Realtime signaal synchronisatie met accoustic 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 accoustic 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

Todo: voorwoord schrijven

Ward Van Assche, juni 2016

Toelating tot bruikleen

Realtime signaal synchronisatie met accoustic 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

Todo: samenvatting schrijven

Trefwoorden

synchronisatie, realtime, datastromen, musicologie, accoustic fingerprinting

Realtime signal synchronization with accoustic fingerprinting

Ward Van Assche

Supervisor(s): Joren Six, Marleen Denert

Abstract—**Sed nec tortor in libero rutrum pellentesque et gravida turpis. Phasellus gravida neque vitae elit fringilla, a efficitur purus sollicitudin. Proin lacus est, suscipit sed nibh ac, hendrerit eleifend leo. Suspendisse quis semper leo. Duis non elit commodo, sodales ex non, venenatis diam. Sed libero tortor, hendrerit et sollicitudin ut, facilisis vitae odio. Fusce vitae mi odio.**

Keywords—**kernwoord1, kernwoord2, kernwoord 3, kernwoord 4**

I. Introduction

SED Sed nec tortor in libero rutrum pellentesque et gravida turpis. Phasellus gravida neque vitae elit fringilla, a efficitur purus sollicitudin. Proin lacus est, suscipit sed nibh ac, hendrerit eleifend leo. Suspendisse quis semper leo. Duis non elit commodo, sodales ex non, venenatis diam. Sed libero tortor, hendrerit et sollicitudin ut, facilisis vitae odio. Fusce vitae mi odio. Cras vitae quam bibendum, elementum velit ut, varius enim. Donec sagittis elit ligula, laoreet viverra felis rhoncus nec. Donec mattis metus pretium, pulvinar enim a, luctus nunc. Pellentesque quis suscipit leo.

II. Sectie

A. Subsectie

Sed nec tortor in libero rutrum pellentesque[1] et gravida turpis. Phasellus gravida neque vitae elit fringilla, a efficitur purus sollicitudin. Proin lacus est, suscipit sed nibh ac, hendrerit eleifend leo. Suspendisse quis semper leo. Duis non elit commodo, sodales ex non, venenatis diam. Sed libero tortor, hendrerit et sollicitudin ut, facilisis vitae odio. Fusce vitae mi odio. Cras vitae quam bibendum, elementum velit ut, varius enim. Donec sagittis elit ligula, laoreet viverra felis rhoncus nec. Donec mattis metus pretium, pulvinar enim a, luctus nunc. Pellentesque quis suscipit leo.

B. Andere subsectie

Sed nec tortor in libero rutrum pellentesque et gravida turpis. Phasellus gravida neque vitae elit fringilla, a efficitur purus sollicitudin. Proin lacus est, suscipit sed nibh ac, hendrerit eleifend leo. Suspendisse quis semper leo. Duis non elit commodo, sodales ex non, venenatis diam. Sed libero tortor, hendrerit et sollicitudin ut, facilisis vitae odio. Fusce vitae mi odio. Cras vitae quam bibendum, elementum velit ut, varius enim. Donec sagittis elit ligula, laoreet viverra felis rhoncus nec. Donec mattis metus pretium, pulvinar enim a, luctus nunc. Pellentesque quis suscipit leo.

Sed nec tortor in libero rutrum pellentesque et gravida turpis. Phasellus gravida neque vitae elit fringilla, a efficitur purus sollicitudin. Proin lacus est, suscipit sed nibh ac, hendrerit eleifend leo. Suspendisse quis semper leo. Duis non elit commodo, sodales ex non, venenatis diam. Sed libero tortor, hendrerit et sollicitudin ut, facilisis vitae odio. Fusce vitae mi odio. Cras vitae quam bibendum, elementum velit ut, varius enim. Donec sagittis elit ligula, laoreet viverra felis rhoncus nec. Donec mattis metus pretium, *pulvinar* enim a, luctus nunc. Pellentesque quis suscipit leo.

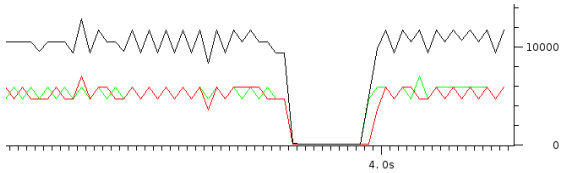


Fig. 1. Detailed capture of the stream at the moment of a handover between two simulated AP's with a strong signal. Notice the gap of 100 ms.

III. Sectie

Aenean auctor congue nisi, volutpat porta urna lobortis in. Donec accumsan fermentum lectus, sed aliquet lectus gravida eget. Ut turpis quam, fermentum eget sem sed, euismod facilisis sem. Etiam a sollicitudin purus. Vestibulum quis nisl et nibh condimentum suscipit tristique eget quam. Aenean eget varius lectus. Maecenas sit amet mi augue. Nullam semper ex et facilisis ullamcorper. Cras volutpat ornare arcu, ac suscipit nisi pellentesque iaculis. Vivamus sit amet ipsum consequat, egestas massa varius, aliquam lorem. Aenean ornare iaculis dolor, eget efficitur elit. Maecenas ut massa ac tortor hendrerit pulvinar a in ante.

IV. Conclusion

The simulation results show a nice advantage for the moving cell[3] concept. The traditional handover problem can be avoided so a workable model for broadband access on trains seems realistic. But there needs to be done a lot of research to make RAU's and RoF as reliable and cheap as possible.

References

[1] 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.

[2] 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.

[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] A. L. Wang, "An industrial-strength audio search algorithm," in *ISMIR 2003, 4th Symposium Conference on Music Information Retrieval*, 2003, pp. 7–13.

Inhoudsopgave

Gebruikte afkortingen

IPEM	Instituut voor Psychoakoestiek en Elektronische Muziek
DSP	Digital Signal Processing
FFT	Fast Fourier transform
ECG	Elektrocardiogram
DTW	Dynamic timewarping
USB	Universal Serial Bus
ADC	Analog-to-digital converter
PCM	Pulse-code modulation

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. [1]

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 klassieke experiment wordt onderzocht wat de invloed is van muziek op de lichamelijke activiteit van een persoon. Alle bewegingen worden geregistreerd met een videocamera en verschillende sensoren.

Hierbij moeten minstens drie datastreams worden geanalyseerd: de videobeelden, de data van de accelerometer(s) 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 van elke opname 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 vergezeld van 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 kan gebruikt worden als referentie, aangezien dit uiteraard al een perfecte weergave is van het omgevingsgeluid.

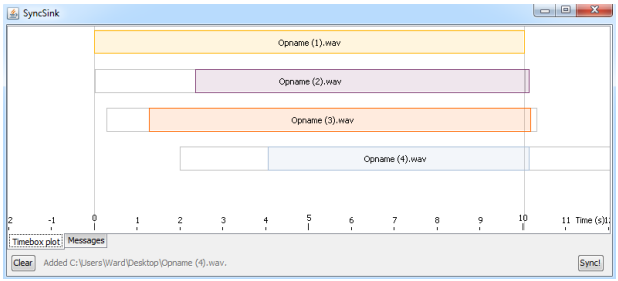
Na het uitvoeren van het experiment beschikt men dus over de gegevens van drie datastreams, waarbij er aan elke datastream een quasi perfect synchrone opname van het omgevingsgeluid is gekoppeld. Aangezien het experiment in één ruimte is uitgevoerd zijn de verschillende opnames van het omgevingsgeluid 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 audiostromen te synchroniseren. Bij het IPEM heeft men een systeem ontwikkeld dat in staat is om verschillende audiostreams te synchroniseren.

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, en dit volledig handmatig. De opgenomen audiobestanden moet worden verzameld op een

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

computer, vervolgens kan met behulp van de audiobestanden de latency van elke datastream worden berekend. Vervolgens kunnen de datastreams worden gesynchroniseerd. 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: Huidige werkwijze om streams te synchroniseren: Een drag and drop interface waarin de opgenomen fragmenten gesleept kunnen worden na afloop van het experiment. Vervolgens wordt de latency berekend.

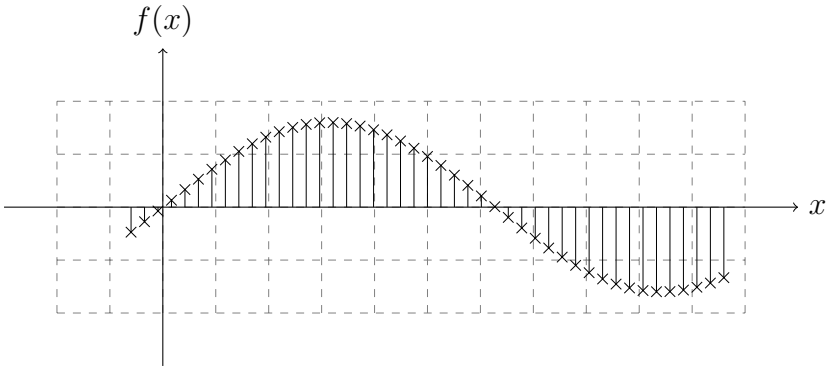
Een ander probleem is iets vager en minder duidelijk te omschrijven. De resultaten van het kruiscovariantie algoritme bevatten soms afwijkingen die moeilijk te verklaren zijn. De precieze oorzaak hiervan, en hoe dit kan worden opgelost zal ook worden onderzocht. Ook is het kruiscovariantie algoritme in vergelijking met het accoustic 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 niet over struikelen.

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. PCM audio heeft verschillende parameters die een invloed hebben op de uiteindelijke kwaliteit van de digitale 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 [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 maximumfrequentie van de te converteren informatie. Bij het gebruik van een lagere frequentie treedt er informatieverlies op. Deze stelling wordt in detail besproken in het originele artikel [15] 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 bepaald 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 [2] en boek [12].

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 het tijdens het uitvoeren van het experiment mogelijk moet zijn om de huidige latencies van de streams op te vragen.

Het is moeilijk om het opvragen van deze gegevens echt in realtime mogelijk te maken. Veel 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. Deze tijd is de maximaal mogelijke achterstand ten opzichte van de realtime latency.

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 wijziging in 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 detectie is mogelijk vanaf het moment dat de buffer voor meer dan de helft gevuld is met data gegenereerd na het gegevensverlies. Rekening houdend met het eerste criterium zou een wijziging van de latency binnen vijf seconden gedetecteerd moeten kunnen worden.

Detecteren van drift

Elke stream heeft een bepaalde 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 minder nauwkeurig gekend 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 samplefrequentie van 8000 Hz komt dit overeen met 7,5 ms.[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 eerder beschreven evaluatiecriteria.

1.4.1 Event-gebaseerde synchronisatie

Deze methode wordt beschreven in [7, 22] en is een eenvoudige, 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

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

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 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 [13] 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 (**??**) 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 door de manier waarop de tijdreeksen niet-lineair worden kromgetrokken ten opzichte van de tijdas[18]. De minimale kost kan in kwadratische tijd berekend worden door gebruik te maken van dynamisch programmeren [10]. DTW is een veelgebruikte techniek in domeinen zoals spraakherkenning, bio-informatica, data-mining, etc [17].

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 [10] 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 we willen zo snel mogelijk de latency tussen de streams achterhalen.

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

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

performantie van het algoritme in beschouwing werd genomen.

1.4.4 Accoustic fingerprinting

Accoustic fingerprinting is een techniek die in staat is om gelijkenissen te vinden tussen verschillende audiofragmenten. Het is 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 accoustic fingerprinting extraheert en vergelijkt fingerprints van audiofragmenten. Een accoustic 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 accoustic 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.

[4]Het grootste voorbeeld hiervan is de smartphone app Shazam. Deze app is de eerste toepassing dat gebruik maakte 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 data-streams 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 deze materie verder in detail behandeld.

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 audiostromen te synchroniseren. Deze bibliotheek moet vanuit andere software kunnen worden opgeroepen en gedetailleerde informatie teruggeven over de synchronisatie van de verschillende audiostreams.

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 binnenkomende datastromen (vergezeld met audiostream) te synchroniseren en op één of andere manier 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.

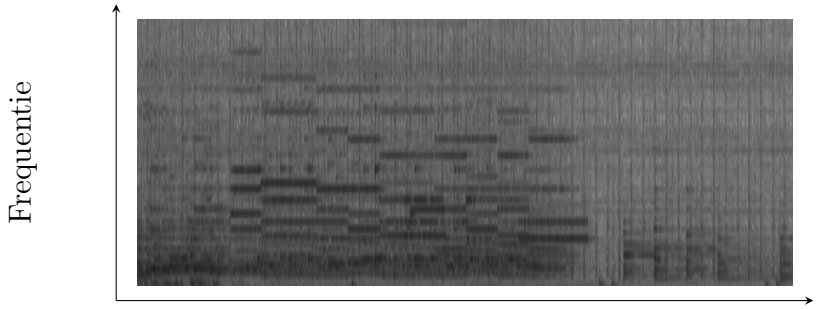
2.1.1 Accoustic fingerprinting

Zoals in de introductie al is beschreven maakt het accoustic fingerprinting algoritme gebruik van fingerprints geëxtraheerd uit audiofragmenten. Het op zoek gaan naar gelijkenissen gebeurt door deze fingerprints met elkaar te vergelijken.

Features

Een cruciale stap bij de ontwikkeling van een accoustic 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. Een 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 toch veel informatie over het opgenomen audiofragment waardoor de kans op foutieve resultaten door *collissions* kleiner wordt.

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



Werking

Een accoustic 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 complexe functies in het frequentiedomein[1]. Deze verzameling functies kan gezien worden als het tijd-frequentiedomein van het audio-

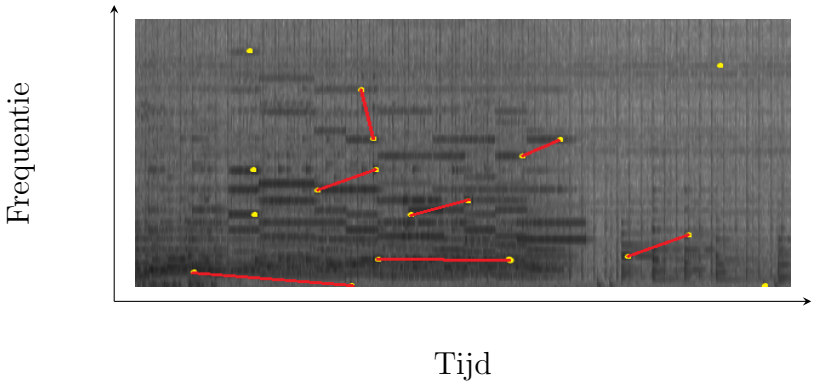
[1]Een veelgebruikt algoritme hiervoor is het Fast Fourrier Transformation algoritme (FFT). In artikel [16]. wordt deze methode uitgebreid besproken.

fragment. De grafische voorstelling van deze verzameling functies wordt het spectrogram genoemd. Meestal wordt op de x-as de tijd weergegeven en op de y-as de frequentie. De intensiteit waarmee een bepaalde frequentie voorkomt kan worden aangeduid door gebruik te maken van verschillende kleuren of contrasten.

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 alle aanliggende tijd-frequentie punten[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.

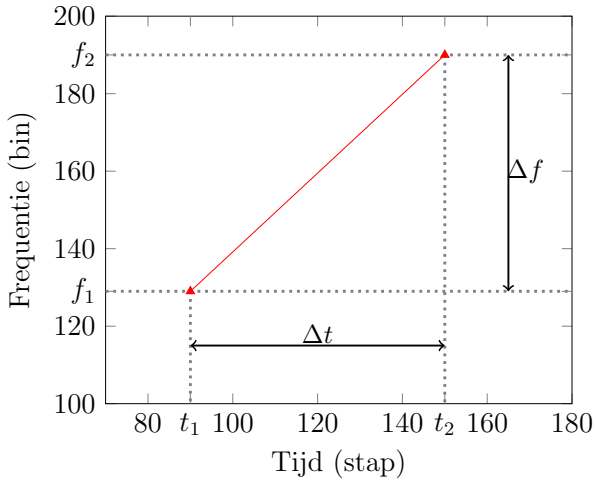
Figuur 2.2: De kandidaat-pieken (gele stipjes) en fingerprints (rode lijnen) van *Talk Talk - New Grass*.



Na het bepalen van de fingerprints worden ze opgeslagen in een datastructuur waarin snel naar matches kan worden gezocht. Om dit mogelijk te maken moeten er van de fingerprints enkele parameters 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: 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:

Ten eerste kan er van worden uitgegaan dat twee overeenkomende fingerprints uit twee geluidsfragmenten dezelfde frequenties ($f1$ en $f2$) zullen hebben. Bijgevolg zal dus ook het verschil in frequentie (Δf) gelijk zijn.

In tegenstelling tot de frequenties is de kans zeer klein dat ook de tijd van de spectrale pieken ($t1$ en $t2$) van twee overeenkomende fingerprints zal overeenkomen. Bij 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. Ook bij het syn-

chroniseren van streams zal de latency er voor zorgen dat de begintijd van de fingerprints uit beide audiofragmenten verschillen.

Hoewel de tijd ($t1$ en $t2$) van twee fingerprints meestal verschilt is dit niet het geval voor het verschil ervan (Δt). Bij twee overeenkomende fingerprints van twee audiofragmenten is het verschil in frequentie inherent gelijk.

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.

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 eerst 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. Wat de precieze waarde is van “een groot aantal” is een parameter van het algoritme.

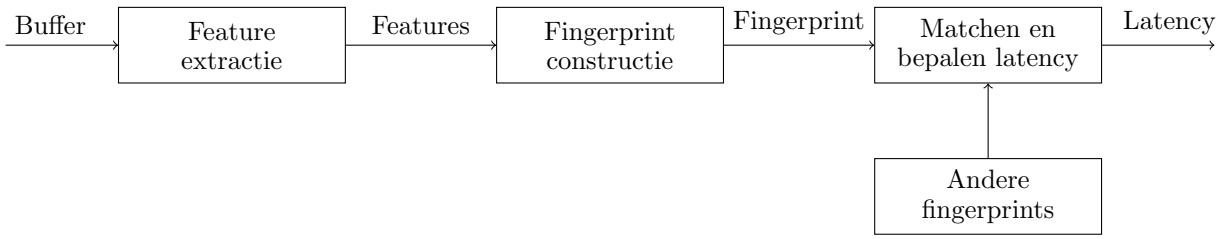
Bepalen van de latency

Accoustic 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 overeenkomstige buffers met elkaar te matchen. Wanneer er een offset

(het verschil tussen $t1$ van elke fingerprint) opvallend veel voorkomt (aantal ligt boven een drempelwaarde), dan kan er van worden uitgegaan dat de huidige latency tussen de streams gelijk is aan de offset. Dit is logisch aangezien het verschil tussen de $t1$ waarden de verschuiving tussen de geluidsfragmenten voorstelt.

Een uitgebreidere beschrijving is te vinden in artikel [24]. 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].

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



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 is zéér interessant maar 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 (variabele s). Deze audioblokken worden telkens cyclisch één sample verschoven tot wanneer de kruiscovariantie waarde voor elke mogelijke verschuiving berekend werd. De variabele i stelt de huidige verschuiving voor en gaat van 0 tot s. De kruiscovariantie waarde wordt berekend met volgende formule:

$$\sum_{j=0}^{s} a_j \cdot b_{(i+j) \ mod \ s} \tag{2.1}$$

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 s samples kan gebeuren in $O(s^2)$. 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 accoustic 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.[2] 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 accoustic 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

[2]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.

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 heel erg 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 accoustic fingerprinting algoritme kan zorgen voor de synchronisatie tot op enkele tientallen milliseconden nauwkeurig. Dit resultaat laat toe dat we het kruiscovariantie algoritme kunnen uitvoeren op zeer korte stukjes audio (een honderdtal milliseconden volstaat).

2.2 Bufferen van streams

In deze scriptie is al ettelijke malen vermeld geweest dat het belangrijk is dat de streams gebufferd worden. Zonder het bufferen is het onmogelijk om de synchronisatiealgoritmen uit te voeren.

De grootte van de buffer heeft invloed op de kwaliteit van de geretourneerde 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.

Naïeve implementatie

Indien er buffers gebruikt worden die 10 seconden audio kunnen bevatten, dan zal het bij een naïeve implementatie in het slechtste geval pas mogelijk zijn om een wijziging van de latency na 15 seconden te detecteren. Een dergelijke wijziging kan gedetecteerd worden wanneer meer dan de helft van de buffer gevuld is met audio met de nieuwe latency. Wanneer er samples gedropt worden net na het moment dat de buffer voor de helft gevuld is, dan zal het algoritme uitgevoerd op de huidige buffer de wijziging niet kunnen detecteren. De volgende buffer zal wel gevuld zijn audio met de nieuwe latency, het duurt echter nog een

bijkomende 10 seconden vooraleer deze buffer gevuld is. De detectietijd bedraagt bijgevolg in het slechtste geval net geen 15 seconden.

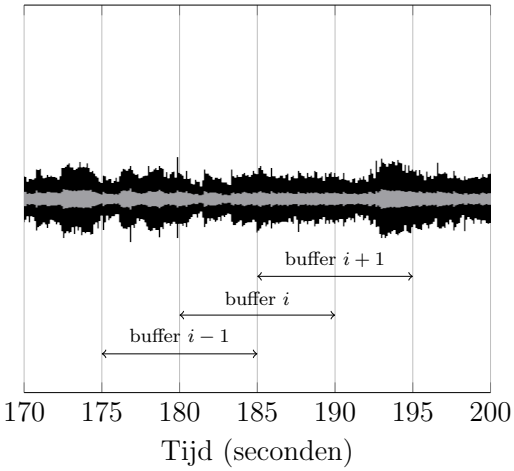
Deze implementatie kan een wijziging van de latency in het beste geval na 5 seconden detecteren. Wanneer er samples gedropt worden net voor het moment dat de buffer voor de helft gevuld is, dan kan het algoritme de nieuwe latency wel onmiddellijk detecteren.

Sliding window

Een meer doordachte manier van bufferen maakt gebruik van een *sliding window*. In onderstaande beschrijving zal ter illustratie ook gebruik gemaakt worden van een buffer met 10 seconden capaciteit. In de voorbeelden zal een stapgrootte van 5 seconden gehanteerd worden.

Het verschil met de naïeve methode is dat de buffer niet pas na 10 seconden wordt opgeschoven. Door de buffer al na 5 seconden op te schuiven zal een wijziging van de latency sneller gedetecteerd kunnen worden; dit terwijl het algoritme toch nog steeds tien seconden audio kan analyseren. In figuur 2.5 wordt grafisch weergegeven hoe de buffer precies verschoven wordt.

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



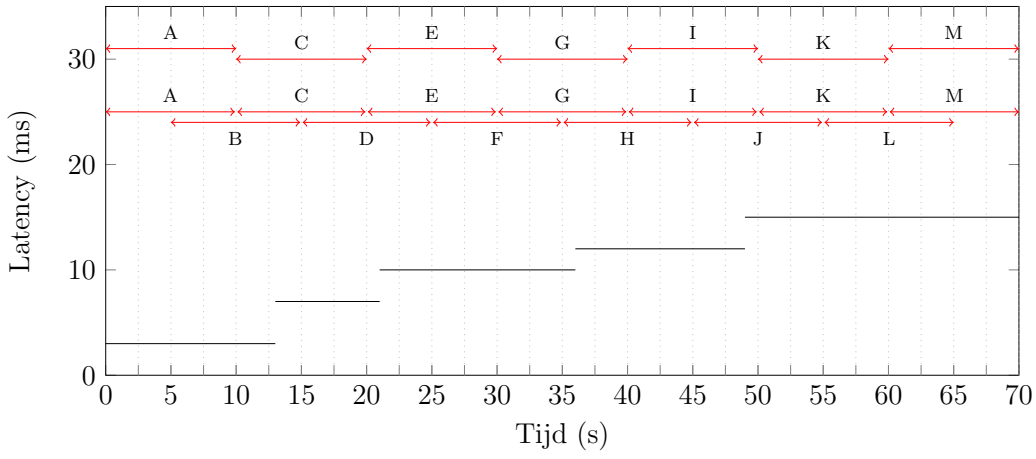
Door de buffer al na 5 seconden (de helft van de buffergrootte) op te schuiven wordt het slechtste geval sterk verbeterd. In het slechte geval wordt een wijziging van de latency gedetecteerd na 10 seconden. Het beste geval blijft wel nog steeds 5 seconden.

Door de stapgrootte nog verder te verkleinen kan het slechtste geval nog verbeterd worden. Aangezien het algoritme per hoeveelheid audio veel frequenter moet worden uitgevoerd heeft dit een negatieve invloed op de performantie.

Voorbeeld

Een praktisch voorbeeld zal bovenstaande beschrijving wat verduidelijken. In het voorbeeld worden twee audiostreams van 70 seconden geanalyseerd. Door het droppen van samples neemt de latency tussen de streams periodiek toe. Figuur 2.6 geeft in het zwart weer hoe de latency gedurende de verwerking evolueert. De opeenvolgende buffers van de twee besproken methode's worden in het rood aangeduid.

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.



De initiële latency van 3 milliseconden wordt zowel met de naïeve methode als met het sliding window gedetecteerd na de analyse van buffer A, 10 seconden na aanvang van de

analyse. De eerste verhoging tot 7 milliseconden vindt te laat plaats om gedetecteerd te kunnen worden door buffer B van de sliding window methode. De nieuwe latency kan wel gedetecteerd worden na het vollopen buffer C van beide methode's. Deze detectie gebeurt dus 7 seconden na de wijziging. De verhoging naar 10 milliseconden vindt net zoals de vorige wijzing net te laat plaats om vroeger gedetecteerd te kunnen worden bij de sliding window methode. De wijziging wordt bij beide methodes na 9 seconden gedetecteerd in buffer E. De derde verhoging kan net niet in buffer G gedetecteerd worden maar wel in buffer H. De sliding window methode levert na 9 seconden een resultaat, dit is beduidend beter dan de 14 seconden van de naïeve methode. De laatste wijziging wordt door de sliding window methode na 6 seconden gedetecteerd en door de naïeve methode na 11 seconden.

Conclusie

De detectiesnelheid van een latencywijziging hangt af van twee parameters: de bufferlengte (b) en de staplengte (s). Het beste geval (T_b) heeft volgende ondergrens:

$$T_b = b/2 \tag{2.2}$$

De beste mogelijke detectiesnelheid bij een buffer van 10 seconden is bijgevolg 5 seconden.

Het slechtste geval (T_s) is ook afhankelijk van de stapgrootte. De bovengrens wordt als volgt bepaalt:

$$T_s = b/2 + s \tag{2.3}$$

De slechts mogelijke detectiesnelheid bij de naïeve methode ($b = s = 10$) is dus 15 seconden. Bij een stapgrootte van 5 seconden is de detectiesnelheid begrensd tot 10 seconden.

Hoofdstuk 3

Implementatie

3.1 Technologieën en software

3.1.1 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 op een bepaalde manier verwerkt kan worden. Een processing pipeline wordt voorgesteld als instantie van de klasse `AudioDispatcher`. Het aanmaken gebeurt met behulp van de klasse `AudioDispatcherFactory`. Deze bevat statische methodes om een `AudioDispatcher` aan te maken van een audiobestand, een array van floating-point getallen of een microfoon. Een processing pipeline kan bewerkt of verwerkt worden met behulp van één of meerdere `AudioProcessors`. Een `AudioProcessor` is een interface met de methodes `process` en `processingFinished`. De `process` methode heeft als enige parameter een `AudioEvent`. Dit object bevat een audio blok, voorgesteld als array van floating-point getallen met waarden tussen -1.0 en 1.0. De grootte van dit blokje

audio, en de mate van overlapping tussen de opeenvolgende blokjes audio is instelbaar. Verder bevat dit object nog andere metadata zoals onder meer een *timestamp*.

Afhankelijk van de implementatie van de `process` methode kan de audiostroom op een bepaalde manier verwerkt, geanalyseerd of gewijzigd worden.

TarsosDSP bevat verder nog een groot aantal klassen met allerhande tools en audioverwerkings algoritmen. Het merendeel hiervan maakt gebruik van de zojuist beschreven processing pipeline.

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
- FFT transformaties

3.1.2 Panako

Panako is net zoals TarsosDSP een Java bibliotheek, door de zelfde auteurs ontwikkeld aan het IPEM. Panako bevat buiten implementaties van algoritmen ook enkele applicaties die hiervan gebruik maken. Deze bibliotheek wordt beschreven in artikel [21].

Panako bevat een open-source implementatie van het accoustic fingerprinting algoritme beschreven in de paper van Avery Li-Chun Wang[24]. Dit algoritme is verder uitgebreid zodat audio waarbij de toonhoogte verhoogd of verlaagd is, of audio die sneller of trager is afgespeeld, toch gedetecteerd kan worden.

Buiten het eigenlijke 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.

Er is ook een applicatie beschikbaar om verschillende geluidsfragmenten te synchroniseren. Deze applicatie maakt behalve van het accoustic fingerprinting algoritme ook nog gebruik van het kruiscovariantie algoritme.

Wanneer de latency tussen de verschillende audiofragmenten bepaald is, dan 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.

3.1.3 FFmpeg

FFmpeg is een command-line multimedia framework dat gebruikt wordt voor encoderen, decoderen, multiplexen, demultiplexen, streamen en afspelen van audio en video. [14]

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

3.1.4 SoX

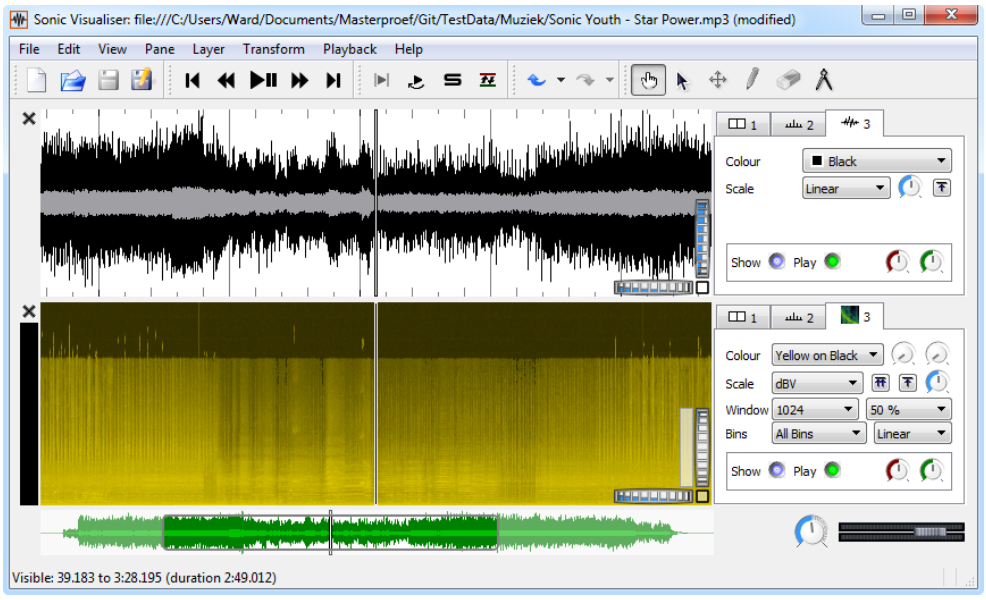
SoX is net zoals FFmeg een command-line tool voor audioverwerking. Buiten de mogelijkheid om audiobestanden te converteren laat SoX ook minder triviale operaties toe. Zo is het onder meer mogelijk om het volume aan te passen, effecten toe te voegen, de bestanden bij te knippen of gegenereerde geluiden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

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

3.1.5 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 waveform als het spectrogram van een audiobestand gevisualiseerd worden. Sonic Visualiser is uitbreidbaar met plug-ins in het Vamp formaat. [9]

Figuur 3.1: De gebruikersinterface van Sonic Visualiser

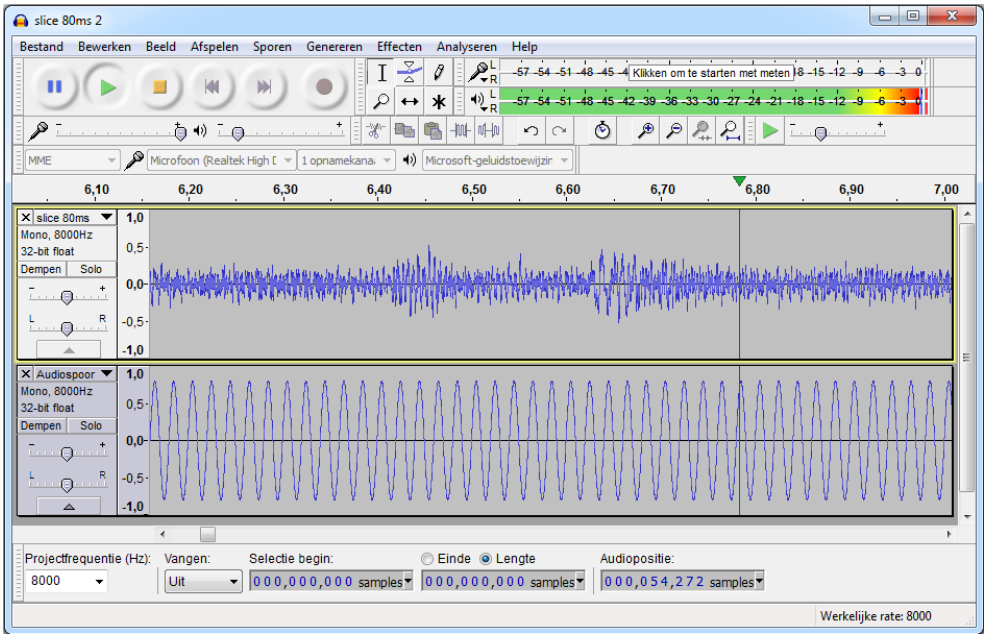


Sonic visualiser werd in dit onderzoek vooral gebruikt om handmatig de latency tussen verschillende audiofragmenten te bepalen. Ook heeft de applicatie dienst gedaan als educatieve tool om verschillende audioverwerkingsalgoritmen visueel voor te stellen.

3.1.6 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.[3]

Figuur 3.2: De gebruikersinterface van Audacity

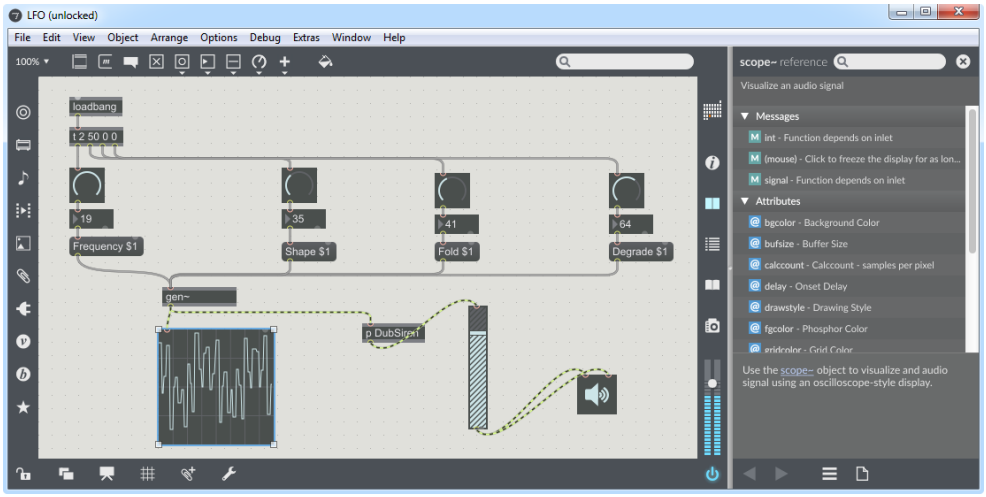


Alle opnames en handmatige bewerkingen op audiobestanden in dit onderzoek zijn uitgevoerd met behulp van Audacity.

3.1.7 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. [4]

Figuur 3.3: De gebruikersinterface van Max/MSP: een *patch panel* met daarop enkele modules die samen een complexere toepassing vormen.



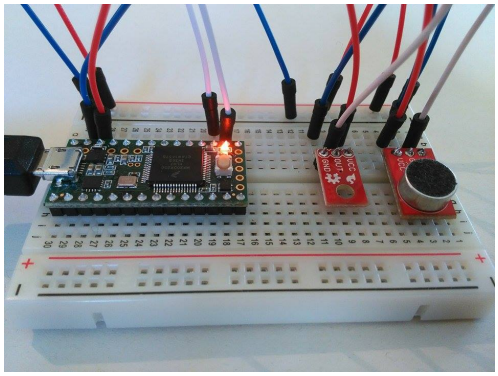
Met Max/MSP is het mogelijk om realtime audio verwerken, daarom zal deze toepassing gebruikt worden voor het ontwikkelen van de gebruikersinterface.

3.1.8 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. [6]

De sensoren die gebruikt worden bij de experimenten van het IPEM zijn meestal aangesloten op Teensy microcontrollers. Om de synchronisatiealgoritmes en het bijhorende systeem in een representatieve situatie te testen zal daarom ook gebruik gemaakt worden van een Teensy microcontroller.

Figuur 3.4: De Teensy microcontroller verbonden met een infraroodsensor en microfoon op een breadboard.



In hoofdstuk 4 wordt deze testopstelling meer in detail besproken.

3.2 Accoustic fingerprinting

De Panako softwarebibliotheek bevat een zeer goede implementatie van het accoustic fingerprinting algoritme. Om wijzigingen mogelijk te maken hebben is de code van het algoritme overgenomen in het project van dit onderzoek. De overgenomen 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 we in onze toepassing enkel geïnteresseerd zijn in de beste offsetwaarde is dit overbodig. De beste offset en bijhorende fingerprints wordt apart bijgehouden. De naverwerking wordt hierdoor vermeden.

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.

`NFFT_STEP_SIZE`

Dit is het aantal samples elke verschuiving in het FFT algoritme. De stepsize beïnvloedt rechtstreeks de nauwkeurigheid van het accoustic fingerprinting algoritme. Wanneer deze parameter is ingesteld op 128 samples en de samplefrequentie 8000hz bedraagt 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 fingerprint matches 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. Bij een hoge waarde moeten meer berekeningen worden uitgevoerd, dit heeft 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

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

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 accoustic fingerprinting algoritme is er een groot verschil tussen de meest "elegante" parameterwaarden en de in de praktisch 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 ingesteld 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*.

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 retourneert 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 lage waarde te geven. 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 maximaal 50 fingerprints per event point degelijke resultaten oplevert.

`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 20 in deze toepassing de beste resultaten zonder de performantie sterk te beperken.

3.3 Kruiscovariantie

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

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

3.3.1 Optimalisaties

3.3.2 Parameters en hun invloed op het algoritme

3.4 Structuur softwarebibliotheek

3.5 Implementatie van een Max/MSP module

Hoofdstuk 4

Evaluatie

4.1 Unit testen

4.2 Stresstesten

4.3 Test in de praktijk

4.4 Usability testen

4.5 Analyse van de complexiteitsgraad

4.6 Praktische bruikbaarheid van het systeem

Hoofdstuk 5

Conclusie

Appendices

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 8000 Hz. 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 [11]. 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 gemiddelde 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.

Referentielijst

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

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

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

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

[5] Dictionary.com unabridged. Mar 2016. URL http://www.dictionary.com/browse/sound-spectrogram.

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

[10] 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.

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

[12] 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.

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

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

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

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

[17] 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.

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

[19] Joren Six. Pitch, pitch interval and pitch ratio representation. Technical report, 2011.

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

Lijst van figuren

Lijst van tabellen

Lijst van codefragmenten