



VST-Synthesizer

Kandidatarbete inom datateknik

Lukas Rahmn, Anton Fredriksson, Sarosh Nasir,
Stanisław Zwierzchowski, Klas Ludvigsson, Andreas Kuszli

KANDIDATARBETE

VST-Synthesizer

Lukas Rahmn, Anton Fredriksson, Sarosh Nasir,
Stanisław Zwierzchowski, Klas Ludvigsson, Andreas Kuszli



CHALMERS

Institutionen för data-och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sweden 2018

VST-Synthesizer

Lukas Rahmn
Anton Fredriksson
Sarosh Nasir
Stanisław Zwierzchowski
Klas Ludvigsson
Andreas Kuszli

© Lukas Rahmn, Anton Fredriksson, Sarosh Nasir, Stanisław Zwierzchowski, Klas Ludvigsson, Andreas Kuszli, 2018.

Handledare: Jan Jonsson, Institutionen för data- och informationsteknik
Examinator: Arne Linde, Institutionen för data- och informationsteknik

Kandidatarbete
Institutionen för data-och informationsteknik
Chalmers Tekniska Högskola
SE-412 96 Göteborg
Telefon +46 31 772 1000

Alla figurer i rapporten är framställda av rapportförfattarna.
Omslag: Bild föreställande syntens användargränssnitt.

Göteborg, Sverige 2018

VST-Synthesizer

Lukas Rahmn, Anton Fredriksson, Sarosh Nasir,
Stanisław Zwierzchowski, Klas Ludvigsson, Andreas Kuszli

Computer Science and Engineering
Chalmers University of Technology
Gothenburg University

Abstract

This project aims to create a virtual synth (short for synthesizer) with the help of the VST specification and the C++ programming language. The group specified functions for the synth since vague specifications for the synth were given. A lot of focus was put into modeling the mathematical models for the synth's functions such as oscillators and filters. To make the work easier, the group used the JUCE framework. An automatic test tool was developed to verify the synth's implementation. The project resulted in a working synth with a well working graphical user interface. All the specified functions were implemented and the synth was extended with some extensions.

Keywords: Virtual Synthesizer, Signal Processing, VST, VST-Plugin, Music Production

VST-Synthesizer

Lukas Rahmn, Anton Fredriksson, Sarosh Nasir,
Stanisław Zwierzchowski, Klas Ludvigsson, Andreas Kuszli

Institutionen för data-och informationsteknik
Chalmers Tekniska Högskola
Göteborgs Universitet

Sammanfattning

Detta projekt syftar till att skapa en virtuell synthesizer (förkortas synt) med hjälp av specifikationen VST och programmeringsspråket C++. Då väldigt vaga specifikationer gavs valdes syntens funktioner av gruppen. Mycket fokus lades på framtagning av matematiska modeller för syntens olika funktioner såsom filter och oscillatorer. För att underlätta arbetet har ramverket JUCE utnyttjats. Ett automatiskt testverktyg utvecklades för att verifiera syntens implementation. Projektet resulterade i en fungerande synt med ett väl fungerande användargränssnitt. Alla specifikationer som ställdes upp i början på projektet implementerades och synten utökades även med ett antal tillägg.

Nyckelord: Virtuel Synthesizer, Signalbehandling, VST, VST-tilläg, Musikproduktion

Innehåll

1	Inledning	1
1.1	Syfte	2
1.2	Historiska syntar	2
1.3	Disposition	3
2	Teori	5
2.1	Oscillator	5
2.1.1	Vågtabeller	7
2.1.2	Lågfrekvensoscillator	8
2.1.3	Vitt brus	8
2.2	Envelope	9
2.3	Butterworthfilter	9
2.4	Reverb	10
2.5	Delay	11
2.6	Distortion	12
3	Mjukvaruutveckling	13
3.1	Specifikation	13
3.2	Systemdesign	14
3.2.1	Pipeline	15
3.2.2	Simultana toner	16
3.2.3	Prestanda	17
3.3	Ljudgeneratorer	18
3.3.1	Multifunktionsoscillator	18
3.3.2	Lågfrekvensoscillator	19
3.3.3	Vitbrusgenerator	20
3.4	Envelope	21
3.5	Butterworthfilter	22
3.5.1	Framtagning av modell för lågpassfilter	22
3.5.2	Framtagning av modell för högpassfilter	23
3.5.3	Teoretisk verifiering	24
3.5.4	Praktisk verifiering	25
3.6	Reverb	26
3.6.1	Faltningsslag	26
3.6.2	Likformigt uppdelad faltning	27
3.6.3	Impulssvar	29

3.7	Delay	29
3.8	Distortion	30
3.9	Grafiskt användargränssnitt	30
3.10	Systemtestning	32
4	Diskussion	35
4.1	Oscillator	35
4.2	Reverb	36
4.3	Filter	37
4.4	Prestanda vs kvalitet	37
4.5	Testning	38
4.6	JUCE	38
4.7	Grafiskt användargränssnitt	39
4.8	Etiska aspekter	40
5	Slutsats	41
A	Fourierserier	I
A.1	Fyrkantsvåg	I
A.2	Sågtandsvåg	III
A.3	Invers Sågtandsvåg	IV
A.4	Trianglevåg	V

Ordlista

VST	“Virtual Studio Technology”. Standardiserat mjukvarugränssnitt för ljudprogram.
SDK	“Software development kit”. Utvecklingsverktyg för programvaruutveckling mot ett visst mjukvarugränssnitt.
MIDI	“Musical Instrument Digital Interface”. Standardgränssnitt för kommunikation mellan bl.a. datorer och kompatibla musikinstrument.
Låg-/hög-/bandpassfilter	Filter som endast låter specificerade frekvenser av en signal att passera brytfrekvensen.
Brytfrekvens	Frekvensen för ett filter där signalen dämpas dubbelt så mycket jämfört med passbandet, d.v.s. dämpas med ca 3dB.
BPM	“Beats per minute” d.v.s. hur många taktslag som går på en minut. Används för att mäta tempo i musik.
Spektrumanalysator	Verktyg för undersökning av en signals amplitud vid olika frekvenser. Ritar ofta upp signalen med amplituden på vertikala axeln och frekvensen på den horisontella.
Sampel	Ett enstaka värde i en digital signal.
Oscillator	Instrument för att fluktuera en signal runt ett värde.
Trådpool	En samling med trådar som väntar på att utföra olika jobb.

Innehåll

Linjär interpolation	En approximativ metod för kurvanpassning.
Överton	En ton som bestäms utav en lägre grundton.
Oktav	Oktav är det intervall som omfattar tolv toner.
API	“Application Programming Interface”. Ett gränssnitt som definierar kommunikation mellan olika mjukvaror.
GUI	“Graphical User Interface” d.v.s. grafiskt användargränssnitt.

1

Inledning

Detta arbetes centrala punkt är digitala synthesizers, förkortas ofta synt. I denna rapport syftar ordet synt på ett elektroniskt musikinstrument som skapar, eller syntetiserar ljud. Klassiskt så innebar detta faktiska elektroniska maskiner som skapade analoga ljudsignaler. Dessa bestod av elektriska oscillator-kretsar sammankopplade med olika typer av styrbara filter, som tillsammans tillät skapandet av en stor mängd ljud. Detta system kompletteras oftast med ytterligare logik för att styra och forma ljudsignalerna [1]. Med utvecklingen av digitala kretsar kom digitala syntar vilket skapade möjlighet för fler unika ljud jämfört med analoga syntar [2].

Med genomslaget av kraftfulla persondatorer så kom möjligheten att köra digitala mjukvaru-syntar. I dessa implementerades hela oscillator- och filterkedjan istället som programkod [3]. Moderna musikproduktionsprogram innehåller nästan alltid minst en synt och tillåter även tredjeparts-utvecklare utveckla egna syntar till programmet. Dessa kan både erbjuda ett eget ljud, efterlikna klassiska hårdvarubaserade syntar eller till och med andra instrument, som piano [4]. För att underlätta interaktionen mellan tredjepartsutvecklarens program och musikproduktionsprogrammen har ett antal standarder vuxit fram som dikterar hur kommunikationen ska ske. En sådan standard är VST [3], som vi återkommer till.

Vad erbjuder studier av digital ljudsyntes ur akademisk synpunkt? Samma signalbehandlingstekniker som appliceras i digitala syntar kan appliceras inom många andra ingenjörsområden som t.ex. bildbehandling, radarbearbetning och analys av biometrisk data. Att applicera denna teknik inom musik erbjuder en möjlighet att fördjupa och utveckla signalbehandlingskunskaper i ett bekant kontext. Det krävs även att kunskap inom flera områden, som programvaruutveckling, musik och signalbehandling förs samman, vilket kan ge nyttiga lärdomar. Som redan nämnt är mjukvarusyntar redan vanligt förekommande, därmed kommer slutprodukten likna existerande produkter men med vissa unika funktioner.

1.1 Syfte

Arbetets syfte är att kombinera kunskaper inom musik, matematik, signalbehandling och programmering för att framställa en mjukvarubaserad digital synt. I projektet ska matematiska modeller framställas för de filter och signaler systemet behandlar, men fokus ligger även på att skapa en fungerande slutprodukt. Synten ska utformas på ett sådant sätt att den är kompatibel med modern musikproduktionsmjukvara, samt att den erbjuder användaren ett grafiskt gränssnitt för att kunna utnyttjas till mer än endast demonstrationssyften. Tilltänkt målgrupp för projektet är vana musikproducenter och entusiaster, inget fokus läggs på att förklara och introducera VST-syntar. Det förutsätts att användaren är bekant med facktermer och har tidigare erfarenhet med syntar.

Projektet ämnar till att utveckla en synt med 16 noters polyfoni, som kan generera ett flertal olika vågformer. Dessa ska kunna manipuleras av diverse effekter såsom lågpas- och högpassfilter, eller en simpel ekoeffekt. Dessa effekter kompletteras med lågfrekvensoscillatorer (LFO) som tillåter automatisering av effekternas inställningar. För att forma signalen över tid ska även synten innehålla ett envelope. Programvaran ska styras av ett grafiskt användargränssnitt alternativt ett fysisk MIDI-tangentbord, via vilka parametrar ändras och olika noter spelas. Konfigurationen av dessa parametrar ska gå att spara.

1.2 Historiska syntar

Vid studie av historiska syntar förekommer vissa namn oftare än andra. Två som har haft stort genomslag är Minimoog Model D och Yamaha DX7.

Minimoog Model D var världens första portabla synt [5]. Den tillverkades på tidigt 70-tal av Robert Moog och blev en enorm succé på grund av sin användarvänlighet samt, för sin tid, låga pris. Några kända artister som under de senaste 40 åren har komponerat musik med hjälp av minimoogen är Dr.Dre, Gary Numan och Keith Emerson [5]. Minimoogen använde sig av en syntesteknik som kallas subtraktiv syntes. Subtraktiv syntes är en teknik som bygger på att en synt har ett antal oscillatorer med olika övertonsrika vågformer (så som trianglevåg eller sågtandsvåg) som kombineras och filtreras för att skapa intressanta ljud.

Yamahas DX7 lanserades på tidigt 80-tal. Denna synt var världens första kommersiella synt som använde sig av frekvensmodulering (FM-syntes) och en av de första digitala syntarna [6]. FM-syntesen bidrog med en enorm mängd olika ljud som tidigare inte varit möjliga. Detta kom till stor användning det kommande årtiondet där DX7:an dominerade syntmarknaden.

1.3 Disposition

Rapporten är strukturerad på följande sätt: I kapitel två introduceras den bakomliggande teori projektet bygger på. Där introduceras de matematiska modellerna och teori för bland annat filter, oscillatorer och effekter. Kapitel tre, mjukvaruutveckling, beskriver i detalj implementation och funktion av syntes olika delar. I kapitel fyra förs en diskussion kring projektet samt de problem och övervägande som gjorts. Ethiska aspekter som har beaktats redogörs för i kapitel fem. Avslutningsvis presenteras gruppens slutsatser i kapitel sex.

2

Teori

En central del av arbetet var att ta fram matematiska modeller för de olika funktionerna och effekterna som ska bygga upp synten. Därav finns det mycket bakomliggande teori för de olika implementationerna som förklaras i detta kapitel.

2.1 Oscillator

Oscillator är den del av synten som skapar ljudvågorna. I detta projekt valdes fyrkants-, sågtand-, sinus- samt triangelvåg, se figur 2.1. Det finns en mängd sätt att generera dessa vågformer, t.ex. kan det tänkas att sågtandsvågen skulle gå att beräkna som en modulusräknare:

$$n_{k+1} = (n_k + s) \bmod 1, s \in \mathbb{R}, 0 < s < 1 \quad (2.1)$$

där s avgör frekvensen av vågformen. Problemet med denna oscillatorimplementation är att den ej är bandbegränsad. Eftersom systemet är ett samplat system kommer frekvenskomponenter som ligger över halva samplingsfrekvensen, den så kallade nyquistfrekvensen, vikas ner i området nedanför nyquistfrekvensen [7, s.241]. Då uppstår oönskade frekvenskomponenter, kallat aliasing, som kan upplevas som irriterande missljud när vågformen spelas upp. För att undvika detta kan en mängd olika metoder användas för att minska eller ta bort aliasing. Den metod som används i detta projekt är additiv syntes, vilket är en teknik som bygger på att skapa vågor genom att summera sinusvågor. Detta kan beräknas med hjälp av fourierserier och användas för att generera en bandbegränsad våg som endast innehåller frekvenskomponenter under nyquistfrekvensen [7, s.150]. På trigonometrisk form kan en periodisk signal $x(t)$ med vinkelfrekvens ω_0 beskrivas på följande sätt [7, s.158]:

$$x(t) = A_0 + \sum_{k=1}^{\infty} [A_k \cos \omega_0 k t + B_k \sin k \omega_0 t]. \quad (2.2)$$

Se tabell 2.1 för värden på A och B för olika vågformer. Genom att begränsa k kan en bandbegränsad approximation uppnås av vågformen $x(t)$. T.ex. sätts $k = 100$ så kan signalen endast innehålla frekvenserna $0, \omega_0, 2\omega_0, 3\omega_0 \dots 100\omega_0$ eftersom sinus och cosinus endast representerar en frekvens. Värdet på koefficienterna erhålls med hjälp

av följande förhållande mellan de trigonometriska koefficienterna och de komplexa exponentialbegränsningarna [7, s.158]:

$$2C_k = A_k - j \cdot B_k, \quad (2.3)$$

där C_k representerar koefficienterna för en alternativ form att uttrycka fourierserier [7, s.159].

$$x(t) = \sum_{k=1}^{\infty} C_k \cdot e^{j\omega_0 t},$$

för denna form finns en explicit formel för C_k koefficienten [7, s.158]:

$$C_k = \frac{1}{T_0} \int_{T_0} x(t) e^{-j\omega_0 t} dt. \quad (2.4)$$

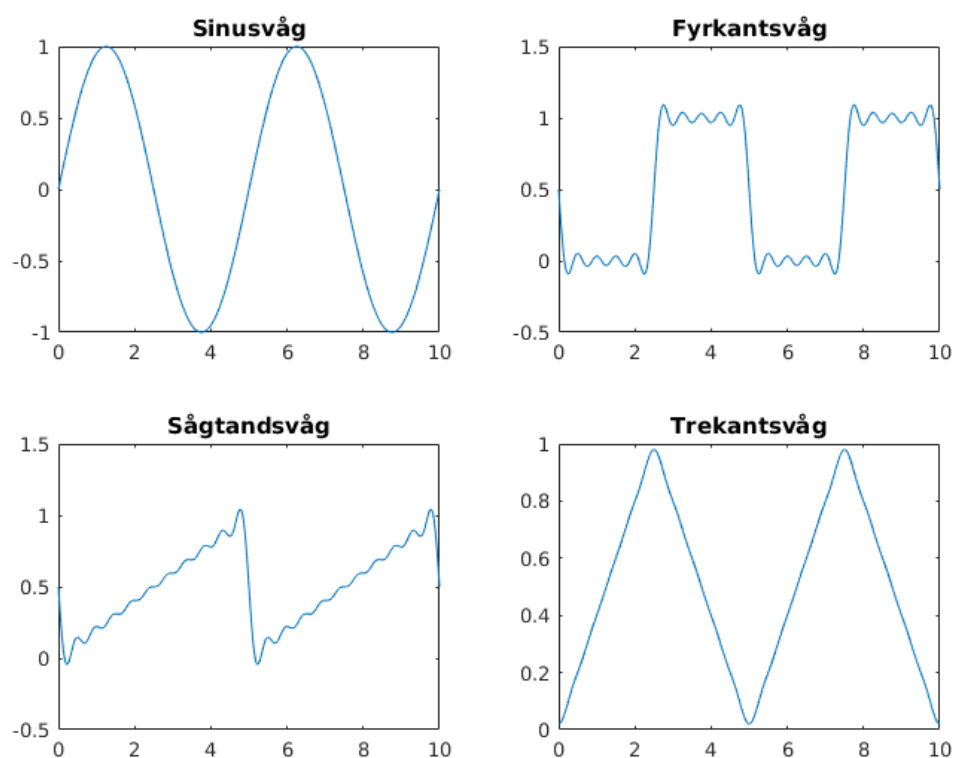
I formel 2.4 representerar T_0 periodtiden för signalen $x(t)$. För härledning av A_k, B_k, C_k för de olika vågformerna med hjälp av formel 2.3 och 2.4, se appendix. Med hjälp av tabell 2.1 och formel 2.2 erhålls en beräkningsalgoritm för en bandbegränsad approximation $x(t)$. För fallet med en bandbegränsad sågtandsvåg med $k = 10$ och en topp till toppamplitud på 1 gäller:

$$x(t) = \frac{1}{2} + \sum_{k=1}^{10} \frac{-A}{\pi k} \cos k\omega_0 t.$$

Plotten av denna vågform ges i figur 2.1. Problemet med att beräkna oscillatorerna på detta sätt är den höga beräkningskomplexiteten, för varje sampel av sågtandsvågsoscillatorn måste uttrycket $\frac{-A}{\pi k} \cos k\omega_0 t$ evalueras k gånger för att sedan summeras. För att undgå denna kostsamma operation används istället något som kallas vågtabeller.

Tabell 2.1: Trigonometriska koefficienter för vågformer med topp till topp amplitud A

Vågform	A_k	B_k
Fyrkantsvåg	$\begin{cases} \frac{A}{2}, & k = 0 \\ 0 & \end{cases}$	$\begin{cases} \frac{-2A}{\pi k}, & k \text{ är udda och } k \neq 0 \\ 0 & \end{cases}$
Triangelvåg	$\begin{cases} \frac{A}{2}, & k = 0 \\ \frac{-4A}{\pi^2 k^2}, & k \text{ udda} \end{cases}$	$0 \forall k$
Sågtandsvåg	$\begin{cases} \frac{A}{2}, & k = 0 \\ 0 & \end{cases}$	$\begin{cases} \frac{-A}{\pi k}, & k \neq 0 \\ 0 & \end{cases}$



Figur 2.1: Vågformer med 10st fourierkoefficienter.

2.1.1 Vågtabeller

När digitalt ljud genereras så samplas en signal exempelvis 44100 gånger varje sekund (CD-kvalité) [8]. När en sinusvåg skall spelas behöver därmed 44100 beräkningar göras varje sekund. Men eftersom ljudvågorna som oscillatorerna skall generera går i perioder är det möjligt att spara endast en period av signalen i en lista och repetera denna. Vid en högre frekvens samplas ljudet med större intervall från listan. Detta ger färre beräkningar i utbyte mot mer minnesanvändning [9].

För att veta vilket element från vågtabellen en oscillator ska sampla används ett index samt en formell för hur mycket indexet skall öka efter varje sampling. Indexet börjar på 0 när en tangent trycks ned och ökar enligt följande formell:

$$i = i + \frac{N \cdot f}{T},$$

där N är storleken på vågtabellen, f är frekvensen som skall samplas och T är samplingshastigheten. Ett index i en lista måste dock alltid vara ett heltal. För att hantera icke heltalsindex så används linjär interpolering mellan de två närliggande värdena. Formeln för interpoleringen följer:

$$\text{sampel} = V(\lfloor i \rfloor)(1 - (i - \lfloor i \rfloor)) + V(\lfloor i \rfloor + 1)(i - \lfloor i \rfloor),$$

där $V(n)$ returnerar värdet på index n i vågtabellen. Bortsett från sinusvågen så innehåller alla vågformer övertoner med högre frekvens än basfrekvensen som genereras. Detta innebär att vid höga frekvenser kan dessa övertoner passera nyquistfrekvensen och skapa aliasing. Då Fourierserier används för att generera de olika vågformerna går antalet övertoner enkelt att anpassa efter vilken ton som ska genereras. Antalet övertoner skall därmed vara högt vid låga frekvenser samt lågt vid höga frekvenser, och eftersom alla vågtabeller skall initieras på förhand så används vanligtvis flera tabeller för varje vågform.

Längden på tabellerna spelar in i hur detaljerade vågformer som kan genereras. Längre tabeller ger mer precision, men kräver också mer minne.

2.1.2 Lågfrekvensoscillator

En oscillator producerar en signal baserad utifrån en viss frekvens, och eftersom den är periodisk kan den uppfattas som aningen livlös. För att göra signalen mer levande kan parametrar ändras baserat på något oberoende av oscillatorns frekvens. Detta går att göra med en så kallad lfo (kort för lågfrekvensoscillator eller low frequency oscillator). En lfo är en oscillator som genererar låga frekvenser, oftast i intervallet 0.2 Hz - 35 Hz, som sedan användas för att modulera en eller flera andra signaler[10, s.32]. Detta kan användas för effekter som exempelvis vibrato, där frekvensen svajar runt originalfrekvensen[10, s.33], eller tremolo, där amplituden svajar runt originalamplituden[10, s.35].

Frekvensen hos en lfo kan även synkroniseras i takt med musiken. För att synkronisera en lågfrekvensoscillators signal med musiken som spelas används låtens takt för att bestämma frekvensen av signalen. Frekvensen bestäms av följande formell:

$$f = \frac{R_{bpm} \cdot r}{60}$$

där r är det förhållandet mellan signalen och takten, och R_{bpm} är låtens takt i bpm. Exempelvis om $r = 1$ så går signalen en period per taktslag. $R_{bpm} \cdot r$ delas sedan på 60 för att gå från minuter till sekunder och stämma överens med cykler per sekund (Hz).

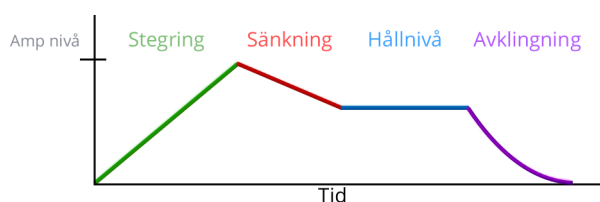
2.1.3 Vitt brus

Vitt brus är en signal vars amplitud är uniform för alla frekvenser [11]. Signalen är icke-periodisk och kan därmed inte samplas med vågtabeller utan måste genereras i realtid.

2.2 Envelope

När en ton spelas är det inte alltid önskvärt att ljudet ska spelas med maximal amplitud under hela tonen. Ofta vill man att den sakta ska öka från nedtryck, minska en aning efter att tonen nått sin maximala amplitud och sedan sakta försvagas när den nedtryckta tonen släpps. För att göra detta möjligt används något som kallas för konturgenerator eller på engelska och kanske mer välkänt inom musikbranschen, envelope.

Figur 2.2 visar hur ett s.k. ADSR-envelope kan se ut, ett envelope som delats in i "Attack", "Decay", "Sustain", "Release", eller "Stegring", "Sänkning", "Hållnivå", "Avklingning". Grafen visar tidslinjen för en genererad ton, där amplituden ökar linjärt från start till precis innan sänkningen. Då sänks den till hållnivån där nivån inte ändras så länge tonen hålls ner. Efter tonen har släppts minskar amplituden icke-linjärt under avklingningen. Ett ADSR-envelope kan ställas in på många olika sätt, inklusive tid, nivå och kurvform för varje del.



Figur 2.2: Exempel på ett ADSR-envelope.

2.3 Butterworthfilter

Ett filter filtrerar bort oönskade frekvenser från en signal. Två vanliga filtertyper är lågpas- och högpasfilter. Dessa låter endast frekvenser under respektive över en viss frekvens att passera [12]. Frekvensen som utgör gränsen kallas för brytfrekvens och har egenskapen att förhållandet mellan in- och utsignalens magnitud ska vara $1/\sqrt{2}$ vid den frekvensen [7, s.296].

Butterworthfilter är en filtersort som kännetecknas med att passbandet, d.v.s. de frekvenser som inte filtreras bort, har en flat frekvensrespons. Antalet poler hos filtrets överföringsfunktion bestämmer filtrets ordning. Ett högre ordningens filter har en brantare kurva vid övergången från passbandet till stoppbandet men ökar även filtrets komplexitet [13]. Överföringsfunktionen i Laplacedomänen [7, s.335] för ett Butterworthfilter utav andra ordningen, där s är komplex frekvens, ser ut som följande [14, s.252]:

$$H(s) = \frac{1}{s^2 + s\sqrt{2} + 1}. \quad (2.5)$$

Filtret i formel 2.5 kan omvandlas till ett lågpasfilter genom att s byts ut mot s/ω_a där ω_a är filtrets brytfrekvens i rad/s [14]. Det ger följande överföringsfunktion:

$$H(s) = \frac{\omega_a^2}{s^2 + s\omega_a\sqrt{2} + \omega_a^2}.$$

På liknande sätt kan filtret i formel 2.5 omvandlas till ett högpasfilter genom att s byts ut mot ω_c/s [14] vilket ger följande överföringsfunktion:

$$H(s) = \frac{s^2}{s^2 + s\omega_a\sqrt{2} + \omega_a^2}.$$

För att möjliggöra implementation av ett filter beskrivet i den kontinuerliga Laplace-domänen måste överföringsfunktionen diskretiseras till \mathcal{Z} -domänen. Detta kan göras med Tustins metod [15] där frekvensen s byts ut enligt enligt formel 2.6 och f_s är samplingsfrekvensen i Hz.

$$s \longrightarrow 2f_s \frac{1 - z^{-1}}{1 + z^{-1}}. \quad (2.6)$$

Då diskretisering förvrider brytfrekvensen måste den förändras enligt formel 2.7, där ω_c är den diskretiserade brytfrekvensen och ω_a den faktiska brytfrekvensen [15].

$$\omega_c = 2f_s \tan\left(\frac{\omega_a}{2f_s}\right) \quad (2.7)$$

2.4 Reverb

När ett ljud låter i ett visst utrymme kan lyssnaren förutom att höra ljudet som kommer direkt från ljudkällan, även höra ljud som har reflekterats en eller ett flertal gånger av utrymmets väggar och eventuella föremål. Reverb (kort för eng. Reverberation) är en ljudeffekt vars uppgift är att få ljud att låta som om det spelades upp i ett visst utrymme som exempelvis en stor hall eller en scen. Detta åstadkoms genom att på något sätt simulera dessa reflektioner [16].

En möjlig mjukvarubaserad implementation av en reverb-effekt går ut på att algoritmer med återkopplade fördröjningar försöker simulera reflektionerna som uppstår

i verkligheten. Denna metoden kräver färre beräkningar men medför högre svårighet för att åstadkomma ett realistiskt resultat [17, s.1].

En annan implementation går ut på att ett impulssvar för ljudet i ett visst utrymme faltas med ljudsignalen som effekten ska appliceras på. Ett sådant impulssvar, det vill säga en inspelning av en kort ljudimpuls i ett utrymme, fångar hur ljudet beter sig i det utrymmet. Utgångssignalen låter då som om ljudsignalen spelades upp i utrymmet vars impulssvar användes vid faltningen [18]. Denna metoden ger ett mycket realistiskt resultat men är relativt beräkningsintensivt [17, s.1].

2.5 Delay

En annan populär effekt inom musikproduktion är delay. Denna effekt fungerar som ett simpelt eko, där ljudet som genereras spelas upp igen efter en viss fördröjning med eventuellt minskad amplitud. Ett simpelt delay L_D av diskret signal $x[n]$ kan beskrivas med följande formell:

$$y[n] = L_D(x[n]) = x[n - D],$$

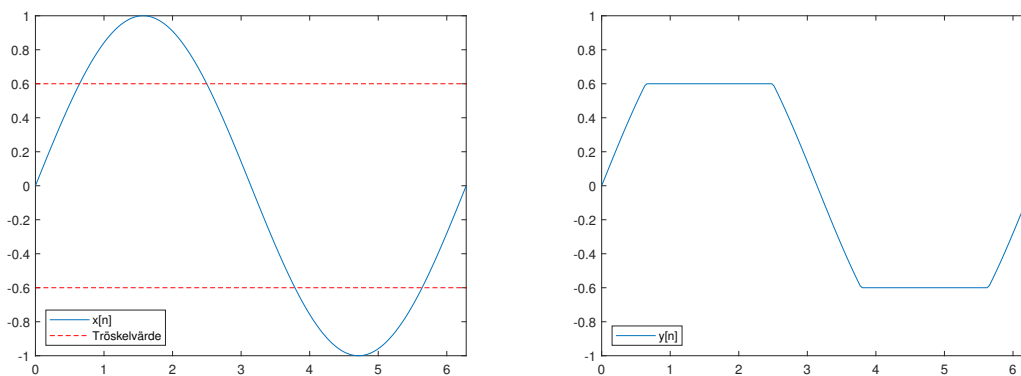
där n är sampelindexet och D är antalet sampel som signalen ska fördröjas [19, s. 30-31].

2.6 Distortion

Distortion är en vanligt förekommande effekt som går ut på att imitera när amplituden av en signal är högre än vad en högtalare eller förstärkare kan spela upp och klipper därmed av signalen vid ett visst tröskelvärde [20]. Funktionen för en simpel distortionseffekt följer:

$$y[n] = \begin{cases} d & x[n] > d \\ -d & x[n] < -d \\ x[n] & \text{annars} \end{cases} \quad (2.8)$$

Där $y[n]$ är utsignalen, $x[n]$ insignalen, d tröskelvärdet och n sampelindexet. Figur 2.3 visar effektens inverkan på en sinusvåg.



(a) Insignal $x[n]$

(b) Utsignal $y[n]$

Figur 2.3: Distortion.

3

Mjukvaruutveckling

Synten följer VST-specifikationen för kommunikation, vilket tillåter att den används som ett tillägg i flera olika värdmjukvaror. VST bygger på en väldefinierad relation mellan värd och tillägg. Värden styr ljudbehandlingen och skickar tangentryckningar och parameterändringar till tillägget. T.ex. kan värden informera vilken not som ska spelas eller via parametrar ändra tilläggets konfiguration. Värden ber sedan tillägget att använda dessa parametrar och behandla ett block av ljudsampels. Storleken av blocket är variabel och väljs av värden. För att underlätta arbetet med VST så används ramverket JUCE, som dels abstraherar bort många implementationsdetaljer kring VST, men tillhandahåller också hjälpfunktioner för ljudbehandling och konstruktion av grafiska gränssnitt.

I detta kapitel beskrivs systemets struktur samt implementationen av de enskilda komponenterna som utgör synten.

3.1 Specifikation

Följande kravspecifikation sattes upp för projektet. Denna lista utgör krav på funktionalitet som skulle uppfyllas, men den utgör inte fullständigt specifikation av slutprodukten.

- ADSR-envelope som ska kunna kopplas till oscillatorernas volym.
- Lågpas-, och högpasfilter med ställbar brytfrekvens.
- Oscillatorer som kan generera sinus-, fyrkant-, triangel-, sågtand- och brus-signaler. Varje oscillator ska ha en parameter som förskjuter den genererade signalen med ett angivet antal oktaver, en parameter för att finjustera den genererade frekvensen samt möjlighet att panorera signalen mellan vänster och höger kanal.
- Pitchbend som ska kunna kontinuerligt förskjuta oscillatorernas frekvens under spelning med hjälp av ett reglage på ett fysiskt tangentbord.

- Minst 16 noter ska kunna spelas samtidigt.
- Lågfrekvensoscillator som kan kopplas till lämpliga parametrar. Den ska kunna ställas in till en fast frekvens med minsta spannet 0,25 - 10Hz eller synkroniseras med musikprogrammets tempo med en multipel av slag med spannet 8 - 1/16 slag.
- Förinställningar ska kunna sparas och senare laddas in.
- Ett användargränssnitt ska kunna kontrollera syntens alla parametrar.

3.2 Systemdesign

Synten är uppbyggd av ett flertal distinkta komponenter, där varje del tillhandahåller en viss funktionalitet till programvara. Följande lista är ett urval av grundkomponenter som existerar. Senare i kapitlet kommer en mer detaljerad specifikation för var och en av dessa komponenter ges.

- Pipeline
- Ljudgenerator
 - Multifunktionsoscillator
 - Vitbrusgenerator
 - Lågfrekvensoscillator
- Envelope
- Effekt
 - Högpasfilter
 - Lågpasfilter
 - Distortion
 - Reverb
 - Delay
- Användargränssnitt

Efter att specifikationerna för delsystemen sammanställts kan flera delsystem utvecklas parallellt av olika individer. Detta medför även att implementationsspecifik kunskap endast krävs av komponentens utvecklare. Komponenterna tillhandahåller

ler även en viss flexibilitet vid implementation tack vare att det går att flytta och återanvända i andra delar av synten.

3.2.1 Pipeline

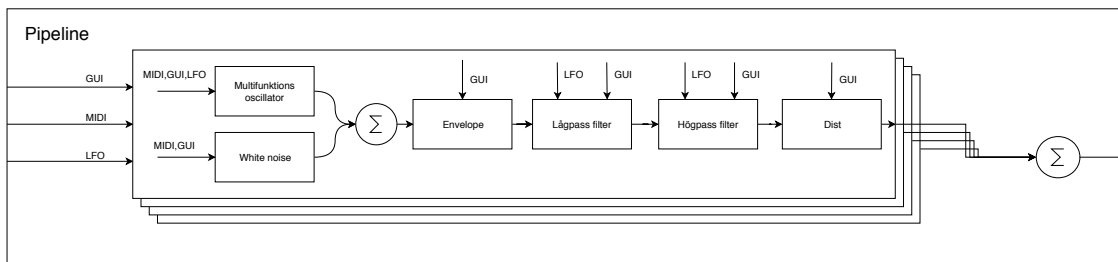
Pipeline är en fundamental del av ljudgenereringen i vår ljudgenereringsprocess. Dess funktion är att behandla inkommande MIDI-kommandon och med hjälp av en mängd delkomponenter producera det efterfrågade ljudet. Det som drev designen av pipeline var en önskan att paketera hela ljudgenereringskedjan i form av en avgränsad delkomponent, i linje med komponentdesignen beskriven ovan.

I figur 3.1 finns en översikt av pipeline, som illustrerar hur de olika delsystemen samverkar. Pilarna märkta GUI indikerar configurationsdata från det grafiska användargränssnittet. Anledningen till att bilden visar flera instanser av den första delen av pipeline är p.g.a. att synten innehåller flera separata ljudblock, vilka kan konfigureras oberoende av varandra. T.ex. kan ett block konfigureras till att generera en hög-passfiltrerad fyrkantsvåg som över tid långsamt avtar i styrka, samtidigt som ett annat block genererar ofiltrerat vitt brus. Det är den stora konfigurations- och kombinationsmöjligheten i pipeline som möjliggör en stor mängd olika ljud att genereras.

Låt oss beskriva hur pipeline fungerar genom ett exempel. All ljudbehandling börjar med något typ av MIDI-meddelande, vilket är en standard för kommunikation mellan instrument [21]. Dessa meddelanden innehåller ingen ljuddata utan endast instruktioner, t.ex. spela not C4 med styrka 23, sluta spela not B3 o.s.v. Kontrollmeddelanden är utformade så att de flesta operationer ett vanligt MIDI-tangentbord stödjer går att beskriva. När pipeline tar emot ett sådant meddelande vidarebefordrar den det till varje vitbrusgenerator, envelope och multifunktionsoscillator. Beror på inställningarna i användargränssnittet genererar vitbrusgeneratorn och multifunktionsoscillatorn var sin ljudsignal. Dessa summeras och den resulterande ljudsignalens amplitud formas därefter av envelopet, baserat på MIDI-datan och inställningar. I nästa steg passerar ljudsignalen serien av effekter, som filter- och distortion-enheterna, som manipulerar ljudsignalen ytterligare. Avslutningsvis summeras ljudsignalerna från de fyra ljudgenereringsblocken, vilket i bilden representeras av det högra summeringsblocket.

I praktiken opererar pipeline ej kontinuerligt utan på ljudblock av varierande storlek. Behandlingen av ljudblock kontrolleras av den externa VST-världmjukvaran som exekverar synten. Från värden erhåller synten MIDI-data samt längden på nästa ljudblock, som vidarebefordras till och startar pipeline. Pipeline exekverar fram tills ett ljudblock av önskad storlek genererats. Därefter stoppar den och returnerar ljudblocket till världmjukvaran. Information om pipeline's interna tillstånd bevaras mellan varje förfrågan, så under nästa block fortsätter pipeline generera ljudsignalen där den slutade.

Det är viktigt att vara medveten om att en pipeline endast kan behandla ett notkommando i taget, d.v.s. den kan endast hantera ett tangent-nedtryck åt gången. Varje oscillator i pipeline mottar samma MIDI-data, så de olika ljudgenereringsblocken i pipeline behandlar alltid samma not. Vilken ljudsignal som varje oscillator producerar till svar på ett not-kommandot kan ändras av användaren, men denna ljudsignal är alltid produkten av ett enda not-kommando samt syntens inställningar.



Figur 3.1: Flödesschema för pipeline.

Figur 3.1 representerar ett förenklat flödesschema över hur pipelinestrukturen ser ut.

3.2.2 Simultana toner

I specifikationen av synten listades ett krav på minst 16 simultana toner. Detta innebär att när användaren trycker ner 16 tangenter samtidigt ska den resulterade ljudsignalen vara summan av ljudet varje tangent ger upphov till. Anledningen till fler än 10 toner är att oftast upphör inte ljudet samtidigt som tangenten lyfts. T.ex. kan användaren valt att med hjälp av en envelope-komponent konfigurera synten så att ljudet ska avta långsamt efter att tangenten lyfts.

Som beskrivet tidigare kan inte pipeline hantera mer än en not/tangent. För att stödja simultana toner använder synten istället multipla pipelines vars resultat adderas. Internt för synten en lista över varje not som ska ljuda, och tilldelar var och en av dem en egen pipeline som ansvarar för att generera den noten. Det totala antalet pipelines kan konfigureras, men är för tillfället satt till 16. När en ny not efterfrågas söker synten efter en ledig pipeline och tilldelar den pipeline noten. Finns ingen ledig pipeline tillgänglig väljs en av de upptagna och den konfigureras om för att generera den nya tonen. De flesta not-kommandona ger upphov till en ljudvåg som med tiden avtar, antingen p.g.a. av att användaren lyfter tangenten eller p.g.a. konturgeneratorn. När detta sker informerar pipeline synten att den inte längre ska generera ljud och är redo att motta nya not-kommandon.

När värdmjukvaran begär ett nytt ljudblock, avgör synten först vilka pipelines som är aktiva. Därefter placerar den de aktiva i en trådsäker arbetskö. Från denna kö hämtar varje av syntens arbetstrådar en pipeline och exekverar den. Synten skapar lika många arbetstrådar som processorkärnor när synten startar. På så sätt paral-

lelliseras exekveringen av pipelines. När kön till slut är tom adderar synten den resulterade ljudsignalblocken från varje pipeline för att returnera det resulterande blocket till värden.

3.2.3 Prestanda

En viktig aspekt av synten är systemprestandan. I värsta fall kan pipelinen exekvera för långsamt för att generera ljud i realtid. Även om exekveringshastigheten är tillräcklig för att generera ljud i realtid är det önskvärt att programmets cpu-användning och minnesanvändning ej är för hög. T.ex. är det önskvärt att det på en laptop går att använda flera instanser av synten och/eller köra andra musikprogram simultant. Därför har ansträngningar gjorts för att optimera ljudgenereringen. En viktig del för att uppnå god prestanda har varit att utnyttja parallellisering till stor grad, optimera kritiska loopar, samt att stänga av onödigt funktionalitet i synten.

Parallelliseringen sker genom att exekvera flera pipelinejobb parallellt. Som standard väljer mjukvaran lika många parallella jobb som det finns tillgängliga kärnor. Eftersom pipelinejobben är helt oberoende krävs ingen kommunikation eller synkronisering mellan trådarna. Synkroniseringen sker först när alla jobb avslutats genom att deras resulterade ljudsignaler adderas. Varje pipelinejobb motsvarar, som beskrivits tidigare, en specifik not och därför sker parallelliseringen på not-nivå.

Med hjälp av prestanda-analyseringsverktyg identifierades de kritiska looparna i programmet, d.v.s. de loopar som programmet spenderar mest tid i. När dessa identifierats togs åtgärder att optimera dem. De största prestandavinsterna uppnåddes genom att minimera externa funktionsanrop, minska antalet instruktioner och utnyttja konstanta deklARATIONER effektivt, så att kompilatorn sedan kunde generera snabb kod för dessa loopar.

Den kanske viktigaste optimering av programmet är att synten endast exekverar de delar som krävs. Varje pipeline övervakar vilka av dess komponenter som behöves för att producera ett visst ljud och stänger av onödiga komponenter. När komponenterna kedjas ihop utgör de även en hierarki som kan optimeras. T.ex. beror ett filters utsignal på dess inkommande ljudsignal, vars källa kan vara en ljudgenerator, och om den källan stängs av, d.v.s. den skapar en signal som endast består av nollor, så är det slöseri med resurser att behandla signalen. Dock är det ej möjligt att omedelbart stänga av alla komponenter som beror på källan, t.ex. kan filterkomponenterna producera en nollskild signal en tid efter att deras insignal övergått till endast nollor. Systemet kan beskrivas som att det har en viss tröghet. För att lösa detta problem låter systemet delkomponenterna själva propagera avstängningssignalerna. Låt oss använda pipelinen i figur 3.1 som exempel. Om envelopet i pipelinen upptäcker att den endast genererar nollsignaler så stängs den av och informerar lågpasfiltret om detta. Lågpasfiltret fortsätter att producera ljud fram tills även det endast producerar en nollsignal varpå filtret stängs och högpassfiltret informeras. Om avstängningssignalen mottas från alla ljudblock i pipelinen avslutas hela pipe-

linen och synten informeras. Om alla syntens pipelines är avstängda så går hela synten ner i viloläge fram tills att ett nytt MIDI-meddelande väcker en pipeline. Till exempel att användaren trycker ner en tangent.

3.3 Ljudgeneratorer

Ljudgeneratorer är de komponenter i pipeline som genererar ljudsignaler, vilket skiljer dem från effektkomponenter som endast manipulerar befintliga signaler.

3.3.1 Multifunktionsoscillator

Multifunktionsoscillatorerna skapar, manipulerar och adderar ihop alla ljudsignaler som genereras. När en av dessa oscillatorer ska generera en signal utifrån en viss frekvens börjar den med att förskjuta frekvensen utifrån ett antal parametrar:

Octave: Heltal i intervallet $[-3, 3]$ som beskriver hur många oktaver frekvensen ska förskjutas.

Offset: Heltal i intervallet $[-11, 11]$ som beskriver hur många tonsteg frekvensen ska förskjutas.

Detune: Flyttal i intervallet $[-1.0, 1.0]$ som beskriver hur mycket frekvensen skall förskjutas mellan de två närliggande noterna.

Overtone: Heltal i intervallet $[1, 7]$ som beskriver hur många multipler frekvensen skall förskjutas med.

Den förskjutna frekvensen beräknas sedan med följande formell:

$$f' = f \cdot 2^{P_{octave} + \frac{P_{offset} + P_{detune}}{12}} \cdot P_{overtone},$$

där $P_{parameter}$ är värdet på de parametrarna, f är originalfrekvensen och f' är frekvensen efter förskjutning. Sedan används den förskjutna frekvensen för att hämta ett sampel från de förkalkylerade vågtabellerna.

Varje vågform har en egen mängd vågtabeller med olika frekvenskomponenter (förklarade i 2.1). Efter testning med 20 Hz som lägsta- samt 20480 Hz som högsta tillåtna frekvens valdes tio tabeller, med en tabell per oktav, att vara ett lämpligt antal vågtabeller för varje vågform. Dessa tabeller numrerades från 0-9, med 0 som lägsta frekvens (20 - 40 Hz). För att få ut rätt tabellindex från en frekvens användes följande formell:

$$i = \lfloor \log_2\left(\frac{f}{f_{min}}\right) \rfloor,$$

där f_{min} är den lägsta tillåtna frekvensen och i är tabellindexet.

I oscillatorn har varje vågform också en volymparameter i intervallet $[0.0, 1.0]$, som bestämmer hur mycket av den vågformen som skall ingå i den resulterande ljudsignalen. För varje vågform hämtas därmed ett sampel som multipliceras med dess tillhörande volymparameter. Dessa parametrar normaliseras så att summan av dem inte överskrider 1, för att den summerade ljudsignalens amplitud inte ska överskrida 1.

3.3.2 Lågfrequensoscillator

I projektet skulle flera oscillatorers parametrar vara kapabla att kopplas till en och samma lågfrequensoscillator. För att möjliggöra detta skapades två stycken lfo:er som ett antal relevanta parametrar skulle kunna kopplas till.

När en oscillator skall fylla ett ljudblock behöver den lätt kunna kalla på en lfo för det sampel oscillatorn skall kalkylera. På grund av detta har en lfo sin egen buffer med samma längd som de ljudblock som oscillatorerna skall fylla. Lfo-buffern behöver också fyllas innan oscillatorerna, så att den informationen finns tillgänglig när oscillatorerna behöver den. Därför fylls lfo-buffrarna direkt vid mottagning av ett nytt MIDI-meddelande. En lfo fyller sin buffer på samma sätt som oscillatorerna, d.v.s. den använder vågtabellerna.

Varje lfo har fem inställningar som användaren kan justera:

Ratio: Ett förhållande mellan frekvensen och låtens takt i *bpm*.

Type: Vilken av syntens fyra vågtyper lfo:er skall fylla sin buffer med.

Gain: Ett flyttal i intervallet $[0,1]$ som används för att skala en lfo:s sampelvärden.

Invert: Ett boolskt värde som inverterar vågformen.

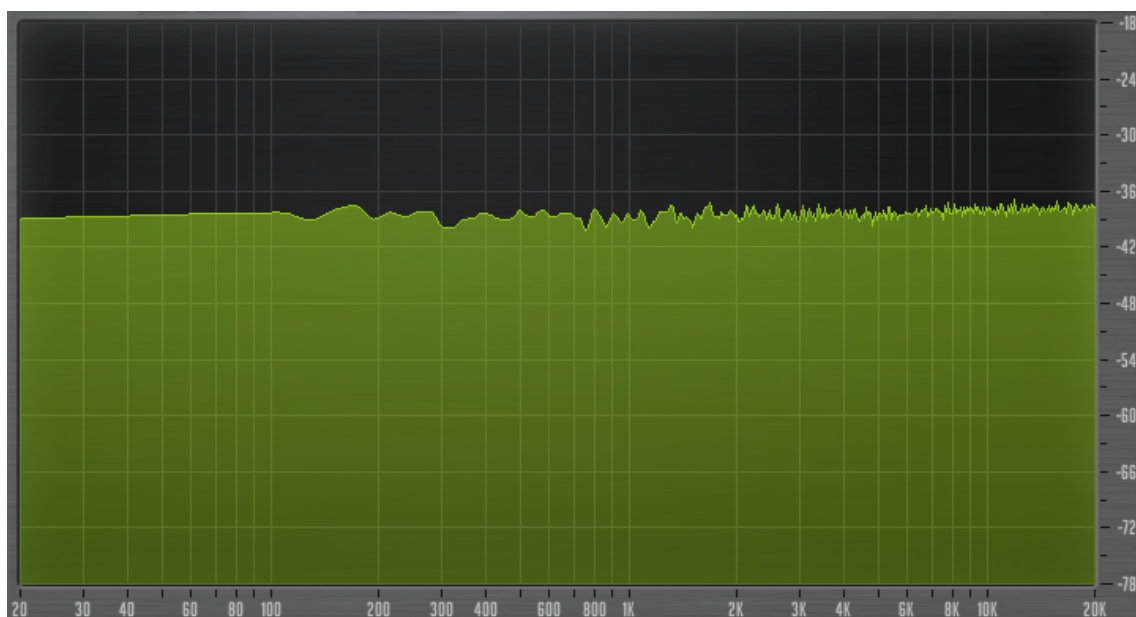
OnPress: Ett boolskt värde som avgör hur fasen av lfo:er skall räknas ut.

Fasen på signalen kan bestämmas på två olika sätt: Synkronisera fasen med den låt som spelas eller sätta fasen till noll varje gång en ny tangent trycks ned. Om *OnPress* är sann skall fasen nollställas varje gång ett nytt tangentnedtryck registreras, och om *OnPress* är falsk skall fasen synkroniseras med låten. Om lfo:n ska synkroniseras med låten behöver den få information om när en låt startar eller stoppar. Denna information kan erhållas från värden, tillsammans med informationen om låtens nuvarande *bpm*, då den kan ha ändrats, samt vart i en takt låten just nu befinner sig. Det enda som lfo:n behöver hålla reda på är hur många takter som har spelats och addera detta till värdet som beskriver positionen i takten, för att sedan kunna räkna ut fasen. T.ex. om låten går i tretakt och lfo:n är inställd på att gå en period var fjärde takt så kommer lfo:ns fas befinna sig på olika ställen i takten beroende på hur många takter som har spelats. Om *OnPress* är falsk och det inte spelas någon låt så fortsätter fasen bara öka i takt med den givna hastigheten.

De parametrar som valdes ut att kunna påverkas av de två lågfrekvensoscillatorerna var: amplitud, frekvens, lågpasfilter samt högpasfilter. De olika parametrarna påverkas av lfo:n på olika sätt. Lågpasfiltret och högpasfiltret utgår från brytfrekvensen som högsta värde och $\omega - (\omega - f_{min}) \cdot P_{Gain}$ som lägsta värde, där ω är filtrets brytfrekvens, f_{min} filtrets lägsta tillåtna frekvens och P_{Gain} är lfo:ns flyttalsparameter som nämns ovan. Amplituden påverkas på samma sätt, men har istället dess nuvarande volym som högsta värde och 0 som lägsta värde. Frekvensen påverkas däremot annorlunda. Den önskade effekten är att frekvensen skall höjas och sänkas med originalfrekvensen som utgångspunkt. För att uppnå detta används följande formell: $f = f \cdot 2^{s \cdot P_{Gain}}$, där f är originalfrekvensen och s är värdet på det sampel som hämtats från lfo-buffern.

3.3.3 Vitbrusgenerator

Vitt brus kan genereras digitalt genom att varje sampel sätts till ett slumpmässigt värde med hjälp av en slumpgenerator. Figur 3.2 visar spektrum från spektrumanalysatorn SPAN [22] för det genererade vita bruset vid en viss tidpunkt.



Figur 3.2: Spektrum för vitt brus vid en viss tidpunkt.

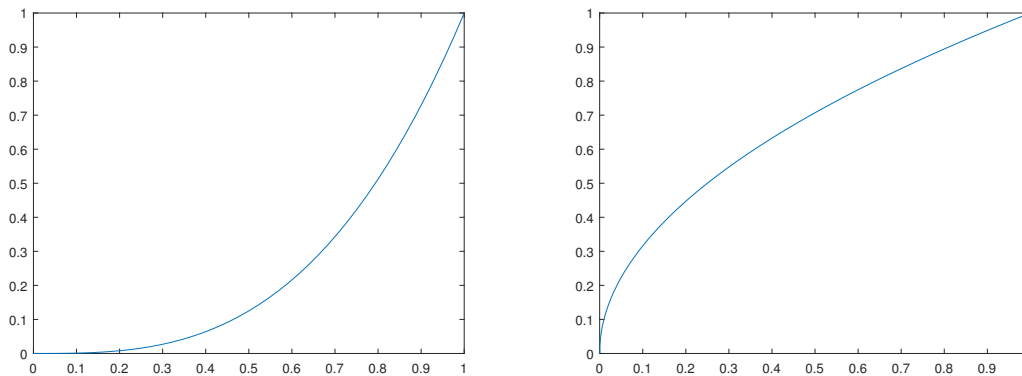
3.4 Envelope

Envelopet utvecklades till en början som en enkel tillståndsmaskin som använde linjära kurvor för att modulera amplituden. Varje steg räknar från 0 upp till längden av steget. I varje iteration så appliceras en kurva likt $amplitude = k \cdot x + m$. När räknaren, i detta fallet x , var lika med längden av steget så sattes den till 0 och tillståndsmaskinen hoppade till nästa steg. Ett speciellt tillstånd är sustain, eftersom synten ska producera ljud så länge som användare håller inne en tangent eller håller ned en sustain pedal. Detta gör att sustaintillstånden inte kan ha en tidsbegränsning, tillståndsmaskinen lämnar inte sustain förrän noten ska sluta spelas. Problemet med den enkla implementationen var att den enda inställningen en användare kunde göra var att ändra längden på attack, decay och release. För att kunna efterlikna mer komplexa ljud så utvecklades ett mer komplext envelope, där förutom längden, både amplituden och kurvformen av varje steg kan ändras. För att åstadkomma detta så används följande formel för att beskriva varje steg:

$$amplitud = \left(\frac{x}{time \cdot level^{\left(\frac{-1}{curve}\right)}} \right)^{curve},$$

där $time$ är längden av steget, $level$ är amplituden och $curve$ är kurvformen.

Figur 3.3 visar hur kurvan ändras när $curve = 3$ och $curve = 0.5$



(a) $curve = 3$

(b) $curve = 0.5$

Figur 3.3: Amplitudenkurvan vid olika $curve$ värden.

Genom att applicera samma regler på tillståndsmaskinen som i det enkla envelopet tillåts nu ett mer komplex envelope där användaren kan konfigurera amplitudkurvan mer precist. Som tidigare nämnt så är sustain ett speciellt steg, detta gäller även i fallet med det mer komplexa envelopet. För att fler ljud ska kunna efterliknas lades även en valfri tidsbegränsning in på sustain. Detta gör att ljudet kan avta även om en tangenten trycks ned. Ett typisk exempel på ett sådan ljud är ett piano där

vibrationen i strängarna långsamt avtar även när pianisten håller ned en tangent eller sustainpedalen. Hur snabbt ljudet avtar och kurvformen på den kurvan kan ställas in på samma sätt som i de andra stegen.

Stödet för att användaren både ska kunna spela mjukt och hårt på synten valdes att implementeras i envelopet. Detta ansågs vara en lämplig placering då envelopet redan hade full kontroll av amplituden av signalen som kommer ut ifrån oscillatorerna. Implementeringen gjordes genom att en konstant mellan 0.0 och 1.0 bestäms varje gång en tangent trycks ner. Denna konstant multipliceras sedan med *level* i alla steg. Detta gör att om användaren trycker ned en tangent löst så kommer en mer dämpad ton spelas. Anledningen till att konstanten appliceras på *level* istället för *amplitud* är för att hastigheten på envelopet ska upplevas som lika när en tangent trycks ned hårt respektive löst.

När alla dessa beräkningar är gjorda sätts värdet på amplituden som sedan multipliceras med ett sampel från oscillatoren. Denna processen appliceras på alla sampels som oscillatoren genererar.

3.5 Butterworthfilter

Typen av filter som används som låg- och högpasfilter är ett butterworthfilter. Egenskapen för butterworthfilter, att passbandet har en flat frekvensrespons, ansågs vara önskvärd för projektet då frekvenserna inom passbandet inte bör förändras av filtret, utan låta på samma sätt som om filtret inte applicerades på signalen.

Första steget var att designa ett teoretiskt filter. För att hålla nere på filtrets komplexitet valdes ett andra ordningens butterworthfilter, d.v.s. ett med två poler. Ett filter med fler än två poler skulle ha en brantare kurva vid övergången från pass- till stoppbandet men implementationen skulle kräva fler beräkningar. Ett filter utav andra ordningen ansågs därmed vara lämpligt eftersom det inte finns några specifika krav på övergången från pass- till stoppbandet.

3.5.1 Framtagning av modell för lågpasfilter

Efter framtagning av en överföringsfunktion för ett lågpasfilter i Laplace-domänen och diskretisering enligt avsnitt 2.3, fås följande överföringsfunktion i \mathcal{Z} -domänen:

$$H_{LP}(z) = \frac{z^{-2} \left(\frac{\omega_c^2}{a_0} \right) + z^{-1} \left(\frac{2\omega_c^2}{a_0} \right) + \frac{\omega_c^2}{a_0}}{z^{-2} \left(\frac{4f_s^2 - 2f_s\omega_c\sqrt{2} + \omega_c^2}{a_0} \right) + z^{-1} \left(\frac{2\omega_c^2 - 8f_s^2}{a_0} \right) + 1}, \quad (3.1)$$

där $a_0 = 2f_s\omega_c\sqrt{2} + 4f_s^2 + \omega_c^2$. Låt oss nu införa beteckningar för koefficienterna i överföringsfunktionen (se formel 3.2). Notera att alla koefficienter förutom a_0 divideras med a_0 för att få ettan i nämnaren som syns i formel 3.1.

$$\begin{aligned} b_2 &= \frac{\omega_c^2}{a_0} & a_2 &= \frac{4f_s^2 - 2f_s\omega_c\sqrt{2} + \omega_c^2}{a_0}, \\ b_1 &= \frac{2\omega_c^2}{a_0} & a_1 &= \frac{2\omega_c^2 - 8f_s^2}{a_0}, \\ b_0 &= \frac{\omega_c^2}{a_0} & a_0 &= 2f_s\omega_c\sqrt{2} + 4f_s^2 + \omega_c^2. \end{aligned} \quad (3.2)$$

Efter inverstransformering utav filtrets system $Y(z) = H(z)X(z)$ (där $Y(z)$ är utsignalen, $H(z)$ är överföringsfunktionen enligt funktion 3.1 och $X(z)$ insignalen) från \mathcal{Z} -domänen till tidsdomänen fås följande formel:

$$y[n] = b_2x[n-2] + b_1x[n-1] + b_0x[n] - a_2y[n-2] - a_1y[n-1], \quad (3.3)$$

där $x[n]$ är insignalen, $y[n]$ är utsignalen och n betecknar nuvarande sampelindex. Notera att formeln innehåller två föregående insignals- och utsignalsvärden då filtret är utav andra ordningen. Ett filter utav högre ordning skulle innehålla fler föregående värden (med fler koefficienter) och därmed skulle antalet beräkningar öka.

3.5.2 Framtagning av modell för högpasfilter

På liknande sätt som för lågpasfiltret, tas en överföringsfunktion fram enligt avsnitt 2.3 som ser ut som följande:

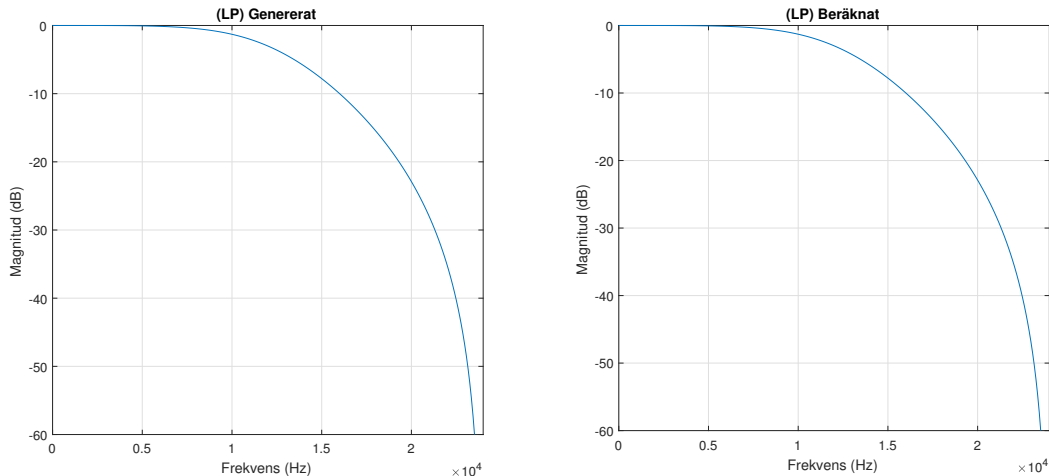
$$H_{HP}(z) = \frac{z^{-2} \left(\frac{4f_s^2}{a_0} \right) + z^{-1} \left(\frac{-8f_s^2}{a_0} \right) + \frac{4f_s^2}{a_0}}{z^{-2} \left(\frac{4f_s^2 - 2f_s\omega_c\sqrt{2} + \omega_c^2}{a_0} \right) + z^{-1} \left(\frac{2\omega_c^2 - 8f_s^2}{a_0} \right) + 1},$$

där $a_0 = 2f_s\omega_c\sqrt{2} + 4f_s^2 + \omega_c^2$. Detta ger i slutändan koefficienterna i formel 3.4 som kan användas för implementering enligt formel 3.3.

$$\begin{aligned}
 b_2 &= \frac{4f_s^2}{a_0} & a_2 &= \frac{4f_s^2 - 2f_s\omega_c\sqrt{2} + \omega_c^2}{a_0} \\
 b_1 &= \frac{-8f_s^2}{a_0} & a_1 &= \frac{2\omega_c^2 - 8f_s^2}{a_0} \\
 b_0 &= \frac{4f_s^2}{a_0} & a_0 &= 2f_s\omega_c\sqrt{2} + 4f_s^2 + \omega_c^2
 \end{aligned} \tag{3.4}$$

3.5.3 Teoretisk verifiering

De teoretiska filtren verifierades genom implementering i Matlab och jämförelse med filter genererade med en Matlab-funktion `butter` som automatiskt genererar butterworthfilter [23]. Figur 3.4 visar frekvensresponskurvan för det beräknade respektive genererade filtret med brytfrekvensen $f_c = 12\text{kHz}$.

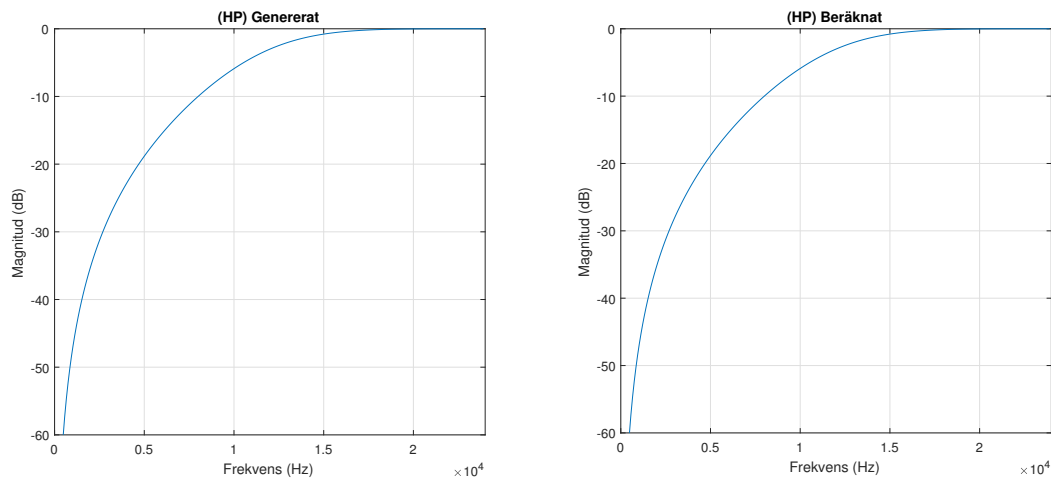


(a) Genererat med Matlab

(b) Implementerat enligt beräkningar

Figur 3.4: Frekvensresponskurvor för lågpasfilter med brytfrekvensen 10kHz.

Vid brytfrekvensen ska förhållandet mellan in- och utsignalens magnitud vara $1/\sqrt{2}$, alltså skall responskurvas magnitud vid f_c vara $20 \cdot \log_{10}(1/\sqrt{2}) \approx -3\text{dB}$ vilket stämmer i figur 3.4. Samma gäller även högpasfiltren vars frekvensresponskurvor syns i figur 3.5.



(a) Genererat med Matlab

(b) Implementerat enligt beräkningar

Figur 3.5: Frekvensresponskurvor för högpasfilter med brytfrekvensen 12kHz.

3.5.4 Praktisk verifiering

Efter faktisk implementering av båda filtertyperna i synten, användes vitt brus för att undersöka responskurvan i praktiken. Ljudet analyserades med virtuella spektrumanalysatorn SPAN [22] efter det hade filtrerats (se figur 3.6). Även här kontrollerades att ljudets magnitud vid brytfrekvensen är 3dB lägre än vid passbandet. Mätresultat för fyra brytfrekvenser för låg- och högpasfilter syns i tabell 3.1 respektive 3.2. Skillnaden mellan amplituden vid passbandet och brytfrekvensen ansågs vara tillräckligt nära 3dB för att vara ett acceptabelt resultat. Figur 3.6 visar även en flat frekvensresponskurva för passbandet.

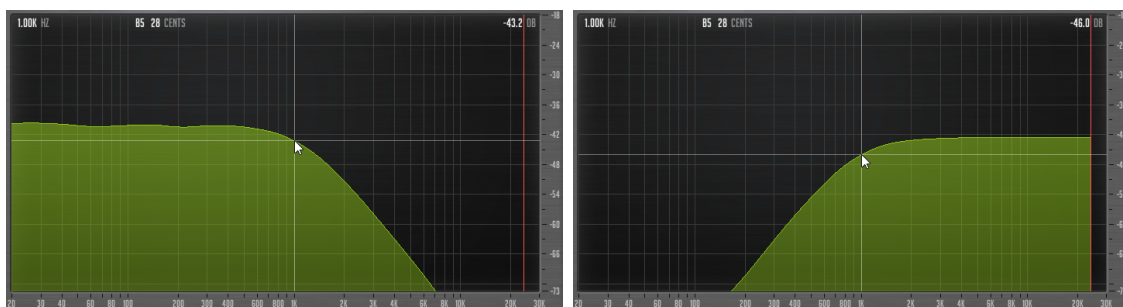
Som tabellerna visar har avvikelser uppstått i de olika testfallen, troligtvis på grund av den manuella avläsningen. Denna verifieringsmetoden kompletteras därmed med automatiserade tester för att verifiera filtrens korrekthet utan manuella avläsningar och därmed högre säkerhet (se avsnitt 3.10).

Tabell 3.1: Mätresultat för lågpasfilter.

Brytfrekv. (kHz)	Vid brytfrekv. (dB)	Vid passband (dB)	Skillnad (dB)
0,2	-43,2	-40,3	2,9
1	-43,4	-40,3	3,1
6	-43,4	-40,1	3,3
12	-43,5	-40,1	3,4

Tabell 3.2: Mätresultat för högpasfilter.

Brytfrekv. (kHz)	Vid brytfrekv. (dB)	Vid passband (dB)	Skillnad (dB)
0,2	-46,5	-42,6	3,9
1	-46,0	-42,6	3,4
6	-46,0	-42,6	3,4
12	-46,0	-42,7	3,3



(a) Lågpasfilter

(b) Högpasfilter

Figur 3.6: Frekvensresponskurvor för filter med brytfrekvensen 1kHz.

3.6 Reverb

Eftersom faltnings-metoden ansågs, baserat på gruppmedlemmarnas kunskap, vara enklare att implementera och ger dessutom ett mer realistiskt resultat valdes den metoden.

3.6.1 Faltningsalgoritm

Faltning av en insignal $x[n]$ med ett ändligt impulssvar $h[n]$ av längd N definieras enligt [24, s.11]:

$$y[n] = x[n] * h[n] = \sum_{k=0}^{N-1} x[n-k] \cdot h[k] \quad (3.5)$$

En direkt implementation av definitionen är enkel men medför en kvadratisk tidskomplexitet $\mathcal{O}(N^2)$ [24, s.15]. Detta skulle vara acceptabelt vid korta impulssvar men eftersom ett impulssvar för ljudet kan ha en storleksordning på exempelvis 10s, vilket med en samplingsfrekvens $f_s = 48000$ ger ett $N = 10 \cdot f_s = 480000$ långt impulssvar, sökes en snabbare lösning.

En allmänt känd egenskap hos faltningsoperationen är att den i frekvensdomänen representeras av multiplikation. Faltningsprocessen kan alltså genomföras med transformering av insignalen och impulssvaret, multiplikation och därefter inverstransformering av resultatet (se ekvation 3.6).

$$y[n] = \mathcal{Z}^{-1}(\mathcal{Z}(x[n]) \cdot \mathcal{Z}(h[n])) \quad (3.6)$$

Transformeringen och inverstransformeringen kan utföras med FFT-algoritmen (Fast Fourier Transform) som har tidskomplexiteten $\mathcal{O}(N \log N)$ [24, s.38]. En FFT-implementation som ingår i JUCE-ramverket skulle därmed kunna användas. En direkt implementation av metoden enligt ekvation 3.6 är dock inte lämplig för en realtidsapplikation med långa impulssvar. Faltning med ett impulssvar av längd N kräver nämligen en lika lång insignal. Operationen utförs utöver det på ett helt block av längd N för att få ut en utsignal av samma längd. Om impulssvaret är exempelvis 10s långt måste alltså 10s av insignalen buffras innan någon utsignal kan beräknas vilket ger en fördröjning på 10s, något som inte är lämpligt för en realtidsapplikation som en synt.

En metod för att komma undan fördröjningen kallas Overlap-Add [24, s.76] och innebär att insignalen delas upp i mindre block av storlek B som innan transformering fylls ut med nollor till storlek $B + N - 1$. Därefter kan faltningen ske i frekvensdomänen med ett impulssvar som också är fyllt med nollor till samma storlek. Resultatet är en utsignal med längden $B + N - 1$ [24]. Då utsignalen är längre än blockstorleken B måste sista $N - 1$ sampel sparas och adderas när nästa utsignalsblock beräknas [24]. Eftersom beräkningar i ett VST-program sker blockvis med ställbar blockstorlek B_{in} (som med JUCE självständiga läge kan vid samplingsfrekvensen 48kHz ställas mellan 144 och 2048 sampel) skulle denna uppdelning kunna användas tillsammans med Overlap-Add metoden. Nackdelen med denna metoden är att en transformering och inverstransformering av $B_{in} + N - 1$ värden krävs vid varje beräkning av en utsignal av längd B_{in} . Eftersom det antas att ett impulssvar med längden N kommer vara betydligt längre än B_{in} sökes en metod som inte kräver en lika lång transformering.

3.6.2 Likformigt uppdelad faltning

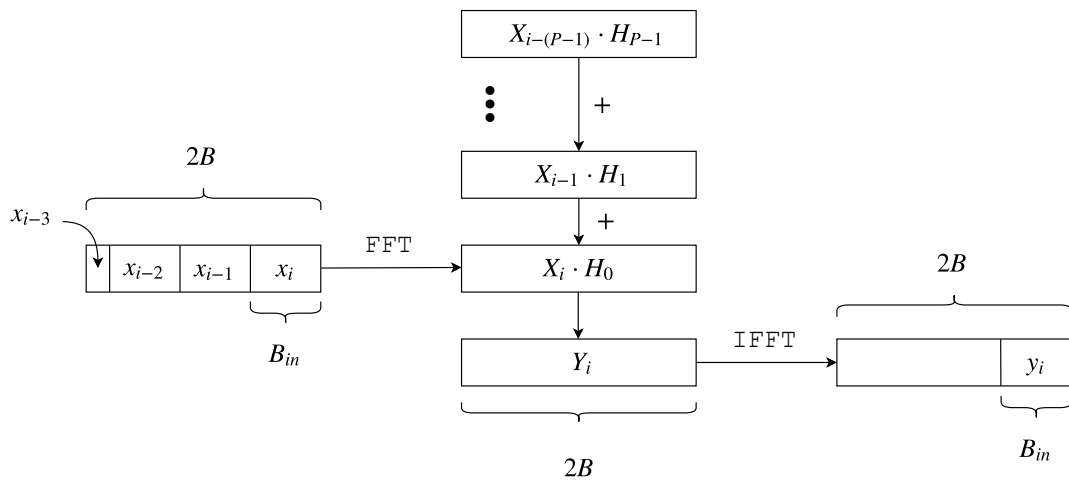
Metoden som är lämplig för vårt ändamål, d.v.s. med mycket längre impulssvar än insignalsblocken är likformigt uppdelad faltning (eng. uniformly partitioned convolution) [24, s.105]. Den liknar Overlap-Add-metoden med skillnaden att även impulssvaret delas upp i lika stora block som insignalen. Metoden illustreras i figur 3.7 och fungerar som följande:

- Impulssvaret H med längden N delas upp i P block $\{H_0, H_1, \dots, H_{P-1}\}$ av storleken B där B är närmaste större eller lika tvåpotens till insignalens blockstorlek B_{in} . Om N inte är delbart med B fylls impulssvaret ut med nollor tills

dess längd blir delbar med B . Blocken transformeras med FFT av storlek $2B$ och lagras [24]. Detta behöver endast ske en gång då ett impulssvar läses in.

- Ett insignalsblock x_i med storlek B_{in} kommer in och placeras längst till höger i en buffer med storleken $2B$. Till vänster om det nya insignalsblocket placeras tidigare insignalsblock tills buffern är full. Buffern transformeras sedan med FFT av storlek $2B$ till X_i och lagras [24].
- Det nyss transformerade insignalsblocket X_i multipliceras i frekvensdomänen med impulssvarets första block H_0 . Därefter multipliceras i ordning tidigare lagrade insignalsblock med respektive impulssvarsblock. Resultaten adderas i frekvensdomänen, inverstransformeras och resultatets sista B_{in} värden blir till den nuvarande utsignalen y_i [24].

Notera att endast $P - 1$ transformerade insignalsblock X samt $\lceil \frac{2B}{B_{in}} \rceil$ insignalsblock x behöver lagras.



Figur 3.7: Likformigt uppdelad faltning med en viss blockstorlek B_{in} .

Denna metod använder sig endast utav en transform och inverstransform av storlek $2B$ för beräkning av varje utsignalsblock. Det existerar snabbare algoritmer för vårt ändamål som exempelvis icke-likformigt uppdelad faltning (eng. non-uniformly partitioned convolution) [24, s.147] men på grund av dess högre implementationskomplexitet bestämdes att ovan beskrivna algoritmen ska användas.

Det visar sig dock i samband med antagandet att insignalsblocket har fast blockstorlek B_{in} , att den valda algoritmen medför vissa nackdelar. VST-gränssnittet garanterar nämligen inte att insignalsblocket har fast storlek. Den ställbara storleken B_{in} utgör snarare en maximal blockstorlek B_{max} . Beroende på VST-värd kan den faktiska blockstorleken för insignalen B_{in} vara mindre än B_{max} och dessutom ständigt variera. Reverb-effekten fungerar i dessa fall inte. Genom tester har det exempelvis visat sig att musikproduktionsprogrammet Reaper använder fasta blockstorlekar

medan ett annat, FL Studio, inte gör det. FL Studio kan ställas in till att använda fasta blockstorlekar vilket får reverb-effekten att fungera men informerar då användaren att tillägget är felaktigt implementerat. Problemet skulle kunna lösas enkelt genom buffring av insignalen men skulle då resultera i en icke-önskvärd fördröjning. Försök gjordes för att anpassa algoritmen för hantering av godtyckliga blockstorlekar $B_{in} \leq B_{max}$ utan fördröjning men problem stöttes då på. JUCE-ramverkets medföljande FFT-algoritm visade sig prestera dåligt vid små blockstorlekar. På grund av problemets höga komplexitet och tidsbrist beslutades att inte implementera stöd för varierande blockstorlekar. Reverb-effekten informerar istället användaren att fasta blockstorlekar måste användas om varierande storlekar upptäcks.

3.6.3 Impulssvar

I implementationen får användaren möjlighet att välja bland ett antal medföljande impulssvar eller använda egna, exempelvis hämtade från internet. De medföljande impulssvaren är dels hämtade från internet, licenserade under "Public domain"-licensen [25] men ett antal är självinspelade klappljud i diverse utrymmen, som i ett badrum eller en trappuppgång. I synten finns möjlighet för att ställa in mixning av originalsignalen och den faltade signalen vilket låter användaren justera effektens inverkan.

3.7 Delay

I projektet implementerades ett simpelt delay som simulerar ett ekot på högst fyra sekunder. Delaykomponenten har därför ett ljudblock med storlek $4 \cdot f_s$, där R_s är samplingshastigheten. Ljudblocket uppdateras sedan varje gång ett nytt ljudblock har genererats. Komponentens tre stycken parametrar:

Rate: Ett flyttal i intervallet $[0.125, 4.0]$ som beskriver längden på ekot.

Time: Ett flyttal i intervallet $[0.0, 1.0]$ som beskriver hur snabbt ekot avtar.

Sync: Ett boolskt värde som bestämmer om *Rate* räknas i sekunder eller taktslag.

När delaykomponenten startar adderar den det genererade ljudblocket till dess egna ljudblock i intervallet $[0, Rate \cdot f_s]$, multiplicerar värdena med *Time* samt adderar dem till det genererade ljudblocket. Om *Sync* är sann representerar *Rate* taktslag per sekund. Om *bpm* är mindre än 60.0 och *Rate* är satt till 4.0 kommer dock inte ljudblocket vara tillräckligt stort. Eftersom *bpm* under 60.0 är en ovanlig förekomst så stödjer inte delaykomponenten synkning med takter under 60.0 *bpm*.

3.8 Distortion

Implementationen av distortioneffekten i synten följde formell 2.8 och gjordes på oscillatornivå, d.v.s. varje oscillator fick en egen distortionkomponent. Effekten appliceras innan signalen filtreras, då den adderar en övertoner till signalen. Något som är värt att belysa är att dessa övertoner kan överstiga nyquistfrekvensen.

3.9 Grafiskt användargränssnitt

Det grafiska användargränssnittet är det som låter en användare att interagera med synten på ett intuitivt sätt. Gränssnittets design är inspirerad av både hårdvaru- och virtuella syntar. Implementationen av gränssnittet gjordes iterativt allteftersom ny funktionalitet implementerades.

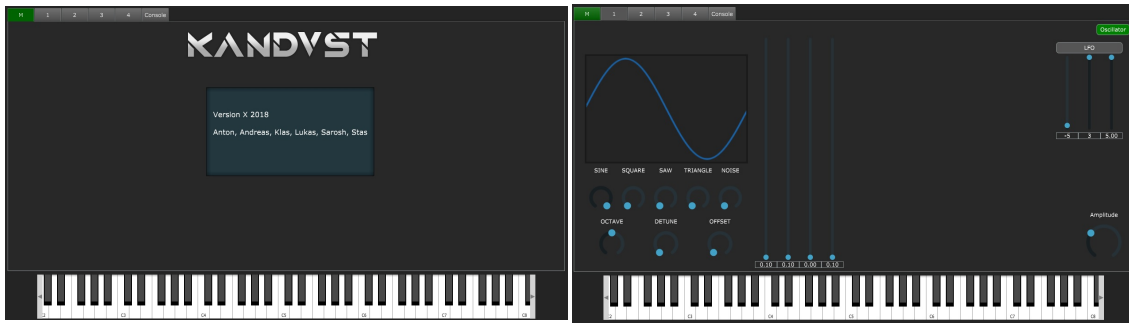
Då de preliminära kraven för syntens funktioner var specificerade så ritades en skiss på hur det grafiska gränssnittet skulle se ut. Denna skiss inspirerades av befintliga synthesizers som Sylenth1 [26] och Serum [27], men designen influerades även starkt av gruppens önskemål och preferenser.



Figur 3.8: Den första skissen av det grafiska användargränssnittet.

Under utvecklingen förändrades önskemålen för användargränssnittet. Exempelvis fanns ett önskemål att två oscillatorer skulle finnas, som man kan se i skissfiguren (figur 3.8). Efter fortsatt utveckling av synten ändrades det till fyra oscillatorer. Detta ledde till att gränssnittet består av ett flertal flikar för de olika oscillatorerna och en huvudflik. Huvudfliken hanterar de funktioner som är gemensamma för oscillatorerna och innehöll även en informationsruta. Det var tänkt att rutan skulle innehålla information om syntens och även möjliggöra att spara och ladda förinställningar.

Vidare under utvecklingen gjordes valet att varje oscillator ska kunna kombinera de fem vågformerna till en unik vågform som användaren själv får skapa. Varje



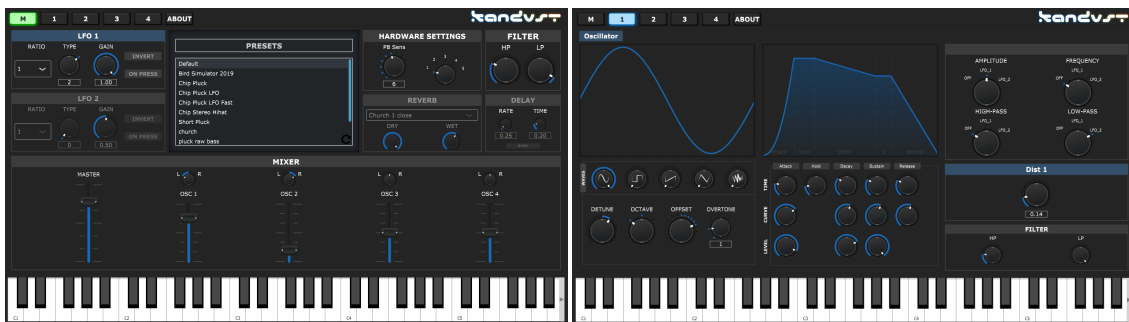
(a) Huvudflik med informationsruta. (b) En oscillatorflik.

Figur 3.9: Första gränssnittet.

oscillatorflik innehåller enskilda inställningar för oscillatorn (se figur 3.9b). Det finns även en konsolflik för att kunna styra vissa funktioner direkt under utvecklingen.

Figur 3.9a visar hur hela synten tillsammans med oscillatorflikarna (figur 3.9b) såg ut i början av utvecklingen.

Det skapades egna vrid- och skjutreglage som skulle användas i gränssnittet (figur 3.10). Gränssnittet förändrades under utvecklingen för att öka användarvänligheten och bli estetiskt tilltalande. I figur 3.10a syns en panel med förinställningar som lades till enligt specifikationen samt våra egna reglage och logotyp. Funktionen att kunna välja tema lades till. Detta syns i figur 3.10c och 3.10d där temafärgen är rosa respektive grön till skillnad från 3.10b där temafärgen är blå, som är standardtemat.



(a) Ny huvudflik med förinställningsruta. (b) Ny oscillatorflik.



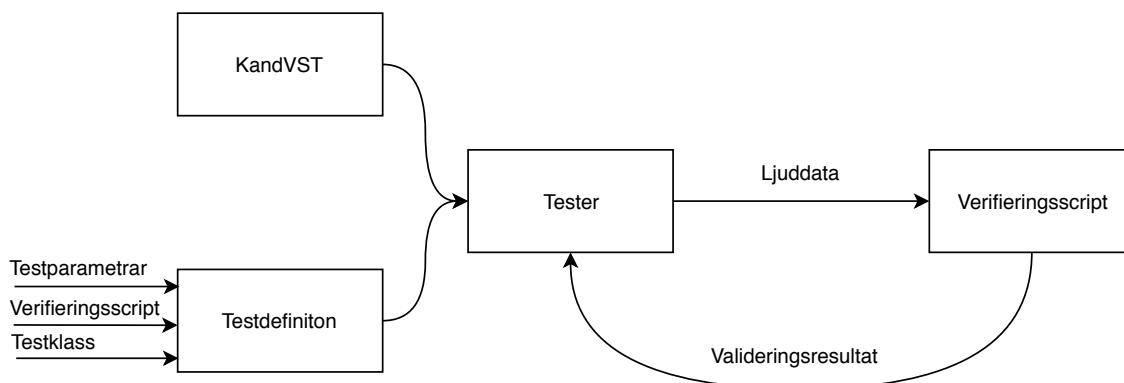
(c) Oscillatorflik med rosa tema. (d) Oscillatorflik med grönt tema.

Figur 3.10: Nytt gränssnitt med egna reglage, logotyp, och teman.

3.10 Systemtestning

En viktig aspekt vid utveckling av mjukvara är testning och verifiering. Som nämnts tidigare har vissa komponenter testats manuellt. Problemet med att manuellt verifiera alla delar av programvaran är dels att det är en tidskrävande process, samt att manuell verifiering endast testar en specifik version av programvaran. Om det genomförs någon ändring av mjukvaran behöver verifieringen göras om.

För att undvika onödigt arbete och öka kvalitén på den färdiga produkten utvecklades ett automatisk testverktyg. Testverktyget är ett fristående program, bestående av en egenutvecklad VST-värd. Det innebär att testverktyget kommunicerar med synten på samma vis som ett musikprogram skulle gjort. VST definierar även metoder för att konfigurera tillägg. Ett VST-tillägg består av två delar, ett användargränssnitt och en ljudbehandlingsdel. Kommunikationen mellan de två delarna sker enligt VST-API:et med hjälp av konfigurerbara parametrar, definierade av tillägget. När ett reglage i användargränssnittet ändras kommuniceras detta till värdmjukvaran i form av en parameterändring, vilken i sin tur vidarebefordrar parameterändringen till ljudbehandlingsdelen. I testverktyget skapas aldrig användargränssnittet. Istället genereras alla parameterändringar av testmjukvaran. Eftersom testverktyget beter sig som en VST-värd, mottar den även alla utgående ljudsignaler som synten genererar. Istället för att spela upp dessa som ett vanligt musikprogram lagras och analyseras de.



Figur 3.11: Testverktygets struktur.

En översiktsbild av testverktygets struktur återfinns i figur 3.11. Som framgår av bilden består indatan till testverktyget dels av det färdiga syntprogrammet, benämnt KandVST, och en eller flera testdefinitioner. Varje testdefinition består av en lista av parametervärden, d.v.s. syntens konfiguration, ett verifieringsscript, samt en testklass. Testklassen är ett kort C++ program som genererar midi-datan för testet, d.v.s. den simulerar en person som spelar på synten. Testverktyget läser testdefinitionen, varpå den konfigurerar KandVST enligt parameterlistan, och därefter exekverar den testklassen för att generera midi-datan som skickas till KandVST. Synten mottager midi-datan och genererar en ljudsignal som sänds tillbaka till testverktyget. Testverktyget mäter tiden det tar för syntens att generera ljudsigna-

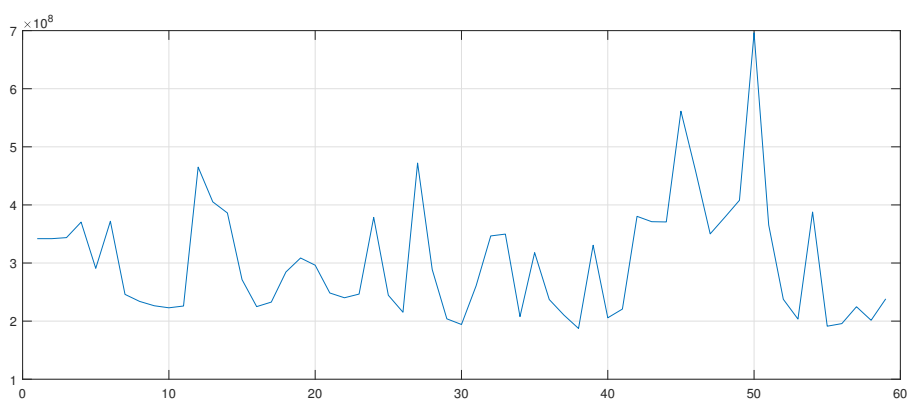
len och sparar de resulterade ljudsignalerna. När KandVST är färdig startar testverktyget det python-script som ingick i testdefinitionen, och skickar ljuddata till scriptet. Verifieringsscriptet analyserar ljuddata i syfte att avgöra huruvida den överstämmer med det förväntade resultatet. När scriptet avgjort om testet lyckats, kommuniceras detta tillbaka till testverktyget. När alla testdefinitioner behandlas av testverktyget rapporteras resultat och tidsåtgång av varje testfall till användaren.

Följande tester utförs av synthesizern

- *Simultana toner*: Testar att 16 simultana tangent nedtryck hanteras. Testet utförs genom att konfigurera synten att endast generera sinusformade ljudvågor, för att sedan simultant trycka ned 16 tangenter. Verifikationscriptet godkänner testet om den finner 16 distinkta toppar vid förväntade frekvenser vid frekvensanalys av ljuddata.
- *Oscillatorfrekvens*: Testar att syntens oscillator genererar önskad frekvens. Synten konfigureras för sinusformade ljudvågor, och en ton spelas. Testet godkänns om verifikationscriptet finner en distinkt topp vid förväntad frekvens i frekvensspektrumet.
- *Högpasfilter*: Testar att högpasfiltret har rätt försvagning vid konfigurerad brytfrekvens. Testet genomförs genom att generera en ljudsignal innehållande två frekvenser, en vid brytfrekvensen och en vars frekvens är mycket större än brytfrekvensen. Testet lyckas om frekvensspektrumet innehåller två toppar, den lägre toppen ska ligga vid gränsfrekvensen och vara $3dB$ längre jämfört med den högre toppen.
- *Lågpasfilter*: Testet fungerar på samma sätt som högpasfiltret, men istället för att den andra frekvenskomponenten är mycket större än brytfrekvensen så är den lägre.
- *Oktavskifte*: Testet fungerar på samma sätt som oscillatorfrekvenstestet, men oktavparametern är satt att transponera frekvensen tre oktaver högre än originalfrekvensen. Verifikationen fungerar också på samma sätt, d.v.s. bekräftar att det finns en topp vid förväntad frekvens i frekvensspektrumet.
- *Tonskifte*: Fungerar på samma sätt som oktavskifte och oscillatorfrekvens, men transponerar frekvensen istället med tonskiftsparametern.
- *Detune*: Fungerar på samma sätt som tidigare nämnda frekvensskiftstester, men transponerar istället med detuneparametern.
- *Dist*: Testar att distortioneffekten förvränger signalen på ett korrekt sätt. Tröskelvärdet sätts med hjälp av distparametern, och därefter spelas en sinuston på högsta amplitud. Verifikationscriptet verifierar sedan att amplituden inte överskrider tröskelvärdet.

För att systematisera användningen av testverktyget konfigurerades en byggserv

kopplad till GitHub, där all källkod förvaras. När koden ändras på Github informeras byggservern. Den kompilarar i sin tur hela kodbasen för att bygga synten och testverktyget. Om kodbasen kompilarar så startar servern testprogrammet som genomför alla tester. Huruvida bygget lyckades och eventuella testresultat rapporteras sedan tillbaka till GitHub. Om bygget lyckas så publicerar även servern de färdigbyggda KandVST-binärerna. Byggservern sparar binärerna och testresultat från bygget, vilket möjliggör en god överblick över projektet. T.ex. eftersom testresultaten loggas och dessa loggar innehåller tiden det tog att utföra testen, kan syntens prestanda över projektets gång övervakas. I figur 3.12 återfinns en graf över antalet klockcykler som åtgått till det simultana ton-testet över tid. Testerna är relativt känsliga för serverns belastning, så enskilda revisioners prestandapåverkan är svår att avgöra, men trender kan upptäckas.



Figur 3.12: Prestandan av simultana tontestet. Y-axeln visar antalet klockcykler som åtgick för att utföra testet och X-axeln visar byggnummer.

4

Diskussion

Strukturen som valdes för synten visade sig fungera bra. Den möjliggjorde ett parallellt utvecklingsarbete där flera olika delar kunde utvecklas individuellt för att sedan föras ihop. Alla val under projektet har inte varit självklara, motivering och en mer grundlig diskussion kommer att göras i detta kapitel.

4.1 Oscillator

Systemdesignsbeslutet att ha vågtabeller som använder sig av fourierserier för att skapa ljudvågor, var något som tydligt definierade syntens ljud. Detta på grund av att alla vågtabeller behöver initieras vid syntens start och sparas i minnet, som i sin tur implicerar att endast ett fåtal ljudvågor fick ingå i synten för att inte en för stor mängd minne skall behöva användas. Dessa ljudvågor kan ändras efter att de har samplats från vågtabellerna, men antalet ljudvågor att utgå ifrån är fortfarande begränsat. Om vågtabeller inte skulle användas i projektet skulle varje sampel behöva räknas ut i realtid och därmed kan funktionen som räknar ut signalen enkelt bytas ut, vilket öppnar upp för en större mängd olika ljudvågor. Eftersom dessa skulle behöva räknas ut i realtid skulle det också innebära färre kombinationer av olika ljudvågor. I detta fall ansågs ett bestämt antal ljudvågor vara tillräckligt för att skapa intressanta ljudvågskombinationer.

Valet att i varje oscillator ha möjligheten att mixa ihop alla fem vågformer är något som är, av oss känt, unikt (men inte på något sätt banbrytande). Denna funktionalitet gör varje oscillator till en egen synt, som sedan mixas ihop med andra ekvivalenta syntar innan de spelas upp. Det som möjliggör denna funktionalitet var att vi använder oss av vågtabeller, vilket gör att utökningen av att en oscillator producerar en signal till fem är bara fyra extra additioner, då värdena redan är uträknade. Att kunna mixa ihop dessa signaler med olika volym utökar också oscillatorns beräkningar med fem extra multiplikationer. Fyra additioner samt fem multiplikationer ansågs vara ett litet pris att betala för funktionaliteten det medförde.

I synten finns två globala lågfrekvensoscillatorer som det går att koppla flera parametrar från olika oscillatorer till. Till en början var tanken att alla parametrar i synten skulle kunna kopplas till lfo:n. Detta märktes snabbt att det var en svår

funktionalitet att uppnå, eftersom lfo:n inte kan ändra på parametrar samtidigt som de ändras av användaren. Utöver detta skulle också olika parametrar bete sig annorlunda när de var kopplade till ett lfo. Exempelvis för en vibrato-effekt vill man att frekvensen skall pulsera med originalfrekvensen i mitten, men om man vill påverka amplituden skall den helst bara sänkas från originalamplituden, då en höjning kan skapa förvrängning av ljudet. Detta ledde till den slutliga implementationen med bara ett fåtal lfo-kopplingsbara parametrar, där alla har sin egen implementation av hur de påverkas. Valet av två globala lågfrekvensoscillatorer innebar också att syntens olika parametrar endast kan "pulsera" i två olika hastigheter och med två olika vågformer. Alternativa implementationer skulle vara att antingen ha fler lågfrekvensoscillatorer eller att lägga en separat lfo för varje oscillatorkomponent. Det förstnämnda ansågs överflödigt både i funktionalitetssyfte samt den plats det skulle ta upp i användargränssnittet. Ett separat lfo för varje komponent skulle vara ett alternativ som gör det svårare att skapa en synkron helhet i det ljud man vill skapa. Valet att synkronisera lågfrekvensoscillatorerna med hjälp av information från den värd som använder synten, och inte bara internt hålla koll på fasen, var ett beslut som gav vissa konsekvenser. För det första så tillhandahåller inte alltid värden denna information. Om värden inte tillhandahåller denna information kommer lfo:ns fas bara fortsätta, vilket ger olika resultat beroende på när användaren trycker ned en tangent. Om en användare exempelvis har en sekvens med midi-noter som skall spelas, så kommer den låta annorlunda beroende på när den sätts igång. Detta är inte ett önskvärt beteende, men utan informationen att en sekvens har börjat spelas är det svårt att anpassa fasen. Dock så fungerar det att synka vid tangentnedtryck, men det är inte alltid den efterlängttade effekten.

4.2 Reverb

Användningen utav faltning med ett impulssvar för att åstadkomma en reverb-effekt gav ett mycket realistiskt låtande resultat, men implementationen av faltningsalgoritmen visade sig vara mer komplicerad än förväntat. Därav valet av en enklare faltningsalgoritm som medför tidigare nämnda begränsningar. Med mer tid och kunskap inom området skulle en effektivare algoritm kunna implementerats med färre begränsningar. Samma skulle även kunna åstadkommas genom användning av en färdig faltningsalgoritm. Viktigt att notera är att den slutgiltiga implementationen fungerar och att en bättre algoritm inte skulle medföra ett betydligt mer realistiskt resultat utan endast förbättra kompatibilitet.

Valet av annan metod än faltning skulle troligtvis leda till ett tillräckligt bra resultat, dock inte lika realistiskt. Det skulle även kunna medföra fler inställningsmöjligheter för användaren, då ljudet faltningsreverb ger helt och hållet beror på impulssvaret, utan möjlighet för annan justering. Utöver det skulle inte behovet för anskaffning av impulssvar finnas.

Slutligen bör det noteras att reverb-effekten appliceras sist i syntens pipeline. Vid

användning av synten i ett musikproduktionsprogram kan effekten stängas av och syntens ljudsignal kopplas till ett annat reverbtillägg om användaren så önskar.

4.3 Filter

Valet av butterworthfilter för låg- och högpasfilter resulterade i, som förväntat, en flat frekvensresponskurva i passbandet. Andra filtertyper skulle kunna medföra en brantare kurva vid övergången från passbandet till stoppbandet vid samma komplexitet men en icke-flat frekvensresponskurva skulle då förändra frekvenser i passbandet vilket inte anses vara önskvärt. Dessutom finns inga specifika krav på hur övergången från passband till stoppband ska se ut och eftersom effekten som det implementerade filtret har på ljudet anses vara acceptabel, är Butterworthfiltret det rätta valet för detta ändamål. En brantare övergång från pass- till stoppbandet skulle kunna åstadkommas, om så önskades, genom användning av ett filter utav högre ordning.

Filtren implementerades i tidsdomänen men skulle även kunna implementeras i frekvensdomänen. På grund av filtrets låga komplexitet var det enkelt att implementera i tidsdomänen och en implementation i frekvensdomänen skulle därmed antagligen bli svårare. En implementation i frekvensdomänen skulle även troligtvis vara mer beräkningsintensiv på grund av transformeringsoperationerna.

4.4 Prestanda vs kvalitet

Under projektet gjordes hela tiden avvägningar kring kvalitet av ljudet mot prestanda av synten. Det kanske tydligaste beslutet var att använda vågtabeller istället för att i realtid generera vågformerna. Genom att använda vågtabeller så kan prestandakravet minska men en viss precision och flexibilitet i vågformerna ges upp. Prestandavinsten väger dock upp den minskade precisionen. En stor nackdel med vågtabeller är den förlängda uppstarttiden på synten då tabellerna genereras. Om istället ett mer realtidsbaserat genereringssätt valts så hade synten haft kortare uppstartstid men varit mer prestandakrävande när den användes. Beaktar man på dessa aspekter så är vågtabeller det tillvägagångssätt som passar bäst. För att få ökad precision i signalen så används interpolation, på grund av detta så kan vågtabellerna innehålla ett mindre antal värden utan att förlora märkbar precision vilket resulterar i mindre minnesanvändning och en kortare uppstartstid. Nackdelen med interpolation är att det krävs fler klockcykler per resulterade ljudsämpel än om interpolation inte skulle användas. Skulle dock inte interpolation användas så skulle vågtabellerna behöva vara betydligt större för att få samma upplevda resultat.

Något som testades men sedan inte användes var cachning av signalen. När en ton spelades så kunde en våglängd av den exakta tonen sparas för att användas när

den spelades igen. Detta gjorde att en del tid kunde sparas eftersom signalen inte behövde räknas fram utifrån vågtabellerna utan istället räckte det att läsa från minnet.

De automatiska testerna visade att cachningen förde in ett fel i signalens frekvens. Även om del cpu-tid kunde sparas så avvecklades det tillgångssättet eftersom kvalitén av signalen ansågs vara viktigare än att spara cpu-tid.

4.5 Testning

Testningsverktyget har visat sig värdefullt då det pekat ut flera brister i syntmjukvaran. Eftersom testningen skett kontinuerligt upptäcktes även kodförändringar som introducerade problem, och dess orsaker kan snabbt identifieras då endast en begränsad mängd kodförändringar behöver undersökas. Dock är det viktigt att vara medveten om att alla problem inte upptäcks av testverktyget. Speciellt så testas inte användargränssnittet automatiskt. Detta p.g.a. att svårigheter att konstruera en VST-värd som kan rendera användargränssnittet. Mot denna bakgrund är det tydligt att testverktyget inte ersätter manuell testning, men reducerar behovet.

Om projektet skulle genomföras igen hade det vara önskvärt att introducera automatisk testning tidigare. Det tog många veckor innan testningen implementerades, så det var endast möjligt att utnyttja dess fördelar under projektets två sista månader. Dock tog testverktyget lång tid att utveckla, ca. tre veckor, och stor kunskap om VST krävdes för att implementera en VST-värd. Den officiella VST-dokumentationen var svår att tyda och tredjepartsdokumentation beskrev endast hur VST-tillägg konstrueras, inte VST-värdar. Det är möjligt att testverktyget skulle tagit längre tid att utveckla om det påbörjades tidigare, p.g.a. att det fanns lägre förståelse för VST i början av projektet. Under konstruktionen av synten abstraherade JUCE-ramverket bort det mesta av VST detaljerna, men dessa abstraktioner kunde inte utnyttjas till testverktyget, då JUCE primärt är designat för att implementera VST-tillägg och inte VST-värdar.

4.6 JUCE

Beslutet att använda ramverket JUCE vid utveckling av synten har genomsyrat de allra flesta aspekter av dess konstruktion. Multitrådning, ljuddatastrukturer och funktioner för att rita grafiska användargränssnitt är blott ett litet urval på den funktionalitet JUCE tillhandahåller. Denna funktionalitet tillsammans VST-abstraktionen besparade mycket tid och tillät projektet att tidigt producera en fungerande prototyp. Dock understryker vi att i ljudbehandling delarna som pipeline, envelope och multifunktionsoscillatorer osv. användes JUCE-ramverket restriktivt. JUCE datastrukturer utnyttjades, men ingen av JUCE befintliga ljudbehandlingsfun-

kionalitet, förutom FFT, användes. På så sätt behölls full kontroll och förståelse för projektets centrala delar. Utan användningen av JUCE skulle med stor sannolikhet arbetet inte fortlöpt i samma tempo, vilket erfarades vid utveckling av testverktyget där JUCE valdes bort.

Att använda JUCE påverkar även slutprodukten. Eftersom JUCE abstraherar bort VST-detalyer samtidigt som det stödjer både version två och tre av VST-specifikationen, så stödjer även synten båda. I kontrast stödjer testverktyget endast version tre, då den radikalt annorlunda utformade version två ej är kompatibel utan signifikanta förändringar av tester-koden.

En andra aspekt av att använda JUCE är licensiering av mjukvaran. JUCE är en kommersiell mjukvara som kostar pengar att använda. Utöver det tillkommer krav på att visa en JUCE-logotyp vid uppstart samt att sända anonym användardata tillbaka till företaget bakom ramverket [28]. Dessa avgifter och krav kan dock undgå då JUCE även erbjuds under licensen GPLv3 [29], men det innebär att även synten måste licensieras under GPLv3. GPLv3 innebär många åtaganden, men det kanske största är kravet på fri distribution av källkoden. Distribueras den färdiga synten, måste även användaren erbjudas en kopia av källkoden. Kraven för GPLv3 bedömdes dock acceptabla då vi redan från projektet början ämnade att göra källkoden tillgänglig. Då även VST version tre är tillgängligt under GPLv3 kan hela projektet licensieras under den licensen utan svårigheter.

4.7 Grafiskt användargränssnitt

Vid implementation av användargränssnittet prioriterades ej användarvänlighet, istället lades fokus på att implementera ny funktionalitet och först därefter reflektera kring hur och vart den skulle placeras i gränssnittet. Det var först då användarvänligheten beaktades. För att göra gränssnittet mer intuitivt och användarvänligt kunde det ha anordnats användartester. Den upplevda användarvänligheten för personer utanför gruppen bedöms därför som osäkert.

Det är möjligt att i framtiden fortsätta utvecklingen av gränssnittet, och då inkludera ny funktionalitet samt lägga fokus på användarvänlighet. Tanken bakom gränssnittet är att användaren ska testa sig fram för att skapa egna ljud. Därför har dokumentation om användning av synten inte prioriterats, något som skulle kunna förändras i framtiden.

Det finns väldigt få animationer i gränssnittet vilket skulle kunna utökas. Gränssnittet är det första man ser när synten används och det kan påverka upplevelsen av synten positivt om det var fler animerade komponenter i gränssnittet. Till exempel en volymmätare eller att färgen på vissa komponenter blinkar i takt med synten. Några grafiska animationer finns, som när man ändrar på vågformer eller på envelopet, men dessa är alla av en utilistisk karaktär, bortsett från möjligheten att kunna ändra temafärg. Syntens gränssnitt är inte långt ifrån första skissen (figur 3.8) även

om det saknas planerade funktioner som skulle kunna märkas om synten jämfördes med skissen.

4.8 Etiska aspekter

Det utvecklade musikinstrumentet har målet (och endast möjligheten) att generera ljudvågor som sedan kan användas för musikproduktion. En sådan handling kan inte på något sätt utgöra skada för någon individ eller samhället, där projektets produkt används som verktyg av någon som utgör skada.

Ett framtidsscenario som kan vara värt att utforska är om synten skulle bli så kompetent att den skulle konkurrera ut andra syntar som redan finns på marknaden. Det behöver inte leda till att andra företag går i konkurs, men det skulle kunna sätta press på dem. Trots dessa spekulationer så anses detta vara ett väldigt avlägset scenario då det idag existerar syntar med liknande funktionalitet, som i vissa fall också är gratis att använda vilket gör det svårt att slå ut dem från marknaden.

En positiv aspekt skulle kunna vara musiken i sig och hur den kan påverka människor. Dock är det då snarare artisten och inte instrumentet som ger upphov till den påverkan. Därmed anser gruppen att projektet inte innehåller några etiska aspekter värda att ta hänsyn till.

5

Slutsats

Resultatet av arbetet anses vara lyckat, då all funktionalitet som planerades finns med i synten och utöver det implementerades reverb effekten. Det grafiska gränssnittet anses tillfredsställande och funktionellt, trots det relativt begränsade fokus som lagts på det. Det tros att en person i den tilltänkta målgruppen, med minimal ansträngning, kan utnyttja synten. Dock har detta påstående inte verifierats. Utöver det blev ett antal utökningsfunktioner implementerade. Något som har underlättat arbetet är att gruppen i tidigt stadie redan hade en ganska klar bild över vad slutprodukten skulle innehålla samt att tidigare kunskap inom ämnet fanns. Tack vare att dessa idéer snabbt omvandlades till en grundläggande kodstruktur fortlöpte arbetet redan i en tidig fas i ett högt tempo. Tillsammans med utnyttjande av automatiska tester och JUCE-ramverket kunde en produkt av god kvalitet och funktionalitet skapas. Det finns dock funktioner och ändringar som inte fick plats inom projektets tidsram. Något gruppen hade önskat vore att implementera alternativa syntestekniker såsom FM-syntes.

Studierna och skapandet av den digitala synten har lett till utökad kunskap och insikt i ämnet. All teoretiska framtagningen av filter, faltningsekvationer för reverb, uträkningar av vågtabeller m.m. har gett djupare förståelse för signalbehandling och dess matematiska modeller. Även processen att överföra teori och modeller till kod och faktiskt programvara har varit lärorikt.

Vi är nöjda men den produkt och funktionalitet vi tillhandahåller. Delar av synten som den steglösa mixningen av vågformer i varje pipeline utmärker sig vid jämförelse med andra syntar. Hela källkoden är tillgänglig under öppen källkods-licensen GPLv3, den intresserade kan besöka projektet Github-sida [30].

Litteratur

- [1] (2009). A Brief History of the Synthesizer, URL: <https://documentation.apple.com/en/logicstudio/instruments/index.html#chapter=A>.
- [2] (2018). FM Tone Generators and the Dawn of Home Music Production (Chapter 2), URL: https://usa.yamaha.com/products/contents/music_production/synth_40th/history/chapter02/index.html.
- [3] M. Walker. (1999). Steinberg Cubase VST 3.7, URL: <https://web.archive.org/web/20150609081456/http://www.soundonsound.com/sos/sep99/articles/cubase37.htm>.
- [4] (2018). VST-instrument av Steinberg, URL: <https://www.steinberg.net/en/products/vst/absolute/start.html>.
- [5] (2018). ModelD, URL: <https://www.moogmusic.com/products/minimoog/minimoog-model-d%5C%C2%5C%AE>.
- [6] N. Magennis. (2008). The DX7, URL: <https://search.proquest.com/docview/213499919?accountid=10041>.
- [7] C. Phillips, J. Parr och E. Riskin, *Signals, Systems, and Transforms*. Pearson Education, 2011. URL: <https://books.google.se/books?id=YekuAAAAQBAJ>.
- [8] J. Watkinson. (2008). Explanation of 44.1 kHz CD sampling rate, URL: <http://www.cs.columbia.edu/~hgs/audio/44.1.html>.
- [9] R. Arora och W. Sethares. (2007). Adaptive Wavetable Oscillators, URL: <http://ieeexplore.ieee.org/document/4291842/>.
- [10] D. Crombie, *The Complete Synthesizer: A Comprehensive Guide*. Omnibus Pr & Schirmer Trade Books, 1982.
- [11] B. Academic. (april 2018). White noise, URL: <https://academic-eb-com.proxy.lib.chalmers.se/levels/collegiate/article/white-noise/76834>.
- [12] D. P. Havens och L. P. Huelsman. (april 2014). Electric filter, URL: <https://www.accessscience.com:443/content/electric-filter/216100>.
- [13] S. Butterworth, "On the theory of filter amplifiers", *Wireless Engineer*, årg. 7, nr 6, s. 536–541, 1930.
- [14] C. Nordling och J. Österman, *Physics Handbook for Science and Engineering*. Professional Publishing House, 2006. URL: <https://books.google.se/books?id=xBAWGQAACAAJ>.

- [15] S. C. Pei och H. J. Hsu, "Fractional Bilinear Transform for Analog-to-Digital Conversion", *IEEE Transactions on Signal Processing*, årg. 56, nr 5, s. 2122–2127, maj 2008, ISSN: 1053-587X. DOI: 10.1109/TSP.2007.912250.
- [16] C. M. Harris. (april 2014). Reverberation, URL: <https://www.accessscience.com:443/content/reverberation/584900>.
- [17] M. Chemistruck, K. Marcolini och W. Pirkle, "Generating matrix coefficients for feedback delay networks using genetic algorithm", i *Audio Engineering Society Convention 133*, Audio Engineering Society, 2012.
- [18] E. M. Wenzel. (april 2014). Virtual acoustics, URL: <https://www.accessscience.com:443/content/virtual-acoustics/757401>.
- [19] T. Laasko, *Splitting the Unit Delay*. IEEE Signal Processing Magazine, jan. 1996, vol. 13.
- [20] J. Smith och N. Lee. (2008). Elementary Digital Waveguide Models for Vibrating Strings, URL: https://ccrma.stanford.edu/realsimple/SimpleStrings/Nonlinear_Overdrive.html.
- [21] MIDI Association. (2018). MIDI 1.0 Specification, URL: <https://www.midi.org/specifications-old/item/table-1-summary-of-midi-message>.
- [22] A. Vaneev. (2018). Voxengo SPAN, URL: <http://www.voxengo.com/product/span/>.
- [23] The MathWorks, Inc. (2018). butter - Butterworth filter design, URL: <https://se.mathworks.com/help/signal/ref/butter.html>.
- [24] F. Wefers, *Partitioned convolution algorithms for real-time auralization*. Logos Verlag Berlin GmbH, 2015, vol. 20.
- [25] Creative Commons. (2018). Public Domain Certification, URL: <https://creativecommons.org/licenses/publicdomain/>.
- [26] (2018). Sylenth1 | LennarDigital, URL: <https://www.lennardigital.com/sylenth1/>.
- [27] (2018). Serum, URL: <https://www.xferrecords.com/products/serum>.
- [28] ROLI. (2017). JUCE 5 End User License Agreement, URL: <https://juce.com/juce-5-license>.
- [29] Free Software Foundation. (2007). GNU GENERAL PUBLIC LICENSE Version 3, URL: <https://www.gnu.org/licenses/gpl-3.0.en.html>.
- [30] (2018). KandVST Repository, URL: <https://github.com/VirtualRaven/KandVST>.

A

Fourierserier

A.1 Fyrkantsvåg

Defintion av fyrkantsvåg:

$$x(t) = \begin{cases} 0, & 0 < t < sq \cdot T_0 \\ A, & sq \cdot T_0 \leq t < T_0 \end{cases}$$
$$sq \in \mathbb{R} \wedge 0 \leq sq \leq 1$$

Beräkning av fourierkoefficienter

$$C_k = \frac{1}{T_0} \int_{T_0} x(t) e^{-j\omega_0 kt} dt = \frac{1}{T_0} \int_{sqT_0}^{T_0} A e^{-j\omega_0 kt} dt$$
$$\{k = 0\} \rightarrow C_0 = \frac{1}{T_0} \int_{sqT_0}^{T_0} A dt = \frac{AT_0(1 - sq)}{T_0} = A(1 - sq)$$

$$C_k = \{k \neq 0\} \rightarrow \frac{1}{T_0} \int_{sqT_0}^{T_0} A e^{-j\omega_0 kt} dt = \left[\frac{A j e^{-j\omega_0 kt}}{T_0 \omega_0 k} \right]_{sqT_0}^{T_0} = \left[\frac{A j e^{-j\omega_0 kt}}{2\pi k} \right]_{sqT_0}^{T_0}$$

$$C_k = \frac{A}{2\pi k} j (e^{-j\omega_0 k T_0} - e^{-j\omega_0 k T_0 sq}) = \frac{A}{2\pi k} j (e^{-j2\pi k} - e^{-j2\pi k sq})$$

$$C_k = \frac{A}{2\pi k} j (1 - (\cos 2\pi k sq - j \sin 2\pi k sq)) = \frac{A}{2\pi k} (j(1 - \cos 2\pi k sq) - \sin 2\pi k sq)$$

Sammanfattning av ovan beräkningar ger

$$C_k = \begin{cases} A(1 - sq), & k = 0 \\ \frac{A}{2\pi k} (j(1 - \cos 2\pi k sq) - \sin 2\pi k sq), & k \neq 0 \end{cases}$$

$$2C_k = \frac{A}{\pi k} \left(-\sin 2\pi ksq - j(\cos 2\pi ksq - 1) \right) = A_k - jB_k \rightarrow$$

$$A_k = \begin{cases} \frac{-A \sin 2\pi ksq}{\pi k}, & k \neq 0 \\ A(1 - sq), & k = 0 \end{cases}$$

$$B_k = \begin{cases} \frac{A(\cos 2\pi ksq - 1)}{\pi k}, & k \neq 0 \\ 0, & k = 0 \end{cases}$$

Notera att $sq = \frac{1}{2}$ ger att:

$$C_k = \begin{cases} \frac{A}{2\pi k} j(1 - e^{-j\pi k}) = \frac{A}{\pi k} j, & k \text{ udda} \\ \frac{A}{2\pi k} j(1 - e^{-j\pi k}) = 0, & k \text{ jämn} \end{cases}$$

$$B_k = \begin{cases} \frac{-2A}{\pi k}, & k \text{ udda} \wedge k \neq 0 \\ 0, & k = 0 \end{cases}$$

$$A_k = \begin{cases} 0, & k \neq 0 \\ \frac{A}{2}, & k = 0 \end{cases}$$

A.2 Sågtandsvåg

$$x(t) = \frac{A}{T_0}t, \quad 0 < t < T_0$$

$$C_k = \frac{1}{T_0} \int_0^{T_0} \frac{A}{T_0} t e^{-j\omega_0 k t} dt = \frac{A}{T_0^2} \int_0^{T_0} t e^{-j\omega_0 k t} dt$$

$$C_0 = \frac{A}{T_0^2} \int_0^{T_0} t dt = \frac{AT_0^2}{2T_0^2} = \frac{AT_0}{2}$$

$$C_k = \{k \neq 0\} \rightarrow \frac{A}{T_0^2} \int_0^{T_0} t e^{-j\omega_0 k t} dt = \frac{A}{T_0^2} \frac{1 - e^{-j\omega_0 k T_0} (j\omega_0 T_0 k + 1)}{(j\omega_0 k)^2}$$

$$C_k = \frac{-A}{4\pi^2 k^2} \left(1 - e^{-j2\pi k} (j2\pi k + 1) \right) = \frac{Aj2\pi k}{4\pi^2 k^2} = \frac{A}{2\pi k} j$$

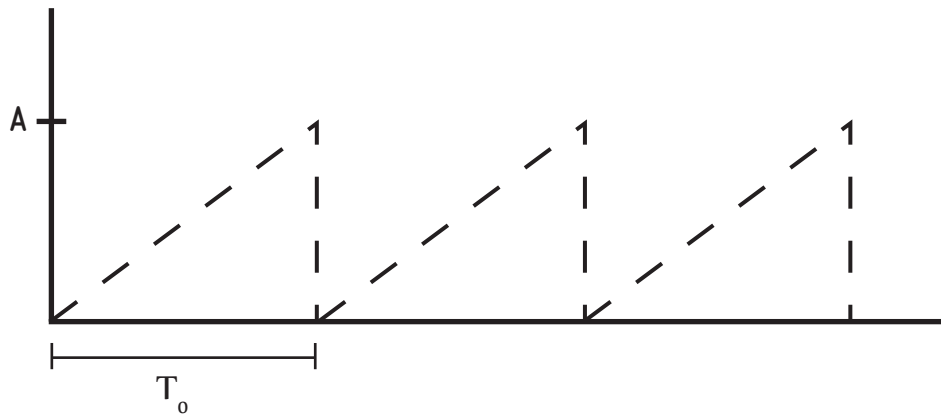
Sammanfattning

$$C_k = \begin{cases} \frac{A}{2\pi k} j, & k \neq 0 \\ 0, & k = 0 \end{cases}$$

$$2C_k = A_k - jB_k \rightarrow$$

$$A_k = \begin{cases} A_k = 0, & k \neq 0 \\ A_k = \frac{AT_0}{2}, & k = 0 \end{cases}$$

$$B_k = \begin{cases} \frac{-A}{\pi k}, & k \neq 0 \\ 0, & k = 0 \end{cases}$$



A.3 Invers Sågtandsvåg

Definition av invers sågtandsvåg:

$$x(t) = A - \frac{A}{T_0}t$$

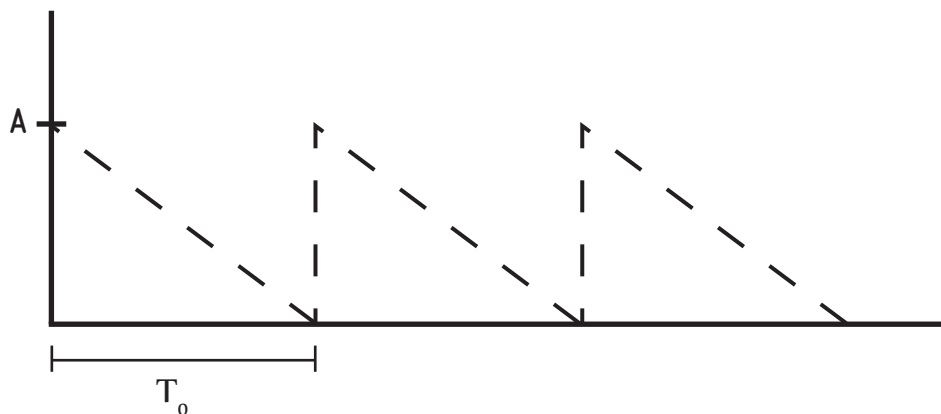
Beräkning av C_k

$$\begin{aligned} C_k &= \frac{1}{T_0} \int_0^{T_0} \left(A - \frac{A}{T_0}t \right) e^{-j\omega_0 kt} dt = \frac{A}{T_0} \int_0^{T_0} e^{-j\omega_0 kt} dt - \frac{A}{T_0^2} \int_0^{T_0} t e^{-j\omega_0 kt} dt = \\ &= 0 - \frac{A}{T_0^2} \int_0^{T_0} t e^{-j\omega_0 kt} dt \\ C_0 &= \frac{A}{T_0} T_0 - \frac{A}{2} = \frac{A}{2} \end{aligned}$$

Vi finner att $C_k = 0 - C_{k2}$ där C_{k2} är de komplexa fourier koefficienterna från del A1.2, dvs $C_{k2} = \frac{-A}{2\pi k}$.

Återanvänder vi beräkningar från tidigare sida så får vi:

$$A_k = \begin{cases} 0, & k \neq 0 \\ \frac{A}{2}, & k = 0 \end{cases}$$
$$B_k = \begin{cases} \frac{A}{\pi k}, & k \neq 0 \\ 0, & k = 0 \end{cases}$$



A.4 Triangelväg

$$x(t) = \begin{cases} \frac{2A}{T_0}t, & 0 < t < \frac{T_0}{2} \\ 2A - \frac{2A}{T_0}t, & \frac{T_0}{2} < t < T_0 \end{cases}$$

$$\begin{aligned} C_k &= \frac{1}{T_0} \int_0^{T_0} x(t) e^{-j\omega_0 kt} \delta t = \frac{1}{T_0} \int_0^{\frac{T_0}{2}} \frac{2A}{T_0} t e^{-j\omega_0 kt} \delta t + \frac{1}{T_0} \int_{\frac{T_0}{2}}^{T_0} 2A \left(1 - \frac{t}{T_0}\right) e^{-j\omega_0 kt} \delta t = \\ &= \frac{2A}{T_0} \left(\int_0^{\frac{T_0}{2}} \frac{t}{T_0} e^{-j\omega_0 kt} \delta t + \int_{\frac{T_0}{2}}^{T_0} e^{-j\omega_0 kt} \delta t - \int_{\frac{T_0}{2}}^{T_0} \frac{t}{T_0} e^{-j\omega_0 kt} \delta t \right) = \end{aligned}$$

I följande beräkningar låt $\{c = -j\omega_0 k\}$

$$\begin{aligned} &= \frac{2A}{T_0} \left(\frac{1 - \frac{1}{2} e^{-\frac{cT_0}{2}} (cT_0 + 2)}{T_0 c^2} - \frac{e^{-cT_0} \left(e^{\frac{cT_0}{2}} (cT_0 + 2) - 2(cT_0 + 1) \right)}{2c^2 T_0} + \frac{e^{-cT_0} \left(e^{\frac{cT_0}{2}} - 1 \right)}{c} \right) = \\ &= \frac{2A}{T_0} \left(\frac{2 - e^{-\frac{cT_0}{2}} (cT_0 + 2) - e^{-\frac{cT_0}{2}} (cT_0 + 2) + 2e^{-cT_0} (cT_0 + 1)}{2c^2 T_0} + \frac{e^{-\frac{cT_0}{2}} - e^{-cT_0}}{c} \right) = \\ &= \frac{2A}{T_0} \left(\frac{1 - e^{-\frac{cT_0}{2}} (cT_0 + 2) + e^{-cT_0} (cT_0 + 1)}{c^2 T_0} + \frac{cT_0 e^{-\frac{cT_0}{2}} - cT_0 e^{-cT_0}}{c^2 T_0} \right) = \\ &= \frac{2A}{T_0} \left(\frac{1 + e^{-\frac{cT_0}{2}} + e^{-cT_0} (cT_0 - cT_0 + 1)}{c^2 T_0} \right) = \frac{2A}{(cT_0)^2} \left(1 - 2e^{-\frac{cT_0}{2}} + e^{-cT_0} \right) = \\ &= \{cT_0 = -j2\pi k\} = \frac{-2A}{4\pi^2 k^2} \left(1 - 2e^{-j\pi k} + 1 \right) = \frac{-A}{\pi^2 k^2} \left(1 - e^{-j\pi k} \right) \end{aligned}$$

$$\frac{-A}{\pi^2 k^2} \left(1 - e^{-j\pi k} \right) = \begin{cases} \frac{-2A}{\pi^2 k^2}, & k \text{ udda} \\ 0, & k \text{ jämn} \end{cases}$$

$$C_k = \begin{cases} \frac{-2A}{\pi^2 k^2}, & k \text{ udda} \\ \frac{A}{2}, & k = 0 \\ 0, & \text{annars} \end{cases}$$

$$A_k = \begin{cases} \frac{A}{2}, & k = 0 \\ \frac{-4A}{\pi^2 k^2}, & k \text{ udda} \\ 0, & k \text{ jämn} \end{cases}$$

$$B_k = 0 \quad \forall k$$

