Aalto-yliopisto
Sähkötekniikan korkeakoulu

12.12.2022

ELEC-C5341
Äänen- ja puheenkäsittely

Pitch Shifter -ääniefekti

Artur Brander, 894973

Jaakko Karhu, 792431

Pessi Lyytikäinen, 792376

Peter Parviainen, 830173

Tiivistelmä

Aiheenamme on pitch shifter -ääniefekti, jolla muutetaan äänen sävelkorkeutta. Tarkoituksemme on käyttää sitä lähtökohtaisesti kitaran äänisignaaliin. Aluksi lähestymme ongelmaa rengaspuskurin avulla ja tuloksena on Matlabissa toimiva algoritmi. Ongelmaksi muodostuu kuitenkin epäjatkuvuuskohdat, jolloin muokattuun ääneen syntyy “poksahduksia” (englanniksi “pops”).

Lopulliseksi ratkaisuksi syntyy toteutus kokonaan erilaisella algoritmilla: vaihevokooderilla. Vaihevokooderi poistaa signaalin epäjatkuvuuskohdat ja sen äänenlaatu on kiitettävä. Lopputulos on kahden oktaavin välillä toimiva, useita erilaisia musikaalisia efektejä sisältävä ääntä toistava ohjelma.

1 JOHDANTO

Pitch shifter eli äänenkorkeuden muunnin on nimensä mukaisesti ääniefekti, jolla äänenkorkeutta muutetaan. Äänenkorkeuden muuntajaa käytetään paljon erilaisissa sovelluksissa esimerkiksi musiikissa ja puheenkäsittelyssä. Äänenkorkeutta voi muuttaa digitaalisesti monella eri tekniikalla, joilla on hyvät ja huonot puolensa. Tutustuimme projektia tehdessä muutamaan eri tapaan, joita kuvaamme raportissa ennen kuin saimme toimivan ratkaisun vaihevokooderilla, joka on kuvattu kokonaisuudessaan. Tavoitteenamme oli saada toteutettua sävelkorkeuden muunnin, joka toimii reaaliaikaisesti ja säilyttää äänenlaadun hyvänä.

2 METODIT

Alussa pähkäilimme ja tutustuimme kurssimateriaaleihin, jotka käsittelivät sävelkorkeuden muuttamista. Ensimmäiseksi toteutustavaksi valikoitui rengaspuskuri kurssin projektimateriaaleista (linkki vaatii kurssille ilmoittautumisen).

Rengaspuskuri on tietorakenne, joka käyttää yhtä kiinteän kokoista puskuria ikään kuin se olisi kytketty päästä päähän ja sen rakenne soveltuu helposti tietovirtojen puskurointiin. Tämä oli entuudestaan tuttu yhden tämän kurssin viikkotehtävän ansiosta ja päädyimme suunnilleen samanlaiseen lopputulokseen. Rengaspuskurissa on kaksi sen ympäri kiertävää “päätä”: kirjoittava pää, joka tallentaa ohjelmaan luetun signaalin arvoja rengaspuskuriin ja lukeva pää, joka vastaavasti lukee näitä arvoja. Kirjoittava pää kulkee aina tietyllä normaalilla nopeudella.

Jos sävelkorkeutta halutaan nostaa, rengaspuskurin alkioita luetaan nopeammalla lukunopeudella. Sävelkorkeuden madaltaminen tehdään vastakkaisesti hidastamalla lukemisnopeutta.

Sävelkorkeuden muuttaminen onnistui, mutta muunnetussa äänessä oli luonnotonta rätinää, jotka johtuivat tuotettuun äänisignaaliin syntyvistä epäjatkuvuuskohdista. Yritimme muokata rengaspuskurin tuottamaa ääntä jatkuvaksi, mutta emme lopulta saaneet poksahduksia pois myöskään kahden rengaspuskurin yhdistelmällä.

Tässä vaiheessa etenimme projektisuunnitelman mukaan: halusimme aluksi tehdä Matlab-ohjelmointiympäristössä toimivan toteutuksen ennen kuin siirtäisimme sen JUCE-sovelluskehykseen, joka käyttää C++ -ohjelmointikieltä. Oivalsimme, että rengaspuskurit eivät ole meille optimaalinen vaihtoehto, joten projektisuunnitelman mukaisesti meille tulisi kiire uuden menetelmän uupuessa. Uusi menetelmä löytyi onneksi pian.

Tämä menetelmä oli kehystämisen ja ikkunoinnin yhdistelmä, joka toimi vain tietyillä taajuuksilla. Tämä oli hyvä edistys vaihevokooderin suuntaan, koska lopullisessa toteutuksessa tultaisiin hyödyntämään samoja asioita. Tämä on siis eräänlainen toteutus OLA-menetelmästä (overlap-add method), jossa kehykset ja ikkunointi ovat oleellisia tekijöitä: Käytännössä kehykset ikkunoidaan ja kasataan päällekkäin, jonka jälkeen ne kompressoidaan oikean

pituiseksi.

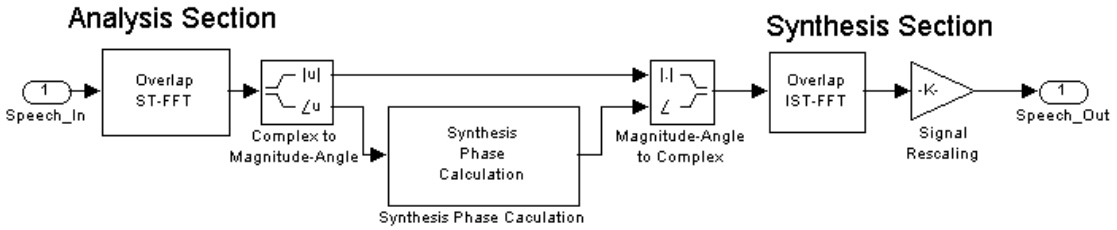
Emme olleet vieläkään tyytyväisiä lopputulokseen, koska tietyn suuruiset muunnokset eivät kuulostaneet hyvältä. Lopulta löysimme projektiassistentin avustuksella hyvältä kuulostavan demonstraation vaihevokooderista, joka sopisi täydellisesti tarkoituksiimme: DSP Final Project Phase Vocoder (Youtube), 2022.

Vaihevokooderi käydään vielä tarkemmin läpi toteutus-osiossa, mutta yksinkertaisesti se on yhdistelmä OLA-menetelmää, STFT:tä (Short-Time Fourier Transform) ja vaiheprosessointia.

Projektin aikana tutustuimme myös musiikin teoriaan ja erilaisiin kitaraefekteihin. Musiikin teoriassa pääpainona oli se, miten 12-säveljärjestelmäinen tasaviritys toimii ja miten eri nuottien taajuudet suhtautuvat toisiinsa.

3 TOTEUTUS

Lopullinen toimiva toteutus on suoritettu Matlab-ohjelmointiympäristössä käyttämällä vaihevokooderia. Vaihevokooderi on algoritmi, joka hyödyntää signaalin taajuustason vaiheinformaatiota (Sethares; William, A.): Taajuustasossa tehdään signaaliin muutoksia, joilla voidaan joko venyttää tai kompressoida digitaalisen äänitiedoston kokoa siten, että epäjatkuvuuskohtia ei synny. Vaiheinformaatiota käytetään siten, että muunnetussa äänitiedostossa on samat vaihemuutokset kuin alkuperäisessä signaalissa suhteutettuna uuteen suurempaan tai pienempään kokoon. Systeemin lohkokaavio on esitetty kuvassa 1. Kuvassa on käytetty tulo- ja lähtösignaalina puhetta, mutta sillä ei ole merkitystä. Vaihevokooderi koostuu kahdesta sektiosta, joita kutsutaan analyysi- ja synteesisektioksi.



Kuva 1. Vaihevokooderin lohkokaavio toteutettuna Matlabissa
A. D. Gotzen, N. Bernardini and D. Arfib, 2000
https://se.mathworks.com/help/audio/ug/pitch-shifting-and-time-dilation-using-a-phase-vocoder-in-matlab.html;jsessionid=ba698c7111a5b2e711d6cc6d981a

Alussa tarvitaan tieto siitä, kuinka paljon alkuperäisen signaalin sävelkorkeutta halutaan muuttaa. Tämä hoituu musiikin teoriasta löytyvällä 12-säveljärjestelmällä, kuten se on selitetty Wikipediassa: Equal Temperament, kohdasta “Mathematics”.

Soittimien sävelet voidaan määrittää monella tavalla, mutta kuten samassa kohtaa sanotaan, tämä tapa tuottaa yhtä hyviä (tai huonoja) tuloksia eri asteikoilla soitettaessa. Yksi oktaavi jaetaan 12 säveleen, joiden välillä on aina sama kerroin. Sävelten ollessa oktaavin päässä toisistaan, on niistä pienempi taajuudeltaan kaksi kertaa niin iso kuin suurempi. Tästä saadaan kaava

$$P_n = P_a (2)^{n/12}, \tag{1}$$

missä n on puolisävelaskeleiden määrä, P_n on muunnetun säveltaajus hertseinä, P_a on sävelkorkeus hertseinä, joka muutetaan uudeksi. n:n ollessa negatiivinen muunnettu säveltaajuus on pienempi, eli sävel madaltuu, kun taas n:n ollessa positiivinen taajuus nousee aiheuttaen korkeampaa säveltä. Vaihevokooderi luo termistä $(2)^{n/12}$ alfa-kertoimen kaavalla

$$\alpha(n) = (2)^{n/12},$$

missä n on muunnetun puolisävelaskeleiden muutoksen suuruus. Tämän käyttö on selitetty alempana.

Vaihevokooderin ensimmäinen prosessointivaihe on kehysten luonti. Tässä vaiheessa tarvitaan kehyksille ja ikkunoille koko. Ikkunafunktiona käytetään Matlabin hann-funktiota. Hann-ikkunan amplitudi on suurin ikkunan keskellä ja reunoissa se laskee nollaan. Kehyksiin jaetun signaalin ikkunointi on välttämätöntä jotta saadaan pehmennettyä kehysten välille syntyviä epäjatkuvuuskohtia. Testasimme eri ikkunakokoja ja valitsimme niistä parhaimman kuuloiset. Hann-ikkunan koko osoittautui olevan parhaimmillaan 1024, analyysi-ikkunan 256 ja synteesi-ikkunan koko saadaan kertomalla analyysi-ikkunan koko alfa-kertoimella (1). Täytetään kehykset alkuperäisellä ääninäytteillä peräkkäin siten, että ne yhdistyvät toisiinsa 75 prosenttisesti. Analyysi-ikkunan koko on 25 prosenttia ikkunan koosta, joten tämän avulla hyppäys 75 prosenttiseen päällekkäisyyteen onnistuu.

Tässä vaiheessa on aika siirtyä taajuustasoon STFT:n avulla. (Driedger, J; Meinard, M. 2016. *A Review Of Time-Scale Modification Of Music Signals)* STFT tarkoittaa Fourier-muunnosta, joka otetaan tietyn kokoisista kehyksistä. Signaalille *x* saadaan STFT-muunnos,

$$X(m,k) = \sum_{r=-N/2}^{N/2-1} x_m(r)\ w(r)\ \exp(-2\pi ikr/N) \tag{2}$$

jossa *m* on kehyksen indeksi, *k* on taajuusindeksi, *N* on kehyksen pituus, x_m kehyksen indeksin mukainen kehys ja *w* on ikkunafunktio. Hyödynsimme Matlabin FFT-funktiota tämän toteuttamiseen: kerroimme aluksi kehyksen ikkunafunktiolla ennen FFT:n suorittamista.

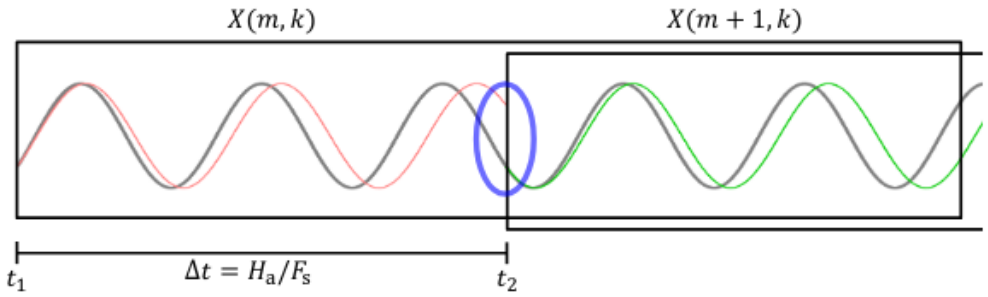
Taajuustasossa tavoitteena on selvittää jokaisen kehyksen todelliset taajuuskomponentit kun otetaan huomioon edellisen kehyksen vaihe ja taajuus. Kuva 2 havainnollistaa tätä esittämällä kaksi peräkkäistä kehystä joissa samantaajuiset sinisignaalit (vihreä ja punainen) ovat eri vaiheessa kehysten rajalla. Kuvassa 3 havainnollistetaan näiden kahden signaalin vaihe-eroa hetkellä t_2. STFT:n avulla saadaan kehyksen taajuudet, jonka avulla puolestaan voidaan tehdä arvio vaiheen muutoksesta kehyksen aikana. (Driedger, J; Meinard, M. 2016. *A Review Of Time-Scale Modification Of Music Signals)* Vertaamalla havaittua vaihetta arvioituun vaiheeseen saadaan laskettua näiden ero. Peräkkäisten kehysten vaihe-eron avulla saadaan selville oikeat taajuuskomponentit.(Nagapuri, Srinivas; Mohith, Amara; Puli, Kishore, Kumar; *Implementation of pitch shifter using Phase Vocoder Algorithm on Artix-7 FPGA*). Kuvassa 2 harmaalla on esitetty sinisignaali, jolla on oikea taajuus ja kehysten välillä ei ole epäjatkuvuutta. Modifioitu signaali taajuustasossa saadaan kaavasta,

$$X^{\mathrm{Mod}}(m,k) = |X(m,k)|\ \exp(2\pi i\ \varphi^{\mathrm{Mod}}(m,k)) \tag{3}$$

jossa X^{Mod} on vaiheiden avulla modifioitu STFT-kehys ja φ^{Mod} on modifioitu vaihe.

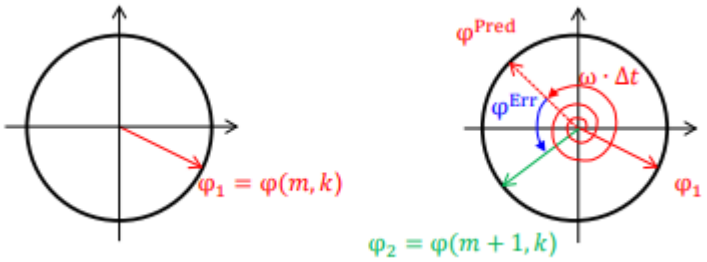
Kun signaalin vaiheet on modifioitu oikeiksi taajuustasossa, täytyy signaali muuttaa aikatasoon. ISTFT suoritettiin Matlabin IFFT-funktiolla. Aikatasoon muuttaminen tapahtuu käyttämällä kaavaa (3) ja ottamalla siitä ISTFT kaavan mukaisesti:

$$x_m^{\mathrm{Mod}}(r) = \frac{1}{N}\sum_{k=0}^{N-1} X^{\mathrm{Mod}}(m,k)\ \exp(2\pi ikr/N) \tag{4}$$



Kuva 2. Edellisen (punainen) ja nykyisen (vihreä) kehyksen signaalit. Sinisellä merkityn kohdan alla havaitaan epäjatkuvuuskohta kehysten välillä. Harmaa signaali kuvaa lopputulosta, jossa ei havaita epäjatkuvuuskohtia.

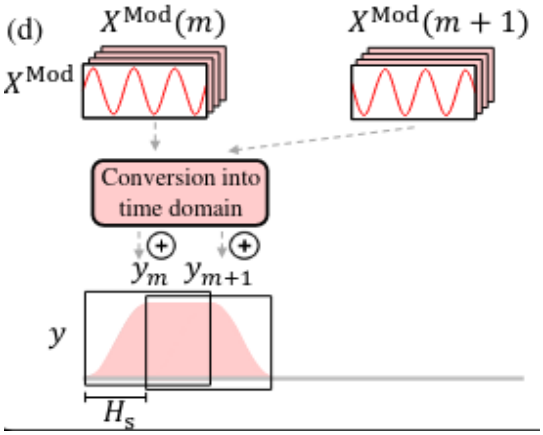
Driedger, J; Meinard, M. 2016. *A Review Of Time-Scale Modification Of Music Signals*



Kuva 3. Vasemmanpuoleisessa ympyrässä on kuvan 1 punaisen sinisignaalin vaihe hetkellä t_1. Oikeanpuoleisessa ympyrässä punaisella merkitty vaihe-estimaatti φ_{pred} , joka on arvio punaisen sinisignaalin vaiheesta hetkellä t_2. Vihreällä merkitty φ_2 on seuraavan kehyksen vaihe hetkellä t_2 ja sinisellä kehysten välinen vaihe-ero.

Driedger, J; Meinard, M. 2016. *A Review Of Time-Scale Modification Of Music Signals*

ISTFT:n jälkeen suoritetaan Overlap-Add, jolla tarkoitetaan kahden kehyksen yhdistämistä siten, että tietyt komponentit summataan päällekkäin. Tämä prosessi havainnollistetaan kuvassa 3. Tämän avulla saadaan joko kompressoitua tai pidennettyä ääninäytteitä, jotta niiden sävelkorkeutta voidaan muuttaa. Seuraavassa vaiheessa ne muokataan oikean pituisiksi. Kuvassa näkyvä H_s tarkoittaa synteesi-ikkunan kokoa.



Kuva 3. Signaalin rekonstruktio taajuustasossa prosessoidusta kehyksestä. Kuvan alaosassa toteutetaan overlap-add.

Driedger, J; Meinard, M. 2016. *A Review Of Time-Scale Modification Of Music Signals*

Overlap-addin avulla muodostuneesta taulukosta halutaan samanpituinen kuin alkuperäinen äänitiedosto. Tämä suoritetaan lineaarisella interpolaatiolla. Lineaarinen interpolointi on menetelmä käyrän sovittamiseksi käyttämällä lineaarisia polynomeja uusien näytepisteiden muodostamiseksi tunnettujen näytepisteiden alueella. Tähän löytyi erinomainen funktio Matlabista nimeltään *interpl*. Lineaarisen interpolaation jälkeen saadaan lopullinen muunnettu äänitiedosto, jonka Matlab-koodi palauttaa kuunneltavaksi.

4 TULOKSET

Lopullisen efektin äänenlaatu on todella hyvä. Esimerkiksi kitaran yksittäisen sävelen ääni muunnettuna toiseen säveleen on mahdoton erottaa alkuperäisestä kitarasta, joka soittaisi samaa säveltä, kunhan muunnettu sävel ei ole liian kaukana muunnettavasta sävelestä.

Tarkemmin sanottuna muunnettu ääni kuulostaa aidolta ylös- sekä alaspäin kahdeksan puolisävelaskelta. Tämän ulkopuolella ääni on vielä oktaaviin asti ylös- ja alaspäin hyvälaatuinen, mutta vaihevokooderi ei muuta täydellisesti kaikkia taajuuksia tuottaen epätäydellistä ääntä. Tuloksena on lähempänä tunnistettavan epäaidolta kuulostava ääni ja kahden oktaavin muunnoksen jälkeen ääni, jonka sävel ei ole enää tunnistettavissa lainkaan.

Äänenlaatua arvioitiin esimerkiksi soittamalla sähkökitaran matalinta kieltä ja muuntamalla siitä saadun äänen oktaavilla alaspäin toteutuksellamme. Tulokseksi saatiin ääni, joka kuulosti sähköbassokitaralta.

Toinen äänenlaadun arviointitapa oli ensin soittaa vapaata kitaran kieltä, joka muunnettaisiin korkeammaksi vaihevokooderin algoritmilla. Tämän jälkeen pitämällä kitaran kaulalta kieltä lyhentäen sitä saaden korkeampi ääni soitettaessa. Tämän äänen ja vaihevokooderin tuottamaan äänen ero ei ole huomattavissa ja värähtelevä kappale on sama.

Näillä kahdella esimerkillä ääni kuulostaa aiemmin mainitulla kuudentoista puolisävelaskeleen välillä täysin aidolta sekä kahteen oktaaviin asti silti laadultaan tarpeeksi hyvältä, että niitä pystyy käyttämään erilaisiin efekteihin.

Toteutus ei ole reaaliaikainen. Jos esimerkiksi halutaan saada bassokitaran ääntä sähkökitaralla ja soittaa muiden kanssa, ei se ole mahdollista nykyisellä toteutuksella.

5 JOHTOPÄÄTÖKSET

Työ onnistui kokonaisuutena hyvin ottaen huomioon sen, että aloitimme aikatason rengaspuskuri-toteutuksella ja lopulta siirryimme taajuustason vaihevokooderiin kiristäen aikataulua. Kiireestä huolimatta saimme toimivan toteutuksen, jolla äänenkorkeutta saadaan muutettua halutun arvon mukaan säilyttäen äänen hyvälaatuisena. Aluksi vaikeaa ja uutta oli vaihevokooderin toteuttamiseen vaadittava teoria, mutta lähteistä löytyneen hyvän tiedon ja kaavojen avulla sen pystyi oppimaan.

Kun koitimme siirtää toteutusta reaaliaikaiseksi JUCE-ympäristössä huomasimme, että taidot sekä aika alkoivat loppumaan. Saimme kuitenkin JUCE:ssa tehtyä rengaspuskuriin perustuvan äänenkorkeutta muuntavan ääniliitännäisen (vrt. “plugin”), jota testasimme kitaralla reaaliaikaisesti. Tulos toteutuksessa oli kuitenkin äänenlaadultaan huono ja vääristi erilaisia taajuuksia tuottaen huonon äänen.

Tulevaisuudessa työtä voisi jatkaa tekemällä äänenlaadultaan hyvän ääniliitännäisen, joka toimisi reaaliaikaisesti. Liitännäisessä olisi selkeä ja käyttäjäystävällinen käyttöliittymä, jossa voisi säätää kuinka monta puolisävelaskelta ääntä halutaan nostaa tai laskea. Lisäksi käyttäjä voisi valita erityylisiä efektejä, kuten harmonisoijan tai arpeggiaattorin. Tämän toteutuksen voisi tehdä Matlabissa Audio Toolboxilla, JUCE:lla tai vaikkapa oman Python-ohjelman.

Koska reaaliaikaista toteutusta ei tehty, jäi myös jotain kysymyksiä vastaamatta: Riittääkö käytetyn ohjelman, esimerkiksi Matlabin, laskentateho tekemään sekunnissa useita

vaadittuja laskentoja äänisignaalille?

Esimerkiksi minimiviivellä 25ms täytyy iteraatioita tehdä sekunnissa vähintään 40. Neljälläkymmenellä iteraatiolla viive ei ikinä voisi olla 25ms, vaan 25ms äänittämisen jälkeen tehty laskenta aiheuttaisi lisää viivettä.

Nykyisessä koodissa puolenkaan sekunnin viive ei haittaa toteutusta merkittävästi, mutta reaaliaikaisena muiden kanssa soitettuna puolen sekunnin viive on selvästi liikaa. Yleisesti puhuttu maksimiviive soittamisessa on 30ms.

Muutamien testien perusteella ohjelmalla kestää ajamisessa pari sekuntia tehdä noin 20 muunnosta, joten hyppy reaaliaikaan ei ole välttämättä mahdollista nykyisen toteutuksen jatkamisella. Nämä muunnokset ovat kuitenkin pidemmälle ääninäytteille: Noin 5s eli 44kHz näytteenottotaajuudella tarkoittaa aikaa 220 000 näytteen äänisignaalin laskemiselle, kun taas esimerkiksi raaliaikaisen toteutuksen 25ms äänisignaalissa on vain 1 100 näytettä.

Tällöin vertailu ei ole täysin realistinen ja tarkempaa testausta tarvitaan siihen, riittääkö nykyisen koodin tehokkuus reaaliaikaistukseen.

LÄHTEET

pitchshifter.pdf projektimateriaalit-kansiossa MyCoursesissa: luettu 22.11.2022

<URL:Course: ELEC-C5341 - Äänen- ja puheenkäsittely, Luento-opetus, 5.9.2022-5.12.2022, Topic: Projektimateriaalit (aalto.fi)>.

Ian Greene. 20.04.2019. DSP Final Project Phase Vocoder (Youtube). 2min58s
<URL:https://www.youtube.com/watch?v=h5o7jeddU0c>

Driedger, J; Meinard, M. 2016. *A Review Of Time-Scale Modification Of Music Signals*

Sethares; William, A. A Phase Vocoder in Matlab. luettu 13.12.2022

<URL:https://sethares.engr.wisc.edu/vocoders/phasevocoder.html>

Wikipedia. Equal Temperament. luettu (viimeisen kerran) 13.12.2022

<URL:https://en.wikipedia.org/wiki/Equal_temperament>

Nagapuri, Srinivas; Mohith, Amara; Puli, Kishore, Kumar; Implementation of pitch shifter using Phase Vocoder Algorithm on Artix-7 FPGA. *International Journal of Control Theory and Applications Volume 10, Number 6. Department of Electronics and Communication Engineering National Institute of Technology Patna, Bihar, India 2017*

<URL: https://serialsjournals.com/abstract/70898_67-nagapuri_srinivas.pdf>

Code

DemoApp.mlapp

```
classdef sasp2022DemoApp < matlab.apps.AppBase
    % Properties that correspond to app components
    properties (Access = public)
        UIFigure            matlab.ui.Figure
                        KyttohjeetvalitseasetusnauohitaSsekkuuntelenauhoitusLabel
matlab.ui.control.Label
        SASP2022DemoLabel   matlab.ui.control.Label
        NormaaliButton      matlab.ui.control.Button
        ArpeggioButton      matlab.ui.control.Button
        MolliButton         matlab.ui.control.Button
        DuuriButton         matlab.ui.control.Button
        BassoButton         matlab.ui.control.Button
        Lamp                matlab.ui.control.Lamp
        OctaverButton       matlab.ui.control.Button
    end
    % Callbacks that handle component events
    methods (Access = private)
        % Button pushed function: OctaverButton
        function OctaverButtonPushed(app, event)
        app.Lamp.Color = [0 1 0];
        run('testausta01_octaver.m')
        pause(5)
        app.Lamp.Color = [1 0 0];

        end
        % Button pushed function: BassoButton
        function BassoButtonPushed(app, event)
        app.Lamp.Color = [0 1 0];
        run('testausta02_basso.m')
        pause(5)
        app.Lamp.Color = [1 0 0];
```

```
        end
        % Button pushed function: DuuriButton
        function DuuriButtonPushed(app, event)
        app.Lamp.Color = [0 1 0];
        run('testausta03_major.m')
        pause(5)
        app.Lamp.Color = [1 0 0];
        end
        % Button pushed function: MolliButton
        function MolliButtonPushed(app, event)
        app.Lamp.Color = [0 1 0];
        run('testausta04_minor.m')
        pause(5)
        app.Lamp.Color = [1 0 0];
        end
        % Button pushed function: ArpeggioButton
        function ArpeggioButtonPushed(app, event)
        app.Lamp.Color = [0 1 0];
        run('testausta05_majorarpeg.m')
        pause(5)
        app.Lamp.Color = [1 0 0];

        end
        % Button pushed function: NormaaliButton
        function NormaaliButtonPushed(app, event)
        app.Lamp.Color = [0 1 0];
        run('testausta06_normaali.m')
        pause(5)
        app.Lamp.Color = [1 0 0];
        end
    end
    % Component initialization
    methods (Access = private)
        % Create UIFigure and components
        function createComponents(app)
            % Create UIFigure and hide until all components are created
            app.UIFigure = uifigure('Visible', 'off');
            app.UIFigure.Position = [100 100 523 272];
```

```
            app.UIFigure.Name = 'MATLAB App';
            % Create OctaverButton
            app.OctaverButton = uibutton(app.UIFigure, 'push');
                  app.OctaverButton.ButtonPushedFcn = createCallbackFcn(app,
@OctaverButtonPushed, true);
            app.OctaverButton.Position = [55 126 100 23];
            app.OctaverButton.Text = 'Octaver';
            % Create Lamp
            app.Lamp = uilamp(app.UIFigure);
            app.Lamp.Position = [426 188 20 20];
            app.Lamp.Color = [1 0 0];
            % Create BassoButton
            app.BassoButton = uibutton(app.UIFigure, 'push');
                    app.BassoButton.ButtonPushedFcn = createCallbackFcn(app,
@BassoButtonPushed, true);
            app.BassoButton.Position = [54 64 100 23];
            app.BassoButton.Text = 'Basso';
            % Create DuuriButton
            app.DuuriButton = uibutton(app.UIFigure, 'push');
                    app.DuuriButton.ButtonPushedFcn = createCallbackFcn(app,
@DuuriButtonPushed, true);
            app.DuuriButton.Position = [209 126 100 23];
            app.DuuriButton.Text = 'Duuri';
            % Create MolliButton
            app.MolliButton = uibutton(app.UIFigure, 'push');
                    app.MolliButton.ButtonPushedFcn = createCallbackFcn(app,
@MolliButtonPushed, true);
            app.MolliButton.Position = [208 64 100 23];
            app.MolliButton.Text = 'Molli';
            % Create ArpeggioButton
            app.ArpeggioButton = uibutton(app.UIFigure, 'push');
                  app.ArpeggioButton.ButtonPushedFcn = createCallbackFcn(app,
@ArpeggioButtonPushed, true);
            app.ArpeggioButton.Position = [371 126 100 23];
            app.ArpeggioButton.Text = 'Arpeggio';
            % Create NormaaliButton
            app.NormaaliButton = uibutton(app.UIFigure, 'push');
                  app.NormaaliButton.ButtonPushedFcn = createCallbackFcn(app,
@NormaaliButtonPushed, true);
            app.NormaaliButton.Position = [372 64 100 23];
            app.NormaaliButton.Text = 'Normaali';
```

```
            % Create SASP2022DemoLabel
            app.SASP2022DemoLabel = uilabel(app.UIFigure);
            app.SASP2022DemoLabel.Position = [19 1 102 22];
            app.SASP2022DemoLabel.Text = 'SASP 2022 Demo';
            % Create KyttohjeetvalitseasetusnauohitaSsekkuuntelenauhoitusLabel
               app.KyttohjeetvalitseasetusnauohitaSsekkuuntelenauhoitusLabel =
uilabel(app.UIFigure);

app.KyttohjeetvalitseasetusnauohitaSsekkuuntelenauhoitusLabel.Position  = [51
174 375 48];
            app.KyttohjeetvalitseasetusnauohitaSsekkuuntelenauhoitusLabel.Text
= 'Käyttöohjeet: valitse asetus -> nauohita 5 sek -> kuuntele nauhoitus';
            % Show the figure after all components are created
            app.UIFigure.Visible = 'on';
        end
    end
    % App creation and deletion
    methods (Access = public)
        % Construct app
        function app = sasp2022DemoApp
            % Create UIFigure and components
            createComponents(app)
            % Register the app with App Designer
            registerApp(app, app.UIFigure)
            if nargout == 0
                clear app
            end
        end
        % Code that executes before app deletion
        function delete(app)
            % Delete UIFigure when app is deleted
            delete(app.UIFigure)
        end
    end
end
```

DSP functionalities

```
function [audio_converted] = phase_vocoder(audio, halfSteps)
%% Alustus
% Luetaan audio sisään
%audio = audio(:,1);      % Stereotiedostosta mono (toimii myös monon kanssa)
pitch    =    2^(halfSteps/12);%    Pitch-kerroin    (määrittää    lopullisen
sävelkorkeuden)
windowSize = 1024;                          % Ikkunan koko
analysisHopSize = 256;                      % Analyysi-ikkunan koko
synthHopSize = round(pitch*analysisHopSize); % Synteesi-ikkunan koko
window = hann(windowSize);                  % Ikkunan alustus
%% Kehysten luonti
% Luodaan kehyksille oma taulukko "frames". numOfFramesCutted kertoo, kuinka
% monta kehysleikkausta saadaan audiotiedostosta.
numOfFramesCutted = floor((length(audio)-windowSize)/analysisHopSize);
frames = zeros(floor(length(audio)/analysisHopSize),windowSize);
frameSizes = size(frames);
numOfFrames = frameSizes(1);
% Täytetään kehykset  alkuperäisellä audiolla peräkkäin siten,
% että ne overlappaa toisensa 75 prosenttisesti. (Tämä prosenttiluku
% osoittautui parhaimmaksi lähteiden mukaan.)
% Esimerkkinä:
% ---1-2-3-4
% -----2-3-4-5
% -------3-4-5-6
% jne.
for i = 1:numOfFramesCutted
   start = 1 + (i - 1) * analysisHopSize;
   finish = (i - 1) * analysisHopSize + windowSize;
   frames(i,:) = audio(start:finish);
end
% Parametrit vaihevokooderia varten
prevPhase = 0;
predictedPhase = 0;
frequencyBins = 2*pi*(0:(windowSize-1)) / windowSize;
%% STFT & Vaiheiden muokkaus vaihevokooderin mukaisesti
for i=1:numOfFramesCutted
   % Tässä vaiheessa suoritetaan STFT kaikille kehyksille:
   fftFrame = fft(frames(i,:) .* window');
```

```
    % Nykyinen vaihe Matlabin valmiilla funktiolla:
    currentPhase = angle(fftFrame);

    % Vaihe-ero nykyisen ja edellisen vaiheen kanssa
    % Päivitetään samalla edellinen vaihe
    deltaPhi = currentPhase - prevPhase;
    prevPhase = currentPhase;

    % Edellisessä kohdassa lasketut vaiheet on kuvattu välille
    % [-pi, pi] angle-funktion takia, joten muokataan välit oikeaksi
    deltaPhiTrue = deltaPhi - analysisHopSize * frequencyBins;

    % Tämän jälkeen uudelleenwrapataan ja tulkitaan taajuus
    deltaPhiWrapped = mod(deltaPhiTrue+pi, 2*pi) - pi;
    predictedFreq = frequencyBins + deltaPhiWrapped / analysisHopSize;
    % Lopussa saatu vaihe lisätään predictedPhase muuttujaan. Tämän avulla
    % saadaan muodostettua muokattu kehys STFT:llä.
    predictedPhase = predictedPhase + synthHopSize * predictedFreq;
      frames(i,:) = real(ifft(abs(fftFrame) .* exp(1i*predictedPhase))) .*
window';
end
%% OLA (Overlap-add)
overlapAdded = zeros(numOfFrames*synthHopSize-synthHopSize+frameSizes(2),1);
start = 1;
% Toteutetaan Overlap-add:
for i=1:numOfFrames
     overlapAdded(start:start + windowSize-1) = overlapAdded(start:start +
windowSize-1) + frames(i,:)';
   start = start + synthHopSize;
end
% Lineaarisella interpolaatiolla saadaan kompressoitua tai pidennettyä
% audio oikeanpituiseksi. Tämä määräytyy pitch-kertoimen mukaan.
% Ensimmäisessä parametrissa on näytepisteet;
%   Toisessa parametrissa on vastaavat arvot ensimmäisen parametrin
näytepisteille;
% Kolmannessa parametrissa on lopulliset pisteet;
% Neljäs parametri kertoo metodin, joka meillä on lineaarinen.
sound_out = interp1((0:(length(overlapAdded) - 1)), overlapAdded,
(0:pitch:(length(overlapAdded)-1)), 'linear');
audio_converted = sound_out;
```