toelaat [21].

Nauwkeurigheid

Zowel de snelheid waarmee wijzigingen van de latency bepaald kunnen worden als de nauwkeurigheid van de latency zelf hangt af van heel wat verschillende parameters van het algoritme.

De detectiesnelheid is vooral afhankelijk van de buffergrootte waarop het algoritme wordt uitgevoerd. Met deze instelling moet echter omzichtig worden omgegaan: een te kleine buffergrootte kan er toe leiden dat het algoritme niet meer in staat is om voldoende matches te vinden. Het kan helpen om andere parameters te wijzigen waardoor het vinden van een groot aantal matches gegarandeerd blijft. Deze parameters worden in sectie 3.2.2 in detail besproken.

De nauwkeurigheid van de latency van het algoritme hangt af van de parameters van het FFT algoritme. Een nauwkeurigheid van 16 ms of 32 ms is standaard. De precieze werking van het FFT algoritme valt buiten de scope van deze scriptie.

2.1.2 Kruiscovariantie

Deze methode bepaalt de gelijkenis tussen twee audiofragmenten en resulteert in één getal. Dit getal is een soort van score die aangeeft in welke mate twee signalen overeenkomen. De latency tussen twee audiofragmenten kan bepaald worden door deze berekening uit te voeren voor **elke mogelijke verschuiving**. De verschuiving waarbij het resulterend getal het hoogst is bepaalt de latency.

Werking

Stel twee audioblokken a en b bestaande uit een gelijk aantal samples (n). Deze audioblokken worden telkens cyclisch één sample verschoven. De variabele i stelt de huidige verschuiving voor.

$$k_i = \sum_{j=0}^{n} a_j \cdot b_{(i+j) \bmod n} \qquad i = 0, 1, .., n \tag{2.1}$$

De kruiscovariantie waarde (k_i) wordt voor elke mogelijke verschuiving berekend. De waarde van i waarbij de kruiscovariantie het hoogst is stelt de latency voor tussen beide audioblokken in aantal samples. De latency in seconden kan bepaald worden door dit resultaat te delen door de samplefrequentie.

De methode kan de latency **tot op één sample nauwkeurig** bepalen. De maximaal bereikbare nauwkeurigheid hangt dus af van de samplefrequentie van de audioblokken. Bij een samplefrequentie van $8000Hz$ is dit $1/8000Hz = 0.125ms$. Dit is ruim voldoende voor het huidige probleem.

Een nadeel aan deze methode is de performantie. Het berekenen van de beste kruiscovariantie van twee audioblokken bestaande uit n samples kan gebeuren in $O(n^2)$. Het is dus belangrijk om bij deze berekening de grootte van de audioblokken te beperken.

In artikel [22] wordt deze techniek meer in detail besproken.

Toepassing in realtime

Het bufferen van de audiostreams maakt ook dit algoritme in realtime toepasbaar. In tegenstelling tot acoustic fingerprinting is het niet de bedoeling dat de berekeningen op de volledige buffer wordt uitgevoerd. Door de kwadratische tijdscomplexiteit zou het algoritme onnoemelijk veel rekenkracht vragen.[3] Er moet dus een manier gevonden worden waarmee het mogelijk is om het aantal samples waarop het algoritme wordt uitgevoerd beperkt wordt.

2.1.3 Toepasbaarheid

Het acoustic fingerprinting algoritme is zeer snel en robuust en kan gebruikt worden om gebufferde audiostreams te synchroniseren tot enkele tientallen milliseconden nauwkeurig (afhankelijk van de parameters van het FFT algoritme).

Het kruiscovariantie algoritme kan eveneens gebruikt worden om (gebufferde) audiostreams te synchroniseren. De grootste troef van dit algoritme is haar nauwkeurigheid: in de beste omstandigheden kan het algoritme resultaten bekomen tot op één sample nauwkeurig. Het bereiken van een dergelijke nauwkeurigheid is onmogelijk met eender welk ander besproken algoritme. De keerzijde is de performantie van het algoritme. Bij het synchroniseren van grote audioblokken kan dit problematisch zijn.

De kenmerken van deze algoritmen zijn complementair. De gemakkelijkste manier om een robuust, snel én nauwkeurig systeem op te bouwen is door het beste van de twee werelden te combineren. Het acoustic fingerprinting algoritme kan zorgen voor de synchronisatie tot op enkele tientallen milliseconden nauwkeurig. In een tweede stap kan het kruiscovariantie algoritme worden uitgevoerd op zeer korte stukjes audio (een honderdtal samples volstaan).

[3] Voor het berekenen van de kruiscovariantie tussen twee buffers met $10s$ audio en een samplefrequentie van $8000hz$ zijn er asymptotisch $6.4 \cdot 10^9$ berekeningen vereist.

2.2 Bufferen van streams

Aangezien de algoritmes een bepaalde hoeveelheid audio nodig hebben vooraleer ze kunnen worden uitgevoerd is het noodzakelijk om de streams eerst te bufferen. Dit proces moet herhaald worden aangezien er mogelijk samples gedropt worden of drift kan ontstaan. In dit deel zal worden uitgelegd hoe het bufferen precies in zijn werk gaat. Om verwarring met andere soorten buffers te vermijden zal dit type buffer verder in deze scriptie een *streambuffer* genoemd worden. Met het woord *slice* zal naar de inhoud verwezen worden.

Buffergrootte

De grootte van de buffer (t) heeft invloed op de kwaliteit van de resultaten. Het spreekt voor zich dat het algoritme beter kan presteren wanneer er 10 seconden in plaats van 1 seconde audio geanalyseerd wordt. Een nadeel is echter dat het langer duurt vooraleer een wijzing van de latency gedetecteerd kan worden. De buffergrootte bepaalt ook de maximale lengte tot wanneer de latency te detecteren is.

Naïeve implementatie

De naïeve implementatie kan een wijziging van de latency in het beste geval na $\frac{1}{2}t$ seconden detecteren. Wanneer er samples gedropt worden net na het moment dat de buffer voor de helft gevuld is duurt de detectie veel langer: $\frac{3}{2}t$ seconden. Een betere implementatie kan dit inkorten.

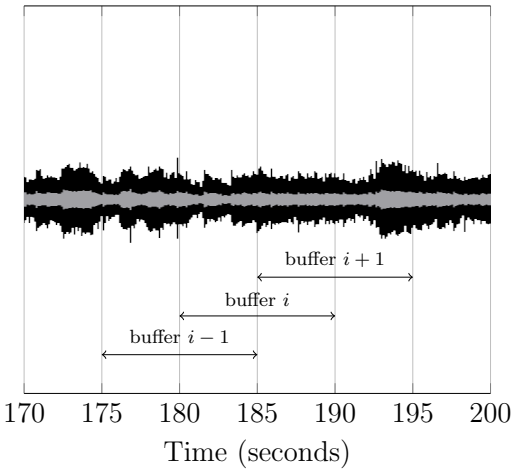
Sliding window

Een meer doordachte manier van bufferen maakt gebruik van een *sliding window*. Onderstaande beschrijving gebruikt een buffer van t seconden en een stapgrootte van s seconden. Hierbij geldt dat $s \leq t$.

Het verschil met de naïeve methode is dat de buffer niet pas na t seconden wordt opgeschoven. Door de buffer al na s seconden op te schuiven zal een wijziging van de latency

sneller gedetecteerd kunnen worden, dit terwijl het algoritme toch nog steeds t seconden audio analyseert. In figuur 2.5 wordt grafisch weergegeven hoe de buffer precies verschoven wordt met $t = 10$ en $s = 5$.

Figuur 2.5: Schematische weergave van een *sliding window* buffer over een audiostream.



Door de buffer al na s seconden op te schuiven wordt het slechtste geval sterk verbeterd. In het slechtste geval wordt een wijziging van de latency gedetecteerd na $\frac{t}{2} + s$ seconden. Het beste geval blijft wel nog steeds $\frac{t}{2}$ seconden.

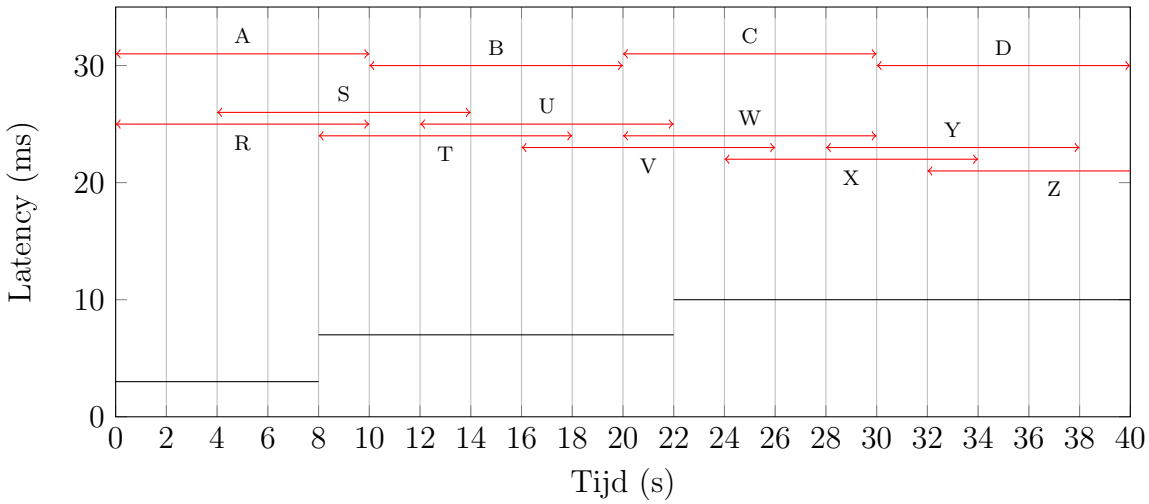
Het verkleinen van de stapgrootte zorgt ervoor dat het algoritme per hoeveelheid audio frequenter moet worden uitgevoerd. Een te kleine stapgrootte heeft bijgevolg een negatieve invloed op de performantie.

Voorbeeld

Een praktisch voorbeeld zal bovenstaande beschrijving wat verduidelijken. In het voorbeeld worden twee audiostreams van 40 seconden geanalyseerd. Door het droppen van samples neemt de latency tussen de streams stapsgewijs toe. Figuur 2.6 toont in het zwart hoe de latency gedurende de verwerking evolueert. De opeenvolgende buffers van de twee besproken methode's worden in het rood aangeduid.

De initiële latency van 3 milliseconden wordt zowel met de naïeve methode als met het sliding window gedetecteerd na de analyse van de allereerste buffer (A of R) 10 seconden

Figuur 2.6: Grafisch weergave van de methode's waarop gebufferd kan worden. De zwarte lijn stelt de huidige latency voor. In het rood worden de opeenvolgende buffers weergegeven.



na aanvang van de analyse. De eerste verhoging tot 7 milliseconden vindt te laat plaats om gedetecteerd te kunnen worden door de eerste buffer van beide methodes. Bij deze verhoging van de latency wordt het verschil tussen beide methodes zichtbaar: bij de sliding window methode vindt de detectie 6 seconden na de wijziging plaats. Bij de naïeve methode moet er echter gewacht worden tot wanneer buffer B is volgelopen 12 seconden na de wijziging. De tweede verhoging naar 10 milliseconden wordt zowel door de naïeve methode als door de sliding window methode gedetecteerd 8 seconden na de wijziging (buffer C of W).

Conclusie

De detectiesnelheid van een latencywijziging hangt af van twee parameters: de bufferlengte (t) en de staplengte (s). De snelheid waarmee een wijziging gedetecteerd kan worden (T) kan als volgt worden samengevat:

$$\frac{t}{2} < T < \frac{t}{2} + s \tag{2.2}$$

2.3 Synchroniseren van streams

In deze toepassing wordt de latency van elke stream bepaald ten opzichte van een referentiestream. Het is niet geweten welke stream voorloopt of achterloopt. Wanneer de referentiestream een bepaalde vertraging heeft ten opzichte van een andere stream dan zal de latency van de andere stream negatief zijn. Bij de daadwerkelijke synchronisatie is het belangrijk dat hiermee rekening gehouden wordt.

Algoritme

Elke keer wanneer er een nieuwe verzameling latencies (ten opzichte van de referentiestream) bepaald is moeten de streams worden aangepast. Dit kan gebeuren door het toevoegen van stilte (samples met waarde 0.0). Het volgende algoritme berekent hoeveel stilte aan welke streams moet worden toegevoegd.

```
1:  function CORRECTIES(L, P)        ▷ L: lijst van latencies, P: lijst van vorige latencies
2:      n ← lengte van L en P
3:      h ← −∞                                          ▷ h: de huidige hoogste correctie
4:      C_i ← lege lijst                          ▷ lijst met de correcties van elke stream
5:      for i ← 1..n do
6:          l ← aantal samples in L_i
7:          p ← aantal samples in P_i
8:          c ← l − p                          ▷ c: correctie in samples van stream i
9:          if c > h then
10:             h ← c                                  ▷ wijzigen van hoogste correctie
11:         end if
12:         C_i ← c
13:     end for
14:     for i ← 1..n do
15:         C_i ← h − C_i              ▷ correctie wordt relatief t.o.v. hoogste correctie
16:     end for
17:     return C
18: end function
```

In het algoritme wordt op basis van de nieuwe en vorige verzameling van latencies per stream de correctie bepaald. Dit is het aantal nul-samples die aan de overeenkomstige stream moet worden toegevoegd. Bij de eerste berekening (regel 8) is het mogelijk dat de correctie negatief is. Aangezien er geen audio verloren mag gaan door samples weg

te knippen wordt de correctie omgezet zodat deze relatief is ten opzichte van de hoogste correctie (regel 16).

Door het berekende aantal nul-samples aan elke stream toe te voegen worden de streams gesynchroniseerd.

Hoofdstuk 3

Implementatie

3.1 Technologieën en software

3.1.1 Java 7

Er zijn verschillende redenen waarom er gekozen is om de synchronisatie bibliotheek in Java te implementeren. De belangrijkste reden is dat de bestaande audio bibliotheken (Panako en TarsosDSP) van het IPEM ook ontwikkeld zijn in Java. Deze kunnen enkel aangeroepen worden via andere Java applicaties.

Het is ook de bedoeling is om de gebruikersinterface te ontwikkelen met behulp van Max/MSP modules. Het ontwikkelen van dergelijke modules is mogelijk in twee programmeertalen: C en Java. De eenvoud en hoge graad van abstractie gaf de doorslag om hierbij Java te gebruiken. De voordelen die C biedt op vlak van snelheid wegen (in deze toepassing) hier niet tegenop.

Hoewel de laatste versie van Max/MSP (7) het toelaat om Java 8 te gebruiken wordt deze versie in dit project om compatibiliteitsredenen vermeden.

3.1.2 JUnit

JUnit is een unit testing framework voor Java. JUnit heeft in dit onderzoek een belangrijke rol gespeeld bij het bepalen van de optimale parameters van de verschillende algoritmen. JUnit maakte het mogelijk om de algoritmes herhaald maar met verschillende parameters op een dataset los te laten. JUnit liet toe om in één oogopslag te zien hoe het algoritme heeft gepresteerd.

3.1.3 TarsosDSP

TarsosDSP is een Java bibliotheek voor realtime audio analyse en verwerking ontwikkeld aan het IPEM. De bibliotheek bevat een groot aantal algoritmes voor audioverwerking en kan nog verder worden uitgebreid. Deze bibliotheek wordt beschreven in artikel [23].

TarsosDSP is voornamelijk gebouwd rond het concept *processing pipeline*. Dit is een abstractie van een audiostream die via programmacode verwerkt kan worden. Een processing pipeline wordt voorgesteld als instantie van de klasse `AudioDispatcher`. Een `AudioDispatcher` kan aangemaakt worden van een audiobestand, een array van floating-point getallen of een microfoon en kan bewerkt of verwerkt worden met behulp van één of meerdere `AudioProcessors`. Objecten van klassen die de interface `AudioProcessor` implementeren kunnen aan de processing pipeline worden toegevoegd. In de implementatie van de `process` methode kan de audiostream op een bepaalde manier verwerkt, geanalyseerd of gewijzigd worden.

TarsosDSP bevat verder nog een groot aantal klassen met allerhande tools en audioverwerkings algoritmen. Een greep uit de features van TarsosDSP:

- Toevoegen van geluidseffecten (delay, flanger,...)
- Toevoegen van filters (low-pass, high-pass, band-pass,...)
- Conversie tussen verschillende formaten
- Toonhoogte detectie
- Wijzigen van de samplefrequentie

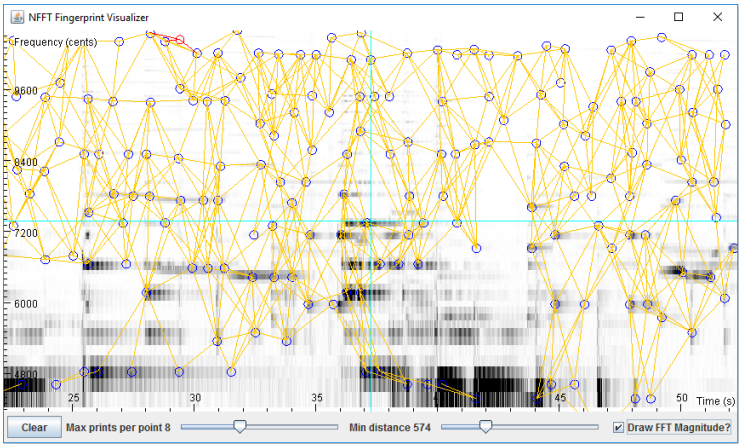
3.1.4 Panako

Panako is net zoals TarsosDSP een Java bibliotheek (door de zelfde auteur) ontwikkeld aan het IPEM. Panako bevat een arsenaal aan algoritmen voor het matchen of synchroniseren van audiofragmenten of audiostreams. Deze bibliotheek wordt uitgebreid beschreven in artikel [21].

Dit onderzoek gebruikt Panako's open-source implementatie van het acoustic fingerprinting algoritme dat beschreven is in artikel [25]. Panako bevat ook een uitbreiding van het algoritme dat overweg kan met audio waarbij de toonhoogte verhoogd of verlaagd is, of waarbij de audio sneller of trager is afgespeeld.

Buiten het algoritme bevat de bibliotheek ook verschillende toepassingen die hiervan gebruik maken. Zo is het mogelijk om de fingerprints van een geluidsfragment te bekijken, matches tussen verschillende geluidsfragmenten te visualiseren, en grafisch te experimenteren met de verschillende parameters. Figuur 3.1 toont een screenshot van deze toepassing.

Figuur 3.1: De gebruikersinterface van Panako's NFFT Fingerprint Visualiser. Onderaan kunnen de parameters van het algoritme met behulp van sliders gewijzigd worden.



Er is ook een applicatie beschikbaar (SyncSink) om verschillende geluidsfragmenten (niet in realtime) te synchroniseren. Deze applicatie gebruikt naast het acoustic fingerprinting algoritme ook het kruiscovariantie algoritme.

Na het bepalen van de latency tussen de verschillende audiofragmenten kan de applicatie

een shell script genereren dat met behulp van *FFmpeg* stukjes van de geluidsbestanden wegknipt of er stilte aan toevoegt. Het resultaat is dat na het uitvoeren van het script de geluidsbestanden gesynchroniseerd zijn.

3.1.5 FFmpeg

FFmpeg is een command-line multimedia framework dat gebruikt wordt voor encoderen, decoderen, multiplexen, demultiplexen, streamen en afspelen van audio en video. [15]

In dit onderzoek wordt FFmpeg voornamelijk gebruikt in scripts bij het geautomatiseerd genereren van testdata.

3.1.6 SoX

SoX is net zoals FFmeg een command-line tool voor audioverwerking. Buiten de mogelijkheid om audiobestanden te converteren laat SoX ook minder triviale operaties toe. Zo is het onder meer mogelijk om het volume aan te passen, effecten toe te voegen, de bestanden bij te knippen of gegenereerde geluiden in een audiobestand te mixen. [8]

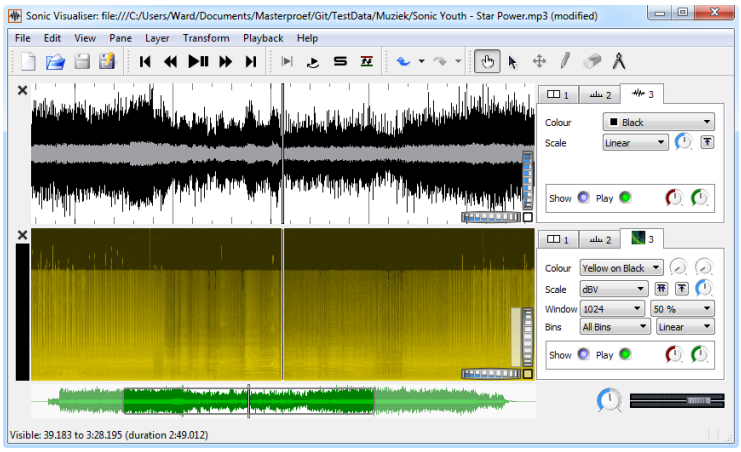
In dit onderzoek wordt SoX gebruikt in scripts bij het manipuleren van de testdata.

3.1.7 Sonic Visualiser

Sonic Visualiser is een gebruiksvriendelijke desktopapplicatie voor de analyse, visualisatie van audiobestanden. Sonic Visualiser laat toe om audiobestanden vanuit verschillende perspectieven te analyseren. Zo kan zowel de golfvorm als het spectrogram van een audiobestand gevisualiseerd worden. Sonic Visualiser is uitbreidbaar met plug-ins in het *Vamp* formaat. [10]

Sonic Visualiser werd in dit onderzoek vooral gebruikt als testtool om handmatig de latency tussen verschillende audiofragmenten te bepalen. De handmatig bepaalde latency werd vervolgens vergeleken met de berekende resultaten.

Figuur 3.2: De gebruikersinterface van Sonic Visualiser

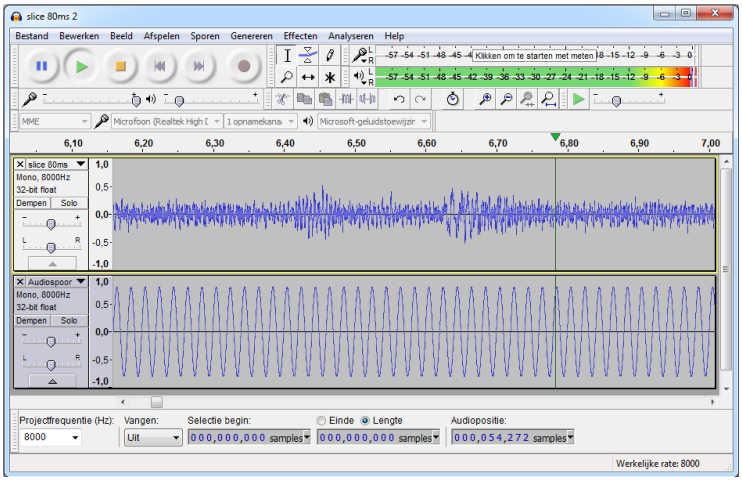


3.1.8 Audacity

Audacity is een open-source desktopapplicatie voor het bewerken, opnemen en converteren van audio. Met Audacity is het ook mogelijk om tal van effecten en filters aan audio toe te voegen.[1]

Figuur 3.3 toont de gebruikersinterface van dit programma.

Figuur 3.3: De gebruikersinterface van Audacity

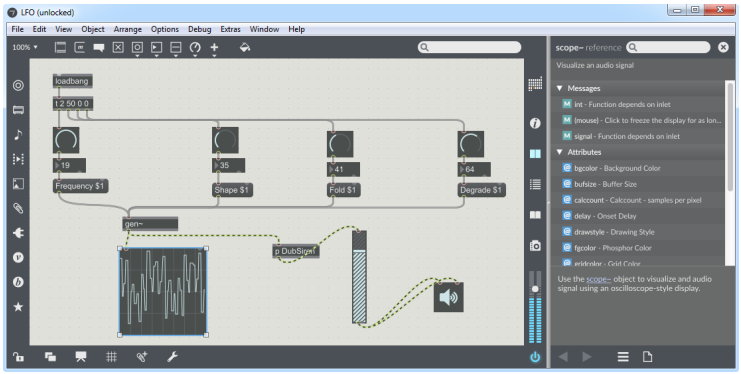


Alle opnames en handmatige bewerkingen op audiobestanden in dit onderzoek zijn uitgevoerd met behulp van Audacity.

3.1.9 Max/MSP

Max/MSP is een visuele programmeertaal voor muziek en multimedia. Het is een systeem waarbij modules met elkaar verbonden kunnen worden om zo complexere systemen op te bouwen. Max/MSP beschikt ook over een API waarmee in Java of C nieuwe modules ontwikkeld kunnen worden. [2]

Figuur 3.4: De gebruikersinterface van Max/MSP: een *patch panel* met daarop enkele modules die samen een complexere toepassing vormen.



Met Max/MSP is het mogelijk om realtime audio te verwerken, daarom zal deze toepassing gebruikt worden voor het ontwikkelen van de gebruikersinterface.

Figuur 3.4 toont een eenvoudige Max/MSP patch.

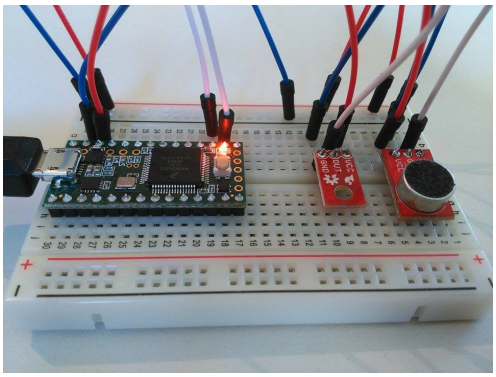
3.1.10 Teensy

De Teensy is een kleine microcontroller die via USB geprogrammeerd kan worden. De Teensy is compatibel met de Arduino software en is hierdoor zeer gebruiksvriendelijk. [5]

De sensoren die gebruikt worden bij de experimenten van het IPEM zijn meestal aangesloten op Teensy microcontrollers.

In hoofdstuk 4 worden de testen die met de Teensy microcontroller zijn uitgevoerd meer in detail besproken. Figuur 3.5 toont een typische testopstelling waarbij twee sensoren zijn aangesloten op de microcontroller.

Figuur 3.5: De Teensy microcontroller verbonden met een infraroodsensor en microfoon op een breadboard.



3.2 Acoustic fingerprinting

De Panako softwarebibliotheek bevat een implementatie van het acoustic fingerprinting algoritme. Om wijzigingen mogelijk te maken hebben is de code van het algoritme overgenomen in het project van dit onderzoek. De aangepaste code blijft wel nog steeds afhankelijk van enkele klassen uit het Panako project.

3.2.1 Optimalisaties

Aan dit algoritme is één vereenvoudiging aangebracht. Het originele algoritme bevatte namelijk de mogelijkheid om alle offsets boven een bepaalde drempelwaarde te verwerken. Deze feature laat toe dat er meerdere matches kunnen gevonden worden binnen één uitvoering van het algoritme. Om dit te ondersteunen moeten alle matches echter wel één voor één worden vergeleken met de drempelwaarde. Omdat deze toepassing enkel de beste offsetwaarde nodig heeft is dit overbodig. De beste offset en bijhorende fingerprints wordt apart bijgehouden. Hierbij wordt vermeden dat de matches op het einde moeten worden overlopen.

3.2.2 Parameters en hun invloed op het algoritme

De werking van dit algoritme is afhankelijk van een aantal parameters die een grote invloed kunnen hebben op de performantie en de nauwkeurigheid van het uiteindelijke resultaat. Daarom is het van belang om voor het uitvoeren van het algoritme de waarde van deze parameters te controleren. De optimale waarde van elke parameter is afhankelijk van verschillende factoren die van situatie tot situatie kunnen verschillen:

- de vereiste nauwkeurigheid van het algoritme;
- de vereiste performantie van het algoritme;
- de mate waarin er omgevingsgeluid aanwezig is;
- de opnamekwaliteit van het omgevingsgeluid.

De meeste parameters worden bijgehouden in een configuratiebestand waardoor ze ook na compilatie wijzigbaar zijn. Dit zijn de belangrijkste parameters uit het configuratiebestand die invloed hebben op het algoritme:

`SAMPLE_RATE`

Deze parameter bepaalt de standaard samplefrequentie die gebruikt wordt tijdens het synchronisatieproces. Het verhogen van deze parameter zorgt voor een tragere verwerking maar een betere nauwkeurigheid. Afhankelijk van op welke manier de synchronisatie wordt opgestart (via Max of met een `AudioDispatcher`) worden de binnenkomende streams geresamplet of wordt al een correcte samplefrequentie verondersteld.

`NFFT_BUFFER_SIZE`[1]

Dit is de grootte van de verschuivende buffer die gebruikt wordt in het FFT algoritme. Deze parameter is cruciaal aangezien de frequentiesterktes op een bepaalde plaats op de tijdas worden berekend per buffer. Deze parameter wordt uitgedrukt in aantal samples.

[1] De letter N in NFFT heeft geen noemenswaardige betekenis. De naam is overgenomen van de gelijknamige parameter uit de Panako bibliotheek.

`NFFT_STEP_SIZE`

Dit is het aantal samples van elke verschuiving in het FFT algoritme. De stepsize beïnvloedt rechtstreeks de nauwkeurigheid van het acoustic fingerprinting algoritme. Wanneer deze parameter is ingesteld op 128 samples en een samplefrequentie van $8000Hz$ dan is de maximale nauwkeurigheid $128/8000Hz = 0.016s = 16ms$.

`MIN_ALIGNED_MATCHES`

Een match tussen twee audiofragmenten wordt pas als geldig beschouwd wanneer er een bepaald aantal matchende fingerprints met dezelfde offset gevonden zijn. Dit aantal wordt ingesteld met deze parameter.

`NFFT_MAX_FINGERPRINTS_PER_EVENT_POINT`

Deze parameter bepaalt het maximum aantal fingerprints waaraan een event point (een punt op het spectrogram) kan deelnemen. Hoe hoger dit maximum hoe vlugger er matches kunnen gevonden worden. Het hanteren van een hoge waarde heeft een negatieve invloed op de performantie.

`NFFT_EVENT_POINT_MIN_DISTANCE`

Dit is de minimale afstand tussen twee event points op het spectrogram die samen een fingerprint kunnen vormen.

Verder maakt het algoritme nog gebruik van twee hardgecodeerde parameters die niet instelbaar zijn via het configuratiebestand: `MIN_FREQUENCY` en `MAX_FREQUENCY`. Deze parameters bepalen binnen welke frequentiebereik er naar fingerprints gezocht worden. De waarden waarop deze ingesteld staan bevinden zich op de rand van de frequenties die door muziek of stemgeluid geproduceerd worden.

3.2.3 Optimale instellingen

Het bepalen van de optimale waarden voor de parameters is geen exacte wetenschap maar eerder een probleem dat proefondervindelijk moet worden aangepakt.

Bij het acoustic fingerprinting algoritme is er een groot verschil tussen de meest “elegante” parameterwaarden en de in de praktijk best presterende waarden. Dit verschil zal duidelijk

worden in volgende opsomming waarin elke parameter zal worden besproken.

`SAMPLE_RATE`

Bij deze parameter is het van belang om een goede balans te vinden zodat de geluidskwaliteit aanvaardbaar blijft zonder een hypotheek te plaatsen op de performantie van het algoritme. De praktijk heeft uitgewezen dat bij een samplefrequentie van $8000Hz$ het algoritme goed presteert. Deze waarde wordt bevestigd in artikel [22].

`NFFT_BUFFER_SIZE` en `NFFT_STEP_SIZE`

De ingestelde waarden van de samplefrequentie, buffergrootte en stapgrootte zijn afhankelijk van elkaar. Om een goede werking van het FFT algoritme te garanderen bij een samplefrequentie van $8000Hz$ worden de buffergrootte en stapgrootte respectievelijk ingesteld op 512 en 128 (of 256) samples. Deze waarden worden eveneens vermeld in artikel [22].

`MIN_ALIGNED_MATCHES`

In een toepassing zoals het detecteren van liedjes ten opzichte van een database is het secuur instellen van deze parameter erg belangrijk. Deze parameter heeft namelijk een grote invloed op het voorkomen van *false positives* of *false negatives*. De standaardwaarde in Panako is 7.

Bij het synchroniseren van streams is de situatie echter helemaal anders. Het binnenkrijgen van een false positive (=foute latency) is veel minder erg dan het helemaal niet binnenkrijgen van (mogelijk correcte) resultaten. Aangezien het algoritme per buffer enkel het beste resultaat teruggeeft is de kans dat bij geluidsfragmenten van behoorlijke kwaliteit eenzelfde foute latency meer voorkomt dan de correcte latency zéér klein.

Bij geluidsfragmenten van mindere kwaliteit kan het gebeuren dat er toch foute resultaten door de mazen van het net glippen. Om dit te vermijden is het mogelijk om nog een extra filtering toe te passen. In deze extra stap worden eventuele uitschieters geëlimineerd.

Bovenstaande argumenten stellen duidelijk dat het beter is om deze parameter een

lagere waarde te geven dan de standaardwaarde. In deze toepassing is gekozen voor de waarde 2 in plaats van het absolute minimum 1: hierdoor wordt pure willekeur bij het matchen van audiofragmenten van extreem slechte kwaliteit vermeden.

`NFFT_MAX_FINGERPRINTS_PER_EVENT_POINT`

In Panako is het standaard dat een event point uitmaakt van maximaal 2 fingerprints. Het hanteren van deze waarde leidt ertoe dat het matchen van audiofragmenten van behoorlijke kwaliteit zeer snel kan worden uitgevoerd.

In deze toepassing is het aantal te vergelijken audiofragmenten meestal erg beperkt. Ook is de kwaliteit van deze fragmenten vaak van ondermaats (bv. de opnames op microcontrollers). Daarom is het in dit geval een goed idee om de waarde van deze parameter zéér hoog in te stellen. Hierdoor verhoogt de kans sterk dat er bij zeer slechte audiofragmenten toch enkele overeenkomende fingerprints gevonden worden. Testen hebben uitgewezen dat de negatieve invloed op de performantie beperkt blijft en dat de resultaten sterk verbeteren.

Bij het zoeken naar matches tussen geluidsopnames opgenomen op microcontrollers heeft de praktijk uitgewezen dat het toelaten van 50 fingerprints per event point degelijke resultaten oplevert. De testresultaten uit sectie 4.2 bevestigen dit.

`NFFT_EVENT_POINT_MIN_DISTANCE`

In tegenstelling tot vorige parameter zorgt het verhogen van deze waarde ervoor dat er minder fingerprints worden gecreëerd. De argumenten die bij vorige parameter zijn aangehaald gelden bijgevolg in omgekeerde zin ook voor deze parameter. Hoewel de Panako standaard 600 is leveren waardes rond het getal 10 in deze toepassing de beste resultaten zonder de performantie sterk te beperken.

3.3 Kruiscovariantie

De Panako bibliotheek bevatte bij aanvang van dit onderzoek al een implementatie van het kruiscovariantie algoritme. In tegenstelling tot het acoustic fingerprinting algoritme

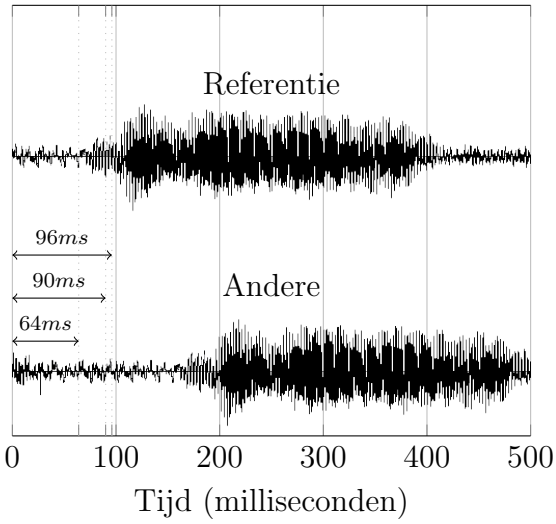
was het echter minder grondig afgewerkt. Om degelijke resultaten te garanderen was het noodzakelijk om enkele anomalieën in de geleverde resultaten te analyseren en de oorzaak hiervan op te lossen. Om te kunnen begrijpen wat er precies fout ging moet eerst de integratie met het acoustic fingerprinting algoritme besproken worden.

Net zoals het acoustic fingerprinting algoritme is de code overgenomen in het project van dit onderzoek. De code is niet meer afhankelijk van de Panako bibliotheek.

3.3.1 Integratie met acoustic fingerprinting

In sectie 2.1.3 is er geschreven dat de latency zeer nauwkeurig kan bepaald worden door het berekenen van de kruiscovariantie. Vanwege de performantie is het wel noodzakelijk om eerst de ruwe latency met acoustic fingerprinting te bepalen.

Figuur 3.6: Twee audiofragmenten: het tweede audiofragment heeft een latency van 90 milliseconden ten opzichte van het eerste.



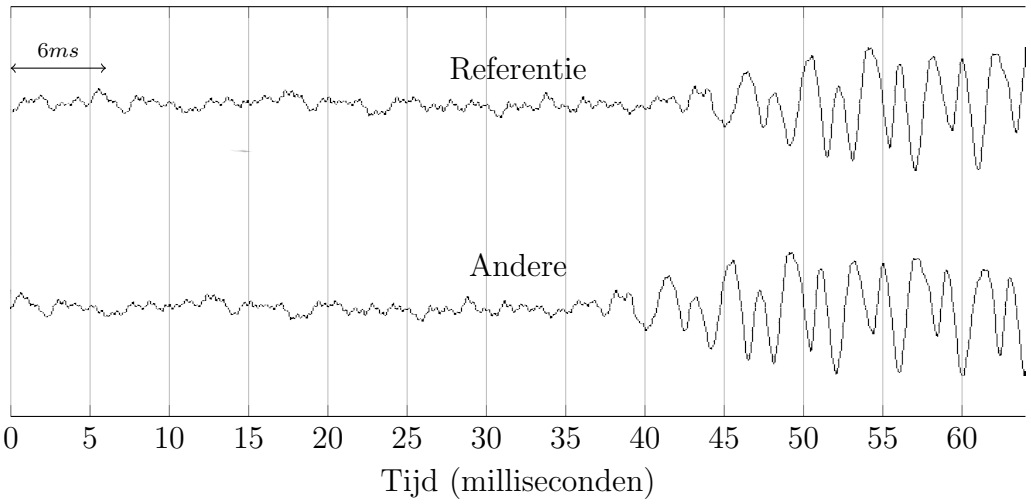
Ter illustratie twee geluidsopnames waarbij de tweede opname een vertraging heeft van 90 ms ten opzichte van de eerste referentieopname. Het uitvoeren van het acoustic fingerprinting algoritme met standaard parameters (tot op $32ms$ nauwkeurig) kan twee resultaten opleveren: $64ms$ of $96ms$. Na het berekenen van de kruiscovariantie kan bepaald worden of er zich een onderschatting (eerste waarde) of overschatting (tweede waarde) heeft voorgedaan, dit is van belang om het resultaat correct te verfijnen. Figuur 3.6 toont de mogelijke

resultaten van het algoritme, in dit voorbeeld $32ms$.

Als de ruw bepaalde latency positief is wordt deze waarde van het tweede fragment weggeknipt. Na deze stap is de latency tussen de resterende audiofragmenten zeker minder dan de minimale nauwkeurigheid van het acoustic fingerprinting algoritme.

Vervolgens worden een aantal samples van elk audiofragment gekopieerd naar een buffer. De latency tussen deze twee buffers wordt berekend met het kruiscovariantie algoritme (beschreven in sectie 2.1.2). Figuur 3.7 toont de buffers waarop het algoritme kan worden uitgevoerd bij een buffergrootte van 512 samples en samplefrequentie van $8000Hz$.

Figuur 3.7: Kruiscovariantie buffers na het wegknippen van de $96ms$ latency bepaald door het acoustic fingerprinting algoritme. Nu heeft de referentie audio $6ms$ latency ten opzichte van het andere audiofragment.



Wanneer het acoustic fingerprinting algoritme de werkelijke latency heeft onderschat wordt de verfijnde latency bekomen door de ruwe latency en het resultaat van het kruiscovariantie algoritme op te tellen. Bij een overschatting is dit niet het geval.

Aangezien het kruiscovariantie algoritme de latency zoekt van de “andere” buffer ten opzichte van de referentiebuffer zal het resultaat niet $6ms$ maar $58ms$ zijn. Dit komt doordat het algoritme de “andere” buffer cyclisch (maar in de verkeerde zin) verschuift.

Om in een dergelijke situatie de verfijnde latency te berekenen moet men van de som van de resultaten ($96ms + 58ms$) nog de lengte van de gebruikte buffer ($64ms$) aftrekken. De

verfijnde latency is dus $154ms - 64ms = 90ms$.

3.3.2 Optimalisaties

Bugfixes

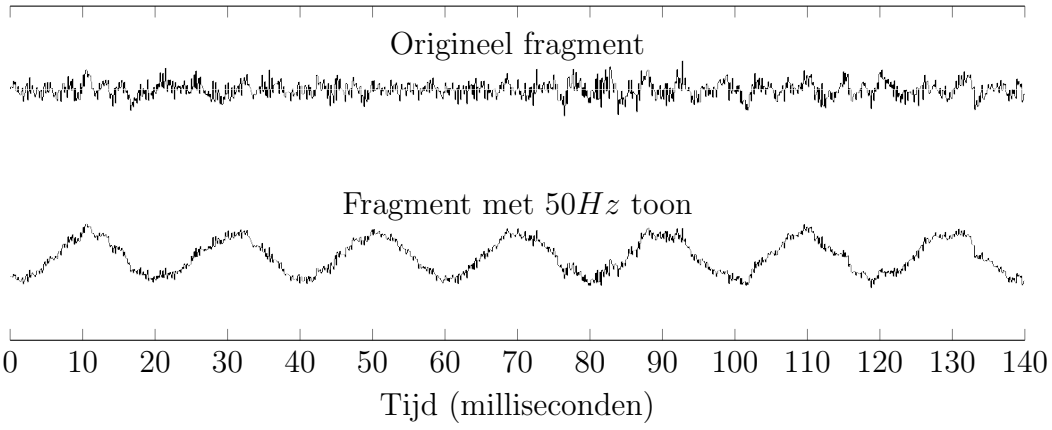
In de originele Panako implementatie van het algoritme werd geen rekening gehouden met het feit dat het acoustic fingerprinting algoritme het de werkelijke latency kan overschatten of onderschatten. De latency bepaald door het kruiscovariantie algoritme werd telkens opgeteld bij de ruwe latency bepaald door het acoustic fingerprinting algoritme. In 50% van de gevallen leidde dit probleem tot foutieve resultaten.

Meerdere malen uitvoeren van het algoritme

In tegenstelling tot acoustic fingerprinting is het bepalen van de latency met kruiscovariantie veel meer foutgevoelig. Bij opnames van slechte kwaliteit worden vaak foute resultaten teruggegeven.

Hoewel bij acoustic fingerprinting mogelijk storende factoren (een zoemende toon, geruis,...) zichtbaar zijn in het spectrogram zullen er naar alle waarschijnlijkheid nog steeds voldoende fingerprints gegenereerd worden op basis van geluiden die wel in beide opnames voorkomen.

Figuur 3.8: Twee gelijke audiofragmenten. Aan het tweede audiofragment is een achtergrondtoon van $50Hz$ toegevoegd.



Dergelijke storende elementen hebben veel meer invloed op het kruiscovariantie algoritme. Een ongewenste zoemende bastoon zorgt voor een grote verandering van de geluidsgolf van het audiofragment. Aangezien de kruiscovariantie rechtstreeks tussen twee geluidsgolven berekend wordt kan dit serieuze gevolgen hebben. In figuur 3.8 wordt dit probleem visueel verduidelijkt.

De invloed van storende factoren zoals zoemende tonen en ruis is gemakkelijk te beperken. Aangezien het kruiscovariantie algoritme over een zéér klein deeltje van het audiofragment wordt uitgevoerd is het zeer eenvoudig om het algoritme tientallen keren, maar telkens op een andere plaats, uit te voeren. Na elke iteratie wordt de latency in het geheugen opgeslagen. Ten slotte wordt er gezocht naar de latency die het meeste voorkomt. Die latency wordt gebruikt om het resultaat van het acoustic fingerprinting algoritme te verfijnen.

3.3.3 Parameters en hun invloed op het algoritme

De parameters van het kruiscovariantie algoritme zijn net zoals die van het acoustic fingerprinting algoritme instelbaar via het configuratiebestand.

`NFFT_BUFFER_SIZE`

Hoewel het FFT algoritme eigenlijk niets te maken heeft met het berekenen van de kruiscovariantie speelt deze parameter bij dit algoritme toch een belangrijk rol. Deze parameter bepaalt namelijk de grootte van de buffers waartussen de kruiscovariantie berekent wordt. Om een goede werking van het kruiscovariantie algoritme te verzekeren is het een vereiste dat de gebruikte buffers groter zijn dan de minimale nauwkeurigheid van het acoustic fingerprinting algoritme (anders is het mogelijk dat na het knippen de audiofragmenten er geen gelijkenissen te vinden zijn tussen beide buffers). Aangezien deze nauwkeurigheid bepaald wordt de `NFFT_STEP_SIZE` parameter en de `NFFT_BUFFER_SIZE` hier altijd een veelvoud van is, is het gemakkelijk om deze parameter ook voor het kruiscovariantie algoritme te gebruiken.

`CROSS_COVARIANCE_NUMBER_OF_TESTS`

Deze parameter bepaalt het aantal keren dat het kruiscovariantie algoritme per slice (zie 2.2) zal worden uitgevoerd. De meest voorkomende latency wordt als resultaat teruggegeven.

`CROSS_COVARIANCE_THRESHOLD`

Deze parameter bepaalt het minimum aantal keer dat een latency moet voorkomen om als geldig beschouwd te mogen worden.

3.3.4 Optimale instellingen

`NFFT_BUFFER_SIZE`

De optimale waarde van deze parameter werd in sectie 3.2.3 besproken.

`CROSS_COVARIANCE_NUMBER_OF_TESTS`

Bij audiofragmenten van goede kwaliteit is het niet noodzakelijk om deze parameter een hoge waarde te geven. Als het algoritme 5 à 10 keer wordt uitgevoerd zal de correcte latency hoogst waarschijnlijk al verschillende malen gevonden zijn.

Bij audiofragmenten van slechte kwaliteit is het van belang om deze parameter zo hoog mogelijk in te stellen. De praktijk heeft uitgewezen waardes vanaf 30 zorgt voor correcte resultaten.

De praktische bovengrens van deze parameter (N_B) is het aantal keren dat een kruiscovariantie buffer (bepaald door `NFFT_BUFFER_SIZE`) past in een streambuffer (zie 2.2) waarin de streams worden ingelezen (bepaald door `SLICE_SIZE_S`). Dit kan worden berekend met volgende formule:

$$N_B = \frac{\texttt{SLICE\_SIZE\_S}}{\texttt{NFFT\_BUFFER\_SIZE}/\texttt{SAMPLE\_RATE}} \tag{3.1}$$

De parameter kan hoger worden ingesteld maar dan zullen opeenvolgende kruiscovariantie buffers overlappen, wat vrij nutteloos is.

`CROSS_COVARIANCE_THRESHOLD`

Het hoog instellen van deze waarde heeft enkel nut als het van belang is dat het kruiscovariantie algoritme zeker een correct resultaat teruggeeft. In de huidige implementatie wordt het ruwe resultaat gebruikt als het aantal gelijke waarden niet boven deze drempel komt. Daarom wordt deze waarde meestal op 1 ingesteld. De best mogelijke verfijning van het resultaat wordt dan toegepast, ongeacht het aantal keer dat het resultaat voorkomt.

3.4 Filteren van de resultaten

Het is moeilijk om met zekerheid te bepalen of een bepaalde latency correct is. Toch kan er een soort van statistische optimalisatie worden uitgevoerd op de opeenvolgende latencies. De kans is namelijk klein dat de werkelijke latency verandert en na een korte tijd terugkeert naar de originele waarde.[2] Ook is de kans klein dat een foute latency toevallig verschillende malen na elkaar gedetecteerd wordt.

Deze eigenschappen laten toe om toch een bepaalde vorm van "foutherstelling" te implementeren. Fouten kunnen namelijk weggefilterd worden door eventuele pieken in opeenvolgende latencies af te vlakken.

3.4.1 Werking

Het afvlakken van pieken kan in realtime geïmplementeerd worden door gebruik te maken van verschillende *sliding windows*. Per audiostream wordt een wachtrij bijgehouden met daarin een aantal opeenvolgende latencies. Wanneer een wachtrij haar maximumcapaciteit heeft bereikt gaat het toevoegen van een nieuwe latency gepaard met het verwijderen van de oudste latency.

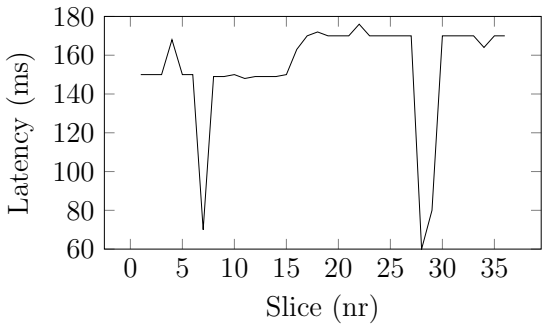
[2] Theoretisch is een dergelijke situatie toch mogelijk: In dit voorbeeld wordt de latency bepaalt tussen de audiostreams ($8000Hz$ samplefrequentie) van twee Teensy microcontrollers. Wanneer er 50 samples gedropt worden van de referentie audiostream zorgt dit voor een latencyverhoging van de andere audiostream van $6ms$. Indien er enkele seconden later 50 samples gedropt worden van de andere audiostream dan leidt dit tot een vermindering van de latency van $6ms$. Visueel is dit een piek in de opeenvolgende latencies.

Het afvlakken van pieken wordt verwezenlijkt door in plaats van de meest recente latency het gemiddelde of de mediaan van de buffer te gebruiken. De mate waarin pieken worden afgevlakt hangt af van de grootte van de buffer en de gebruikte methode (gemiddelde of mediaan).

3.4.2 Voorbeelden

In het ongefilterde verloop van 32 latencies (zie figuur 3.9) doet er zich één blijvende correcte verhoging van de latency voor. De twee grote en drie kleine pieken zijn (waarschijnlijk) foutief.

Figuur 3.9: Grafische weergave van het ongefilterde verloop van de latency.

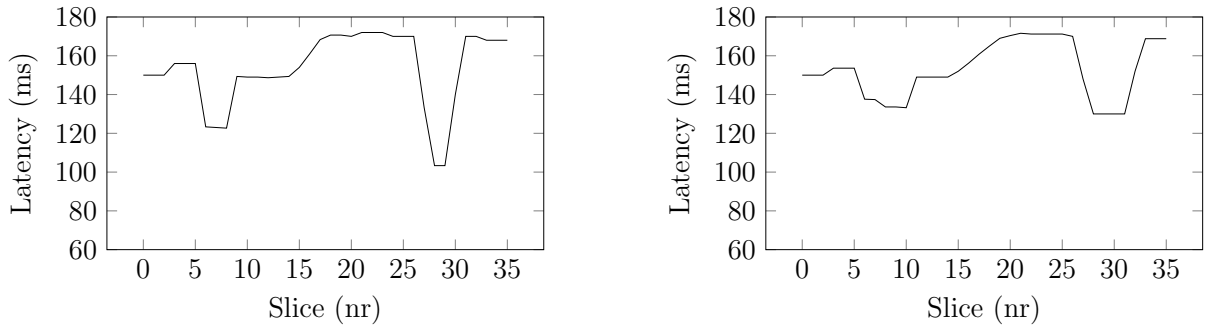


De filter moet foute pieken wegfilteren en elke juiste wijziging intact laten.

Moving average filter

Deze methode berekent de gemiddelde latency van de wachtrij. Figuur 3.10 toont het resultaat voor een wachtrij met maximumgrootte 3 en 5.

De kleinste pieken zijn weggefilterd maar de grootste pieken zijn nog steeds zichtbaar. De oorzaak hiervan is dat deze pieken door hun grootte toch sterk doorwegen op het gemiddelde. Om er voor te zorgen dat grote pieken nog meer worden weggefilterd zou de grootte van de wachtrij nog moeten worden verhoogd. In 3.4.4 zal duidelijk worden waarom dit geen goed idee is.



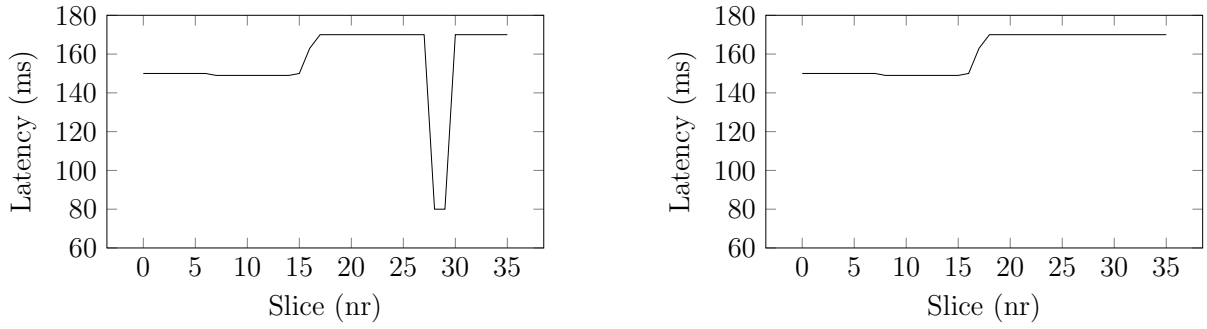
(a) Buffergrootte 3 **(b)** Buffergrootte 5

Figuur 3.10: Het verloop van de latencies na het toepassen van een moving average filter.

In de gefilterde resultaten valt ook op dat de hellingshoek van de latencyverhoging vermindert naarmate een grotere buffer gehanteerd wordt. Dit is geen goede eigenschap: het duurt namelijk veel langer tot er na een wijziging terug een stabiele latency bereikt wordt.

Moving median filter

De filter berekent de mediaan van elke wachtrij. Figuur 3.11 toont het resultaat voor een wachtrij met maximumgrootte 3 en 5.



(a) Buffergrootte 3 **(b)** Buffergrootte 5

Figuur 3.11: Het verloop van de latencies na het toepassen van een moving median filter.

De pieken veroorzaakt door 1 (eventueel) foute latency worden bij de moving median filter onmiddellijk weggefilterd. In de grafiek is er geen enkel spoor meer van te vinden. De piek veroorzaakt door 2 (eventueel) foute latencies wordt enkel volledig weggefilterd wanneer

buffergrootte 5 gehanteerd wordt.

De wijziging van de latency wordt niet afgevlakt zoals bij de moving average filter. De hellingshoek blijft onaangetast.

3.4.3 Parameters

`LATENCY_FILTER_TYPE`

Deze parameter bepaalt welke soort filter zal worden toegepast op de binnenkomende latencies. Mogelijke waarden zijn: `average`, `median` en `none`.

`LATENCY_FILTER_BUFFER_SIZE`

Dit is de grootte van de buffer waarin de meest recente latencies zullen worden opgeslagen.

3.4.4 Gevolgen

Het filteren van de latency heeft een negatief effect op de snelheid waarmee wijzigingen van de latency gedetecteerd kunnen worden. Zowel de grootte van wachtrij als de slicegrootte heeft invloed op de snelheid waarmee nieuwe latencies gedetecteerd worden. De grootte van de streambuffers (parameter `SLICE_SIZE_S`) bepaalt namelijk met welk interval nieuwe latencies binnenkomen.

Een verhoging van de latency zal bij de moving average filter een onmiddellijke maar beperkte invloed hebben op het gefilterde resultaat. De tijd die nodig is om tot een stabiel resultaat te komen kan berekend worden met volgende formule:

$$(\texttt{SLICE\_SIZE\_S}) \cdot (\texttt{LATENCY\_FILTER\_BUFFER\_SIZE}) \tag{3.2}$$

Bij de moving median filter wordt een wijziging van de latency gedetecteerd wanneer meer dan de helft van de buffer de gewijzigde latency bevat. De tijd tot wanneer een detectie plaatsvindt kan met volgende formule berekend worden:

$$\frac{(\texttt{SLICE\_SIZE\_S}) \cdot (\texttt{LATENCY\_FILTER\_BUFFER\_SIZE})}{2} \tag{3.3}$$

3.5 Ontwerp van de softwarebibliotheek

De Java bibliotheek voor het synchroniseren van streams bestaat uit verschillende onderdelen onderverdeeld in `packages`.

`be.signalsync.stream`

Deze package bevat klassen die verantwoordelijk zijn voor het abstract voorstellen en verwerken van streams. Streams kunnen namelijk via verschillende kanalen worden ingelezen. Met behulp van deze klassen wordt een abstractere verwerking mogelijk.

`be.signalsync.slicer`

De klassen uit deze package zorgen voor het bufferen van de binnenkomende streams (zie sectie 2.2) zodat het bepalen van de latency in realtime mogelijk wordt.

`be.signalsync.syncstrategy`

Deze package bevat de implementatie van de synchronisatiealgoritmen. Deze algoritmen worden aangeroepen van uit het `be.signalsync.sync` package. Welk algoritme precies gebruikt wordt kan worden ingesteld via het configuratiebestand.

`be.signalsync.datafilters`

Deze package bevat de implementaties van de verschillende soorten filters besproken in sectie 3.4.

`be.signalsync.sync`

Deze package roept klassen uit het `be.signalsync.slicer` aan om binnenkomende streams op te splitsen in slices. Vervolgens worden de latency-detecterende algoritmes gebruikt om de latency tussen de opeenvolgende buffers te bepalen. Met behulp van het package `be.signalsync.datafilters` worden eventuele pieken uit de resulterende latencies weggefilterd.

`be.signalsync.msp`

Deze package bevat de implementatie van enkele Max/MSP modules. Met behulp van deze modules is het mogelijk om de streams van microcontrollers in te lezen en de data ervan te synchroniseren.

De complete bespreking van het ontwerp is vrij uitgebreid. Enkel het ontwerp van de verschillende soorten streams zal behandeld worden. De bespreking van het volledige ontwerp van de softwarebibliotheek bevindt zich in bijlage B.

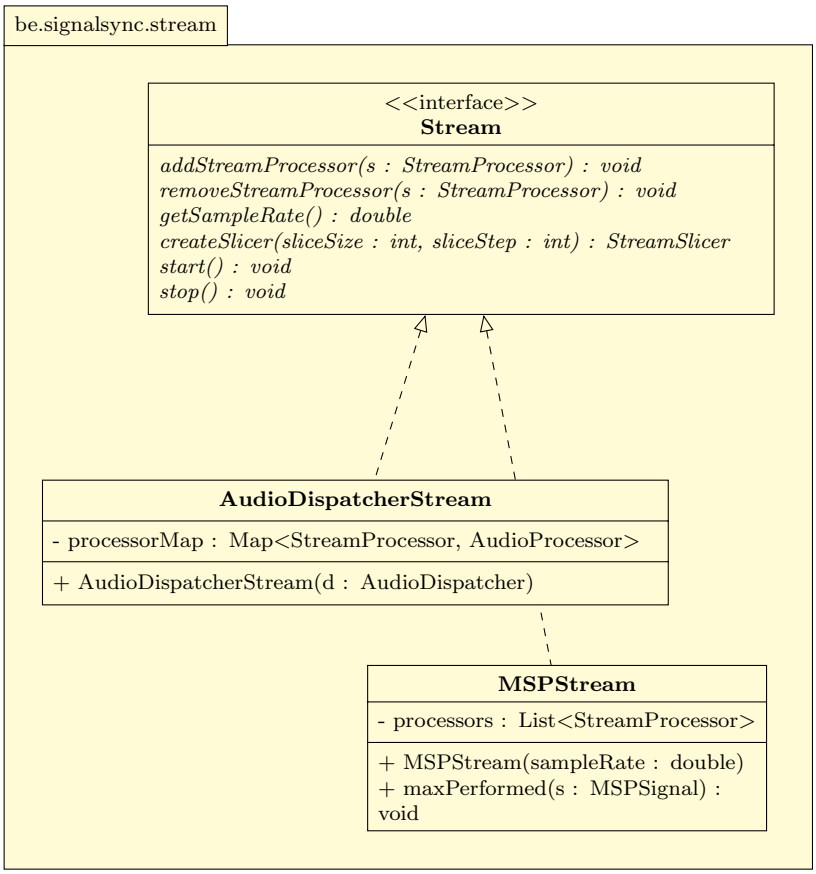
3.5.1 Streams

Streams kunnen van verschillende bronnen afkomstig zijn: een microfoon, een bestand, een virtuele Max/MSP patchkabel,... Streams afkomstig van een microfoon of bestand kunnen met behulp van de TarsosDSP klasse `AudioDispatcher` worden ingelezen (besproken in sectie 3.1.3). Deze klasse laat op eenvoudige wijze verdere verwerking toe. Het inlezen van data uit Max/MSP gebeurt op een fundamenteel andere manier die niet door de klasse `AudioDispatcher` ondersteund wordt. Daarom is er in dit onderzoek nog een extra abstractielaag ontwikkeld waarmee het mogelijk wordt om zowel `AudioDispatchers` als Max/MSP streams op dezelfde manier aan te spreken en te verwerken. Figuur 3.12 toont de `Stream` interface en de klassen die deze interface implementeren. De methodes van de interface worden in de concrete klassen niet herhaald maar zijn uiteraard wel aanwezig.

AudioDispatcherStream

De klasse `AudioDispatcherStream` is een typisch voorbeeld van het *Adapter* ontwerppatroon [24]. Een bestaande klasse (`AudioDispatcher`) wordt namelijk verbonden met een nieuwe interface (`Stream`) met behulp van een adapter of *wrapper* (`AudioDispatcherStream`). In de adapterklasse wordt er geen echte logica toegevoegd. De enige logica die de klasse bevat heeft als doel het mappen van de methode's van de `Stream` interface naar de klasse `AudioDispatcher`. Het attribuut `processormap` is bijvoorbeeld een `HashMap` die gebruikt wordt voor het mappen van `StreamProcessors` (eigen implementatie) naar AudioProces-

Figuur 3.12: UML diagram: de `Stream` interface en haar implementaties.



sors (TarsosDSP implementatie). De werking van `StreamProcessors` wordt verderop in sectie 3.5.1 besproken.

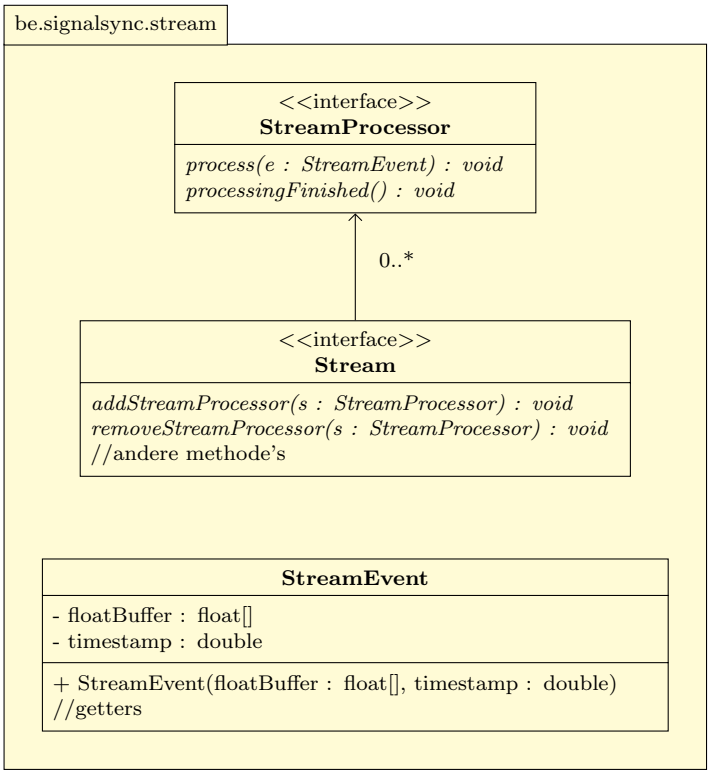
MSPStream

Een `MSPStream` is geen typische implementatie meer van het adapter pattern. Deze klasse voorziet namelijk een extra methode: `maxPerformed` met als parameter een `MSPSignal`. Dit is een belangrijk object in Max/MSP modules geschreven in Java. Een `MSPSignal` bevat een buffer van samples die via een virtuele Max/MSP patchkabel binnenkomen of buitengaan. Elke keer wanneer een Max/MSP module zo'n object binnenkrijgt moet het worden gedelegeerd aan de corresponderende `MSPStream`. Dit delegeren gebeurt met behulp van de `MaxPerformed` methode. Als alle `MSPSignal` objecten correct gedelegeerd worden dan kan het `Stream` object op exact dezelfde manier verwerkt worden als een stream afkomstig

van een bestand of microfoon.

Het verwerken van streams

Figuur 3.13: UML diagram van de klassen en interfaces die een rol spelen bij het verwerken van streams.

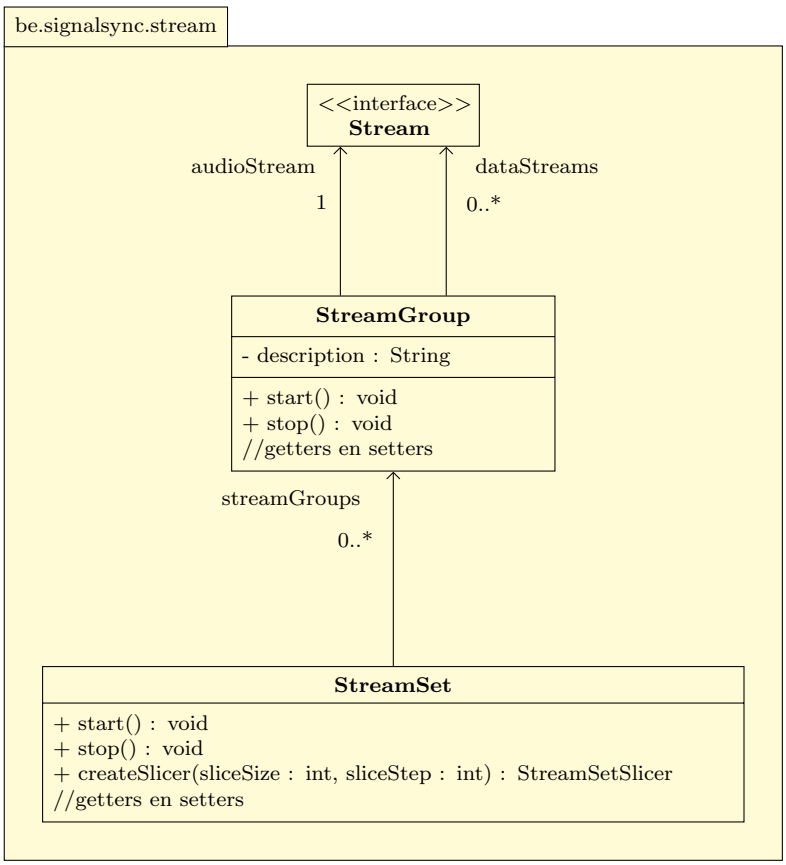


Streams kunnen door één of meerdere klassen worden verwerkt door de methode's van de interface `StreamProcessor` te implementeren. Na het inlezen van een bepaalde hoeveelheid data wordt `process` opgeroepen met als parameter een `StreamEvent` object. Dit object bevat metadata en een buffer van samplewaarden. Figuur 3.13 toont een UML diagram van de klassen die met dit proces te maken hebben.

Het bijhouden van streams

In dit onderzoek is er altijd een onderscheid gemaakt tussen datastreams en audiostreams. Datastreams zijn streams afkomstig van sensoren of videocamera's, audiostreams zijn de

Figuur 3.14: UML klassendiagram waarop wordt verduidelijkt hoe de verschillende streams worden bijgehouden alvorens ze gesynchroniseerd kunnen worden.



opnames van het omgevingsgeluid waaraan de datastreams "gekoppeld" zijn. Puur softwarematig wordt er echter geen verschil gemaakt tussen deze verschillende soorten streams. Of een digitaal signaal nu afkomstig is van een microfoon of van een sensor, in beide gevallen is het niet meer dan een opeenvolging van samples.

Om de synchronisatiealgoritmes correct aan te roepen moet er toch een onderscheid gemaakt worden tussen de audiostreams en datastreams. Daarom is de klasse `StreamGroup` in het leven geroepen. Deze klasse bevat één audiostream en nul of meer gekoppelde datastreams. Meestal zijn dit streams die vanuit dezelfde microcontroller zijn ingelezen (zie sectie 1.1).

Een verzameling van `StreamGroups` wordt een `StreamSet` genoemd. Een `StreamSet` is een soort van wrapper voor een lijst van `StreamGroups`. Om verschillende `StreamGroups` met

elkaar te synchroniseren is het noodzakelijk dat er een `StreamSet` object van wordt aangemaakt. Vervolgens kan het worden doorgegeven aan de `RealtimeSignalSync` klasse, dit is de klasse die verantwoordelijk is voor het bepalen van de latency tussen de verschillende audiostreams.

Figuur 3.14 toont een klassendiagram van hoe de streams intern worden bijgehouden.

3.6 Max/MSP modules

In de introductie van deze thesis (sectie 1.5) is er geschreven dat er voor het volledige synchronisatieproces een gebruiksvriendelijke interface ontwikkeld zal worden. Deze interface moet het voor musicologen/onderzoekers mogelijk maken om `StreamSets` aan te maken en te synchroniseren zonder het schrijven van één lijn code.

Een heel flexibele manier om dit te doen is door het schrijven van één of meerdere Max/MSP modules. Hierbij wordt de gebruiker niet in een hokje geduwd van wat een stream precies moet zijn of hoe het moet worden ingelezen. Bij deze benadering is het een garantie dat elk Max/MSP signaal vergezeld van een synchrone audiostream gesynchroniseerd kan worden. Max/MSP ondersteunt standaard het inlezen van bestanden en microfoons.

Problematisch is het feit dat de streams afkomstig van Teensy microcontrollers standaard niet kunnen worden aangesproken vanuit een Max/MSP context. Aangezien dit voor dit onderzoek een vereiste werd hiervoor een module geschreven.

3.6.1 Inlezen van de Teensy microcontroller

De klasse `TeensyReader` bevat de implementatie van een Max/MSP module waarmee signalen afkomstig van Teensy microcontrollers kunnen worden ingelezen. Het aanmaken van het object kan met volgende code:

```
mxj~ be.signalsync.msp.TeensyReader <<parameters>>.
```

De module verwacht één optionele en vier verplichte parameters:

Poort

De COM-poort waarmee de Teensy microcontroller communiceert met de computer. Wanneer de naam van de poort bekend is kan deze hier letterlijk worden opgegeven. Het is ook mogelijk om een index (vanaf 0) op te geven. Indien er 3 microcontrollers zijn aangesloten kan er 0, 1 of 2 worden opgegeven. Deze parameter kan ook worden weggelaten. In dat geval wordt automatisch de Teensy met index 0 gekozen.

Samplefrequentie

Dit is de samplefrequentie in Hertz waaraan de Teensy microcontroller met de computer communiceert. Alle binnenkomende signalen worden geresampled naar de samplefrequentie van Max/MSP. De meest courante waarden zijn 8000 of 11025.

Het is aangeraden (maar geen vereiste) om de Max/MSP samplefrequentie in te stellen op een veelvoud van de Teensy samplefrequentie. Dit vereenvoudigt het resamplingproces en heeft een positieve invloed op de geluidskwaliteit.

Startkanaal

De index van de eerste analoge pin waarvan het signaal moet worden ingelezen. Indien het eerste in te lezen signaal zich op pin A3 bevindt, dan moet deze parameter worden ingesteld met waarde 3.

Audiokanaal

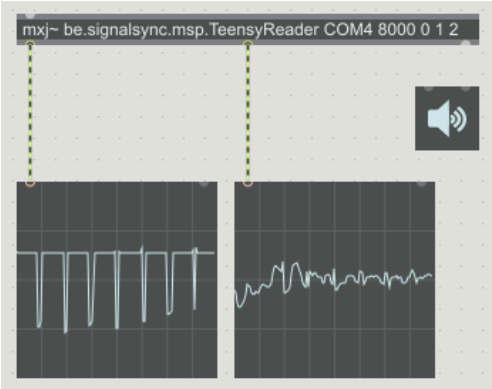
De index van het audiokanaal dat gebruikt zal worden voor de synchronisatie. De index begint te tellen vanaf het startkanaal. Indien er gestart wordt op pin A3 en het audiosignaal zich op pin A5 bevindt, dan zal deze parameter waarde 2 moeten krijgen. Het signaal van dit kanaal ondergaat na het inlezen een extra filtering waarbij het geluid geoptimaliseerd wordt. Bij het invullen van een ongeldig getal (te hoog of negatief) wordt de filtering op geen enkel kanaal toegepast.

Aantal kanalen

Deze parameter stelt het aantal in te lezen kanalen in.

Na het valideren van de parameters wordt de module aangemaakt. Het aantal uitgaande verbindingen van deze module wordt bepaald door de laatste parameter.

Figuur 3.15: Screenshot van de `TeensyReader` in Max/MSP aangesloten op een infraroodsensor en microfoon.



Figuur 3.15 toont de `TeensyReader` waarbij de Teensy samplet aan een frequentie van $8000Hz$. De microfoon is aangesloten op pin A1, de infraroodsensor is aangesloten op pin A0. De links scope module toont het signaal van de infraroodsensor, de rechtse module toont de geluidsgolf.

Implementatie

Een Max/MSP module kan in Java geschreven worden door een klasse te laten overerven van `MSPPerformer`. Bij de constructie worden de *outlets* (de uitgaande verbindingen) geïnitialiseerd en wordt er een `TeensyDAQ` object aangemaakt. Dit is een object uit afkomstig uit de `TeensyDAQ` bibliotheek ontwikkeld aan het IPEM. Deze bibliotheek biedt een low-level interface naar de Teensy microcontroller aan. Na het aanmaken van dit object registreert de module zich als *handler*. Om dit mogelijk te maken wordt de methode `handle` van de interface `DAQDataHandler` geïmplementeerd. Na het registreren wordt deze methode consequent opgeroepen met samples afkomstig van de Teensy microcontroller. Deze samples worden vervolgens gebufferd.

Een klasse die overerft van `MSPPerformer` moet de methode `perform` implementeren. In deze methode worden de samples uit de buffers gehaald, geresampled naar de Max/MSP samplefrequentie en verstuurd naar de outlets.

3.6.2 De synchronisatiemodule

De tweede module zal de synchronisatie van de streams verzorgen en is geïmplementeerd in de klasse `Sync`. De module maakt gebruik van `RealtimeSignalSync` om de latencies tussen de audiostreams te bepalen. De module kan in Max/MSP met volgende code worden aangemaakt: `mxj~ be.signalsync.msp.Sync <<parameter>>`

Instellen van de stream structuur

De module verwacht één parameter van het type `String` die beschrijft hoe de streams gestructureerd zijn. De structuur wordt voorgesteld als een door komma's gescheiden reeks van de letters '`a`' en '`d`'. Elk door komma's gesplitst deel bepaald de structuur van een `StreamGroup` waarin de letters '`a`' en '`d`' respectievelijk voor audiostream en datastream staan. Per deel mag maar eenmaal de letter '`a`' voorkomen.

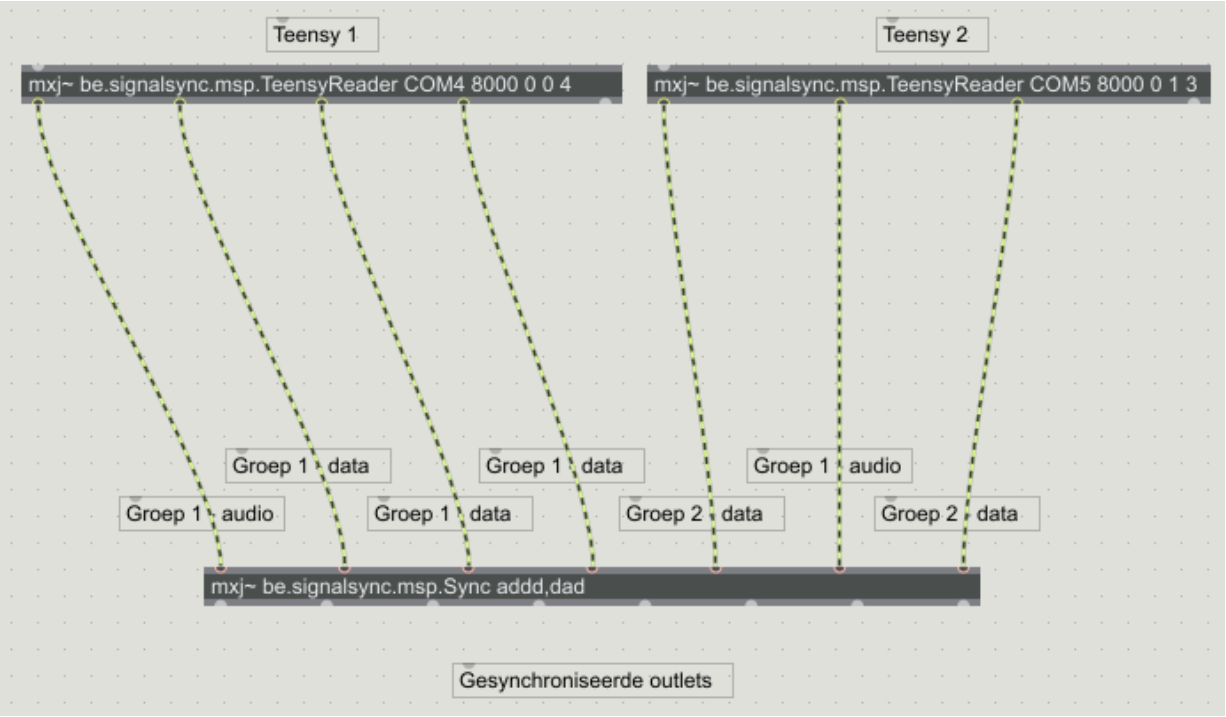
Een mogelijke waarde voor de parameter is `addd,dad`. De eerste `StreamGroup` bestaat uit 4 streams waarbij de synchronisatie met de eerste stream zal worden uitgevoerd. De tweede `StreamGroup` bestaat uit 3 streams met de tweede stream als audiostream. De volledige tekenreeks (de verzameling van alle `StreamGroups`) stelt de structuur van een `StreamSet` voor onderverdeeld in `StreamGroups`.

Figuur 3.16 toont hoe deze module gebruikt moet worden in combinatie met de `TeensyReader`. De audiostream *outlets* (bepaald door de vierde parameter van de twee `TeensyReaders`) worden verbonden met de audiostream *inlets* van de `Sync` module (bepaald door de letters '`a`' van de parameter).

Implementatie

Net zoals `TeensyReader` erft deze module over van `MSPPerformer`. Ook wordt de interface `SyncEventListener` geïmplementeerd. In de constructor wordt op basis van de parameter van de module een `StreamSet` van `MSPStream` objecten aangemaakt. In de perform methode van de module worden de signaalvectoren doorgegeven aan de `MSPStream` objecten.

Figuur 3.16: De synchronisatie van streams in Max/MSP afkomstig van twee Teensy microcontrollers.



De synchronisatie wordt verwezenlijkt door in de constructor een `RealtimeSignalSync` object aan te maken waarop de klasse zich registreert. Aan het `RealtimeSignalSync` object wordt de `StreamSet` meegegeven. Elke keer wanneer er informatie over de latency beschikbaar is wordt de methode `onSyncEvent` opgeroepen met alle latencies.

Met behulp van deze latencies en de methode beschreven in 2.3 worden de Max/MSP streams gesynchroniseerd.

3.6.3 Andere modules

Aangezien Max/MSP al een groot aantal modules bevat was het niet nodig om bepaalde features in de eigen modules te implementeren. Enkele modules die zeer bruikbaar zijn voor deze toepassing:

`sfplay~`

Met deze module is het mogelijk om streams van verschillende bestandstypen in

te lezen. Dit kan handig zijn om de synchronisatie van een experiment toch als naverwerking uit te voeren.

`sfrecord~`

Deze module laat toe om een Max/MSP signaal weg te schrijven naar een bestand. De gesynchroniseerde streams kunnen met behulp van deze module op schijf worden opgeslagen.

`scope~`

Deze module toont de golfvorm van een Max/MSP signaal. Dit is handig tijdens het uitvoeren van een experiment om de signalen te controleren.

Hoofdstuk 4

Evaluatie

Om de kwaliteit van de softwarebibliotheek te kunnen garanderen zijn er verschillende soorten testen uitgevoerd.

De eerste soort testen zijn geschreven voor het analyseren van de kwaliteit van de algoritmes. De algoritmes worden hierbij blootgesteld aan audiofragmenten waartussen de latency bepaald moet worden. Door te kijken naar het foutenpercentage en de performantie zijn de optimale parameterwaarden voor de geteste situatie bepaald. In dit hoofdstuk zullen enkele illustratieve testen omschreven worden.

Buiten de testen voor de algoritmes zijn er ook enkele unit testen geschreven voor het testen van enkele cruciale elementen van de softwarebibliotheek.

De performantie wordt uitgedrukt in het aantal keer dat het algoritme sneller is dan realtime. Een algoritme met een performantie van $2 \times$ realtime verwerkt n seconden audio in $n/2$ seconden. De testen zijn uitgevoerd op een *Intel Core i7-4510 2.0 GHz* processor. Het is aangeraden om de performantie resultaten relatief te bekijken. Deze zijn namelijke erg afhankelijk van de processor waarop de testen zijn uitgevoerd.

Het is vereist dat de performantie boven $1 \times$ realtime blijft. Bij een lagere performantie komt de audio sneller binnen dan de verwerkingssnelheid van de algoritmen. Het bepalen van de latency zal dan vertraging oplopen wat kan leiden tot slechte resultaten.

4.1 Testen van de algoritmes

Het testen van de algoritmes werd uitgevoerd door de JUnit testcase `SynchronizationTest` uit de package `be.signalsync.test`. Deze testcase laat toe om de slices van verschillende audiofragmenten met elkaar te matchen en te analyseren waar de algoritmes precies in de fout gaan. De volgende tabel toont de instellingen die in alle testen van toepassing zijn:

Instelling	**Waarde**
`MIN_ALIGNED_MATCHES`	2
`CROSS_COVARIANCE_THRESHOLD`	1
Fingerprinting nauwkeurigheid	$32ms$
Kruiscovariantie nauwkeurigheid	$0.1ms$

4.1.1 Aanmaken van de dataset

Bij het uitvoeren van deze testen is het de bedoeling om enkel de algoritmes te testen. Om niet afhankelijk te zijn van andere softwareonderdelen wordt de dataset op voorhand aangemaakt. Het aanmaken van deze dataset gebeurt in twee stappen. Eerst wordt het originele audiofragment gewijzigd door er latency en eventueel een ander geluid aan toe te voegen. Dit gebeurt met behulp van een Perl script. Vervolgens worden de verschillende audiofragmenten in slices geknipt en opgeslagen. In de testcase worden de algoritmes rechtstreeks op deze slices uitgevoerd zonder andere softwareonderdelen aan te roepen.

4.1.2 Toevoegen van latency

In het meest eenvoudige scenario wordt de latency berekend tussen twee identieke audiofragmenten. Eén audiofragment is hierbij bewerkt door het toevoegen van stilte of het wegknippen van een stukje audio. Aangezien de audiofragmenten buiten deze wijziging identiek zijn zouden de algoritmes in theorie geen enkele fout mogen maken.

Voor de test worden 12 varianten voorzien van een audiofragment. Het originele audiofragment is de referentie. Er zijn zowel varianten met een positieve als met een negatieve

latency voorzien.

Dit is de verzameling van de geteste latencies:

$$\{20ms,\ -20ms,\ 80ms,\ -80ms,\ 90ms,\ -90ms,\ 300ms,$$
$$-300ms,\ 2000ms,\ -2000ms,\ 6000ms,\ -6000ms\ \}$$

Van elk audiofragment (referentie en elke variant) worden (ongeveer) 55 slices aangemaakt (lengte: 10s, overlap: 5s). Elke slice wordt gematcht met de corresponderende slice van het referentie audiofragment.

Acoustic fingerprinting

Bij het testen van het acoustic fingerprinting algoritme werden de volgende parameters gehanteerd:

Instelling	**Waarde**
`NFFT_EVENT_POINT_MIN_DISTANCE`	100
`NFFT_MAX_FINGERPRINTS_PER_EVENT_POINT`	10

Dit waren de resultaten:

Resultaat	**Waarde**
Totaal aantal testen	324
Totaal aantal geslaagd	324
Slaagpercentage	100%
Uitvoeringstijd	55 × realtime

Kruiscovariantie

De hierboven vermelde parameters blijven gelijk. `CROSS_COVARIANCE_NUMBER_OF_TESTS` wordt wel gewijzigd. Het eenmalig uitvoeren van het algoritme resulteerde in volgende resultaten:

Resultaat	Waarde
Totaal aantal testen	324
Totaal geslaagd	306
Slaagpercentage	94%
Uitvoeringstijd	53 × realtime

Bij het 5 maal uitvoeren zijn dit de resultaten:

Resultaat	Waarde
Totaal aantal testen	324
Totaal geslaagd	324
Slaagpercentage	100%
Uitvoeringstijd	53 × realtime

Bespreking

Het acoustic fingerprinting heeft de verschillende testen foutloos doorstaan. Het kruiscovariantie algoritme maakte echter enkele fouten bij het eenmalig uitvoeren van het algoritme. Dit is logisch te verklaren. Het kruiscovariantie werd namelijk op een zeer klein stukje audio uitgevoerd. Bij het eenmalig uitvoeren kan het gebeuren dat één van de buffers toevallig enkel stilte bevat (alle samples hebben de waarde 0.0). In dat geval is de kruiscovariantie voor elke verschuiving 0. De latency kan dus niet bepaald worden. Aangezien het toch aangeraden is om het algoritme meerdere malen op verschillende plaatsen uit te voeren vormt dit niet echt een probleem (zie 3.3.2).

De performantie van de algoritmen is zeker aanvaardbaar. Het 5 maal uitvoeren van het kruiscovariantie algoritme heeft amper invloed op de uitvoeringstijd.

4.1.3 Toevoegen van een sinusgolf

In sectie 3.3.2 is geschreven dat het kruiscovariantie algoritme het moeilijk krijgt wanneer de te matchen geluidsgolven visueel erg van elkaar verschillen. Dit wordt gesimuleerd door

het toevoegen van een lage toon aan één van de audiofragmenten. Door het kruiscovariantie algoritme verschillende keren uit te voeren wordt de invloed van dit probleem beperkt.

Bij dit experiment wordt de eerder beschreven dataset uitgebreid. Van elke latency worden twee varianten voorzien. Aan elke variant wordt één van deze sinusgolven toegevoegd waardoor de golfvorm er helemaal anders gaat uitzien:

$$\{\ 50Hz,\ 100Hz\ \}$$

Van deze aangepaste audiofragmenten worden op voorhand opnieuw slices aangemaakt.

Acoustic fingerprinting

Bij het uitvoeren van het acoustic fingerprinting algoritme op de gemodificeerde dataset werden volgende parameters gehanteerd:

Instelling	**Waarde**
`NFFT_EVENT_POINT_MIN_DISTANCE`	100
`NFFT_MAX_FINGERPRINTS_PER_EVENT_POINT`	10

Dit waren de resultaten:

Resultaat	**Waarde**
Totaal aantal testen	648
Totaal aantal geslaagd	648
Slaagpercentage	100%
Uitvoeringstijd	55 × realtime

Kruiscovariantie

De hierboven vermelde parameters blijven gelijk. `CROSS_COVARIANCE_NUMBER_OF_TESTS` werd wel gewijzigd. Er zijn 3 testen uitgevoerd waarbij het aantal kruiscovariantie uitvoeringen werd ingesteld op respectievelijk 1, 5 en 20 maal. Dit zijn de resultaten:

Resultaat	Waarde
Totaal aantal testen	648
1 uitvoering	
Totaal aantal geslaagd	465
Slaagpercentage	72%
Uitvoeringstijd	55 × realtime
5 uitvoeringen	
Totaal aantal geslaagd	642
Slaagpercentage	99%
Uitvoeringstijd	53 × realtime
20 uitvoeringen	
Totaal aantal geslaagd	648
Slaagpercentage	100%
Uitvoeringstijd	52 × realtime

Bespreking

Het valt op dat het kruiscovariantie algoritme in moeilijkheden komt na het toevoegen van de sinusgolf. Het meerdere malen uitvoeren van het algoritmen vangt dit probleem goed op. Na 5 uitvoeringen zijn de meeste resultaten al correct. Wanneer dit aantal tot 20 wordt opgetrokken werkt het algoritme foutloos.

Het meerdere malen uitvoeren van het algoritme heeft een zeer beperkte invloed op de performantie.

4.1.4 Conclusie

Deze testen tonen aan dat de algoritmen in staat zijn om zelf gemodificeerde audiofragmenten te synchroniseren. De resultaten geven een beeld van hoe de algoritmes presteren onder welke parameters.

Een heel belangrijke vaststelling is dat de uitvoeringstijd relatief constant blijft ondanks

grote wijzigingen aan de parameters. De oorzaak hiervan is dat de parameters weinig invloed hebben op het asymptotische gedrag van de algoritmen. Het kruiscovariantie algoritme heeft als tijdscomplexiteit $O(n^2)$. Het wijzigen van het aantal uitvoeringen wijzigt niets aan n. Aangezien de grootte van n standaard heel beperkt is blijft het algoritme goed presteren.

De wijzigingen aan de parameters van het acoustic fingerprinting algoritme hebben ook geen grote invloed op de uitvoeringstijd. Het zoeken naar matchen van fingerprints gebeurt namelijk met behulp van hashes in een hashtabel. Aangezien zoeken in een hashtabel een $O(1)$ operatie is, is de tijdscomplexiteit lineair met het aantal fingerprints.

4.2 Praktijktest: bepalen van de latency

De ontwikkelde toepassing zal door het IPEM gebruikt worden tijdens experimenten om de datastreams afkomstig van sensoren te synchroniseren. Meestal worden de sensoren samen met een microfoon aangesloten op één of meerdere Teensy microcontrollers. Het is dus belangrijk om uit te testen dat het mogelijk is om de latency tot op één milliseconde nauwkeurig te bepalen bij een geluidsopname afkomstig van een Teensy.

4.2.1 Opstelling

Bij deze test zullen de algoritmes niet rechtstreeks worden aangeroepen. Het bepalen van de latency gebeurt met behulp van de reeds besproken Max/MSP modules. De werking van volgende componenten wordt hierdoor gecontroleerd:

- De synchronisatie algoritmes
- De slicers
- De `TeensyReader` Max/MSP module
- De `Sync` Max/MSP module

In deze test zal één audiostream worden ingelezen van een Teensy microcontroller. De andere audiostream is afkomstig van de standaard microfoon van een laptop. Beide opna-

mes zijn van relatief slechte kwaliteit. Daarom is dit een goede test om te bepalen of de volledige toepassing voldoende robuust is.

De gedetailleerde opstelling en de resultaten worden beschreven in bijlage C

4.2.2 Instellingen

Door de slechte geluidskwaliteit moeten de parameters extremer worden ingesteld. De volgende tabel bevat de gewijzigde instellingen.

Parameter	**Waarde**
Algoritme	Kruiscovariantie
`NFFT_EVENT_POINT_MIN_DISTANCE`	10
`NFFT_MAX_FINGERPRINTS_PER_EVENT_POINT`	50
`MIN_ALIGNED_MATCHES`	2
`CROSS_COVARIANCE_NUMBER_OF_TESTS`	30
`CROSS_COVARIANCE_THRESHOLD`	1

4.2.3 Uitvoering en resultaten

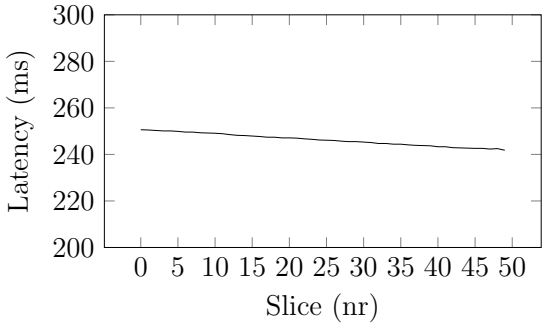
Tijdens het uitvoeren van de test worden enkele liedjes afgespeeld op een normaal volume. Tijdens de test wordt er ook gesproken. De test duurt ongeveer 4 minuten. De latencies worden niet gefilterd.

Grafiek 4.1 toont de latency van de Teensy ten opzichte van de laptopmicrofoon. Het valt op dat er drift is opgetreden. De oorzaak hiervan is niet verder onderzocht. De uitvoeringstijd is vastgesteld op $44 \times$ realtime.

4.2.4 Conclusie

De belangrijkste conclusie die kan worden getrokken is dat alles werkt. De latency wordt in de Max/MSP console geprint en vertoont geen afwijkende resultaten. Ook heeft de drift bewezen dat het mogelijk is om dit verschijnsel te detecteren.

Figuur 4.1: Grafische voorstelling van het verloop van de latency.



De uitvoeringstijd toont aan dat er nog veel marge is om de parameters eventueel nog extremer in te stellen bij het synchroniseren van audiofragmenten van nog slechtere kwaliteit.

4.3 Praktijktest: synchroniseren van streams

Met deze test is getracht te bepalen dat de streams daadwerkelijk correct gesynchroniseerd worden. De test is uitgevoerd in Max/MSP waarbij twee audiobestanden werden ingelezen. Aan de geluidsfragmenten zijn op willekeurige plaatsen korte stiltes toegevoegd. Elke stilte zorgt voor een wijziging van de latency tussen de geluidsfragmenten. Door de gesynchroniseerde streams weg te schrijven naar een wave-bestand kon het resultaat handmatig gecontroleerd worden. Ook kan de synchronisatie auditief geverifieerd worden door elke gesynchroniseerde stream langs één zijde van een hoofdtelefoon af te spelen.

4.3.1 Conclusie

Deze test heeft aangetoond dat de synchronisatie correct wordt uitgevoerd. De gesynchroniseerde streams kunnen bij bepaalde samplefrequenties in Max/MSP een afwijking van enkele milliseconden bevatten. Deze fout ontstaat wanneer de samplefrequentie niet mooi kan worden omgezet. In de gebruikershandleiding wordt er uitgelegd hoe hiermee moet worden omgegaan.

4.4 Testen van de softwarecomponenten

Buiten de synchronisatiealgoritmes bevat de geschreven software nog enkele componenten met vrij complexe logica. Om de kwaliteit van de software te garanderen zijn er enkele unittesten geschreven. Deze testen zijn met succes uitgevoerd.

4.4.1 Testen van de StreamSlicer

De klasse `StreamSlicerTest` bevat de vereiste testlogica. De test gebruikt een verzonnen stream waarvan handmatig de groottes van de slices zijn berekend. Deze worden vergeleken met de slices die de `StreamSlicer` teruggeeft. De samples van de stream zijn de afgeronde timestamps, hierdoor kan ook de inhoud van de slices getest worden.

4.4.2 Testen van de Datafilters

Deze testen bevinden zich in de klasse `DataFilterTest`. De test bevat een hardgecodeerde opeenvolging van latencies. Van deze latencies zijn handmatig de verwachte waarden na toepassing van verschillende soorten datafilters berekend. In de test worden de datafilters op de originele reeks latencies toegepast. De resultaten worden vergeleken met de handmatig berekende waarden.

Hoofdstuk 5

Conclusie

Het ontwikkelde systeem zal in dit hoofdstuk worden getoetst aan de doelen en criteria beschreven in hoofdstuk 1. Ook zullen enkele mogelijke toekomstige verbeteringen besproken worden.

5.1 Doelen

Het eerste doel was dat de algoritmes aangepast en/of geoptimaliseerd moeten worden zodat ze de latency tussen audiofragmenten opgenomen met een basic microfoon kunnen bepalen tot op minstens één milliseconde nauwkeurig. De praktijktest beschreven in 4.2 toont aan dat dit mogelijk is.

Het tweede doel is de implementatie van de softwarebibliotheek. Met behulp van deze bibliotheek moet het mogelijk zijn om programmatisch streams te kunnen aanmaken en synchroniseren. Het ontwerp van deze softwarebibliotheek is omschreven in 3.5. Dit toont aan dat het doel ook is verwezenlijkt.

Het derde doel is ook succesvol voltooid. De gebruikersinterface kan door musicologen zelf worden samengesteld in Max/MSP. In de praktijktest is aangetoond dat het mogelijk is om streams in te lezen en de latency tussen de streams te bepalen. Met behulp van deze latency kan de module de streams synchroniseren.

5.2 Beoordeling algoritmes

In 1.3 werden enkele meer specifieke criteria opgelegd waaraan de algoritmes moet voldoen.

Het eerste criterium stelt dat het mogelijk moet zijn om met een buffergrootte (de grootte van de slices) van maximaal 10 seconden de synchronisatie uit te voeren. De testen omschreven in 4.1 en 4.2 zijn uitgevoerd met de vooropgestelde buffergrootte en zijn geslaagd. Het huidige systeem voldoet dus aan dit criterium.

Het tweede criterium bepaalt de snelheid waarmee gedropte samples gedetecteerd kunnen worden. In de fase van het onderzoek waarin deze criteria zijn opgesteld was het nog onduidelijk welke factoren hier allemaal invloed op hebben. In sectie 2.2 is onder meer omschreven welke invloed de grootte en stapgrootte van de buffers hebben op de detectiesnelheid. Ook het eventueel filteren van de resultaten (beschreven in 3.4) heeft een invloed op de detectiesnelheid. De testen die in hoofdstuk 4 omschreven werden voldoen aan de vooropgestelde maximale detectiesnelheid van 10 seconden. Er werd namelijk een slicegrootte van 10 seconden en stapgrootte van 5 seconde gehanteerd. Deze maximale detectiesnelheid kan met deze instellingen niet worden bereikt indien er aan foutcorrectie wordt gedaan.

Volgens het laatste criterium moet het mogelijk moet zijn om drift te detecteren. De detectiesnelheid hiervan is gelijk aan de snelheid waarmee gedropte samples gedetecteerd kunnen worden. In de praktijktest trad dit verschijnsel op. Het is dus bewezen dat dit mogelijk is.

Het is duidelijk dat van de besproken algoritmen uit de introductie (sectie 1.4) acoustic fingerprinting kruiscovariantie het meest geschikt waren voor het onderzochte probleem. Dat het systeem zo goed presteert is grotendeels te danken aan de robuuste fundamenten waarop alles gebouwd is.

5.3 Mogelijke verbeteringen en uitbreiding

Het experiment uit de probleemschets maakte gebruik van een accelerometer, een videocamera en afgespeelde muziek. Met het huidige systeem is het mogelijk om de datastream van de accelerometer te synchroniseren met de afgespeelde muziek. Deze signalen kunnen namelijk verwerkt worden in Max/MSP.

Hoewel het geluid van de beelden als Max/MSP signaal verwerkt kan worden is dit niet het geval voor de bijhorende beelden. Een mogelijke uitbreiding van het huidige systeem omvat het ontwikkelen van een Jitter variant van de huidige synchronisatiemodule. Jitter is een uitbreiding van Max waarmee realtime video verwerkt kan worden. Ondanks dat er nog geen verder onderzoek gedaan is naar de haalbaarheid hiervan kan het ontwikkelen van een Jitter module eventueel een oplossing bieden voor het hierboven beschreven probleem.

Een ander soort data dat bij bepaalde experimenten gebruikt wordt is MIDI. In de huidige implementatie wordt dit nog niet ondersteund. Het ontwikkelen van een Max/MSP omzetmodule die MIDI gegevens van en naar Max/MSP signalen converteert zou dit probleem kunnen oplossen. Artikel [4] legt uit hoe MIDI naar een Max/MSP signaal gemapt kan worden.

5.4 Terugblik

Bij het vergelijken van het bereikte resultaat met de originele opdracht heb ik het gevoel dat ik de masterproef met succes volbracht heb. Ik ben er namelijk in geslaagd om het synchroniseren van datastreams zonder nabewerking mogelijk te maken. Ook heb ik een flexibele interface ontworpen in Max/MSP.

Het uitwerken van de eerste uitbreiding (het sleutelen aan de algoritmes) was soms noodzakelijk om de kwaliteit van het nieuwe systeem te garanderen.

Bijlagen

Bijlage A

Resultaten: DTW experiment

In dit experiment proberen we de nauwkeurigheid van het DTW algoritme te bepalen wanneer streams gebufferd worden. Hiertoe bepaalden we eerst de latency tussen twee audiofragmenten. Vervolgens verkleinden we iteratief de duur van het fragment met 10 seconden waarop we het algoritme opnieuw uitvoerden. Tenslotte vergeleken we de buffergrootte en nauwkeurigheid van de resultaten.

We hebben gebruik gemaakt van twee audiofragmenten waarbij het ene fragment 2.390 seconden vertraging heeft ten opzichte van het andere fragment. Beide fragmenten hebben samplefrequentie van $8000Hz$. Eén van de twee fragmenten is een opname van het origineel en bijgevolg van matige kwaliteit.

Het experiment is uitgevoerd in *Sonic Visualiser* met behulp van de *Match Performance Aligner* plug-in. Deze plug-in laat synchronisatie toe met behulp van het DTW algoritme. De implementatie wordt uitgebreider besproken in artikel [12]. Voor dit experiment hebben we de default instellingen gebruikt. De plug-in bepaalt elke twintig milliseconden de latency tussen beide fragmenten.

De volgende tabel geeft de resultaten van het experiment weer. De eerste kolom bevat de lengte van de vergeleken fragmenten in seconden. Deze lengte stelt de buffergrootte voor van een audiostream. De tweede kolom geeft aan hoeveel seconden van de stream moet worden verwerkt tot er een stabiel resultaat wordt bekomen. De derde kolom geeft het

gemitdoddelde weer van de gevonden latencies. Deze waarde wordt berekend vanaf dat het algoritme een stabiel resultaat heeft gevonden. De vierde kolom bevat de standaardafwijking van dit resultaat.

Lengte	**Tijd tot stabiel**	**Gemiddelde latency**	**Standaardafwijking**
60s	2.540s	2,393s	0.048s
50s	2.540s	2,390s	0.095s
40s	2.540s	2,394s	0.020s
30s	2.540s	2,384s	0.145s
20s	2.540s	2,390s	0.108s
10s	2.540s	2,395s	0.025s

Uit bovenstaande resultaten kunnen we verschillende zaken concluderen. Ten eerste zien we aan de standaardafwijking dat de individuele resultaten (die iedere 20ms gegenereerd worden) niet nauwkeurig genoeg zijn om te gebruiken in onze toepassing. De gemiddelde waarde komt wel in de buurt van de werkelijke latency maar is nog steeds niet zo nauwkeurig. Ook moeten we bij de berekening van het gemiddelde rekening houden met het feit dat het algoritme pas na een bepaalde tijd een stabiel resultaat vindt, in dit geval 2.540s.

We hebben dit algoritme ook uitgetest op een fragment waaruit 500 ms hebben weggeknipt om het probleem met gedropte samples te simuleren. Het algoritme reageerde hier zeer snel op: de nieuwe latency werd na 240 ms gevonden. Het probleem is dat we zojuist hebben getracht de nauwkeurigheid te verbeteren door het gemiddelde te nemen van de resultaten. Dit heeft als gevolg dat wanneer er samples gedropt zijn het eindresultaat zich bevindt tussen de initiële en nieuwe latency.

Bijlage B

Ontwerp van de softwarebibliotheek

In deze bijlage zal het ontwerp van de softwarebibliotheek besproken worden. Het ontwerp van de verschillende soorten streams zal niet worden herhaald aangezien deze materie al eerder behandeld is in sectie 3.5.1.

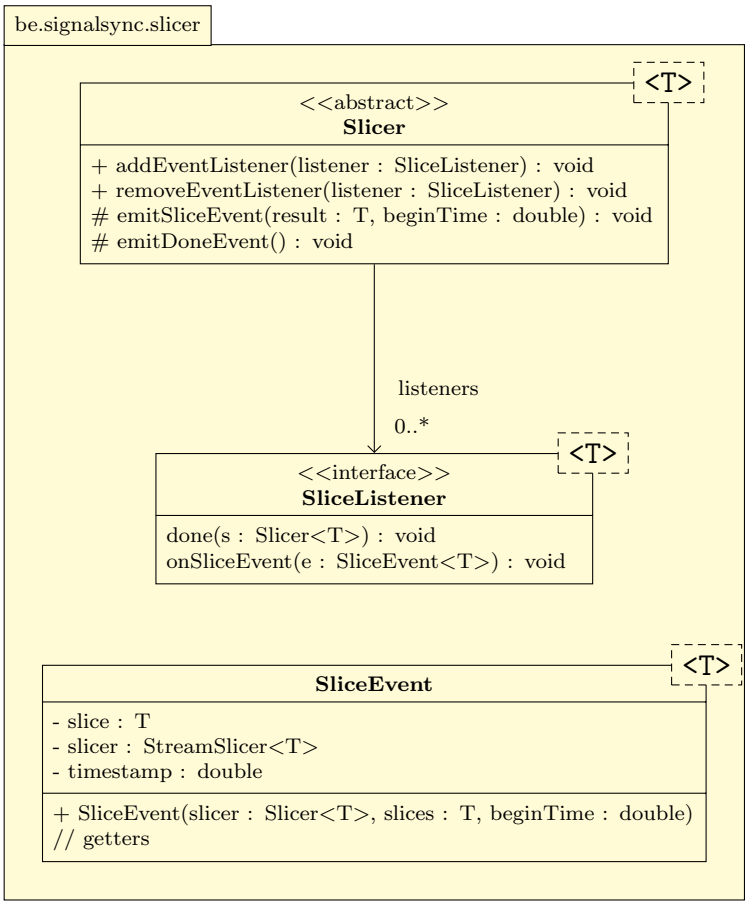
Bufferen van streams

Alle klassen die te maken hebben met het bufferen van streams (het opsplitsen in zogenaamde *slices*) bevinden zich in het package `be.signalsync.slicer`. Het hoe en waarom van deze verwerking werd al uitgebreid behandeld in sectie 2.2.

De abstracte klasse Slicer

Een `Slicer` is een abstracte klasse die een stream of verzameling van streams inleest, in stukjes knipt en vervolgens deze stukjes teruggeeft aan elke geïnteresseerd object. Hoe en wat er precies in stukjes geknipt wordt hangt af van het subtype van `Slicer`. De klasse `Slicer` is enkel verantwoordelijk voor het registreren, bijhouden en verwittigen van objecten die geïnteresseerd zijn in slices. Dit proces wordt verwezenlijk met behulp van het *observer* ontwerppatroon (uitgebreid besproken in boek [24]).

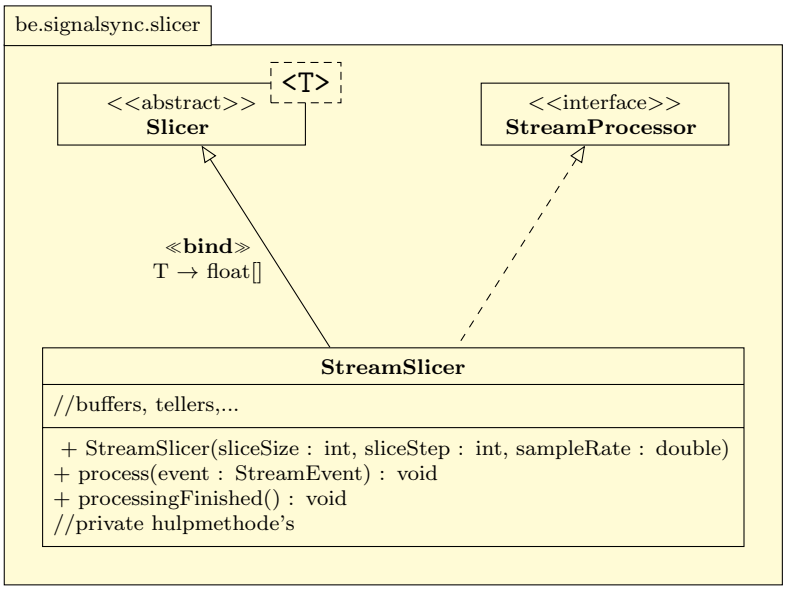
Figuur B.1: UML diagram de observer logica van de klasse `Slicer`



Figuur B.1 toont de drie klassen die een rol spelen het observer mechanisme. Elke klasse is generiek en maakt gebruik van type parameter `T`. Dit is het type van hoe een slice wordt voorgesteld. Bij een slice van een stream is dit type bijvoorbeeld een array van `float` waarden.

Zoals in de figuur te zien is bevat `Slicer` een verzameling van geïnteresseerde objecten. De klasse van een geïnteresseerd object moet de interface `SliceListener` implementeren. Wanneer een object geregistreerd is kan het `SliceEvent` objecten ontvangen. Dit object bevat de laatste nieuwe slice, de `Slicer` vanwaar de slice afkomstig is en een timestamp. Wanneer er geen slices meer verzameld kunnen worden kan de methode `done` van elke geïnteresseerd object worden opgeroepen.

Figuur B.2: UML diagram van de klasse `StreamSlicer` en haar supertypes.

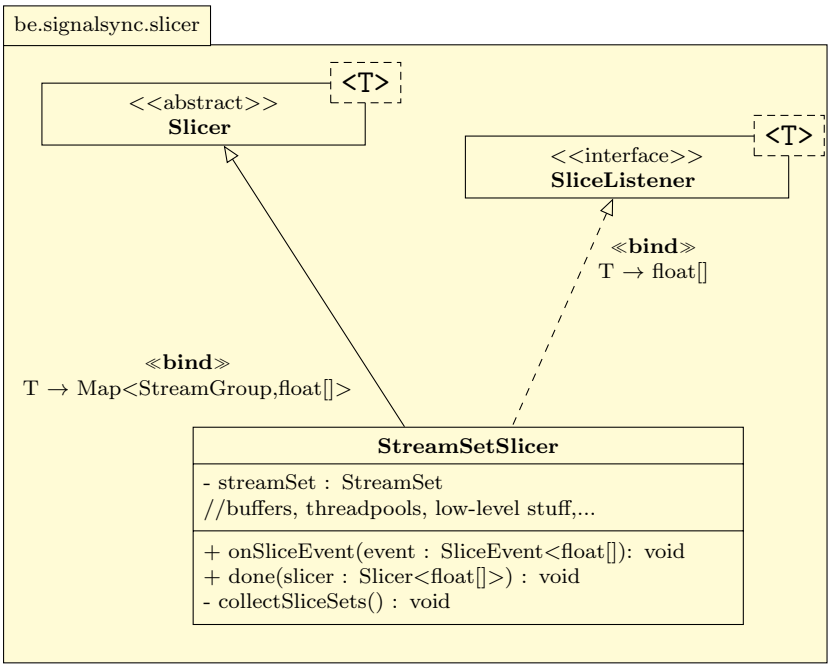


StreamSlicer

Een `StreamSlicer` is een subklasse van `Slicer` die de data afkomstig van een stream opdeelt in `float` arrays van een welbepaalde lengte. Om toegang te krijgen tot de data van een `Stream` implementeert de `StreamSlicer` de interface `StreamProcessor`. Via `process` krijgt deze klasse opeenvolgende `float` arrays met samples binnen die worden opgeslagen in buffers. Wanneer er een slice gereed is worden de buffers samengevoegd tot één float array en wordt `emitSliceEvent` van de superklasse `Slicer` opgeroepen. Wanneer `processingFinished` wordt opgeroepen en de stream dus geen data meer beschikbaar heeft, dan wordt `emitDoneEvent` opgeroepen zodat alle `StreamListeners` hiervan op de hoogte zijn.

Figuur B.2 toont een UML diagram met de besproken klassen en interfaces. Sommige low-level methoden en attributen zijn hierbij weggelaten.

Figuur B.3: UML diagram van de klasse `StreamSetSlicer` en haar supertypes.



StreamSetSlicer

Een `StreamSetSlicer` is een subtype van de klasse `Slicer` die verantwoordelijk is voor het slicen van alle **audiostreams** van een `StreamSet`. Per `StreamGroup` wordt er dus maar één stream in stukjes verdeeld. Dit is logisch aangezien de synchronisatiealgoritmen voor het bepalen van de latency enkel worden uitgevoerd op slices van de audiostream.

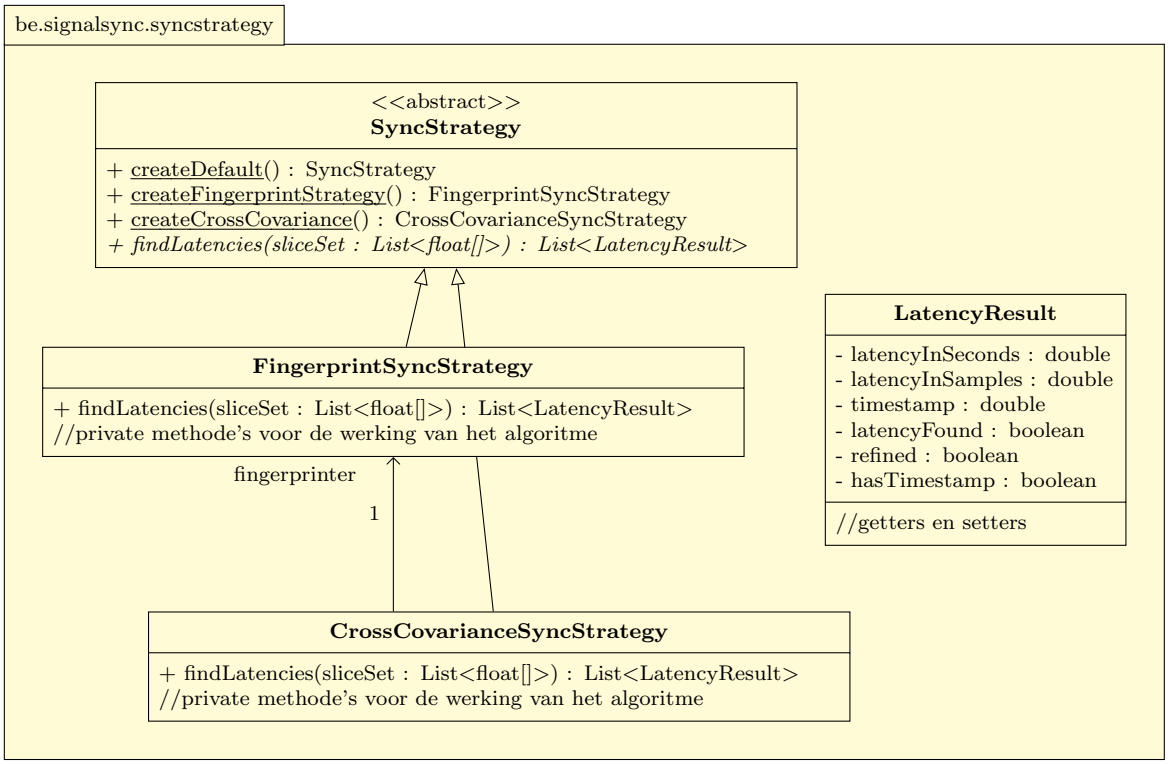
Bij de creatie van een `StreamSetSlicer` wordt van elke `StreamGroup` uit de `StreamSet` de audiostream opgehaald. Op dit `Stream` object wordt vervolgens de methode `createSlicer` opgeroepen waarop er een `StreamSlicer` wordt teruggegeven. De `StreamSetSlicer` voegt zichzelf als geïnteresseerd object aan de `StreamSlicer` toe. Ook wordt de `StreamSlicer` gekoppeld aan de `StreamGroup` door ze als sleutel en waarde toe te voegen aan een `Map`.

Na afloop van deze initialisatie wordt er gewacht tot wanneer alle streams waarop de `StreamSetSlicer` zich geregistreerd heeft een `SliceEvent` (met een `float` array) verstuurd hebben. Al deze slices worden vervolgens samen met hun corresponderende `StreamGroup` via de `emitSliceEvent` methode verstuurd naar de geïnteresseerde objecten. Dit proces wordt herhaald tot er geen enkele `Stream` nog data ter beschikking heeft.

Figuur B.3 toont een vrij abstract klassendiagram van de `StreamSetSlicer` en haar supertypen. Low-level attributen zoals de buffers, threadpools en locks zijn hierop weggelaten.

Oproepen van de algoritmen

Figuur B.4: UML diagram van de klassen met synchronisatiealgoritmen.



De synchronisatiealgoritmen bevinden zich in subklassen van `SyncStrategy`. Deze klasse bevat een abstracte methode `findLatencies`. Deze methode ontvangt als parameter een lijst van `float` arrays waarbij elke array de samples bevat van één slice bevat. Na het uitvoeren geeft de methode een lijst met `LatencyResults` terug.

`SyncStrategy` bevat ook enkele statische methodes waarmee de verschillende mogelijke strategieën gemakkelijk kunnen worden aangemaakt. De parameters die aan de constructoren moeten worden meegegeven worden uit het configuratiebestand gehaald.

De klasse `CrossCovarianceSyncStrategy` bevat een instantie van `FingerprintSyncStrategy`

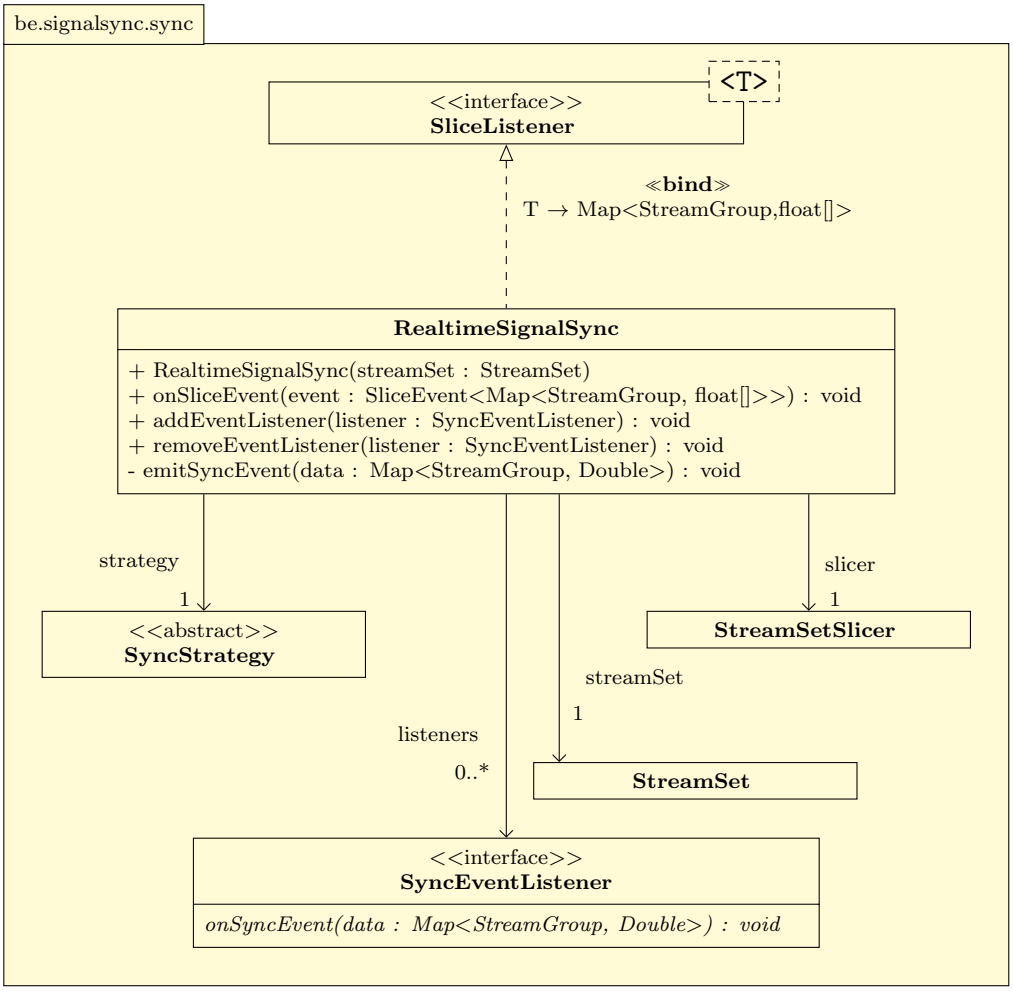
om de geleverde resultaten te kunnen verfijnen. Beide klassen erven over van `SyncStrategy`.

De klasse `CrossCovarianceSyncStrategy` bevat informatie over de bepaalde latency.

Figuur B.4 toont het UML diagram van de besproken klassen.

Bepalen van de latency

Figuur B.5: UML diagram van `RealtimeSignalSync` en alle klassen waarvan ze afhankelijk is.



De klasse `RealtimeSignalSync` is verantwoordelijk voor het aanroepen van de klassen uit verschillende packages waardoor de uiteindelijke synchronisatie kan plaatsvinden.

Een `RealtimeSignalSync` object wordt aangemaakt door aan de constructor een `StreamSet`

object mee te geven dat de te synchroniseren streams bevat. Vervolgens wordt er met behulp van de methode `createSlicer` een `StreamSetSlicer` object aangemaakt. Aangezien `SliceListener` geïmplementeerd wordt kan `RealtimeSignalSync` zich registreren als geïnteresseerd object.

Wanneer er voor elke `StreamGroup` een slice beschikbaar is wordt de methode `onSliceEvent` opgeroepen. In deze methode wordt het `SyncStrategy` object gebruikt om de latencies te bepalen. Optioneel kunnen deze resultaten ook nog gefilterd worden. Ten slotte worden de resultaten naar alle geregistreerde `SyncEventListeners` verstuurd via de private methode `emitSyncEvent`. Geïnteresseerden kunnen zich registreren via de `addEventListener` methode. Deze werkwijze leunt net zoals het *slice event* mechanisme aan bij het *observer* ontwerppatroon.

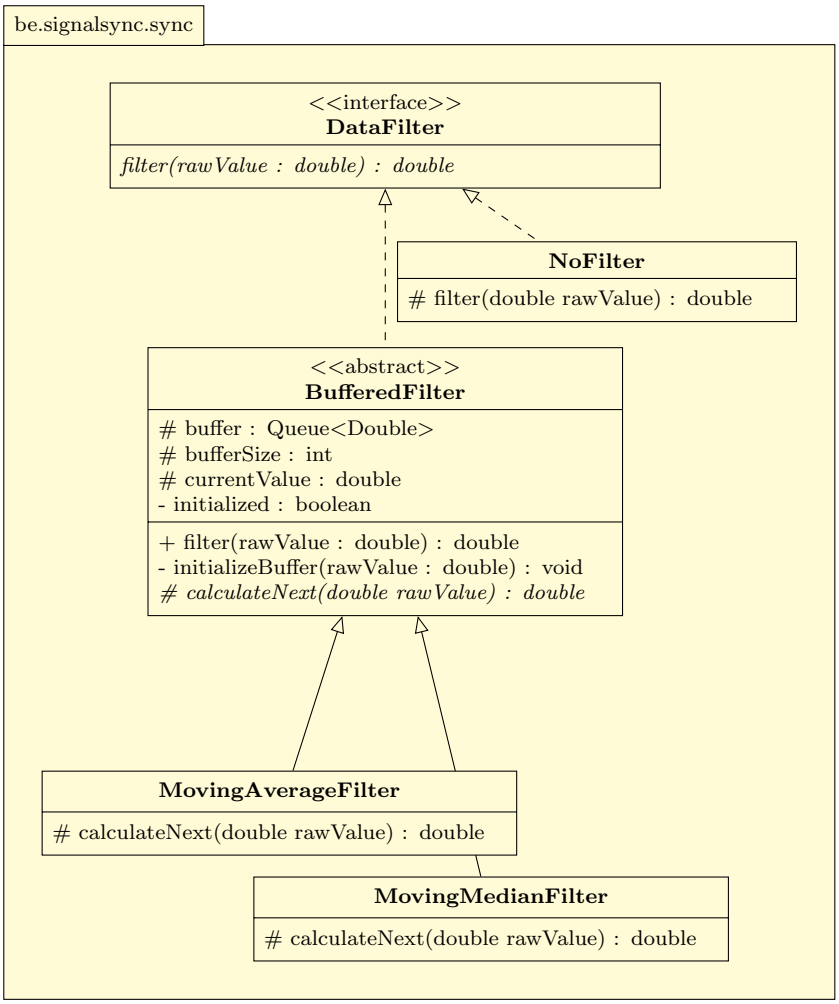
Het starten van de synchronisatie

Het starten van de synchronisatie gebeurt noch in deze klasse noch in een slicer- of strategyklasse. Zowel het slicen als het uitvoeren van de algoritmen is een deterministisch proces en afhankelijk van de samples die door een `Stream` object naar de `StreamProcessors` verstuurd worden. Bij een `AudioDispatcherStream` zal de synchronisatie starten wanneer de `start` methode opgeroepen wordt. Bij een `MSPStream` zal de synchronisatie pas starten wanneer de streams in Max/MSP zelf geactiveerd worden.

Filteren van de resultaten

Zoals in sectie 3.4 is besproken kunnen de latencies gefilterd worden. De meest elementaire filter wordt beschreven door de interface `DataFilter` waarin de methode `filter` wordt beschreven. Deze methode ontvangt een `double` als parameter en geeft een (gefilterde) `double` terug. De meest eenvoudige implementatie van deze klasse is de `NoFilter`. Deze filter geeft rechtstreeks de meegegeven waarde terug zonder iets te wijzigen.

Figuur B.6: UML diagram van de verschillende datafilters.



Gebufferde filters

`DataFilter` wordt ook geïmplementeerd door de abstracte klasse `BufferedFilter`. Dit is een filter die de laatste n waarden in een buffer bijhoudt. De waarde van n wordt bepaalt in het configuratiebestand.

Deze klasse is abstract aangezien de manier waarop de uiteindelijk gefilterde waarde berekent wordt nog niet bepaald is. Hiervoor is de abstracte methode `calculateNext` voorzien. De klasse `MovingAverageFilter` doet dit door het gemiddelde van de waarden uit de buffer te nemen. Bij `MovingMedianFilter` gebeurt dit door de mediaan te berekenen.

Figuur B.6 toont een UML diagram van deze verschillende klassen en interfaces.

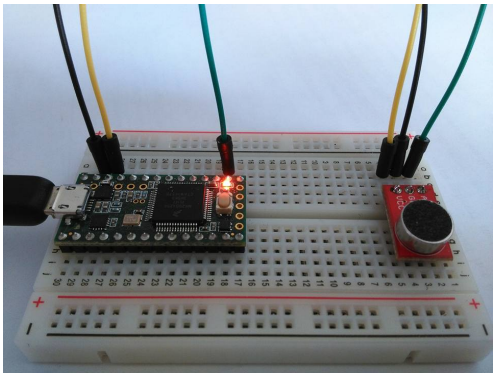
Bijlage C

Praktijktest: opstelling

Teensy

Op de Teensy microcontroller (figuur C.1) wordt een microfoon aangesloten op pin A0. Het programma `TeensyAnalogRead` wordt op de Teensy uitgevoerd. Dit programma is afkomstig van het TeensyDAQ project (IPEM software). Hierdoor worden er maximum 5 analoge waarden aan een frequentie van $8000Hz$ ingelezen.

Figuur C.1: De Teensy microcontroller verbonden met een microfoon op pin A0.

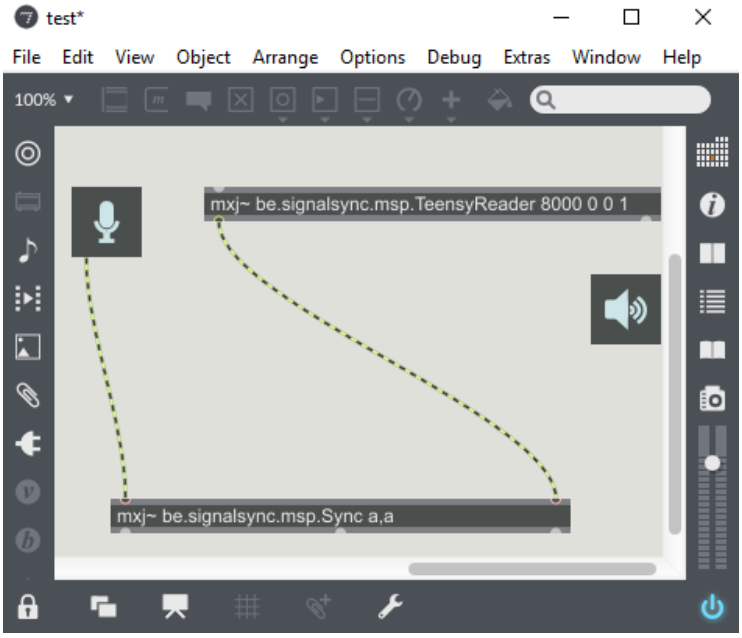


Microfoon

De andere audiostream is afkomstig van een laptopmicrofoon. Alle mogelijke verbeteringen die worden ondersteund (ruisonderdrukking en onderdrukking van akoestische echo) zijn uitgeschakeld.

Max/MSP patch

Figuur C.2: De Max/MSP patch waarmee de test is uitgevoerd.



De laptopmicrofoon wordt ingelezen met behulp van de `ezadc~` module. Deze module is standaard aanwezig in Max/MSP. De audiostream afkomstig van de Teensy microcontroller wordt en ingelezen met behulp van de `TeensyReader` module (geschreven voor dit onderzoek). De module is aangemaakt met volgende code:

```
mxj~ be.signalsync.msp.TeensyReader 8000 0 0 1
```

Het synchroniseren gebeurt met de `Sync` module. Er worden geen datastreams gekoppeld aan de audiostreams. Dit is de code voor het aanmaken van de module:

```
mxj~ be.signalsync.msp.Sync a,a
```

Resultaten

De volgende tabel bevat de opeenvolgende gedetecteerde latencies.

1	0.250621	11	0.249121	21	0.247121	31	0.245371	41	0.24337
2	0.250501	12	0.248991	22	0.247001	32	0.245121	42	0.24337
3	0.250371	13	0.248501	23	0.246741	33	0.244741	43	0.24299
4	0.250121	14	0.248251	24	0.246501	34	0.244741	44	0.24287
5	0.250121	15	0.248121	25	0.246241	35	0.244491	45	0.24275
6	0.249991	16	0.247991	26	0.246121	36	0.244491	46	0.24262
7	0.249621	17	0.247751	27	0.246001	37	0.244121	47	0.24262
8	0.249621	18	0.247491	28	0.245741	38	0.243991	48	0.24237
9	0.249371	19	0.247491	29	0.245501	39	0.243871	49	0.24250
10	0.249241	20	0.247121	30	0.245501	40	0.243751	50	0.24187

Bijlage D

Gebruikershandleiding

Deze handleiding is bedoeld voor mensen (onderzoekers, musicologen,...) die gebruik willen maken van de Max/MSP module zonder enige kennis op het vlak softwareprogrammatie.

Installatie

Eerst moeten de JAR bestanden en het configuratiebestand gedownload worden van het volgende GitHub repository: `https://github.com/wardva/SignalSync/tree/master/Dist`. De bestanden zijn ook rechtstreeks (gezipt) te downloaden via volgende link: `https://github.com/wardva/SignalSync/raw/master/Dist/Dist.zip`. Een gratis tool voor het uitpakken van het archief is 7Zip.

Na het binnenhalen (en eventueel uitpakken) van de bestanden dienen ze te worden verplaatst naar de *lib* map in het installatiedirectory van Max/MSP. Dit is de standaard plaats van deze map onder Windows: `C:\Program Files\Cycling '74\Max 7\resources\packages\max-mxj\java-classes\lib`. Deze map kan echter variëren afhankelijk van eigen gebruikersinstellingen.

Na het herstarten van Max/MSP kunnen de modules gebruikt worden (zie verder).

Het configuratiebestand

Indien het configuratiebestand (`config.properties`) zich in dezelfde map bevindt als de JAR bestanden kunnen de parameters van de algoritmen gewijzigd worden. Na elke wijziging is het noodzakelijk om Max/MSP te herstarten. De betekenis van de parameters zijn al eerder besproken in deze scriptie en zullen daarom niet verder behandeld worden.

Aanmaken van een module

Een in Java geschreven module wordt in Max/MSP aangemaakt met behulp van de `mxj` module. Een module die aan digitale signaalverwerking doet wordt aangemaakt met behulp van de `mxj~` module. De `mxj(~)` module moet worden opgeroepen met als parameter de volledige *klassenaam* van de Java module. Na deze naam volgen de module-specifieke parameters (deze zijn eerder in deze scriptie al besproken).

De namen van de modules die resulteren uit dit onderzoek: `be.signalsync.msp.Sync` en `be.signalsync.msp.TeensyReader`. Beide modules doen aan digitale signaalverwerking en moeten dus worden aangemaakt met `mxj~`.

De TeensyReader module

Deze module is in sectie 3.6.1 al uitgebreid besproken. Bij het inladen van deze module is het aangeraden (maar zeker niet verplicht) om de samplefrequentie van Max/MSP in te stellen op een veelvoud van de samplefrequentie van de Teensy. Dit vereenvoudigt het resampleproces. Soms ontstaat er drift wanneer de frequenties niet eenvoudig kunnen worden omgezet.

De Sync module

Deze module is in sectie 3.6.2 al in detail besproken. Net zoals bij de vorige module moeten de streams worden geresamplet. Het is daarom aangeraden om er voor te zorgen dat de

samplefrequentie van Max/MSP een veelvoud is van de samplefrequentie gebruikt in de softwarebibliotheek (`SAMPLE_RATE` in het configuratiebestand).

De Teensy microcontroller

Om data of audio van een Teensy in te lezen in Max/MSP moet er een programma op de Teensy worden uitgevoerd. De volgende GitHub map bevat twee Arduino sketches die op de Teensy kunnen worden uitgevoerd: `https://github.com/wardva/SignalSync/tree/master/Teensy%20Arduino%20sketches`. Op volgende webpagina wordt uitgelegd hoe Arduino sketches op een Teensy kunnen worden uitgevoerd: `https://www.pjrc.com/teensy/td_download.html`.

De map **TeensyAnalogRead** bevat een sketch waarmee de analoge pinnen van een standaard Teensy met een frequentie van $8000Hz$ kunnen worden uitgelezen. In de map **MicToUSB** bevindt zich de code voor het inlezen van een Teensy uitgerust met een *Audio Adaptor Board* aan een samplefrequentie van $11025Hz$. De geluidskwaliteit van een Teensy uitgerust met dergelijke apparatuur is opmerkelijk beter. Meer informatie hierover is te vinden op volgende pagina: `https://www.pjrc.com/store/teensy3_audio.html`.

Bijlage E

Handleiding voor ontwikkelaars

Deze handleiding is bedoeld voor ontwikkelaars en bestaat uit twee delen. Het eerste deel is bedoeld voor mensen die de synchronisatiebibliotheek willen gebruiken in eigen projecten zonder de broncode te wijzigen. Het tweede deel legt uit hoe de bibliotheek aangepast kan worden.

Het volledige project is openbaar beschikbaar op `https://github.com/wardva/SignalSync`. Het Java-project bevindt zich in de map **SignalSync**.

E.1 Gebruiken van de broncode

De softwarebibliotheek importeren in Eclipse

Vooraleer er van start kan worden gegaan moeten de bestanden `config.properties`, `signalsync.jar`[1] en de jSSC JAR bestanden worden gedownload van het volgende (publieke) GitHub repository: `https://github.com/wardva/SignalSync/tree/master/Dist`. Deze bestanden bevinden zich ook in een Zip bestand dat rechtstreeks gedownload van volgende URL: `https://github.com/wardva/SignalSync/raw/master/Dist/SignalSync.zip`

[1]Het JAR bestand bevat ook alle afhankelijkheden: Panako[21], TarsosDSP[23] en TeensyDAQ.

Maak vervolgens het Java-project aan in Eclipse dat gebruik zal maken van de softwarebibliotheek. Ga hierna naar de properties van het project, selecteer links **Java Build Path** en klik op de tab **Libraries**. Klik vervolgens op de knop **Add External JARs**. Ga nu op zoek naar `signalsync.jar`, `jssc.jar` en `jssc-2.7.0-src.jar` en voeg ze toe aan het project. Let erop dat het configuratiebestand zich steeds in dezelfde map bevindt als de JAR bestanden.

Broncode en documentatie koppelen aan project

`signalsync.jar` bevat de broncode en Javadoc van de softwarebibliotheek. Deze bestanden kunnen aan een bestaand Eclipse project gekoppeld worden waardoor het mogelijk wordt om tijdens het ontwikkelen gebruik te maken van deze informatie. Dit kan worden verwezenlijkt door in de eerder besproken **Libraries** tab op zoek te gaan naar `signalsync.jar` en dit item uit te klappen.

Koppelen van de broncode

Selecteer **Source attachment** en klik op **edit**. Selecteer **External location** en ga vervolgens op zoek naar `signalsync.jar`.

Koppelen van de Javadoc

Selecteer **Javadoc location** en klik op **edit**. Selecteer **Javadoc in archive**. Blader vervolgens bij **Archive path** naar de locatie van `signalsync.jar`. Blader en selecteer bij **Path within archive** het mapje **doc** in het JAR bestand.

Aanmaken en synchroniseren van streams

Hoe de streams aangemaakt en gesynchroniseerd kunnen worden in het gemakkelijkst uit te leggen aan de hand van een voorbeeld. In het volgende voorbeeld worden enkele streams

aangemaakt en wordt de latency ertussen bepaald. Meer informatie over de TarsosDSP `AudioDispatchers` is te vinden in artikel [23].

Om te beginnen moeten er enkele streams worden aangemaakt. In dit voorbeeld zal gebruik gemaakt worden van `AudioDispatcherStreams`.

```
1  //Een AudioDispatcher aanmaken van een microfoon
2  AudioDispatcher micDispatcher =
3      AudioDispatcherFactory.fromDefaultMicrophone(512, 0);
4
5  //AudioDispatcher wrappen in een Stream
6  Stream micStream =
7      new AudioDispatcherStream(micDispatcher);
8
9  //Datastream koppelen aan microfoon
10 Stream dataStreamAttachedToMicrophone = ...
11
12 //Een AudioDispatcher aanmaken van een bestand
13 File file = new File("recorded.wav");
14 AudioDispatcher fileDispatcher =
15     AudioDispatcherFactory.fromFile(file, 512, 0);
16
17 //AudioDispatcher wrapper in een Stream
18 Stream fromFileStream =
19     new AudioDispatcherStream(fileDispatcher);
```

Vervolgens moeten de streams gegroepeerd worden in `StreamGroup` objecten waarin wordt aangegeven welke audiostream gebruikt wordt voor de synchronisatie. Deze objecten worden verpakt in een `StreamSet`.

```
1  //Per gekoppelde groep streams een StreamGroup aanmaken
2  StreamGroup micGroup = new StreamGroup();
3  micGroup.setDescription("The microphone streamgroup");
4  micGroup.setAudioStream(micStream);
5  micGroup.addDataStream(dataStreamAttachedToMicrophone);
6
7  StreamGroup fileGroup = new StreamGroup();
8  fileGroup.setDescription("The file streamgroup");
9  fileGroup.setAudioStream(fromFileStream);
10
11 //Van alle StreamGroups een StreamSet aanmaken
12 StreamSet allStreams = new StreamSet();
13 allStreams.addStreamGroup(micGroup);
14 allStreams.addStreamGroup(fileGroup);
```

De `StreamSet` kan vervolgens worden meegegeven aan een `RealtimeSignalSync` object dat de synchronisatie (bepalen van de latency) op zich neemt. Ook wordt er een anonieme `SyncEventListener` aangemaakt. Op de allerlaatste lijn worden alle streams opgestart.

```
1  RealtimeSignalSync syncer = new RealtimeSignalSync(allStreams);
2
3  //Een anonieme inner-class registeren als SyncEventListener.
4  syncer.addEventListener(new SyncEventListener() {
5      @Override
6      public void onSyncEvent(Map<StreamGroup, LatencyResult> l) {
7          //Resultaten afdrukken
8          for(Entry<StreamGroup, LatencyResult> entry : l.entrySet())
9          {
10             StreamGroup group = entry.getKey();
11             LatencyResult result = entry.getValue();
12             System.out.println(group.getDescription());
```

```
13                System.out.println(result.toString());
14            }
15        }
16    });
17    //Alle streams starten
18    allStreams.start();
```

Wijzigen van de broncode

Project importeren in Eclipse

De gemakkelijkste manier om de broncode van het project binnen te halen is door het volledige GitHub repository te clonen. De kans is wel groot dat hierbij ook heel wat onnodige bestanden gedownload worden (zoals de broncode van deze thesis). De StackOverflow post `http://stackoverflow.com/a/13738951/1264345` beschrijft een methode om dit te omzeilen.

Na het binnenhalen van het project kan de map **SignalSync** geïmporteerd worden in Eclipse. Maak hiervoor een nieuw Java project aan en wijzig de *default location* in het pad van **SignalSync**. Na het importeren zullen enkele foutmeldingen te zien zijn wegens gebroken afhankelijkheden.

Het project is afhankelijk van volgende softwarebibliotheken: TarsosDSP, Panako en TeensyDAQ. De TarsosDSP broncode kan gedownload worden van `https://github.com/JorenSix/TarsosDSP`, de Panako broncode van `http://panako.be/releases/Panako-latest/readme.html` en de TeensyDAQ broncode van `https://github.com/JorenSix/TeensyDAQ`.

Na het importeren van deze projecten in eclipse en het instellen van de projecten als *reference project* moeten er nog enkele JAR bestanden worden toegevoegd.

jssc-2.7.0.jar, deze bibliotheek is te downloaden van: `https://code.google.com/archive/p/java-simple-serial-connector/downloads`.

max.jar, deze bibliotheek bevindt zich in de Max/MSP installatiemap standaard onder

Windows: `C:\Program Files\Cycling '74\Max 7\resources\packages\max-mxj\java-classes\lib`

Na het toevoegen van de *references* en JAR bestanden zouden de foutmeldingen moeten verdwijnen.

Het project koppelen aan Max/MSP

Het zou vrij omslachtig zijn om telkens een JAR bestand te moeten genereren vooraleer een bepaalde module in Max/MSP uitgetest kan worden. Daarom is het ook mogelijk om Max/MSP te koppelen aan een map met classfiles. Hiervoor moet het configuratiebestand `max.java.config.txt` uit de Max/MSP installatiemap worden aangepast. Onder Windows bevindt het bestand zich standaard in volgende map: `C:\Program Files\Cycling '74\Max 7\resources\packages\max-mxj\java-classes`.

Met volgende lijn wordt er verwezen naar een map met classfiles: `max.dynamic.class.dir <<pad>>`. Hierbij moet `<<pad>>` worden vervangen door het pad van de map (backslashes escapen met een backslash). Er moet ook naar elke afhankelijkheid verwezen worden. Verwijs dus ook naar de map met classfiles van TarsosDSP en Panako. Ten slotte moet ook de jSSC JAR worden opgenomen in het bestand. Voeg daarom volgende lijn toe waarbij `<<pad>>` staat voor de map waarin zich de JAR bevindt: `max.dynamic.jar.dir <<pad>>`.

Debuggen van Max/MSP modules

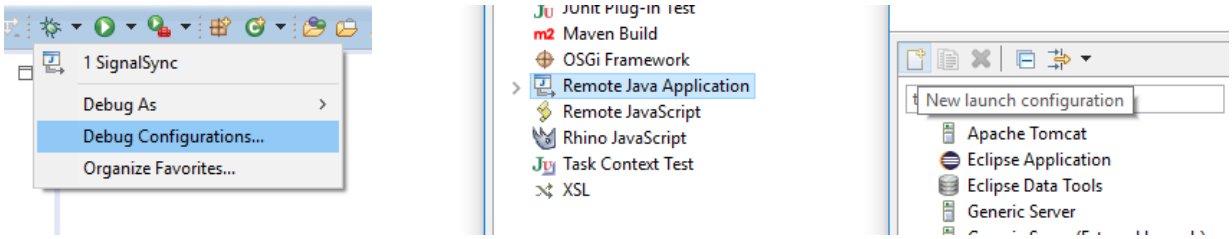
Het is mogelijk om de broncode in Eclipse te debuggen terwijl een module in Max/MSP wordt uitgevoerd. Deze handige feature vereist wel wat configuratie.

Ten eerste moeten volgende lijnen aan het Max/MSP configuratiebestand (zie vorige sectie) worden toegevoegd:

```
max.jvm.option -Xdebug
max.jvm.option -Xnoagent
```

```
max.jvm.option -Xrunjdwp:transport=dt_socket\,address=8074\,server=y\,suspend=n
max.jvm.option -XX:-UseSharedSpaces
```

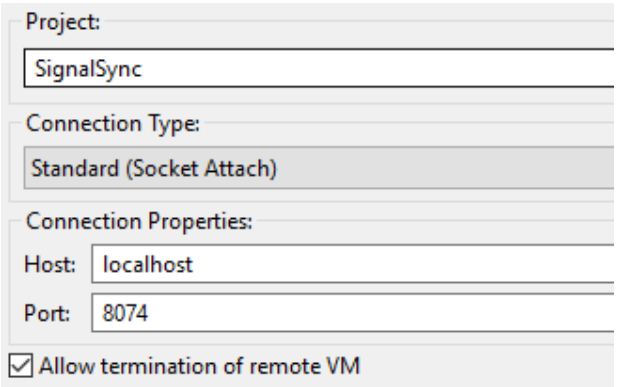
Het is mogelijk dat deze lijnen in het standaard configuratiebestand in commentaar staan. Indien dit het geval zou zijn dan is het voldoende om de puntkomma's voor en achter de regel te verwijderen.



Figuur E.1: Nieuwe debug configuration aanmaken in Eclipse

Ga hierna naar **Debug Configurations** in Eclipse. Selecteer **Remote Java Application** en klik vervolgens op het icoon van **new launch configuration**. Deze stappen worden verduidelijkt in figuur E.1.

Figuur E.2: De debug configuratie in Eclipse



Geef vervolgens de configuratie een naam en blader bij **Project** naar het te debuggen project (SignalSync). Kies bij **Connection Type** voor **Standard (Socket Attach)**. Bij **Connection Properties** moet de poort worden opgegeven die in het Max/MSP configuratiebestand gespecificeerd is (standaard 8074). Aangezien de applicatie lokaal draait moet bij **Host** `localhost` of `127.0.0.1` worden opgegeven. Met **Allow termination of remote VM** kan worden ingesteld of de Max/MSP VM vanuit Eclipse mag worden afgesloten. Figuur E.2 toont een screenshot van de instellingen.

Het debuggen wordt gestart door eerst de Max/MSP patch te openen en vervolgens het debuggen in Eclipse te starten met de zojuist aangemaakt debug configuratie. Bij het starten van de patch is het nu mogelijk om via breakpoints door de code navigeren.

Genereren van testdata

De bestanden die vereist zijn voor het uitvoeren van de testen bevinden zich niet in het GitHub repository maar moeten nog gegenereerd worden.

De eerste stap voor het genereren van de bestanden is het uitvoeren van het Perl script `TestSetGenerator.pl`. Dit script kan gedownload worden van de map **Scripts** in het GitHub repository. Het script maakt van een groep mp3-bestanden verschillende varianten aan (zie 4.1.2 en 4.1.3). In het script moet de bron- en doelmap worden opgegeven.

Met behulp van de Java applicatie `SlicerApp` uit de package `be.signalsync.app` kunnen de gegenereerde bestanden worden omgezet in slices. Deze slices dienen als input voor enkele testen.

Referentielijst

[1] Audacity. http://audacity.sourceforge.net/, 2015. [Online; geraadpleegd 12-maart-2016].

[2] Cycling '74 Max. https://cycling74.com/, 2016. [Online; geraadpleegd 12-maart-2016].

[3] Ipem - systematic musicology. https://www.ugent.be/lw/kunstwetenschappen/en/research-groups/musicology/ipem, 2016. [Online; geraadpleegd 05-maart-2016].

[4] Msp midi tutorial 1: Mapping midi to msp. https://cycling74.com/wiki/index.php?title=MSP_MIDI_Tutorial_1:_Mapping_MIDI_to_MSP, 2016. [Online; geraadpleegd 29-mei-2016].

[5] Teensy USB Development Board. https://www.pjrc.com/teensy/, 2016. [Online; geraadpleegd 19-maart-2016].

[6] A Digital Media Primer for Geeks. https://xiph.org/video/vid1.shtml, 2016. [Online; geraadpleegd 21-maart-2016].

[7] David Bannach, Oliver Amft, and Paul Lukowicz. Automatic event-based synchronization of multimodal data streams from wearable and ambient sensors. In *Smart sensing and context*, pages 135–148. Springer, 2009.

[8] Benjamin Barras. Sox: Sound exchange. Technical report, 2012.

[9] Juan Pablo Bello, Laurent Daudet, Samer Abdallah, Chris Duxbury, Mike Davies, and Mark B Sandler. A tutorial on onset detection in music signals. *Speech and Audio Processing, IEEE Transactions on*, 13(5):1035–1047, 2005.

[10] Chris Cannam, Christian Landone, and Mark Sandler. Sonic visualiser: An open source application for viewing, analysing, and annotating music audio files. In *Proceedings of the 18th ACM international conference on Multimedia*, pages 1467–1468. ACM, 2010.

[11] Simon Dixon. Live tracking of musical performances using on-line time warping. In *Proceedings of the 8th International Conference on Digital Audio Effects*, pages 92–97. Citeseer, 2005.

[12] Simon Dixon and Gerhard Widmer. Match: A music alignment tool chest. In *ISMIR*, pages 492–497, 2005.

[13] B. Fries and M. Fries. *Digital Audio Essentials: A comprehensive guide to creating, recording, editing, and sharing music and other audio*. O'Reilly Digital Studio. O'Reilly Media, 2005. ISBN 9781491925638.

[14] Javier Jaimovich and Benjamin Knapp. Synchronization of multimodal recordings for musical performance research. In *NIME*, pages 372–374, 2010.

[15] Roman Kollár. Configuration of ffmpeg for high stability during encoding.

[16] Harry Nyquist. Certain topics in telegraph transmission theory. 1928.

[17] Alan V Oppenheim. Speech spectrograms using the fast fourier transform. *IEEE spectrum*, 8(7):57–62, 1970.

[18] Chotirat Ann Ratanamahatana and Eamonn Keogh. Everything you know about dynamic time warping is wrong. In *Third Workshop on Mining Temporal and Sequential Data*. Citeseer, 2004.

[19] Stan Salvador and Philip Chan. Toward accurate dynamic time warping in linear time and space. *Intelligent Data Analysis*, 11(5):561–580, 2007.

[20] Joren Six. *Digital Sound Processing and Java*. UGent, IPEM, Sint-Pietersnieuwstraat 41, 9000 Ghent - Belgium, 5 2015.

[21] Joren Six and Marc Leman. Panako - A Scalable Acoustic Fingerprinting System Handling Time-Scale and Pitch Modification. In *Proceedings of the 15th ISMIR Conference (ISMIR 2014)*, 2014.

[22] Joren Six and Marc Leman. Synchronizing Multimodal Recordings Using Audio-To-Audio Alignment. *Journal of Multimodal User Interfaces*, 9(3):223–229, 2015. ISSN 1783-7677. doi: 10.1007/s12193-015-0196-1.

[23] Joren Six, Olmo Cornelis, and Marc Leman. TarsosDSP, a Real-Time Audio Processing Framework in Java. In *Proceedings of the 53rd AES Conference (AES 53rd)*. The Audio Engineering Society, 2014.

[24] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.

[25] Avery Li-Chun Wang. An industrial-strength audio search algorithm. In *ISMIR 2003, 4th Symposium Conference on Music Information Retrieval*, pages 7–13, 2003.