ANDERS ESKILDSEN

KOMPOSITION OG LYDPRODUKTION MED SUPERCOLLIDER

AAU OPEN

AALBORG UNIVERSITY
OPEN PUBLISHING

Anders Eskildsen

Komposition og lydproduktion med SuperCollider

AAU OPEN

AALBORG UNIVERSITY
OPEN PUBLISHING

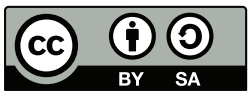
Komposition og lydproduktion med SuperCollider
Af Anders Eskildsen

1. udgave, 2025

Omslag og grafisk layout: Anders Eskildsen

ISBN: 97887-7642-045-1
DOI: 10.54337/aau.kolms2025

Udgivet af:
Aalborg University Open Publishing | www.open.aau.dk



Indhold

Forord

I denne bog introducerer jeg til musik- og lydprogrammering som et redskab til komposition og lydproduktion på grundlæggende niveau. Mit ønske med bogen er, at studerende kan lære at bruge programmering som et kreativt redskab til komposition og lydproduktion, uanset om man på forhånd har kendskab til programmering. Alle kan lære at kode på grundlæggende niveau, og med musik som vores legeplads er det en fornøjelse!

Inden for computermusikken er programmering for nogle musikere og komponister det primære arbejdsredskab. Men andre kan også få glæde af at dykke ned i musik- og lydprogrammering, da det giver adgang til unikke muligheder inden for lyddesign og kompositionsprocesser, som ikke (eller kun meget besværligt) er tilgængelige med mere traditionelle instrumenter og redskaber. Foruden grundlæggende musik- og lydprogrammeringsteknik introducerer bogen til musikalske emner inden for algoritmisk komposition og digital klangdannelse, såsom minimalistisk komposition og beatproduktion samt additiv, subtraktiv og granular syntese.

Den tekniske platform er SuperCollider, som er et fantastisk redskab til musik- og lydprogrammering. Det er tilmed gratis og open source.

Bogen er blevet til som et supplerende materiale til min undervisning i musik- og lydprogrammering på musikuddannelsen ved Aalborg Universitet. Gennem min undervisning har jeg opdaget, at der manglede en introduktion på dansk inden for området. Der findes flere glimrende introduktioner til SuperCollider på engelsk (Fieldsteel, 2024a, 2024b; Magnusson, 2021; Ruviaro, 2015), men det kan være en stor mundfuld at tilegne sig ny viden i computermusikkens felt med ganske megen teknisk jargon, samtidig med at man lærer at læse og skrive et nyt (programmerings)sprog - altsammen på engelsk. Jeg håber derfor, at denne bog, henvendt til begyndere på dansk, kan sænke adgangsbarrieren for studerende og andre nysgerrige sjæle inden for komposition og lydproduktion ved hjælp af programmeringsredskaber.

Når du har studeret denne bog, udført øvelserne, produceret lyd og komponeret på måder du ikke havde forestillet dig fandtes, vil du have et solidt fundament for at kunne arbejde videre med komposition og lydproduktion i SuperCollider.

Heldigvis er der meget mere at komme efter, og en ny verden af muligheder vil gradvist udfolde sig for dig. I slutningen af denne bog udpeger jeg derfor nogle ressourcer til videregående studier.

Praktisk information om bogens kildekode- og lydeksempler

For at gøre det så let som muligt for dig at komme i gang på egen hånd, indeholder denne bog en lang række eksempler, som du selv kan bruge direkte i SuperCollider.

KILDEKODE 🔗
: Alle bokse med kildekode her i bogen kan let kopieres ved at klikke på 🔗-ikonet under kodeboksen, hvorved en webside åbner med den samme kodeboks. Derfra kan koden let kopieres med et klik på en knap øverst til højre.

LYDEKSEMPLER 🎧
: Hvis der også er et 🎧-ikon, findes der på samme webside et lydeksempel, som er produceret med den viste kildekode. I nogle tilfælde vil kildekoden ved produktion af lydeksemplet være eksekveret mere end én gang, så lyden høres flere gange efter hinanden, da dette kan give læseren et bedre indtryk af eksemplet.

Om forfatteren

Anders Eskildsen er adjunkt i musik på Aalborg Universitet, hvor han underviser i musik- og lydprogrammering, grundlæggende indspilnings- og lydstudieteknik, udvikling af digitale musikinstrumenter og interaktive lydinstallationer, kreative praksisser som live coding og Soundpainting, grundlæggende digital musikteknologi, musikkultur og kvalitative metoder, musik og innovation, musikhistorie med fokus på jazz, eksperimentalmusik og meget andet. Denne bog er et af resultaterne af hans undervisning, hvor han har identificeret et behov for dansksproget undervisningsmateriale vedrørende brug af redskaber som SuperCollider.

Eskildsen forsker i redskaber til kreativ musikpraksis, herunder udvikling af gestikbaserede interfaces til software inden for musikproduktion og -performance, hvor SuperCollider spiller en central rolle (Eskildsen & Walther-Hansen, 2020; Walther-Hansen & Eskildsen, 2024). Han arbejder også med kreativ, musikalsk samskabelse og improvisation ved hjælp af tegnsproget Soundpainting (Eskildsen, 2024), som han er certificeret udøver af. Han har skabt interaktive lydinstallationer (Eskildsen & Horvath, 2022) og udvikler open source software, heriblandt SuperCollider-redskaber til aleatorisk komposition (Eskildsen, 2022).

Eskildsens egen musik er ofte skabt med SuperCollider og bevæger sig fra ambiente, generative klange til snigende grooves skabt med granular syntese. Du kan finde den via anderseskildsen.eu/music.

KAPITEL 0

Indledning

Denne bog introducerer til programmering som et kreativt redskab for musikstuderende på de videregående uddannelser. Med afsæt i platformen SuperCollider introducerer bogen på grundlæggende niveau til unikke tilgange til komposition og lydproduktion fra computermusikkens verden. Bogen er henvendt til studerende eller særligt interesserede som ikke nødvendigvis har erfaring med programmering, men som har kendskab til musikteori og -teknologi på grundlæggende niveau.

Dette indledende kapitel introducerer programmering som en musikalsk aktivitet i et historisk perspektiv og forklarer hvordan redskabet SuperCollider passer ind i det musikteknologiske landskab. I bogen arbejdes der på to niveauer - det overordnede kompositionsniveau, hvor vi bruger algoritmer til at generere musikalske forløb, og det mere detaljerede niveau, hvor vi arbejder med at producere lyd med lydsignalernes byggeklodser i form af oscillatorer, samples, envelopes, filtre mm. Det særlige ved SuperCollider er, at man kan kombinere disse to abstraktionsniveauer, hvilket gør redskabet utroligt fleksibelt og righoldigt.

0.1 Programmering og computermusik

Matematikeren og grevinden Ada Lovelace er af computerhistorikere posthumt blevet anerkendt som verdens første programmør, fordi hun i 1842 beskrev det, der sidenhen er blevet betegnet som den første computeralgoritme (Gregersen, 2015). Lovelace var meget optaget af den samtidige forsker Charles Babbages arbejde med hulkort-maskiner, som var forgængere til nutidens digitale computere. Babbage arbejdede blandt andet med at designe en „Analytical engine“, der groft sagt skulle fungere som det vi i dag ville kalde en regnemaskine. Men Lovelace så et meget større potentiale i Babbages hulkortmaskine end den oprindelig var tiltænkt, nemlig at den med abstrakte operationer ville kunne gøre meget andet end foretage udregninger med tal - eksempelvis at komponere musik:

> It might act upon other things besides *number*, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptations to the action of the operating notation and mechanism of the engine. Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent. (Lovelace, 2015, p. 162)

Selvom der skulle gå mere end et århundrede før den computergenererede musik blev udbredt, har Lovelace så afgjort fået ret i sin forudsigelse. I bred forstand kan man sige, at alle de digitale redskaber hvormed vi i dag producerer, distribuerer og forbruger musik, er baseret på computerprogrammering - om end vi interagerer med de fleste af disse redskaber gennem grafiske brugerflader eller fysiske interfaces. I mere konkret forstand er programmering for nogle musikere og komponister inden for computermusikken og den elektroniske musik i dag det primære redskab til at skabe musik, da de nuværende redskaber giver en fantastisk frihed og fleksibilitet, hvis man er villig til at sætte sig ind i programmeringsteknikkerne.

0.1.1 Fra MUSIC-N til SuperCollider

Computermusikkens redskaber har gennemgået en omfattende udvikling siden de tidligste eksperimenter med computerbaseret komposition i midten af det 20. århundrede. Fokuserer vi specifikt på musik- og lydprogrammering, er der også sket ganske meget siden de første forskere i 1950'erne eksperimenterede med at få dyre mainframe-computere til at komponere musik. I disse tidlige eksperimenter

skabte computeren en slags partitur, men dannede ikke nogen egentlige klange. I 1957 opfandt Max Mathews ved AT&T Bell Laboratories programmeringssproget *MUSIC*, der sammen med efterfølgerne *MUSIC II*, *MUSIC III*, *MUSIC V* m.fl. kendes under betegnelsen „*MUSIC-N* languages“ (Wang, 2017, p. 61).

Med *MUSIC-N* og opfindelsen af DAC-enheder, som konverterer fra digitale til et analoge signaler, blev vejen banet for, at computere både kunne generere partiturer og tilmed realisere de lyde, som partituret specificerede. Fra *MUSIC III* var et centralt designprincip på plads, som også efterfølgende musikprogrammeringssprog baserede sig på, nemlig kategoriseringen af tonegeneratorer, filtre, envelopegeneratorer mm. som såkaldte *Unit Generators* (i daglige tale *UGens*) (Wang, 2017, p. 61). Man sammensætter disse UGens til en slags synthesizer, så eksempelvis en envelopegenerator-UGen styrer amplituden/lydstyrken for en sinustone, der er genereret med en tonegenerator-UGen. I *MUSIC-N*-lingo kalder sådan en sammenkobling for et „instrument“ og en samling af sådanne instrumenter for et „orkester“. Dette orkester kan så realisere et „partitur“, analogt til almindelig praksis med musikalsk fremførelse baseret på musiknotation (Wang, 2017, p. 61). Dette skete dog ikke i realtid, da computeren møjsommeligt skulle tygge sig igennem et partitur og derved skabe en lydfil. Ifølge Mathews kunne den computer, han arbejdede med i slutningen af 1950'erne, bruge flere minutter på at generere blot ét sekunds lyd, og det endda med ganske begrænsede klanglige muligheder („Artists' Statements I“, 2017, p. 84).

I forlængelse af *MUSIC-N* blev der skabt mere moderne og avancerede programmeringssprog og -platforme til musik og lyd, herunder først og fremmest *Csound* fra midten af 1980'erne, der som open source-projekt fortsat er i anvendelse i dag. Csound kunne ikke oprindeligt generere lyd i realtid (hvilket dog er blevet tilføjet sidenhen) - der skulle i stedet et andet projekt til for at ændre dette: I 1980'erne skabte Miller S. Puckette ved IRCAM det i dag meget udbredte værktøj *Max*, der var opkaldt efter førnævnte Max Mathews. *Max* var i første omgang designet til at være et MIDI-system, der kunne styre ekstern signalbehandlingshardware i IRCAM's eksperimentelle lydstudier og blev udgivet som kommerciel software. Det innovative ved *Max* var, at der er tale om en grafisk brugerflade, hvor brugeren opretter forskellige små bokse og forbinder dem med virtuelle kabler for at definere signalflowet, lidt ligesom en modulær synthesizer.

Senere udviklede Puckette et nyt system kaldet *Pure Data* (i daglig tale blot *Pd*), der i sin brugerflade var en gentænkning af *Max*, men derudover også inkorporerede en lyd-engine, som man kunne styre fra brugerfladen (M. S. Puckette, 1996). *Pd* er open source, og lyddelen blev derfor også inkorporeret i *Max*, der i den sammenhæng fik navnet *Max/MSP*. *Pd*, som er gratis og open source, og den kommercielle slægtning *Max/MSP* er i dag meget populære og udbredte redskaber inden for computermusik. Begge platforme er gode kandidater til at blive anvendt af musikstuderende i dag, fordi de med den grafiske brugerflade er lette at komme i gang med, hvis man ikke

har arbejdet med programmering før. Når de ikke er valgt som platform til denne bog, er det fordi de mere omfattende projekter, man går i gang med umiddelbart efter begynderstadiet, hurtigt bliver yderst komplekse og vanskelige at håndtere i den visuelle platform. Dertil er en tekstbaseret tilgang bedre egnet, om end dette selvfølgelig også afhænger af personlige præferencer og færdigheder.

SuperCollider, platformen i denne bog, blev oprindeligt skabt af James McCartney og udgivet som kommerciel software i 1996 (McCartney, 2002). I 2002 frigav McCartney imidlertid SuperCollider under en open source-licens, og projektet udvikles og anvendes i dag af musikere, komponister, forskere, lyddesignere m.fl. i et omfattende open source-fællesskab. SuperCollider er i den aktuelle version 3 et af de mest fremtrædende redskaber til algoritmisk komposition og programmatisk lydproduktion vi har til rådighed i dag.

Et af de særlige træk ved SuperCollider er, at det består af tre dele: Et tekstbaseret programmeringssprog, en state-of-the-art lydserver samt en dedikeret teksteditor med indbygget dokumentation og hjælp. På grund af denne tredelte opbygning kan SuperColliders lydserver anvendes som komponent i andre systemer, hvilket er blevet udnyttet i mange specialdesignede programmeringsplatforme som *Sonic Pi*, *TidalCycles*, *Overtone* m.fl. Et andet nyere skud på stammen, der dog ikke har lige så stor anvendelse som Pd, Max/MSP og SuperCollider, er programmeringssproget *ChucK*, som er ganske interessant men knap så oplagt som et redskab til begyndere.

0.2 Om denne bog

Med denne bog ønsker jeg at bidrage til, at flere musikere får adgang til de fantastiske muligheder, musik- og lydprogrammering giver inden for komposition og lydproduktion. I dag kan enhver med en laptop, en DAW og evt. et abonnement på loops og samples relativt let skabe musik. Sammenlignet med de lydstudieressourcer, der var nødvendige for blot få årtier siden, er der sket en omfattende udvidelse af mulighederne for den enkelte. Men når denne almengørelse af musikteknologien giver flere mennesker mulighed for fremstille musik, bliver det vanskeligere for skabende kunstnere at skille sig ud fra mængden. At investere tid og engagement i programmering som redskab giver musikere og komponister mulighed for at skabe unikke musikalske og lydlige træk, som kun vanskeligt eller slet ikke kan frembringes med mainstream, kommerciel musiksoftware. Bogen åbner døren til disse muligheder ved at introducere til teknik, terminologi og kreative tilgange på forståelig og praktisk anvendelig vis.

0.2.1 To fokusområder

Programmering kan i dag foregå på mange forskellige abstraktionsniveauer, da der findes ganske mange programmeringssprog, som er designet med forskellige formål. Den såkaldte maskinkode udgør det laveste niveau og består af sekvenser af binære tal, som indikerer basale operationer, der kan udføres af en computerprocessor. For mennesker er maskinsprog yderst vanskelig at fremstille og læse, fordi det er så fjernt fra menneskers ordinære sprog. Derfor har man opfundet en række programmeringssprog, der opererer på et højere abstraktionsniveau. I det børnevenlige programmeringssprog Scratch kan man eksempelvis lave små spil ved at sammenkoble forskellige klodser med musen.

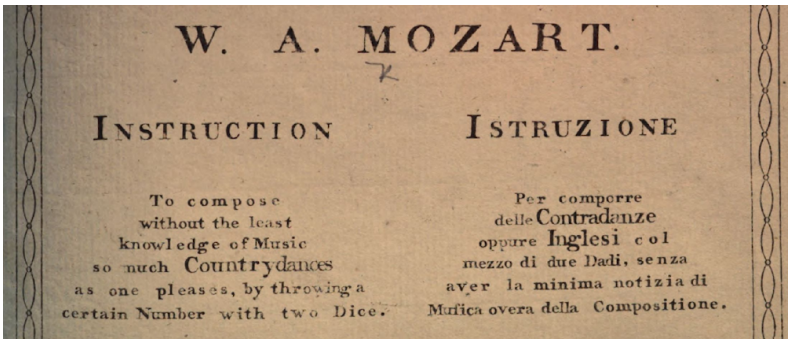
SuperCollider og andre musikfokuserede programmeringssprog fungerer på et relativt højt abstraktionsniveau. Det betyder i denne sammenhæng, at mange tekniske detaljer er abstraheret væk, således at programmøren kan fokusere på de musikalske/lydlige aspekter. Det vi typisk arbejder med her, svarer i nogen udstrækning til de ovenfor nævnte metaforer „partitur“ og „orkester“. Mere specifikt skal vi i bogen arbejde med det, man kalder *algoritmisk komposition* (svarende til „partituret“), dvs. vi skal fremstille algoritmer og mønstre, der kan generere musikalske forløb. Derudover skal vi også arbejde med *klangdannelse*, dvs. vi skal fremstille små programmer, der gennem forskellige teknikker til lydsyntese designer de enkelte klange og lyde, vores kompositioner består af.

0.2.1.1 Algoritmisk komposition

Umiddelbart kunne man godt forestille sig, at algoritmisk komposition er noget, der primært hører til computermusikken. Men en algoritme kan defineres som en foreskrevet proces, der ved at udføre en række konkrete trin har til formål at løse et bestemt problem. Og i den betydning har algoritmen en længere historie som et musikalsk-kompositorisk redskab.

> Within the field of algorithmic composition the algorithm constitutes an abstract model which defines and controls some or all structural aspects of the music. This model can also serve as a generator that is capable of producing the piece as a possible variant within a field of possibilities. The latter approach can be implemented as a computer program, but the underlying idea is much older. (Essl, 2017, p. 105)

Essl sporer algoritmisk kompositionsteknik helt tilbage til den tidlige polyfone musik, som er noget af det ældste nedskrevne musik, der findes i vesten (Essl, 2017). Nyere eksempler findes i J. S. Bachs regelbaserede kompositionsteknikker og såkaldte gådekanon-kompositioner samt i W. A. Mozarts berømte *Musikalische Würfelspiele* (1793), hvor terningekast afgør rækkefølgen af det musikalske materiale og dermed giver mulighed for at skabe forskellige variationer over den underliggende metrik og harmonik (Essl, 2017).



Figur 0.1: Udsnit af forsiden til W. A. Mozarts Musikalische Würfelspiele. Tilhører det offentlige domæne. Kilde: IMSLP.

I det 20. århundredes vestlige kompositionsmusik indgår algoritmisk komposition i flere af hovedstrømningerne. Med først tolvtonemusikken og senere hen serialismen og aleatorikken gør mange komponister forskellige algoritmer til det centrale redskab i deres kompositionsproces. I modsætning til de ofte meget deterministiske algoritmer introducerede komponisten Iannis Xenakis ideen om *stokatisk musik*, hvor tilfældighedsoperationer baseret på matematisk bestemte sandsynligheder

bliver en central del af algoritmerne. I forlængelse af dette samt minimalismen, den amerikanske eksperimentalmusik og den tidlige ambiente musik blev ideen om *generativ komposition* stadfæstet som et af den moderne kompositionsmusiks modus operandi. Hertil kom, at de ovenfor omtalte udviklinger inden for computerteknologi og redskaber til computermusik gradvist blev en del af kompositionsmusikken, og computermusikken vandt samtidig frem som et selvstændigt, men beslægtet felt, hvori algoritmisk komposition spiller en væsentlig rolle.

SuperCollider er et fantastisk redskab til algoritmisk komposition, blandt andet fordi programmeringssproget indeholder et omfattende bibliotek af såkaldte patterns - generative mønstre, der kan kombineres på mange forskellige måder. I denne bog udforsker vi på grundlæggende niveau hvordan vi med disse redskaber kan styre både overordnede formmæssige forhold i musikken, men også hvordan vi kan skabe variationer og musikalsk indhold på mere detaljerede niveauer. Én praktisk tilgang er at bruge SuperCollider som en generativ maskine, der spiller på andre synthesizere eller styrer instrumentplugins i en DAW, fx via MIDI. Men der er mange fordele ved at bruge SuperColliders egen lydserver i stedet, hvor vi selv kan designe klangdannelsen. Man kan på denne måde bruge algoritmisk komposition til ikke blot at styre traditionelle kompositionsparametre som tonehøjde, rytmik og dynamik, men også til at variere og komponere med andre lydlige parametre.

0.2.1.2 Klangdannelse og lyddesign

Klangdannelse sker naturligt, når fysiske objekter bliver sat i vibration og derfor udsender lydbølger. Objektets akustiske egenskaber medvirker til afgøre klangen, hvilket ligger til grund for akustiske instrumenter. Den elektroniske klangdannelse har, ligesom computermusik og algoritmisk komposition, en interessant historie, der kun kort kan antydes her. Den tidlige elektroniske klangdannelse var udelukkende analog, hvor elektriske komponenter sammensat i kredsløb genererede strømsignaler, der blev omdannet til lyd via højttalere. Selv i dag er den analoge klangdannelse lidt af et plusord, da den ofte forbindes med en særligt „varm“ kvalitet, og der produceres derfor også i dag analoge synthesizere. Men digital teknologi har en række fordele, heriblandt fx at instrumentets tonehøjde ikke varierer, når temperaturen stiger og falder (hvilket er tilfældet med mange analoge synthesizere, hvor den elektriske modstand i udstyret varierer med temperaturen). En tilstrækkeligt kraftig digital computer kan uden problemer fremstille lyden af tusindvis af oscillatorer (tonegeneratorer) i realtid, hvor en analog synthesizer, der skulle gøre det samme, ville kræve tusindvis af elektriske komponenter, hvilket er urealistisk i et hardwaresetup.

Størstedelen af redskaberne til elektroniske klangdannelse er derfor i dag digitale, hvad enten de findes i plugin-form eller som hardwareenheder. Men selv digitale lydredskaber søger ofte at emulere lyden af analogt grej ved at tilføje forskellige

former for støj og lydlige artefakter som klanglig farvning af et lydsignal. „Virtual analog“ er blevet en væsentlig salgsrubrik. Samtidig lever den såkaldt modulære synthesizer i bedste velgående, hvor der efterhånden findes tonsvis af lyd- og kontrolmoduler i det populære „Eurorack“-format. Også den modulære synthesizer er blevet emuleret digitalt, blandt andet af programmet *VCV Rack* (og open source-klonen *Cardinal*). Med disse modulære redskaber består kompositions- og lyddesignarbejdet i at koble forskellige moduler sammen med patchkabler og justere på modulernes indstillinger ved hjælp af fadere og drejeknapper.

I SuperCollider og lignende platforme kan vi på samme måde arbejde med enheder, der svarer til elektriske komponenter som tonegeneratorer og hardware-moduler som envelopegeneratorer eller granular synthesis-moduler. Dette gør vi ved hjælp af SuperColliders lydserver og de ovenfor omtalte *UGens*. De fleste UGens har forskellige *input* og *output* samt en række *indstillinger*. Som eksempel kan vi tage sinustonegeneratoren **`SinOsc`**, der er en af de UGens, vi møder senere i bogen. Den har som output et lydsignal i form af en sinusbølge. Den har også et frekvens-input, hvormed vi kan styre hvor mange gange pr. sekund vi gennemløber bølgeformen (og dermed styre hvilken tonehøjde, der spilles). Dette input kan vi indstille til en fast værdi, fx 440 Hz, kammertonen, eller vi kan styre det med en anden UGen, så tonehøjden varierer over tid. I det tilfælde siger vi, at vi *modulerer* oscillatorens frekvens. UGens som **`SinOsc`** har også nogle *indstillinger*, eksempelvis kan vi specificere initialfasen, dvs. hvor i bølgeformen, oscillatoren skal begynde, når den først bliver sat i gang.

Der findes ganske mange UGens, hvormed vi kan skabe og transformere klange, arbejde med samples og så videre. Klassiske klangdannelsesteknikker kan fint implementeres i SuperCollider, hvilket denne bog viser på flere forskellige områder. Men der er ingen faste regler, når det gælder kreative praksisser som komposition og lyddesign, og vi kan derfor også udvide, „hacke“ og eksperimentere inden for klangdannelsen. Ved at gøre det opnå man to vigtige ting:

- For det første kan vi skabe unikke klanguniverser, der ikke lyder som alle de almindelige plugins. Dette betyder ikke, at SuperCollider eller andre lignende platforme skal erstatte de sædvanlige redskaber som DAWs og plugins. Men det er et unikt redskab at have adgang til, når man vil skille sig ud både i lydligt resultat og i kompositionsmetode.
- For det andet, og mindst lige så vigtigt, kan vi ved at eksperimentere med klangdannelse i SuperCollider lære en masse om lyd, klang og digital lydteknologi. Det sidste er vigtig viden i en tid, hvor digital musikproduktion er blevet allemandseje og samtidig er kommercialiseret i en meget udstrakt grad. Ønsker man at navigere i dette komplekse, teknologiske landskab, er det med andre ord meget væsentligt at kunne gennemskue på et grundlæggende niveau, hvordan teknologierne fungerer og hvad de kan anvendes til.

I de følgende kapitler arbejder vi med disse to abstraktionsniveauer først hver for sig, og derefter i kombination. Men først en advarsel: Når man har fået blod på tanden og fornemmer mulighederne ved kombination af disse to stærke redskaber, kan det være svært at lægge fra sig igen. God fornøjelse med komposition og lydproduktion i SuperCollider!

0.2.2 Sådan bruger du denne bog

Bogen er skrevet med henblik på studerende på de videregående uddannelser men kan bruges af alle, der har en forståelse af musikteori og musikteknologi på grundlæggende niveau. Det forudsættes eksempelvis, at man har et elementært kendskab til musikalske skalaer, intervaller og rytmelære samt bølgelære, filtre, oscillatorer og MIDI. For nogle læsere vil det være relevant at opsøge supplerende materiale, da denne bog fx ikke går i dybden med de grundlæggende begreber inden for musikteori.

Hvis man er begynder inden for programmering, er det en god ide at starte med kapitel 1, som introducerer til de grundlæggende begreber og teknikker, man skal kende, for at kunne programmere på grundlæggende niveau i SuperCollider. Man skal, som det gode gamle ordsprog lyder, kravle, før man kan gå. Al erfaring viser nemlig, at det er sjovest at tage fat på de mere avancerede emner, når man har godt styr på de grundlæggende færdigheder. Er man mere erfaren inden for både programmering og computermusik, kan man i stedet bruge bogen mere som et opslagsværk. Bemærk dog, at bogen er bygget op med en progression, således at bogens anden halvdel forudsætter, at man kender til de grundlæggende begreber og teknikker, der er introduceret i den første halvdel.

Uanset hvilken baggrund man har, vil jeg anbefale, at man *undervejs i læsningen selv indtaster og afprøver eksemplerne med kildekode*. Betydningen af denne aktive tilgang kan ikke overdrives! Det er der nemlig flere gode grunde til:

- Man forstår og husker bedre hvordan kildekoden fungerer, når man selv har indtastet den.
- Man lærer et nyt sprog bedst, når man bruger det i praksis (McManus, 2024).
- Og sidst, men bestemt ikke mindst: Det er sjovt at lege med eksemplerne på egen hånd!

Prøv selv at installere SuperCollider, åbne programmet og indsætte nedentstående kildekode i tekst-editoren. Placer derefter cursoren på én af de midterste linjer og tast Ctrl-Enter (på PC) eller Cmd-Enter (på Mac) for at høre lyden. Koden starter SuperColliders lydserver og producerer derefter to sinustoner; en tone i hver stereo-kanal, som begge springer tilfældigt op og ned mellem 220 Hz og 880 Hz, 10 gange i sekundet.

```
(
s.waitForBoot({
    {
        SinOsc.ar(
            LFNoise0.kr(10.dup).exprange(220, 880)
        )
    }.play;
})
)
```

Kodeboks 0.1: En LFO styrer en sinustone-oscillator med tilfældigt valgte frekvenser

Du kan stoppe lyden igen med Ctrl-Punktum (på Windows/Linux) eller Cmd-Punktum (på Mac).

0.2.2.1 Bogens lydeksempler

Mange af bogens kodeblokke har et tilhørende lydeksempel, som kan afspilles i webudgaven. Lydeksemplerne er optaget direkte fra SuperCollider og kun efterbehandlet med en let tilpasning af lydstyrke. De vil derfor i praksis klinge ligesom det, man hører, hvis man selv eksekverer kildekoden. Dog vil der i nogle tilfælde være tale om stokastiske processer, som skaber et unikt resultat, hver gang man eksekverer kildekoden. I lydlige og musikalske egenskaber vil det, man selv kan frembringe ved at eksekvere kildekoden, i disse tilfælde dog være nært beslægtet med lydeksemplet.

I nogle få tilfælde er lydoptagelserne her i bogen behandlet med eksterne effekter som instrument-plugins, rumklang og lignende. I de tilfælde er det beskrevet tydeligt i teksten, hvilke eksterne redskaber der er anvendt til at realisere eksemplet.

0.2.2.2 Tre afsnitstyper

Bogens kapitler indeholder afsnit af tre forskellige typer, som du kan bruge efter behov:

ARTIKLER
: Introducerer til hvordan man kan arbejde med generativ komposition og digital klangdannelse i SuperCollider.

CHEAT SHEETS
: Overskuelige, korte oversigter over centrale redskaber og teknikker. Oplagt at bruge som opslagsværk.

ØVELSER
: Praktiske opgaver, som giver mulighed for at øve sig i de forskellige teknikker.

0.2.2.3 PDF-udgave og webudgave

Den primære udgave af bogen er udgivet i PDF-form af Aalborg University Open Publishing og kan downloades hos AAU.

Bogen findes desuden i en webudgave på sc.anderseskildsen.eu, hvor bogens lydeksempler kan afspilles.

Jeg anbefaler, at man læser bogen i PDF-form, da det er det mest overskuelige format. Til gengæld er webudgaven også god at have ved hånden, fordi den indeholder bogens lydeksempler, og fordi man let kan kopiere kildekode fra bogens mange kodeeksempler og indsætte i SuperCollider.

KAPITEL 1

Grundlæggende programmering

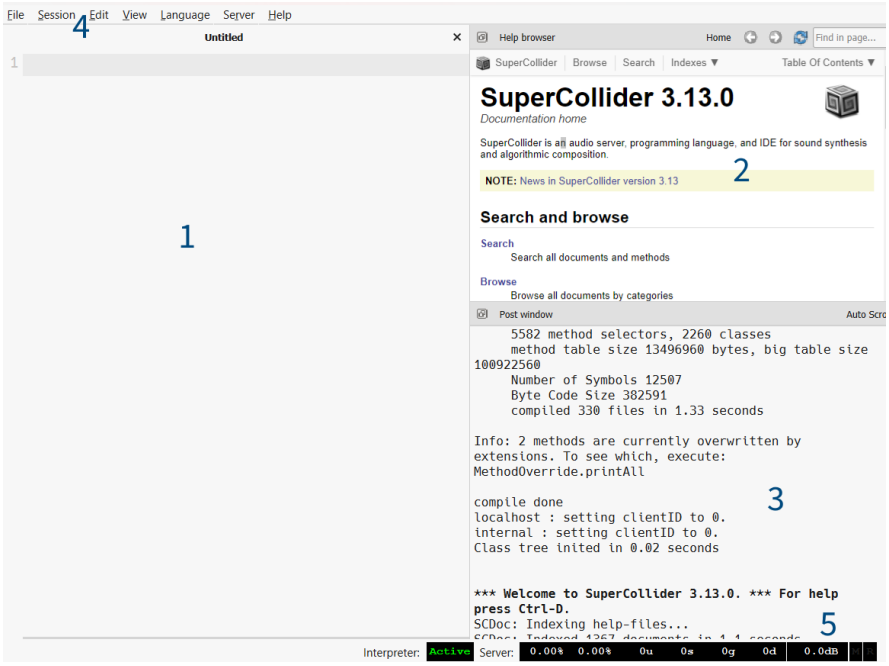
Dette kapitel introducerer til grundlæggende programmering i SuperCollider. Ud over at introducere brugerfladen, kigger vi nærmere på syntaksen i programmeringssproget SuperCollider, så vi kan skrive og eksekvere kildekode. Derudover introduceres nogle grundlæggende begreber og teknikker, som det er vigtigt at have styr på, inden vi går videre. Det drejer sig om brug af såkaldte variabler, methods og argumenter. I slutningen af kapitlet indgår nogle grundlæggende programmeringsøvelser med og uden lyd. Men vores hovedprioritet her i første kapitel er altså at forstå, skrive og eksekvere kildekode, da det er fundamentet for at kunne arbejde med de musikalske emner i de efterfølgende kapitler.

1.1 SuperColliders brugerflade

Første gang man åbner SuperCollider, mødes man af en umiddelbart noget minimalistisk brugerflade. Med mindre man har arbejdet med programmering før, vil det i begyndelsen være lidt uvant, at brugerfladen først og fremmest består af et tekstdokument, hvor man noterer og eksekverer kildekode. Men det ændrer sig hurtigt, når man kommer i gang, og inden længe kommer man til at sætte pris på den enkelhed, brugerfladen også repræsenterer.

Lad os danne os et overblik for at komme i gang. Når man åbner SuperCollider første gang efter en frisk installation, vil brugerfladen typisk indeholde følgende elementer:

1. Den tomme kasse til venstre indeholder dokumenter. Det er her, vi skriver og eksekverer kildekode.
2. Kassen øverst til højre med navnet *Help browser* giver adgang til SuperColliders dokumentation. Det er denne boks, der henvises til, når vi i denne bog taler om at slå noget op i SuperColliders dokumentation.
3. Kassen nederst til højre med navent *Post window* er et vindue, hvor SuperCollider viser nyttige informationer og fejlmeddelelser.
4. Menuen øverst giver adgang til forskellige almindelige funktioner for tekstdokumenter som „Save“, „Open...“, „Cut“, „Paste“ samt en række SuperCollider-specifikke indstillinger og funktioner.
5. Talrækken nederst til højre giver forskellige informationer om SuperColliders tilstand, primært i forhold til lydserveren. Her kan man højreklikke for at få vist en menu med forskellige funktioner.



Figur 1.1: Forskellige dele af SuperColliders brugerflade

Help browser'en og SuperColliders Post window kan flyttes, lukkes eller rykkes til et separat vindue. Men for begyndere kan det være fornuftigt at beholde setup'et som det er indtil videre, så redskaberne er til at finde. Dog kan man nemt genaktivere kasserne via menuen *View*, hvis de er blevet lukket.

1.1.1 Brugerflade, fortolker og lydserver

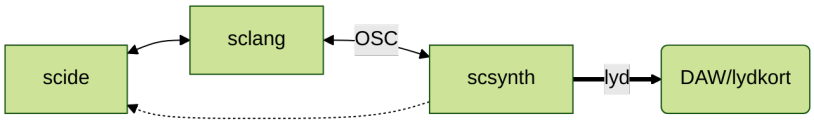
Rent teknisk består SuperCollider faktisk af tre forskellige programmer/processer. Det er ikke nødvendigt at kende alle detaljer om, hvordan de fungerer og samarbejder, men det er nyttigt at vide, at de findes, så man kan følge med, når vi taler om at arbejde med fortolkeren eller lydserveren.

scide, AKA. BRUGERFLADEN
: Den grafiske brugerflade, som er beskrevet ovenfor, kaldes også for SuperColliders IDE (Integrated development environment). Det er dette program, vi interagerer direkte med, når vi arbejder med SuperCollider

sclang, AKA. FORTOLKEREN
: Fortolkeren (på engelsk „the interpreter") kører i baggrunden og er ansvarlig for at fortolke og udføre instrukserne i den SuperCollider-kildekode, vi eksekverer i brugerfladen. Starter automatisk, når vi starter brugerfladen.

scsynth, AKA. LYDSERVEREN
: Lydserveren kører også i baggrunden og er ansvarlig for at fremstille de lyde, vi (gennem fortolkeren) specificerer. Lydserveren starter ikke automatisk sammen med brugerfladen men skal bootes, før vi kan lave lyd i SuperCollider.

Fortolkeren og lydserveren kommunikerer med hinanden ved hjælp af noget, der hedder OSC-meddelelser (Open Sound Control). OSC er et mere præcist og fleksibelt alternativ til MIDI.



Figur 1.2: Dataflow mellem SuperColliders tre komponenter

1.2 Eksekvering af kildekode

Et program består af nedskrevne *instrukser* (på engelsk *statements*), som computeren skal udføre. I SuperCollider kan vi få computeren til at udføre instrukser ved at *eksekvere* udvalgte dele af kildekoden. Man eksekverer kildekode ved at taste Ctrl-Enter på PC eller Cmd-Enter på Mac. Prøv det selv:

- Indtast linjerne herunder i et SuperCollider-dokument.
- Sæt cursoren på en af linjerne og tast Ctrl-Enter (PC) eller Cmd-Enter (Mac).
- Iagttag derefter outputtet i SuperCollider's „Post window" (som ved et nyinstalleret setup vil befinde sig til højre i skærmbilledet).

```
1 5 + 10;
2 Scale.major;
3 rrand(0, 100);
4 "Vekseldominant".postln;
```

Kodeboks 1.1: Eksekvering af kildekode, linje for linje

1.2.1 Flere instrukser ad gangen

Hvis vi gerne vil gøre mere end én ting ad gangen, kan vi adskille vores instrukser ved hjælp af semikolon.

```
1 "Et fantastisk tal:".postln; rrand(0, 100).postln;
```

Kodeboks 1.2: Adskillelse af instrukser ved hjælp af semikolon

Hvis vi udelader semikolon, kan SuperCollider ikke forstå hvor den ene instruks stopper og hvornår den næste starter. Da vi ofte for overskuelighedens skyld skriver én instruks pr. kodelinje, er det en god tommelfingerregel - med nogle få vigtige undtagelser - at afslutte kodelinjer med et semikolon.

Ofte er det en god ide at fordele instrukserne over flere linjer. Hvis vi vil have SuperCollider til at udføre flere linjer med instrukser umiddelbart efter hinanden, kan vi gøre dette ved at afslutte de enkelte linjer med semikolon og omkranse linjerne med parenteser. Dette kaldes en *kodeblok*. Kodeblokken eksekveres ved, at man sætter cursoren på en af linjerne og trykker Ctrl/Cmd-Enter. Det betyder ikke

noget præcis hvilken linje, bare cursoren befinder sig et sted mellem de yderste parenteser.

```
(
"Endnu et fantastisk tal:".postln;
rrand(50, 100).postln;
)
```

Kodeboks 1.3: En kodeblok

I denne bogs kodeeksempler sættes der kun i få tilfælde parenteser omkring kodeblokke. Læseren skal altså selv sætte parenteser på relevante steder, hvis der kopieres kildekode fra bogens eksempler. Heldigvis kan SuperCollider let indsætte matchende parenteser omkring markerede udsnit af kildekoden[1] *Auto insert matching parentheses, brackets, quotes under menuen Edit Preferences Editor Behavior..*

Læg i øvrigt mærke til hvordan begge linjer i kodeblokken ovenfor bliver udført så hurtigt efter hinanden, at det i praksis sker samtidigt, men dog i den noterede rækkefølge.

1.2.2 Fut i lydserveren

For at kunne bruge programmering som et redskab til musikalsk komposition og lyddesign skriver vi ofte kildekode, som genererer lyd. I de tilfælde skal vi først starte SuperColliders lydserver. Den mest enkle måde at starte lydserveren på er at køre følgende linje.

```
s.boot;
```

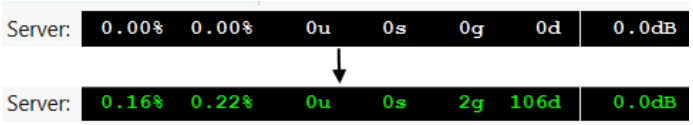
Kodeboks 1.4: Start af SuperColliders lydserver

Alternativt kan man starte lydserveren på andre måder:

- Tastaturgenvejen Ctrl/Cmd-B.
- Gå via menuen „Server“ øverst i vinduet
- Højreklik på tallene i nederste højre hjørne ud for „Server“.

[1] SuperColliders IDE kan indstilles til automatisk at indsætte matchende parenteser ved at sætte flueben ud for

Bemærk, at når lydserveren er bootet, bliver disse tal i nederste højre hjørne grønne. Når serveren bootes ser vi desuden en del tekstbeskeder til højre i SuperColliders post window, herunder hvilket lydkort, lydserveren er forbundet med.



Figur 1.3: Boot af SuperColliders lydserver

Derefter kan vi afspille lyde på serveren. Tast Ctrl/Cmd-Punktum for at slukke lyden.

```
1 { SinOsc.ar(440) * 0.1 }.play;
2 Pbind(\degree, [0, 2, 4]).play;
```

Kodeboks 1.5: Et par simple lyde

Hvis din server ikke booter: Skulle din server mod forventning ikke starte, kan det ofte skyldes indstillingerne i lydkortet eller problemer med lydkortets drivere. Her må du konsultere dokumentationen for det pågældende hardware og styresystem. Du kan også skifte til et andet lydkort eller en anden driver - se hvordan i afsnittet *Audio device selection* i SuperColliders dokumentation, som også omtaler specifikke opmærksomhedspunkter ved de forskellige platforme (Mac, Linux og Windows).

1.2.2.1 Start af lydserveren på Mac med 'sample rate mismatch'

På Mac sker det jævnligt, at styresystemet indstiller input og output til to forskellige samplerates, hvis man bruger computerens indbyggede lydkort. Dette er imidlertid ikke kompatibelt med SuperColliders lydserver, hvilket man kan se, hvis man i post window får fejlmeddelsen „Sample rate mismatch". I den situation er man nødt til at justere på styresystemets indstillinger, og det er heldigvis ganske enkelt: Kør nedenstående for at åbne programmet „Audio MIDI Setup" og indstil input og output til den samme samplerate (fx 44.1 kHz eller 48 kHz).

```
1 // Linjen herunder åbner programmet "Audio MIDI Setup"
2 "open -a 'Audio MIDI Setup'".unixCmd;
```

Kodeboks 1.6: Løsning til 'sample rate mismatch'-fejl på Mac

1.3 Brug af variabler

Vi kan opbevare forskellige former for data i computerens hukommelse ved hjælp af variabler. Man kan forestille sig en skuffe, som man sætter en seddel på med en kort beskrivelse af indholdet. Når man så skal finde indholdet frem igen eller putte noget nyt indhold i, finder man blot skuffen med den rette seddel frem. Variabler fungerer lidt på samme måde: Når man opretter en variabel, gemmer man et objekt under et bestemt navn. Når man så senere skal bruge objektet igen, henviser man blot til variablens navn. Og nu vi er ved variabelnavne: De starter altid med et lille begyndelsesbogstav[2]. Hvis vi prøver at bruge variabelnavnet `~Kaffe`, vil vi derfor få en fejlmeddelelse. Brug i stedet `~kaffe`.

1.3.1 Globale variabler

Der er to typer variabler: *Globale* og *lokale* variabler. Globale variabler kan anvendes (næsten) overalt i et SuperCollider-program. De noteres med tegnet tilde (~), direkte efterfulgt af et passende navn.

```
1 ~trin;
2 ~akkord;
3
4 // Alle enkeltbogstaver fra a til z udgør også globale
  ↪  variabler:
5 a;
6 b;
```

Kodeboks 1.7: Globale variabler

For at bruge variabler skal vi kunne tilgå og definere deres indhold.

- For at tilgå variablen, dvs. finde dens indhold frem og bruge det til noget, bruger vi slet og ret variabelnavnet, fx `~trin`.
- For at definere eller ændre variablens indhold, bruger vi først variabelnavnet efterfulgt af et lighedstegn, og det nye indhold på højre side af lighedstegnet: `~trin = 2`. Dette kaldes også *assignment*, fordi vi i dette tilfælde „assigner“ tallet 2 til variable med navnet `~trin`.

[2] Teknisk set skal variabler begynde med småt begyndelsesbogstav, fordi stort begyndelsesbogstav er forbeholdt de såkaldte klassenavne (se 1.5.2).

```
~trin.postln; // tjek først variablens indhold
~trin = 5;    // gem et tal
~trin.postln; // tjek variablens indhold igen

//Vi kan efterfølgende tilgå og bruge variablens indhold blot ved at bruge dens navn:
~trin - 10;
```

Kodeboks 1.8: Grundlæggende brug af variabler 

En variabel kan efterfølgende let omdefineres ved at foretage en ny assignment.

```
// Vi kan også omdefinere indholdet med endnu en assignment:
~trin = 50;
~trin.postln;

// Variablens nuværende indhold kan anvendes, når man regner en ny værdi ud og gemmer under samme variabelnavn:
~trin = ~trin * 10 + 7;
~trin.postln;
```

Kodeboks 1.9: At omdefinere indholdet af en variabel 

Der findes en mindre teknisk forskel på variabler som `~trin` og `t`, men på begynderniveau er det ikke nødvendigt at skelne mellem de to[3].

1.3.2 Lokale variabler

Vi bruger blandt andet lokale variabler til at gemme data som akkorder og rytmer, men også til at definere signalflowet (se 4.1.3), når vi designer lyde.

Lokale variabler defineres med nøgleordet **`var`** i stedet for ~. De kan udelukkende tilgås inden for en afgrænset kodeblok bestående af almindelige parenteser eller tuborg-parenteser, og de skal oprettes i begyndelsen af den kodeblok, de tilhører.

[3] Rent teknisk udgør variabler som `kaffe` og `the` såkaldte *environment variables*, hvor variabler som `a`, `b` og `c` er globale variabler i mere klassisk forstand. Med environment variables kan man skifte mellem environments og dermed adskille det man i mere avanceret programmering kalder for *namespaces*. Som begynder skal man ikke bekymre sig om dette, da vi på grundlæggende niveau udelukkende arbejder inden for ét environment. Vi kan dermed for enkelhedens skyld betragte environment variables som globale variabler.

```
(
var akkord = [0, 1, 4];
akkord.postln;
// Inden for kodeblokken: viser variablens indhold
)

akkord.postln;
// Uden for kodeblokken: giver en fejlmeddelelse
```

Kodeboks 1.10: En lokal variabel

1.4 Funktioner for den dovne programmør

Vi er nu nået til et emne, som er en gave til den dovne programmør, der ikke gider at skrive det samme sæt af instrukser mere end én gang. Heldigvis er det faktisk ofte en god ide at være doven på denne måde, fordi det er effektivt at indkapsle og genbruge sæt af instrukser, da det gør os i stand til at gøre mange ting med meget lidt kildekode. I programmering er *funktioner* et grundlæggende koncept, som er væsentligt at forstå. I SuperCollider og de fleste andre programmeringssprog er en funktion en defineret række af instrukser, der fungerer lidt ligesom de kodeblokke, vi kiggede på ovenfor (se 1.2.1). De er nyttige, hvis vi gerne vil gøre det samme mange gange, da vi i stedet for at skrive den samme kodeblok mange gange blot kan „kalde“ funktionen.

1.4.1 Hjemmelavede funktioner

Funktioner fungerer som sagt næsten ligesom de kodeblokke vi har set, hvor flere instrukser er omkranset af parenteser. Med funktioner bruger vi blot tuborgklammer ({}) i stedet for parenteser. Kører man linjen `{};` i SuperCollider, vil post window således vise `a Function`. For at gøre brug af funktioner, kan vi gøre følgende:

- Vi skriver mellem tuborgklammerne de instrukser, funktionen skal udføre, adskilt med semikolon.
- Dernæst gemmer vi funktionen under en global variabel (se 1.3.1), så vi kan referere til den igen andre steder i kildekoden.
- Hvis vi vil *kalde* funktionen, dvs. eksekvere de instrukser, som er indeholdt i funktionen, kan vi koble noget ekstra kode på variabelnavnet, nemlig `.value`. Dette er en *method*, hvilket bliver forklaret mere indgående nedenfor (se 1.5).

```
1 // Vi gemmer funktionen under en variabel
2 ~minFunktion = { "Hej fra funktionen".postln; };
3
4 // Vi kan henvise til funktionen
5 ~minFunktion;
6
7 // Og vi kan kalde funktionen med .value
8 ~minFunktion.value;
9 // Se i post window, at instruksen bliver udført
```

Kodeboks 1.11: En funktion

1.4.2 Output fra funktioner

Vi bruger ofte funktioner, fordi vi ønsker at producere „noget“. Med andre ord er vi ofte interesserede i at opfange og bruge en funktions output. Outputtet fra en funktion er altid det, der fremgår i den sidste instruks/kodelinje i funktionen. Som eksempel kan vi lave en funktion, der trækker et tal fra et andet tal. Når vi eksekverer funktionen med `.value` vil vi derfor få resultatet af udregningen, som vi så kan bruge til det formål vi ønsker - herunder bliver den blot gemt under et andet variabelnavn og derefter vist i SuperColliders post window.

```
1 ~minFunktion = {
2     10 - 5;
3 };
4
5 ~resultat = ~minFunktion.value;
6 ~resultat.postln;
7 // -> 5
```

Kodeboks 1.12: En funktion til addition

Fordi der kun er én kodelinje i funktionen, vil resultatet af denne kodelinje være funktionens output. I dette tilfælde er funktionen selvfølgelig ikke særligt nyttig, da den altid vil give samme resultat. Lad os derfor se på, hvordan vi kan bruge funktioner mere fleksibelt ved hjælp af det, der hedder *argumenter*.

1.4.3 Input til funktioner: Argumenter

Funktioner kan også have en slags input, som vi kalder for *argumenter*. Lad os sige, at i stedet for at lægge de samme to tal sammen, hver gang funktionen kører, ønsker at angive de to tal og så få funktionen til at regne summen ud for os. I dette tilfælde kan vi indføre to argumenter, hvilket vi gør på den første kodelinje i funktionen ved hjælp af nøgleordet **`arg`**. Derefter kan vi angive værdier til argumenterne i parenteser, når vi bruger `.value`.

```
~minFunktion = {
    arg kaffe, the;
    kaffe - the;
};

~minFunktion.value(100, 20);
// -> 80
```

Kodeboks 1.13: Argumenter til funktioner

1.4.3.1 Standardværdier

Vi giver ofte argumenter nogle standardværdier, så funktionen kan fungere, selv hvis vi ikke angiver en værdi, når vi kalder den. På den måde får vi mulighed for at arbejde med en standardudgave og variationer derover.

```
~minFunktion = {
    arg kaffe = 10, the = 5;
    kaffe - the;
};

~minFunktion.value;
// -> 5
```

Kodeboks 1.14: Argumenter og standardværdier

Angiver vi værdier, når vi kalder funktionen med `.value`, regner SuperCollider med, at vi skriver dem i den rækkefølge, de er indført i funktionens første linje. I vores tilfælde ovenfor betyder det, at vi skal angive `kaffe`, før vi kan angive `the`. Men med standardværdier kan vi vælge at springe et eller flere argumenter over og i stedet fortælle SuperCollider hvilket argument, vi ønsker at specificere.

```
~minFunktion = {
    arg kaffe = 10, the = 5;
    kaffe - the;
};

~minFunktion.value(the: 20);
// -> -20
```

Kodeboks 1.15: Navngivet argument ved funktionskald

Når vi ikke angiver en værdi til et argument (ovenfor er `kaffe`-argumentet ikke specificeret), bruger funktionen i stedet standardværdien.

1.4.4 Mellemregninger med lokale variabler

I praksis arbejder vi typisk med mere komplekse funktioner end de eksempler vi har set ovenfor. I den sammenhæng er det særligt nyttigt at anvende lokale variabler (se 1.3.2) til at holde styr på data i vores mellemregninger. Lad os eksempelvis tilføje en udregning til vores lille eksempel.

```
~minFunktion = {
    arg kaffe = 10, the = 5;
    var drik = kaffe - the;
    drik * kaffe;
};

~minFunktion.value;
// -> 50
```

Kodeboks 1.16: Lokale variabler i funktioner

Indtil videre har vores funktioner ikke været musikalske, da de primært har skullet illustrere på enkel vis, hvordan vi bruger funktioner. Men én af anvendelserne af funktioner har at gøre med klangdannelse, hvor vi bruger en særlig funktionstype: *UGen-funktioner*.

1.4.5 UGen-funktioner

Funktionerne ovenfor kører i SuperColliders fortolker (se 1.1.1). Men den UGen-funktioner kører på lydserveren. De noteres grundlæggende på samme måde som i eksemplerne ovenfor, men indeholder primært såkaldte UGens, som vi skal se nærmere på i et senere kapitel (se 4.1).

Før vi kan lave lyd med UGen-funktioner skal vi boote lydserveren (se 1.2.2). Derefter noterer vi `.play` umiddelbart efter funktionens afsluttende tuborgklamme. Nedenstående UGen-funktion producerer en sinustone, der svinger ved 440 Hz (stop lyden igen med Ctrl/Cmd-Punktum).

```
{ SinOsc.ar(440) }.play;
```

Kodeboks 1.17: En UGen-funktion

Ligesom ved almindelige funktioner er det sidste linje, der udgør outputtet, altså det vi hører[4]. På samme måde som ovenfor kan vi bruge lokale variabler til at håndtere vores data, som UGen-funktioner svarer til signaler. Vi kan eksempelvis assigne vores sinustonegenerator til en lokal variabel, definere en anden lavfrekvent oscillator unden en anden variabel, og bruge sidstnævnte til at styre lydstyrken for førstnævnte:

```
{
    var tone = SinOsc.ar(440);
    var lfo = SinOsc.ar(2);
    tone * lfo;
}.play;
```

Kodeboks 1.18: Lokale variabler i UGen-funktioner

Bemærk i øvrigt hvad der sker, hvis vi øger frekvensen for LFO'en - så er vi i gang med en klangdannelsesteknik, der kaldes amplitudemodulation (AM).

```
{
    var tone = SinOsc.ar(440);
    var lfo = SinOsc.ar(200);
    tone * lfo;
}.play;
```

Kodeboks 1.19: Simpel AM-syntese

Vi vender grundigt tilbage til UGen-funktioner senere i bogen (se 4.1).

1.4.6 Indbyggede funktioner

Der findes en række indbyggede funktioner, som det kan være nyttigt at kende til. Dem kalder vi ikke ved at bruge `.value`, i stedet bruger vi et sæt parenteser efter

[4] Når vi senere arbejder med SynthDefs (se 5.3), er det værd at notere sig, at outputtet i stedet skal angives med en særlig UGen.

funktionsnavnet. Vi kan fx bruge funktionen `rrand` til at generere et tilfældigt tal eller funktionen `midicps` til at omregne fra MIDI-tonetal til frekvens.

```
1 rrand(0, 10);
2 rrand(100, 200);
3
4 midicps(69);
5 midicps(57);
6
7 // En tilfældig MIDI-tone, omregnet til frekvens:
8 midicps(rrand(0, 127));
```

Kodeboks 1.20: Indbyggede funktioner 

I det sidste eksempel her bliver den „inderste“ funktion udført først, dvs. `rrand` vælger først sit tal, og derefter omregner `midicps` dette tal til en frekvens. Dette kaldes med et fancy udtryk „order of execution“, hvilket såmænd blot betyder, at SuperCollider udregner de „inderste“ funktionskald og matematiske operationer først, når vores kode eksekveres.

1.5 Indbygget funktionalitet i methods

Al indbygget funktionalitet i SuperCollider er knyttet til en særlig gruppe af funktioner, der kaldes *methods*. Hvis vi forstår og kan anvende de forskellige methods, der findes, kan vi groft sagt bruge alle de redskaber, der findes i SuperCollider! Derfor er det relevant at lære hvordan methods fungerer.

1.5.1 Funktionalitet er knyttet til methods

Når vi skal have SuperCollider til at udføre en handling, som fx at sætte en lyd i gang, bruger vi en eller flere konkrete *methods*. En method er en *funktion*, hvis rolle er at udføre en bestemt handling, når den bliver aktiveret.

Lyder det abstrakt? Okay, lad os se på hvordan methods optræder i kildekode: Methods noteret typisk lige efter punktummer. I et fiktivt eksempel som `~car.drive`, er `drive` altså en method. Methods kaldes undertiden også for „messages“. Det skyldes, at methods altid anvendes på „noget“, og dette noget kalder man så en „receiver“ - i det fiktive eksempel her er receiveren det, der er gemt under variablen `~car`. Men lad os kigge på nogle reelle methods i SuperCollider.

```
1 // method'en .postln viser objekter i post window, i dette
  ↪  tilfælde et stykke tekst
2 "Hello world".postln;
3
4 // method'en .minorPentatonic angiver (kombineret med receiveren
  ↪  Scale) en mol-pentaton skala
5 Scale.minorPentatonic;
6
7 // method'en .midicps omregner et tal fra MIDI-tonehøjde til
  ↪  frekvens, målt i hertz
8 69.midicps;
```

Kodeboks 1.21: Tre methods med forskellig funktionalitet

Vi kan ikke gennemgå alle methods i SuperCollider her, da der er alt for mange til at det rent praktisk giver mening. Men vi kommer løbende i bogen til at gå i dybden med hvordan man bruger konkrete methods. Dog er det vigtigt at forstå, at alle methods hører til enten en *class* (klasse) eller en *instance* (forekomst).

1.5.2 Class methods

En klasse er blot en kategori af objekter. Klasser repræsenteres af ord, som er skrevet med stort begyndelsesbogstav, fx **Scale**, **Pbind** og **SinOsc**. *Class methods* er knyttet til, ja, klasser. For at aktivere en class method noterer man førstklassenavnet, derefter et punktum, og til sidst method-navnet.

```
1 TempoClock.tempo; // finder eller angiver tempo
2 Scale.phrygian;   // den frygiske skala
3 SinOsc.ar;        // angiver en sinus-oscillator
4 Pbind.new;        // opretter en ny Pbind (ramme for
  ↪ komposition)
```

Kodeboks 1.22: Eksempler på class methods

Der findes i SuperCollider en særlig class method, som vi bruger hele tiden: `.new`. Denne method skaber et nyt objekt - en „instance“/forekomst af den valgte klasse.

```
1 Pbind.new()     // opretter en ny Pbind
2 SynthDef.new()  // opretter en ny SynthDef
3 Scale.new()     // opretter en ny skala
```

Kodeboks 1.23: Eksempler på .new

`.new` har en særstatus, fordi det at oprette en ny forekomst af noget er så almindeligt, at der for rigtig mange klasser findes en genvej til at bruge method'en - nemlig helt at udelade `.new`. Derfor giver de to nedenstående udtryk samme resultat.

```
1 Pwhite.new(0, 10)   // .new fremgår eksplicit
2 Pwhite(0, 10)       // .new er underforstået
3 // Denne forekomst af Pwhite vil generere værdier mellem 0 og 10
```

Kodeboks 1.24: Eksplicit og implicit .new

Om man bruger eks- eller implicit `.new` er et spørgsmål om personlig præference. Den eksplicitte tilgang er mest tydelig at læse, men ofte vil den implicitte udgave være mest oplagt og give sig selv. Det er fx tilfældet, når vi i næste kapitel skal arbejde med patterns (se 2.1), hvor man meget ofte opretter nye patterns med operationer som **Pwhite**(0, 5) og **Pseq**([1, 2, 3]).

1.5.3 Instance methods

Instance methods bruger vi på „instances“, dvs. konkrete forekomster af bestemte klasser af objekter. Fx er `9` en forekomst af klassen **`SimpleNumber`**. `"kaffe"` er en forekomst af klassen **`String`**. Instance methods noteres ligesom class methods med et punktum først[5], men de forekommer efter instances, ikke klassenavne. Her er nogle eksempler:

`.dup`
: Kopierer det objekt, den anvendes på. Det kan fx være en sinustone-oscillator: **`SinOsc`**`.ar.dup` giver to sinustone-oscillatorer.

`.play`
: Kan bruges i flere forskellige sammenhænge. Med **`Pbind`**`.new().play` kan vi starte med at afspille en ny Pind.

`.midicps`
: En method, som blandt andet kan anvendes på tal (og UGens (se 4.1)). Method'en omregner MIDI-tonetallet 69 til frekvens målt i Hz. `69.midicps` bliver altså til `440`. Bemærk, at dette er præcist det samme som vi så ifm. indbyggede funktioner (se 1.4.6), altså `midicps(69)`.

Det smarte ved at lære methods at kende er, at mange methods kan anvendes på klasser og instanser, som er i familie med hinanden. Fra og med denne bogs introduktion til oscillatorer og beslægtede redskaber (se 4.1) kommer vi fx til at arbejde med forskellige klasser inden for den overordnede **`UGen`**-klasse, fx **`SinOsc`**, som er en oscillator, der genererer sinustone og **`Pulse`**, som genererer firkantede bølgeformer. Alle UGens har en række methods til fælles. Outputtet fra begge oscillatorer kan skaleres med en method, der hedder `.range`, som vi gennemgår senere (se 4.2.2.1). Har man lært disse methods at kende for én UGen, kan man sandsynligvis bruge dem for de fleste andre UGens også.

Flere centrale methods kan bruges med mange forskellige slags objekter. Det gælder eksempelvis for `.play`, som vi bruger jævnligt. Herunder kan du se nogle eksempler (for at høre resultatet af de første to linjer skal lydserveren først bootes (se 1.2.2)).

[5] For nogle (men ikke alle) methods findes en syntaktisk variant, hvor instance methods noteres *før* objektet, adskilt med et simpelt mellemrum. Fx er `play SinOsc.ar;` helt korrekt syntaks, der svarer til `SinOsc.ar.play;`. Jeg anbefaler, at man holder sig til den sidstnævnte form, da punktummet mellem method og objekt tydeliggør, hvad der er method og hvad denne method knytter sig til.

```
Pbind().play;
{SinOsc.ar * 0.1}.play;
Routine({"hej ".post; 1.wait; "med dig".postln;}).play;
```

Kodeboks 1.25: Forskellige resultater med .play

Det er dog ikke alt, der kan „afspilles“ med `.play`. Begge instrukser herunder resulterer derfor i fejlmeddelelser.

```
10.play;
// Det giver ikke mening at afspille et tal

Pbind.play;
// .play findes som instance method for forekomster af Pbind,
↪  ikke som class method
```

Kodeboks 1.26: .play kan ikke anvendes på alle objekter!

1.5.4 Argumenter

I mange tilfælde kan vi styre hvordan en given method fungerer. Det gør vi ved hjælp af *argumenter* - præcis ligesom ved funktioner (se 1.4.3). Vi har ovenfor set en række tilfælde, hvor en method er blevet efterfulgt af et sæt parenteser, nogle gange med nogle tal eller anden tekst noteret mellem parenteserne. Det, der står mellem parenteserne, kalder vi for argumenter. Med vores fiktive eksempel ovenfor kunne man forestille sig, at `~car.drive(50)` og `~car.drive(100)` ville få en bil til at køre med henholdsvis 50 km/t og 100 km/t. Herunder ses nogle reelle eksempler methods med argumenter.

```
// vi beder om en sinustone-oscillator, som svinger ved 220 Hz
↪  audio rate
SinOsc.ar(220);

// vi beder om 10 kopier af en sinustone-oscillator
SinOsc.ar.dup(10);

// vi beder om et pattern, der kan generere tilfældige tal
↪  mellem 0 og 7
Pwhite.new(0, 7);
```

Kodeboks 1.27: Styring af methods med argumenter

1.5.5 Dokumentation for methods

I SuperColliders dokumentation kan man se hvilke methods, der passer til forskellige klasser.

- Slår man en klasse op (fx **Array**, **Pwhite**, **SinOsc** etc.), vil man kunne se hvordan de forskellige class og instance methods fungerer og anvendes.
- Slår man en method op (fx `.new`, `.play`, `.reverse` etc.), vil man kunne se de forskellige klasser, som har denne method tilknyttet. Klik på klassenavnet for at se dokumentationen for den pågældende methods rolle i forhold til den specifikke klasse af objekter.

Placér cursoren sammen med et af kodeudtrykkene herunder og tast Ctrl/Cmd-D for at åbne dokumentationen. Læs selv nærmere om de forskellige methods, og bemærk hvilke resultater, der dukker op.

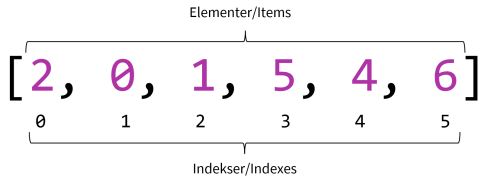
```
1  // Hvilke class og instance methods knytter sig til henholdsvis
   ↪  Array, Pwhite og SinOsc?
2  Array
3  Pwhite
4  SinOsc
5
6  // På hvilke slags objekter (dvs. hvilke klasser) kan vi bruge
   ↪  disse methods?
7  .new
8  .play
9  .trace
10 .ar
11 .reverse
```

Kodeboks 1.28: Undersøg selv dokumentation for methods

1.6 Organisering med lister

Når vi arbejder med musik, har vi ofte brug for at organisere musikalske parametre som rækker af tal. Fx kan man sige, at tonerne i en komposition udgør en lang liste af nodeværdier og tonehøjder. Man kan anskue skalaer og akkorder som lister af intervaller eller skalatrin. Eller man kan se rytmiske sekvenser som en liste af tidsintervaller - osv. Rytmer, akkorder, sekvenser, tonetrin, skalaer, varigheder, overtoner osv. kan altså repræsenteres som rækker af tal. Til at organisere disse rækker af tal bruger vi noget, der kaldes „arrays“, som vi på dansk blot kan kalde *lister*. I denne bog bruges begge termer om det samme[6]. Det er heldigvis ganske enkelt at bruge lister, og det er yderst nyttigt at vide hvordan de fungerer.

En liste/et array er teknisk set en såkaldt datastruktur, som kendetegnes ved, at den indeholder en samling af **elementer**, der er sat i en bestemt **rækkefølge**. Hvert element har en plads i rækkefølgen, som betegnes elementets **indeks**. Det første element har indeks 0, det næste indeks 1, og så fremdeles. Har du bemærket, at kapitlerne i denne bog er nummereret på samme måde?



Figur 1.4: Listens elementer og indekser

1.6.1 Notation og brug af lister

Hvis vi ønsker at notere de tre skalatrin i en akkord uden lister, kan vi gøre det ved hjælp af variabler (se 1.3). Det kunne eksempelvis være på denne måde: `~tone1 = 0; ~tone2 = 2; ~tone3 = 4;` - men det er typisk ikke særligt effektivt i længden. Når man ser et mønster som dette, er det typisk på tide at overveje at bruge en liste. For at notere vores akkord som en liste bruger vi kantede parenteser til afgrænse listen, og så noterer vi elementerne, adskilt med komma. Hvis vi gemmer vores liste som en variabel, kan vi tilgå elementerne enkeltvis ved at angive deres indeks, omkranset af kantede parenteser, efter variabelnavnet.

[6] Lister/arrays hører teknisk set til klassen `Array` i SuperCollider, men der findes også en beslægtet datastruktur, der hedder `List`. Forskellen mellem de to er blot, at sidstnævnte kan udvides dynamisk. Nysgerrige læsere henvises til SuperColliders dokumentation.

```
~akkord = [0, 2, 4];

~akkord[0].postln; // -> 0
~akkord[1].postln; // -> 2
~akkord[2].postln; // -> 4

~akkord[2] = 7;    // -> [ 0, 2, 7 ]
~akkord[2].postln; // -> 7
```

Kodeboks 1.29: Notation af lister

Vi kommer senere (se 2.1) til at se nærmere på, hvordan man bruger en liste som vores akkord ovenfor til faktisk at spille en akkord. Herunder kommer dog en smagsprøve - husk at boote lydserveren (se 1.2.2), før du eksekverer kildekoden.

```
~akkord = [0, 2, 4];

// Akkorder
Pbind(\degree, ~akkord);
Pbind(\degree, ~akkord, \strum, 0.05);

// Akkordbrydninger
Pbind(\degree, Pseq(~akkord));
Pbind(\degree, Pshuf(~akkord, 2));
```

Kodeboks 1.30: Første lydeksempel med lister

1.6.2 Matematik med lister

En af de mange fordele ved at samle vores data i én struktur er, at vi kan arbejde med alle elementerne på én gang. Vi kan eksempelvis foretage matematiske operationer, som bliver udført for alle elementerne. Det skyldes, at når en liste udelukkende indeholder tal (i modsætning til fx tekstbidder eller andre objekter), så understøtter listen som regel også de methods, som knytter sig til tal.

```
[0, 2, 4] + 1;
// -> [1, 3, 5]

[0, 2, 4] * 2;
// -> [0, 4, 8]

[0, 2, 4].pow(2); // .pow(2) betyder "i anden"
// -> [0, 4, 16]
```

Kodeboks 1.31: Matematiske operationer med lister 

Der findes derudover en lang række methods (se 1.5), som er særlige for lister. Vi kan fx bede om en sum, et gennemsnit, et plot og mange andre funktioner.

```
[0, 2, 4].sum;
// -> 6

[0, 2, 4].mean;
// -> 2

[0, 2, 4].plot;
// -> En graf over data
```

Kodeboks 1.32: Methods til lister 

1.6.3 Iteration over lister

Hvis man ønsker at gøre noget for alle elementer i en liste, men ikke kan finde en indbygget method til at opnå det ønskede resultat, kan man angive - you guessed it - en funktion (se 1.4), som kører for hvert element. Inden for programmeringsverdenen siger man i det tilfælde, at man *itererer over listens elementer*. Hertil findes der særligt to nyttige methods for lister: `.do` og `.collect`. Forskellen på de to er, at førstnævnte blot udfører funktionen for hvert element, hvor sidstnævnte skaber et nyt array med outputtet fra hvert funktionskald. Det lyder måske teknisk, men lad os tage nogle eksempler for at forstå sammenhængen.

For at iterere over en liste angiver vi som et argument til `.do` den funktion, vi ønsker at udføre. Inde i den funktion opretter vi et argument (se 1.4.3). Herunder vises de tre elementer blot i post window med et lille stykke tekst efter.

```
~akkord = [0, 2, 4];

~akkord.do({
    arg trin;
    (trin + " er et flot tal.").postln;
});
```

Kodeboks 1.33: Iteration med .do

Ønsker vi at bruge elementernes respektive indeks i funktionen, kan vi også gøre det - her indfører vi endnu et argument, som ved hvert funktionskald vil modtage det aktuelle elements indeks.

```
~akkord = [0, 2, 4];

~akkord.do({
    arg trin, indeks;
    ("I denne iteration kigger vi på indeks " + indeks).postln;
    (trin + " er et flot tal.").postln;
});
```

Kodeboks 1.34: Iteration med indeks

Sidst men ikke mindst kan vi bruge `.collect` til at transformere arrays. Vi kan fx fremstille en akkordrækkefølge, altså teknisk set en liste bestående af lister.

```
~akkord = [0, 2, 4];

// Æblemand-sekvens - trin: I-VI-II-V
~grundtoner = [0, -2, 1, -3];

~akkorder = ~grundtoner.collect({
    arg trin;
    ~akkord + trin;
});

~akkorder.postln;
// -> [ [ 0, 2, 4 ], [ -2, 0, 2 ], [ 1, 3, 5 ], [ -3, -1, 1 ] ]

// Afspil akkorden:
Pbind(\degree, Pseq(~akkorder)).play;
```

Kodeboks 1.35: Iteration med .collect

Iteration er et særdeles nyttigt redskab, der både benyttes i komposition, som antydet ovenfor, og i klangdannelse. Vi bruger eksempelvis iteration til at definere oscillatorbanke, når vi skal arbejde med overtonerækker. Men mere herom senere (se 7.1).

1.6.4 Tricks til skabelse af lister

I stedet for at definere vores lister manuelt kan vi være snedige og oprette dem med en algoritme. Det kan fx være, at vi gerne vil have en lineær talrække, en liste med tilfældige tal, eller en liste med elementer skabt af en funktion, vi selv skriver.

1.6.4.1 Automatiske talrækker

Lineære talrækker, dvs. en liste med tal, hvor der er et bestemt interval mellem på hinanden følgende tal, kan genereres ganske let. Class-method'en `Array.series(antal, start, interval)` skaber en liste med det angivne antal elementer, det første element vil have startværdien, og de efterfølgende værdier vil have den forudgående værdi + det angivne interval. Denne form er meget tydelig og læsbar, men har dog den ulempe/fordel, at man ikke kan se, hvor talrækken slutter (altså hvad der vil være det sidste tal i listen). I stedet kan man bruge den syntaktiske genvej (`start .. slut`).

```
1  // Alle heltal fra 10 til 20
2  Array.series(11, 10, 1);
3  (10..20);
4  // -> [ 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 ]
5
6  // De lige tal fra 10 til 20
7  Array.series(6, 10, 2);
8  (10, 12..20);
9  // -> [ 10, 12, 14, 16, 18, 20 ]
10
11 // 5-tabellen
12 Array.series(11, 0, 5);
13 (0, 5..50);
14 // -> [ 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50 ]
```

Kodeboks 1.36: Automatiske talrækker

Disse tilgange virker i øvrigt også med decimaltal og negative intervaller.

```
// Virker også med decimaltal ...
Array.series(11, 0, 0.1);
(0, 0.1..1);
// -> [ 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0 ]

// ... og negative intervaller
Array.series(11, 0, -1);
(0..-10);
// -> [ 0, -1, -2, -3, -4, -5, -6, -7, -8, -9, -10 ]
```

Kodeboks 1.37: Talrækker med decimaltal og negative intervaller 

Hvilken form er at foretrække? Det kommer an på, om man på forhånd gerne vil vide hvor mange elementer, der er i listen (hvis det fx skal passe med længden af en musikalsk sekvens) - så er `Array.series` bedst. Hvis det er vigtigere at kende start- og slutgrænsen, kan vi bruge genvejen. Der findes dog en beslægtet tilgang, som i nogle tilfælde er handy: `Array.interpolation(antal, start, slut)` genererer en liste med det ønskede antal elementer, fordelt lineært mellem start og slut.

```
Array.interpolation(9, 2, 10);
// -> [ 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0 ]
```

Kodeboks 1.38: Array.interpolation 

1.6.4.2 Lister med tilfældigt genererede tal

Ønsker vi en liste med tilfældigt genererede tal, kan vi bruge `Array.rand(antal, minimum, maksimum)`. Lad os eksempelvis generere en liste med 10 tal mellem 0 og 127 (rækkevidden i MIDI-protokollen).

```
// Kør denne instruks flere gange for at se resultatet
Array.rand(10, 0, 127);
// -> [ 38, 63, 11, 82, 63, 107, 47, 13, 92, 79 ]
// -> [ 67, 99, 121, 78, 93, 67, 87, 85, 102, 32 ]
// -> [ 31, 11, 40, 122, 86, 11, 20, 87, 15, 127 ] osv.
```

Kodeboks 1.39: Lister med tilfældige tal 

Dette kan vi eksempelvis bruge til at spille forskellige toner eller volumen-indstillinger på en ekstern synthesizer via MIDI.

Der findes et par beslægtede methods, som kan være nyttige, blandt andet `Array.rand2()` og `Array.exprand`. Særligt interesserede læsere kan se hvordan de fungerer i SuperColliders dokumentation - se sektionen om class methods under opslaget om Array.

1.6.4.3 Hjemmestrikkede algoritmer til at generere lister

Kan vi ikke finde et indbygget redskab til at skabe vores liste, kan vi altid skrive en funktion, der producerer listen for os. Det kan vi gøre ved hjælp af `Array.fill(funktion)`, der på mange måder minder om de iterationsteknikker, vi arbejdede med ovenfor.

Lad os sige, at jeg vil skabe en liste med 10 frekvenser, der svarer til tilfældigt valgte MIDI-tonehøjder inden for to oktaver. Vi kan også vise MIDI-tonehøjderne i post window, mens vi skaber listen.

```
1 Array.fill(8, {
2     var tone = rrand(48, 72);
3     tone.postln;
4     tone.midicps;
5 });
6 // -> [ 391.99543598175, 164.81377845643, 184.99721135582,
  ↪  195.99771799087, 184.99721135582, 311.12698372208,
  ↪  207.65234878997, 220.0 ]
```

Kodeboks 1.40: Hjemmestrikket liste med Array.fill

Sidst men ikke mindst kan det være nyttigt at bruge elementernes indeks i funktionen. Dertil kan vi oprette et argument, ligesom ovenfor i forbindelse med iteration. Lad os eksempelvis ved hjælp af indekstallene beregne frekvenserne for de første 10 naturlige overtoner for en tone, der klinger ved 220 Hz.

```
(
~overtoner = Array.fill(10, {
    arg indeks;
    // vi lægger 1 til, fordi vi starter ved indeks 0
    var toneNummer = indeks + 1;
    220 * toneNummer;
});
// -> [ 220, 440, 660, 880, 1100, 1320, 1540, 1760, 1980, 2200 ]
)

// Boot lydserveren for at høre tonerne
{ SinOsc.ar(~overtoner).sum * 0.01; }.play;
```

Kodeboks 1.41: Den naturlige overtonerække

I SuperCollider og mange andre programmeringssprog er der ofte mere end én metode til at opnå det samme resultat. Vi kan fx bruge `.collect`, som blev introduceret ovenfor i forbindelse med iteration, sammen med genvejen til at beskrive lineære talrækker, til at opnå samme resultat som ovenfor. Eller vi kan bruge simple matematiske operationer, også som beskrevet ovenfor.

```
(1..10).collect({
    arg toneNummer;
    220 * toneNummer;
});
// -> [ 220, 440, 660, 880, 1100, 1320, 1540, 1760, 1980, 2200 ]

(1..10) * 220;
// -> [ 220, 440, 660, 880, 1100, 1320, 1540, 1760, 1980, 2200 ]
```

Kodeboks 1.42: Alternativer til Array.fill

Der er ikke nogen tilgang, som er mere rigtig end andre. Jeg bruger ofte **`Array`**`.fill` eller `.collect`, når der ikke findes tilsvarende methods eller genveje, eller hvis jeg foretrækker den eksplicitte og tydelige form på grund af tydelighed. Nogle gange bruger jeg også den korte form, som er hurtigere at notere men har den ulempe, at den kan være mere vanskelig at læse, hvis man hurtigt skal gennemskue en større mængde kildekode.

1.7 Strategier til fejlsøgning

Når man er i gang med at lære et nyt sprog, er det uundgåeligt, at man fra tid til anden skriver et ord forkert eller ikke har helt styr på syntaksen. Det er helt normalt, at man løber ind i vanskeligheder, når man lærer at programmere. Det skyldes, at computere - uanset hvor smarte de er - har endog meget svært ved at forstå vores instruktioner. De seneste års fremgang inden for kunstig intelligens og store sprogmodeller implementeret i chatbots kan måske få computeren til at virke intelligent og forstående, selv når vi ikke formulerer os helt klart. Men faktisk har computere i programmeringssammenhæng brug for, at vi følger nogle klare regler om et forudbestemt ordforråd. De kræver også, at vi følger en entydig syntaks til punkt og prikke. Ellers giver de op og giver os en fejlmeddelelse. Og det sker jævnligt, også for erfarne programmører. Derfor er det vigtigt, at du etablerer nogle gode strategier til fejlsøgning: Det vil både være nyttigt i forhold til at løse de problemer, du måtte løbe ind i, men det vil også give adgang til en bredere viden og teknisk forståelse. En fejlmeddelelse er måske ikke lige det sjoveste, vi kan opleve, men den udgør ofte en lejlighed til at lære noget nyt.

1.7.1 Sådan håndterer du kode, der ikke virker

Øv - din kode virker ikke. Enten kommer der slet ingen lyd, når du kører din kode, eller også sker der ikke det, du forventer. Du har nu siddet den sidste halve time og prøvet at finde fejlen. Du har genstartet SuperCollider og din computer, men desværre uden held. Hvad gør man så?

1.7.1.1 Læs fejlmeddelelsen

Hvis der dukker en fejlmeddelelse eller anden relevant information op i SuperColliders post window, er dette det første sted, du bør kigge. SuperColliders fejlmeddelelser er ikke altid lige informative, men ofte giver de et hint til hvad der er på færde. Fejlmeddelelsen vil være fremhævet med en særlig tekstfarve, men nogle gange er det nødvendigt at scrolle op i SuperColliders post window for at finde den.

Figur 1.5: En fejlmeddelelse

1.7.1.2 Tjek de mest typiske fejlkilder

Der findes nogle fejl, som jeg ser igen og igen hos studerende. Og indrømmet, de sniger sig lejlighedsvis også ind i kildekode, jeg selv skriver. Så tjek først, om det kan være et af disse problemer, du står med.

PARENTESER OG KLAMMER
: Er alle dine parenteser og klammer, dvs. `( )`, `{}` og `[]`, men også omkransende tegn som `\**\`, `''` og `""` startet og afsluttet korrekt?

VIRKER CTRL/CMD-ENTER IKKE?
: Dette skyldes rod i parenteserne. Se om du kan finde en manglende parentes et sted. Prøv at flytte en lille del af din kode over i et nyt dokument for at tjekke, om den del virker. Gentag dette, indtil du finder fejlen.

KOMMA/PUNKTUM/KOLON/SEMIKOLON
: Er følgende tegn anvendt korrekt? `,` `.` `:` `;`

STORT/SMÅT BEGYNDELSESBOGSTAV
: Klassenavne som **`Pbind`** og **`Saw`** skal have stort begyndelsesbogstav. Methods og variabler som `.play` og `~minVariabel` har altid små begyndelsesbogstaver.

HVIS DU IKKE HØRER NOGET LYD
: Er lydserveren bootet? - Dette tjekker du ved at se om rækken af tal nederst til højre er grønne (så er serveren bootet) eller grå (serveren er ikke bootet). Brug evt. den lidt morbide kommando **`Server`**`.killAll;` for at slukke alle serverprocesser, før du genstarter serveren.

1.7.1.3 Simplificér din kildekode og isolér problemet

Det er altid fornuftigt at forenkle kildekoden så meget som muligt. Det gør det lettere at gennemskue intentionen med kildekoden samt hvor fejlen kan have sneget sig ind.

DEL KILDEKODEN OP
: Prøv at flytte det stykke kildekode, du arbejder på, over i et nyt dokument. Hvis du har syntaksfejl i dele af dit dokument, kan det påvirke resten af dokumentet.

DEAKTIVER DELE AF KILDEKODEN
: Prøv at deaktivere dele af din kode (med `//` eller `/**/`) og eksekvere igen for at finde den del af koden, som er årsag til problemerne.

SAMMENLIGN DIN KILDEKODE MED ET FUNGERENDE EKSEMPEL
: Find et lignende eksempel fra undervisningsmateriale eller SuperColliders dokumentation, som virker. Kan du få øje på nogle forskelle på din kode og det valgte eksempel?

START FORFRA MED AFSÆT I ET FUNGERENDE EKSEMPEL
: Start forfra i et nyt dokument med et simpelt eksempel fra denne bog eller andre lødige ressourcer, hvor koden fungerer som forventet. Øg kompleksiteten derfra og eksekvér koden så ofte som muligt. På den måde kan du identificere hvor problemet opstår.

1.7.1.4 Tal med en gummiand

Det lyder måske lidt skørt, men der findes faktisk en udbredt debugging-strategi, som kaldes *rubber duck debugging*. Strategien går ud på, at man (til en reel eller metaforisk gummiand) forklarer, hvad man forventer, at kildekoden gør. Det er vigtigt, at man forklarer kildekoden trin for trin, samt at man beskriver det uventede, der sker. At beskrive processen og problemet på denne måde kan give aha-oplevelser og få selv de mest garvede programmører til at få øje på det, de ellers havde stirret sig blinde på (Hunt, 2000, p. 95). Dette kan rent læringsmæssigt betragtes som en form for problemanalyse, der gennem verbalisering giver mulighed for ny indsigt.

En nyere variant af denne strategi kunne være at konferere med en sprogmodel (chatbot). Moderne chatbots kan i nogle tilfælde (men bestemt ikke altid) spotte syntaksfejl eller andre problemer, som vi selv har stirret os blinde på. OBS: Hvis du er studerende, bør du være opmærksom på, om brug af kunstig intelligens til fejlsøgning er tilladt på din uddannelse. Spørg din underviser eller studievejleder, hvis du er i tvivl.

1.7.1.5 Søg information om problemet

Du kan altid konsultere SuperColliders indbyggede dokumentation. Der kan du søge på nøgleord, klassenavne, methods osv. Du kan også slå konkrete klasser eller methods op, hvis du placerer din cursor ved fx klassenavnet og trykker Ctrl/Cmd-D. Så vil SuperCollider slå den markerede klasse eller method op for dig. Det kan være utroligt handy, hvis man fx ikke kan huske hvad argumenterne til en bestemt method går ud på.

Derudover findes der selvfølgelig søgemaskiner og sociale fora online, som kan være udmærkede kilder til at lære af andre brugeres udvekslinger og erfaringer. Søg på de nøgleord, der har med dit problem at gøre, og måske kan du være heldig, at andre har oplevet det samme og løst problemet.

1.7.1.6 Spørg om hjælp

Der findes mange mennesker, der gerne vil hjælpe - især hvis man spørger på en konstruktiv og ordentlig måde. Spørg hellere en gang for meget end en gang for lidt!

Når du spørger din underviser om hjælp vedrørende musik- og lydprogrammering, er det altid en god ide at vedlægge følgende.

- Et minimalt eksempel på den kildekode, som ikke giver det forventede resultat. Hvis man ikke kan se kildekoden, er det næsten altid umuligt at hjælpe.
- En beskrivelse af problemet: Hvad forventer du, og hvordan afviger resultatet fra det forventede?
- En beskrivelse af fejlsøgningen: Hvad har du allerede selv prøvet for at løse problemet?

Hvis disse oplysninger er med, giver du modparten rigtig gode forudsætninger for at komme med et brugbart svar.

Det er helt normalt og forventeligt, at nybegyndere inden for programmering løber ind i syntaksfejl og andre vanskeligheder. Men hvis du bliver god til fejlsøgning, bliver det hurtigt en fornøjelse at løse problemerne.

1.8 Øvelse: Grundlæggende programmering

Denne øvelse omhandler brug af SuperColliders IDE til at eksekvere kildekode, tilføje kommentarer, bruge variabler, samt identificere syntaksfejl i SuperCollider-kildekode.

1.8.1 Forklar kildekoden

› Eksekvér nedenstående kodeblok på én gang. Forklar i kommentarer hvad hver enkelt kodelinje gør.

```
"Else Marie Pade".postln;

TempoClock.tempo.postln;

~akkord = [0, 3, 4];
~akkord.postln;

(3 + 4).postln;
3 + 4.postln;
```

Kodeboks 1.43: Hvad gør kildekoden?

1.8.2 Yndlingstal og variabler

1. Post dit yndlingstal i SuperColliders post window.
2. Gem dit yndlingstal under en global variabel, fx `~mitYndlingstal`.
3. Post indholdet af din variabel.
4. Post indholdet af din variabel ganget med 200.

1.8.3 Nu kører det for dig

1. Eksekvér linjerne i kodeblokken herunder én for én.
2. Eksekvér linjerne i kodeblokken herunder som én kodeblok (se 1.2.1).

```
"Bach is back".postln;
2.pow(24).postln;
[57, 69].midicps.postln;
```

Kodeboks 1.44: Eksekvering af kildekode

1.8.4 En funktionel funktionsforståelse

1. Læs nedenstående kildekode og beskriv hvad hver linje i funktionen gør.
2. Bekræft din forståelse af funktionen ved at køre den med `.value` et par gange.

```
(
~myFunc = {
    arg root = 0;
    var chord = root + [0, 4, 6, 9];
    chord.reverse.mirror;
};
)
~myFunc.value(2);
~myFunc.value(4);
~myFunc.value(-3);
```

Kodeboks 1.45: Funktionens funktionalitet

Boot eventuelt lydserveren med `s.boot` og brug funktionen til at spille en akkordbrydning: `Pbind(\degree, Pseq(~myFunc.value(-3))).play;`. Vi lærer mere om dette i næste kapitel (se 2.1).

1.8.5 Listige liste-methods

1. Undersøg ved hjælp af nedenstående kildekode, hvad de forskellige methods for lister gør.
2. Erstat `~liste` med en liste på mindst 5 tal, som du selv vælger.

Husk følgende:

- Den tekniske term for en „liste“ i SuperCollider er **`Array`**. Du kan også i dokumentationen støde på klassen **`ArrayedCollection`**. For nuværende kan vi blot betragte disse som forskellige navne for lister.
- Listers indeks-tal starter fra 0 (se 1.6). Det første element har derfor indeks 0, det næste indeks 1 og så fremdeles.

```
~liste = [0, 1, 3, 4, 6, 7];

~liste.reverse;
~liste.mirror;
~liste.mirror1;
~liste.scramble;
~liste.size;
~liste.sum;
~liste.mean;
~liste.normalize(0, 100);
~liste.normalizeSum;
~liste.normalizeSum.sum;
~liste.at(4);
~liste[4];
~liste.put(4, -2);
~liste.plot;
~liste.do({ arg num; num.pow(2).postln; });
~liste.collect({ arg num; num * 10; });
~liste.dupEach(3);
~liste.sputter(0.5, 50);
~liste.rotate(1);
~liste.wrapExtend(15);
```

Kodeboks 1.46: Listige liste-methods 

1.8.6 Kan du finde alle de syntaktiske kategorier?

Du er i gang med at lære et nyt sprog. Nedenstående øvelse giver en god forudsætning for at undersøge og forstå kildekode, du ikke har set før.

1. Identificér følgende i kildekoden herunder:
   a) Alle klassenavne. *Hint: Det er dem, der starter med stort begyndelsesbogstav (se 1.5.2).
   b) Alle methods, herunder også implicit (se 1.5.2) `.new`.
   c) Alle lister.
   d) Alle unikke variabelnavne (se 1.3).
   e) Alle funktioner (se 1.4).
   f) Alle argumenter (se 1.4.3).
2. Undersøg ved hjælp af SuperColliders dokumentation følgende:
   a) Hvad dækker de forskellige klassenavne over?
   b) Hvilken funktionalitet knytter sig til de forskellige methods?
   c) Hvilken betydning har argumenterne i de to methods, hvor der er angivet eksplicitte argumenter?

Tip: Du kan slå op i SuperColliders indbyggede dokumentation ved at placere cursoren ved en method eller et klassenavn og taste Ctrl/Cmd-D.

```
~interval = 12;
440 * ~interval.midiratio;

Platform.userExtensionDir;
MIDIClient.init;

(
{
    arg input = 0;
    var output = (2 + input).pow(2);
    output.postln;
}.value(3);
)

s.boot;
{SinOsc.ar(880)}.play;

~trin = [0, 2, 4, 6].mirror;
~komp = Pbind(\degree, Pseq(~trin)).play;
~komp.stop;
```

Kodeboks 1.47: Find Holger, SuperCollider edition 

Der findes i ovenstående kodeblok:

- 5 klassenavne.
- 14 methods (heraf 2 stk. implicit `.new`).
- 1 liste.
- 4 unikke variabelnavne (3 globale og 1 lokal).
- 2 funktioner.
- 6 argumenter.

1.8.7 Find og ret syv fejl

1. Find og ret fejlene i de syv eksempler herunder.
2. Ryd SuperColliders post window med tastaturgenvejen Ctrl/Cmd-Shift-P, inden du starter med et nyt eksempel.
3. Læs fejlmeddelelsen, før du retter fejlen.
4. Forklar i en kommentar for hver fejl hvad du har rettet og hvad problemet bestod i.

```
~alder = 32,5;
~skala = Scale.locrian; \\ her gemmer jeg min yndlingsskala
(
10.postln
20.postln
)
(
~Resultat = 4 * 10;
~Resultat.postln;
)
"57".midicps;
~svar = 10 * (4 + (2 * 1.5);
~svar.postln;
```

Kodeboks 1.48: Find 7 fejl 

1.9 Øvelse: De første bip

Denne øvelse går ud på at producere lyde på SuperColliders lydserver og eksperimentere med at ændre på lyden ved at variere på parametrene. På dette trin er det ikke vigtigt, at du forstår, hvad alle kodelinjerne gør. Til gengæld er det en god ide at eksperimentere med kildekoden for at få en fornemmelse af, hvordan det fungerer. Hvis du får ændret på koden, så den ikke længere fungerer, kan du bare kopiere koden herunder igen og starte forfra.

Start først lydserveren med `s.boot;`

1.9.1 En uendelig sekvens

1. Overvej hvad de enkelte kodelinjer gør ved lyden.
2. Eksekvér kodeblokken og lyt til resultatet. Tryk Ctrl/Cmd-Punktum for at stoppe igen.
3. Prøv at ændre på tallene eller vælge en anden skala, og kør blokken igen for at lytte til resultatet.

```
Pbind(
    // prøv at vælge en anden skala, fx Scale.egyptian
    \scale, Scale.minor,

    // prøv at ændre på de forskellige tal herunder
    \degree, Pseq([0, 3, 2, 1, 4, 5, 6], inf),
    \root, 1,
    \octave, 4,
    \dur, 0.25,    // <-- denne værdi skal være større end 0
    \legato, 1.2,
).play;
```

Kodeboks 1.49: Skalaudforskning

Tip: Kør `Scale.directory;` for at få vist de forskellige indbyggede skalaer. Det er også muligt at definere sine egne skalaer eller bruge alternative stemningssystemer.

1.9.2 Selvkørende oscillatorer

1. Overvej hvad de enkelte kodelinjer gør ved lyden.
2. Eksekvér kodeblokken og lyt til resultatet. Tryk Ctrl/Cmd-Punktum for at stoppe igen.

3. Prøv at ændre på tallene på linje 5-13 og kør blokken igen for at lytte til resultatet.
4. Prøv at ændre **`LFNoise0`** på linje 9 til **`LFNoise1`**, **`LFNoise2`**, **`LFSaw`** eller **`LFPulse`**. Hvordan forandrer det lyden?

```
1  ({
2      var sig, lfo, lfoFreq;
3
4      // prøv at ændre på de forskellige tal herunder
5      var freq = 330;
6      var lfoFreqStart = 2;
7      var lfoFreqEnd = 10;
8      var duration = 7;
9      lfoFreq = Line.kr(lfoFreqStart, lfoFreqEnd, duration,
       ↪  doneAction: Done.freeSelf);
10     lfoFreq = lfoFreq.dup(2);
11     lfo = LFNoise0.kr(lfoFreq);
12     lfo = lfo.bipolar(24);
13     lfo = lfo.round(4);
14
15     // justér ikke herunder
16     lfo = lfo.midiratio;
17     sig = Pulse.ar(freq * lfo);
18     sig = Splay.ar(sig);
19     Limiter.ar(sig * 0.1);
20 }.play;)
```

Kodeboks 1.50: Sjov med LFO'er

KAPITEL 2

En strøm af lyde

Generativ komposition kan være kernen i den kreative proces, når man komponerer, hvilket ofte er tilfældet inden for computermusikken. Men i bredere forstand kan vi også bruge generative processer til at skabe samples, sekvenser, variationer mm. til brug i mere traditionel komposition og lydproduktion.

I SuperCollider udgør de såkaldte patterns et centralt redskab til generativ komposition. Dette kapitel introducerer først til patterns på et grundlæggende niveau, med særligt fokus på hvordan det centrale redskab `Pbind` knytter generative patterns til musikalske parametre som tonehøjde og rytmik. I næste kapitel går vi mere i dybden med hvordan patterns blandt andet ved hjælp af en teknik kaldet *indlejring* kan kombineres til at danne komplekse, generative systemer.

2.1 Introduktion til Pattern-baseret komposition

SuperCollider indeholder et yderst righoldigt bibliotek af patterns, som vi kan kombinere på utallige, kreative måder. Men hvad er patterns egentlig? Patterns er *opskrifter på strømme af værdier*. Det lyder måske abstrakt, men det er egentlig ikke så galt: **Pseq** definerer en sekvens af værdier (ligesom en sequencer), **Pwhite** definerer en strøm af tilfældigt genererede værdier, **Pseries** definerer en lineær række af værdier (fx 1, 2, 3), osv. Og ja, du har nok allerede gættet hvordan man kan spotte et pattern - navnene på SuperColliders pattern-klasser starter belejligt nok altid med P. Det første pattern vi skal forstå er det såkaldte **Pbind**, som udgør rammen for de øvrige patterns, vi anvender.

2.1.1 Rammen for generativ komposition: Pbind

Som udgangspunkt for generativ komposition laver vi en instans af klassen **Pbind**, som vi så kan afspille med method'en `.play` (husk at boote lydserveren (se 1.2.2), hvis du ikke har gjort det allerede).

```
1 // Kør denne linje for at starte kompositionen
2 ~eksempel = Pbind().play;
3
4 // Kør denne linje for at stoppe igen
5 ~eksempel.stop;
```

Kodeboks 2.1: Den simplest mulige Pbind-komposition

Pbind har den funktion, at den „binder" musikalske parametre sammen til en strøm af begivenheder. I Pbind bruger vi på den ene side *nøgler*, angivet med fx `\degree`, `\dur` og `\scale`, til at angive kompositionsmæssige parametre, og vi bruger *patterns* eller *faste værdier* til at angive eller generere disse parametre.

Her er et enkelt eksempel, hvor vi med nøglen `\degree` vælger at knytte den musikalske parameter *skalatrin* sammen med en fast værdi, nemlig værdien 2, som angiver *tredje* skalatrin. Men hov, hvorfor betyder 2 ikke *andet* skalatrin her? Det skyldes, at *første* skalatrin har værdien 0. Dette afspejler en konvention inden for de fleste programmeringssprog, hvor man starter med at tælle ved tallet 0. Ønsker vi *andet* skalatrin, skal vi i stedet angive værdien 1, og så fremdeles.

```
Pbind(
    // Vi vælger 3. skalatrin (i C-dur tonen e)
    \degree, 2,
).play;
```

Kodeboks 2.2: Pbind og skalatrin

Men ovenstående bliver hurtigt lidt ensformigt at lytte til. I stedet for faste værdier kan vi bruge et pattern til at generere forskellige værdier. Vi starter med **Pseq**, som vi kan bruge til at generere en sekvens af værdier - stadig skalatrin, fordi vi bruger \degree-nøglen.

```
Pbind(
    \degree, Pseq([0, 1, 3, 4, 7]),
).play;
// -> 0, 1, 3, 4, 7
```

Kodeboks 2.3: Enkel tonesekvens med Pseq

Alternativt kan vi lade computeren vælge skalatrin for os. Vi kan fx bruge **Pwhite** til at generere 5 tilfældige skalatrin inden for en oktav (trin 0-7).

```
Pbind(
    \degree, Pwhite(0, 7, 5),
).play;
// -> Fx 5, 4, 6, 3, 6
```

Kodeboks 2.4: Tilfældighed med Pwhite

Vi kan også kombinere en fast sekvens med tilfældigt valgte værdier ved at knytte **Pwhite** til \degree og dermed tonehøjde, mens vi knytter **Pseq** til \dur og dermed varighed/rytmik. På den måde får vi en komposition med både faste og tilfældige parametre.

```
Pbind(
    \degree, Pwhite(0, 7),
    \dur, Pseq([0.25, 0.5, 0.25], 4),
).play;
```

Kodeboks 2.5: Kombination af Pwhite og Pseq

Senere i dette kapitel gennemgås de mest almindelige nøgler (se 2.3) og patterns (se 2.4).

Lad os imidlertid først kigge lidt nærmere på nogle af de generative redskaber, vi kan bruge i pattern-baseret komposition. Som eksempel på to forskellige typer af patterns fortsætter vi med at se nærmere på **Pseq** og **Pwhite**.

2.1.2 Pseq - en fleksibel sequencer

Pseq er et *listebaseret* pattern. Når vi bruger **Pseq** til at skabe en sekvens, noterer vi således som det første argument (se 1.4.3) en liste (se 1.6) med de elementer, som skal indgå i sekvensen. **Pbind** knytter derfor én efter én de angivne elementer i listen sammen med den nøgle, **Pseq** er noteret ud for (herunder nøglen for skalatrin, \degree).

```
Pbind(
    \degree, Pseq([7, 4, 2, 0]),
).play;
// -> 7, 4, 2, 0
```

Kodeboks 2.6: En sekvens med Pseq

Vi kan som det næste argument angive hvor mange gange sekvensen skal afspilles.

```
Pbind(
    \degree, Pseq([7, 4, 2, 0], 2),
).play;
// -> 7, 4, 2, 0, 7, 4, 2, 0
```

Kodeboks 2.7: Repetition med Pseq

I stedet for et bestemt antal gentagelser kan vi gentage uendeligt med nøgleordet **inf**.

```
~eksempel = Pbind(
    \degree, Pseq([7, 4, 2, 0], inf),
).play;
// -> 7, 4, 2, 0, 7, 4, 2, 0, 7, 4 ...
~eksempel.stop;
```

Kodeboks 2.8: Uendelig gentagelse med Pseq

Vi kan vælge at bruge **Pseq** til at styre flere forskellige parametre. Selvom den nederste **Pseq** herunder gentager sekvensen i det uendelige, slutter vores samlede sekvens af begivenheder, så snart den første Pseq i Pbind'en er færdig.

```
Pbind(
    \degree, Pseq([0, 1, 2, 7, 4, 3, 1, 2]),
    \dur, Pseq([0.25, 0.5, 0.25, 1], inf),
).play;
```

Kodeboks 2.9: Flere Pseqs på én gang

Det kan nogle gange være en god ide at bruge variabler til at fordele vores kode over flere linjer. Eksemplet herunder giver samme resultat som ovenfor men kan være lettere at overskue, fordi de enkelte linjer er mere simple.

```
~skalatrin = [0, 1, 2, 7, 4, 3, 1, 2];
~varigheder = [0.25, 0.5, 0.25, 1];
Pbind(
    \degree, Pseq(~skalatrin),
    \dur, Pseq(~varigheder, inf),
).play;
```

Kodeboks 2.10: Organisér koden med variabler

Elementerne i vores sekvenser kan være andre patterns, fx **Pwhite**, som vi så ovenfor. På den måde kan man blande variation og dynamik ind i sin komposition, men samtidig repetere og fastholde delelementer, så dele af sekvensen er genkendelig.

```
Pbind(
    \degree, Pseq([
        -7, 7,              // først et par faste toner
        Pwhite(-3, -1, 3), // derefter tre tilfældige, lidt dybe
        ↪ toner
        Pwhite(2, 4, 3),   // derefter tre tilfældige, lidt
        ↪ højere toner
    ], 2).trace,          // afspil hele sekvensen to gange (og
    ↪ vis outputtet med .trace)
    \dur, 0.5,
).play;
```

Kodeboks 2.11: Indlejring af Pwhite i Pseq

Indlejring af patterns på denne måde er en yderst nyttig kilde til mere interessante og subtile variationer. Vi ser nærmere på forskellige teknikker til indlejring af patterns i næste kapitel (se 3.1).

2.1.3 Pwhite - en tilfældighedsgenerator

Pwhite genererer tilfældige tal inden for et minimum og et maksimum.

```
(
~eksempel = Pbind(
    \degree, Pwhite(0, 4),
).play;
)
~eksempel.stop;
```

Kodeboks 2.12: Elementær brug af Pwhite

Modsat **Pseq** kører **Pwhite** som udgangspunkt uendeligt, men vi kan begrænse *antallet* af tilfældige tal med et yderligere argument.

```
Pbind(
    \degree, Pwhite(0, 4, 3),
).play;
```

Kodeboks 2.13: Pwhite med begrænset antal

Angiver vi decimaltal i stedet for heltal som øvre og/eller nedre grænse for **Pwhite**s output, får vi også decimaltal som resultat.

```
Pbind(\degree, Pwhite(0, 7, 4).trace).play;
Pbind(\degree, Pwhite(0.0, 7.0, 4).trace).play;
```

Kodeboks 2.14: Pwhite - decimaltal vs. heltal

Ligesom med **Pseq** kan vi bruge **Pwhite** til at styre en række forskellige andre parametre.

```
Pbind(
    // En fast sekvens af tonehøjder
    \degree, Pseq([0, 2, 4, 5], inf),

    // \db angiver lydstyrke, målt i decibel
    \db, Pwhite(-40, -20),

    // \dur angiver tonevarigheder, målt i antal taktslag
    \dur, Pwhite(0.1, 0.3),

    // \pan angiver panorering, hvor -1 er venstre og 1 er højre
    \pan, Pwhite(-1.0, 1.0, 16),
    // 16 toner
).play;
```

Kodeboks 2.15: Pwhite næsten over det hele

Med **Pwhite** er alle tal mellem minimum og maksimum lige sandsynlige. Dette er dog ikke altid den mest interessante statistiske fordeling - fx kan det være mere oplagt, at værdierne tættere på en bestemt grænse er mest sandsynlige, eller at udfaldene følger en såkaldt normalfordeling omkring en middelværdi. Det kan vi gøre med **Pexprand** og **Pgauss**.

```
Pbind(\legato, Pexprand(0.01, 1, 8).trace).play; // værdier tættere på minimum (0.01) er mest hyppige
Pbind(\legato, Pgauss(1, 0.1, 8).trace).play; // værdier tættest på middelværdi (1) er mest hyppige
```

Kodeboks 2.16: Alternativ sandsynlighedsfordeling med Pexprand og Pgauss

Det kan også være relevant at bevæge sig gradvist op eller ned med tilfældige spring, hvilket kan gøres med **Pbrown**.

```
Pbind(\degree, Pbrown(-7, 7, 1, 8).trace).play; // små spring
Pbind(\degree, Pbrown(-7, 7, 5, 8).trace).play; // større spring
```

Kodeboks 2.17: Trinvis tilfældighed med Pbrown

Vi kigger nærmere på disse forskellige tilfældighedsgenerator i næste afsnit (se 2.2).

2.1.4 Nyttige teknikker til at bearbejde output fra patterns

Når vi arbejder med patterns som kompositionsredskaber, er det typisk relevant at bearbejde outputtet fra patterns på forskellig vis. Det kan fx gøres med almindelige matematiske operationer.

```
Pbind(\degree, Pseq([0, 1, 2])).play;
// -> 0, 1, 2 - ingen transformation

Pbind(\degree, Pseq([0, 1, 2]) + 1).play;
// -> 1, 2, 3 - alle toner flyttes et skalatrin op

Pbind(\degree, Pseq([0, 1, 2]) * 2).play;
// -> 0, 2, 4 - dobbelt så store spring

Pbind(\degree, Pseq([0, 1, 2]) + Pseq([0, 7], inf)).play;
// -> 0, 3, 2 - hvert andet element flyttes en oktav op
```

Kodeboks 2.18: Matematiske operationer med outputtet fra patterns

Afrunding er også muligt med method'en `.round()`. For eksempel kan vi afrunde tilfældigt genererede tal mellem -12 og +12 til nærmeste tal i 3-tabellen og derved få en melodi, der kun springer i intervaller, som er opbygget af små tertser.

```
Pbind(
    \scale, Scale.chromatic,
    \degree, Pwhite(-12, 12).round(3),
).play;
```

Kodeboks 2.19: Afrunding med .round

Vi kan også bruge nogle specifikke pattern-methods til at gentage eller sammenklumpe outputtet fra patterns på forskellig vis. Kør nedenstående kode og gæt selv hvad `.repeat`, `.stutter` og `.clump` gør.

```
Pbind(\degree, Pseq([0, 1, 2]).repeat(3)).play;
// -> 0, 1, 2, 0, 1, 2, 0, 1, 2

Pbind(\degree, Pseq([0, 1, 2]).stutter(3)).play;
// -> 0, 0, 0, 1, 1, 1, 2, 2, 2

Pbind(\degree, Pseq([0, 1, 2]).clump(3)).play;
// [0, 1, 2], [0, 1, 2], [0, 1, 2]
```

Kodeboks 2.20: Pattern-methods: .repeat, .stutter og .clump 

2.2 Patterns som aleatoriske og stokastiske redskaber

I midten af det 20. århundrede inkorporerede vigtige strømninger i kompositionsmusikken statistik og tilfældighed i deres kompositionsteknikker. Musikken gik under betegnelser som *aleatorik*, *stokastisk musik*, *indeterminacy* og *chance music*. Ordet *alea* er latin for terning, og man bruger jo som bekendt terninger til at skabe tilfældige udfald inden for et nøje defineret udfaldsrum. Aleatorik er som musikalsk tradition netop baseret på inkorporering af tilfældighed i kompositionsarbejdet.

Men hvad er egentlig tilfældighed? Man kan måske fra et psykologisk perspektiv lidt pragmatisk sige, at det tilfældige er det, vi opfatter som værende uorganiseret, eller hvor vi ikke kan finde nogen mønstre eller repetition i det, vi observerer. I matematikken og statistikken arbejder man med en lidt anden forståelse, hvor der i stedet er tale om *sandsynligheden* for, at bestemte hændelser sker. Det er denne tanke, som ligger bag ideen om stokastisk komposition, hvor de tilfældige udfald ikke er helt uforudsigelige men kan beskrives med statistiske redskaber.

I SuperCollider findes der mange forskellige patterns, som genererer tilfældige værdier. Herunder sammenlignes de mest almindelige af disse, idet der også gives eksempler på hvornår de forskellige redskaber meningsfuldt kan anvendes. Dog er der ikke nogen endegyldig facitliste for hvilke patterns man skal bruge i hvilke situationer, da de kan anvendes og kombineres frit.

Man kan groft skelne mellem to typer af patterns, som inkorporerer tilfældighed:

Tilfældighedsgeneratorer
: Genererer ligesom **`Pwhite`** værdier ud fra 2-3 parametre såsom øvre/nedre grænser eller grænser for spring, middelværdier eller spredning.

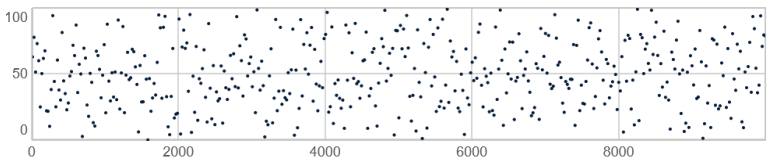
Listebaserede generatorer
: Genererer ligesom **`Pseq`** output baseret på specificerede lister af værdier (eller patterns).

2.2.1 Tilfældighedsbaserede patterns

Blandt de af SuperColliders indbyggede patterns, som på forskellig vis genererer tilfældige værdier, vil jeg fremhæve følgende som et nyttigt udvalg at kende til. De præsenteres herunder med deres standardargumenter (som vist i SuperColliders dokumentation).

`Pwhite``(min, max, antal)`
: Genererer tilfældige tal mellem et minimum (`lo`) og et maksimum (`hi`). Ken-

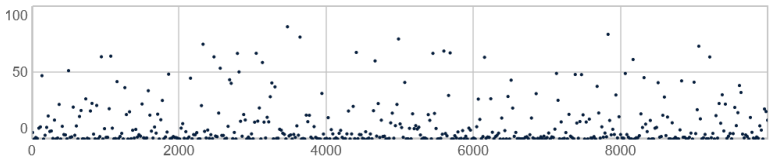
detegnet for Pwhite er, at alle tal mellem disse to grænser er lige sandsynlige. Anvendes typisk når man ønsker „helt“ tilfældige værdier med mulighed for store og små spring, fx ved atonal eller arytmisk komposition. I `Pwhite(0, 10, 5)` er 0 den nedre grænse, 10 den øvre grænse, og 5 antallet af genererede værdier.



Figur 2.1: 10.000 værdier genereret med Pwhite(0, 100, 10000)

`Pexprand``(min, max, antal)`

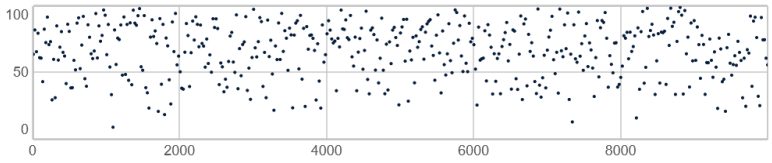
Tal tættest på den nedre grænse er mest sandsynlige, da sandsynligheden svarer til en eksponentialfordeling. Pexprand anvendes typisk hvor man ønsker en Pwhite-lignende fordeling, men inden for fx frekvens eller lydstyrke. Disse parametre skal følge en eksponentiel fordeling i stedet for en lineær fordeling, hvis vi skal tilnærme os, hvordan de perciperes af en menneskelig lytter.



Figur 2.2: 10.000 værdier genereret med Pexprand(0.01, 100, 10000)

`Phprand``(min, max, antal)`

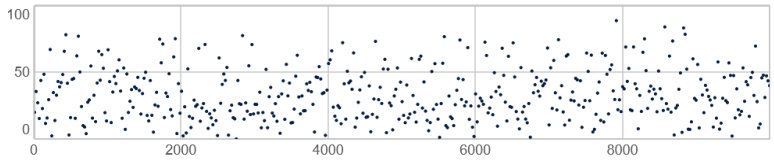
Tal tættest på en øvre grænse er mest sandsynlige.



Figur 2.3: 10.000 værdier genereret med Phprand(0, 100, 10000)

`Plprand``(min, max, antal)`

Tal tættest på en nedre grænse er mest sandsynlige. Anvendes, hvor man kun ønsker få værdier tæt på en øvre grænse.

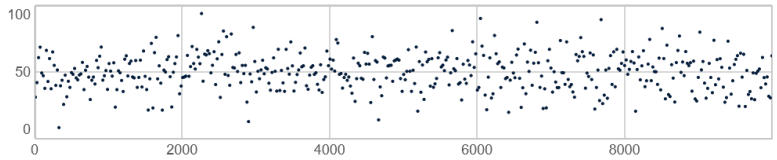


Figur 2.4: 10.000 værdier genereret med Plprand(0, 100, 10000)

Pgauss(middelværdi, standardafvigelse, antal)

Tal tæt på en middelværdi er mere sandsynlige end tal, der ligger længere væk. Baseret på det, man kalder normalfordelingen/Gaussfordelingen. Vi bruger fx Pgauss hvis vi ønsker, at de fleste værdier skal ligge i omegnen af et centrum, fx midt i stereobilledet eller en skala, men hvor der kan være nogle få vildskud.

Med **Pgauss**(10, 3, 5) er 10 middelværdien, 3 er standardafvigelsen, og 5 er antallet af genererede værdier.

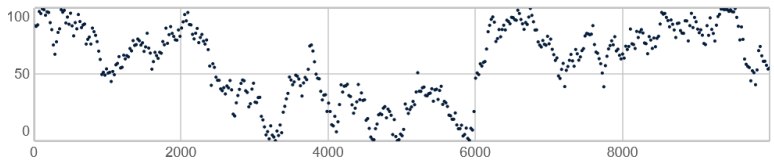


Figur 2.5: 10.000 værdier genereret med Pgauss(50, 17, 10000)

Pbrown(min, max, maksimalt trin, antal)

Genererer ligesom Pwhite tilfældige værdier mellem et minimum og et maksimum, men med en begrænset afstand (trin_max) mellem to på hinanden følgende værdier. Algoritmen kendes også som en random walk/drunkard's walk. Pbrown bruges, når man ønsker en gradvis udvikling i en strøm af tilfældigt valgte værdier, dvs. hvor springene mellem de enkelte skridt er begrænsede. Det kunne fx være ved trinbevægelser i melodier eller ved en cutoff-frekvens, som skal bevæge sig mere organisk mellem forskellige værdier.

I **Pbrown**(0, 10, 3, 5) er 0 den nedre grænse, 10 den øvre grænse, 3 det maksimale spring fra den ene værdi til den næste, og 5 antallet af genererede værdier.



Figur 2.6: 10.000 værdier genereret med Pbrown(0, 100, 2, 10000)

2.2.2 Listebaserede, stokastiske patterns

Disse patterns anvender vi, hvis vi gerne vil definere en række mulige udfald, men vil lade algoritmen bestemme hvordan der vælges mellem valgmulighederne eller hvilken rækkefølge en foruddefineret sekvens afspilles i. Genopfrisk gerne emnet lister (se 1.6).

Prand`(liste, antal)`
: Vælger et bestemt `antal` tilfældige tal fra en given `liste`. Anvendes, når man blot ønsker, at algoritmen vælger blandt en liste med givne valgmuligheder.

Pxrand`(liste, antal)`
: Fungerer præcis ligesom **Prand**, bortset fra, at samme element ikke vælges to gange i træk. Anvendes, hvis vi fx ikke ønsker tone- eller rytmegentagelser.

Pwrand`(liste, antal, sandsynligheder)`
: Vælger ligesom de to foregående patterns tilfældige elementer fra en given liste, men med forskellige sansyndligheder for disse valg. Anvendes, hvis nogle valgmuligheder skal forekomme oftere end andre. Det kunne fx være rytmiske variationer, som er mindre hyppige end mere almindelige grooves. Bemærk, at tallene i listen med sandsynligheder sammenlagt skal give 1, hvilket kan gøres let med array-method'en `.normalizeSum`.

Pshuf`(liste, antal)`
: Fungerer ligesom den almindelige **Pseq**, bortset fra, at den givne liste afspilles i en tilfældig rækkefølge. Kan være nyttigt, hvor man ønsker at gentage en sekvens, selvom selve sekvensen er sat i tilfældig rækkefølge - det kan give en fornemmelse af regelmæssighed, selvom rækkefølgen er ny. Bemærk, at antallet af antagelser her er antallet af gentagelser af hele listen (modsat fx **Prand** eller **Pseries**, hvor antallet angiver antal enkeltelementer).

Der findes andre relevante listebaserede patterns som **Place**, men disse er lidt mere komplicerede og gennemgås derfor først senere (se 3.1.2).

2.3 Tonehøjde, rytmik og dynamik i Pbind

I SuperCollider kan man sammensætte længere forløb af musikalske „begivenheder“ ved hjælp af **Pbind**. Det er ofte inden for Pbind, at vi specificerer musikalske parametre som tonehøjde, dynamik, rytmik og frasering. Det gør vi ved at sammensætte nøgler og værdier i formen `\nøgle, værdi`.

```
1 Pbind(
2     \degree, 4, // 5. skalatrin, et g (fordi udgangspunktet er
      ↪  C-dur)
3     \dur, 0.5,  // varighed på et halvt taktslag (dvs. en
      ↪  ottendedel i almindelig musikterminologi)
4     \db, -25,   // -25 decibel lydstyrke
5 ).play;
```

Kodeboks 2.21: Nøgler og værdier i Pbind

Nøglerne i Pbind'en ovenfor her er altså de tekstbidder, der starter med \ - `\degree`, `\dur` eller `\db`. Værdierne som knyttes til disse nøgler er henholdvis `4`, `0.5` og `-25`.

Herunder gennemgås de mest almindelige standardnøgler, som er nyttige at kende til. Det skal med henblik på de senere kapitler i denne bog bemærkes, at vi kun kan bruge disse standardnøgler sammen med de klangdannelsesopskrifter, som kaldes **SynthDef**s, hvis vi indretter vores SynthDef på en bestemt måde. Nøglerne der nævnes herunder fungerer kun, når man anvender argumenterne `freq`, `amp` og `gate` korrekt i sin SynthDef. Se hertil afsnittet om argumentnavne i SynthDef (se 5.3.5).

2.3.1 Tonehøjde i Pbind

SuperCollider kan automatisk udregne hvilken frekvens, vores toner skal klinge med, hvis vi i Pbind angiver tonehøjde på en genkendelig måde, dvs. ved hjælp af en række konventionelle nøgler[1]. Til at notere tonehøjder anbefaler jeg, at man anvender en kombination af følgende nøgler:

`\degree`
: Angiver *skalatrin*, hvor 0 er det første trin (fx tonen c i en C-dur skala).

[1] Teknisk set er det ikke Pbind, som man måske skulle tro, men i stedet SuperColliders såkaldte *default Event*, som foretager omregningen fra de nøgler og værdier, vi angiver i Pbind, til tekniske parametre som frekvens. Nysgerrige læsere henvises til SuperColliders dokumentation for yderligere uddybning.

`\scale`
: Angiver *skala*, valgt med fx `Scale.minor` eller angivet som en liste (se 1.6) bestående af intervaller. `Scale.major` er standardværdien. For skalavalgmuligheder, kør `Scale.directory;`

`\octave`
: Angiver hvilken MIDI-oktav, tonen findes i.

`\root`
: Skalaens grundtone, målt i antal halvtoners afstand fra c. Dvs. 0 er c, 1 er cis, 2 er d osv. Dette kan også være negative tal, dvs. -3 svarer til a.

```
1  Pbind(
2      \degree, [0, 2, 4, 6],    // en diatonisk firklang
3      \scale, Scale.minor,      // mol-skala
4      \octave, 5,               // oktav 5 er den oktav, som
       ↪  starter ved nøglehuls-c
5      \root, -3                 // 3 halvtoner under nøglehuls-c (i
       ↪  dette tilfælde a)
6  ).play;                       // tilsammen en A-mol7
```

Kodeboks 2.22: Skalatrin, skala, oktav og grundtone

Man kan også vælge at angive tonehøjde på andre abstraktionsniveauer - med MIDI-tal (nøglen `\midinote`) eller oscillatorfrekvens (nøglen `\freq`).

```
1  Pbind(\midinote, 60).play;  // c
2
3  Pbind(\freq, 440).play;     // a, kammertonen
```

Kodeboks 2.23: Alternative nøgler til angivelse af tonehøjde

Vi kan angive transponering og parallelforskydning inden/uden for skala med:

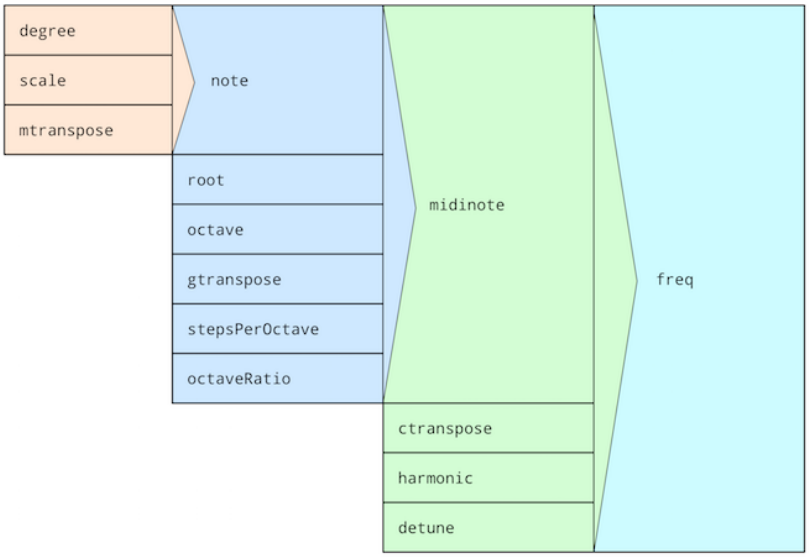
`\mtranspose`
: Modal transponering, dvs. parallelføring inden for skalaen. Måles i antal skalatrin.

`\ctranspose`
: Kromatisk transponering, dvs. uafhængigt af skala. Måles i antal halvtonetrin.

```
Pbind(
    \degree, [0, 2, 4],
    \mtranspose, Pseq([0, 1, 2, 3]),
).play;

Pbind(
    \degree, [0, 2, 4],
    \ctranspose, Pseq([0, 1, 2, 3]),
).play;
```

Kodeboks 2.24: Modal og kromatisk transponering

Til disse kunne man måske indvende, at `\root` og `\ctranspose` giver samme resultat, så hvorfor have to forskellige nøgler, der gør det samme? Når begge nøgler sameksisterer, er det fordi det giver os mulighed for at definere en grundtone med `\root` og en (midlertidig) bevægelse væk fra denne grundtone med `\ctranspose`. Selvom der ikke er nogen funktionel forskel, er der en semantisk forskel for komponisten, som kan være meget væsentlig. Dette naturligvis sammenholdt med, at ingen er tvunget til at anvende disse nøgler til specificering af tonehøjde. Man kan blot angive miditone-tal med `\midinote` eller oscillatorfrekvens med `\freq`, hvis man ønsker at arbejde på det abstraktionsniveau. Man kan endda anvende en række andre nøgler, som for enkelhedens skyld er udeladt her. De kan ses på nedenstående graf, som stammer fra SuperColliders dokumentation (SuperCollider 3 documentation contributors, u.d.).



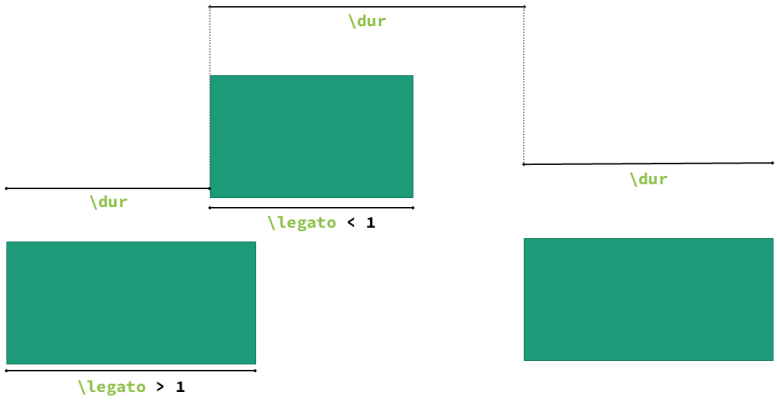
Figur 2.7: Pbind-nøgler til udregning af oscillatorfrekvens. Licens: CC BY-SA 3.0. Kilde: SuperCollider 3 documentation contributors, https://doc.sccode.org/Classes/Event.html

2.3.2 Varighed, frasering og timing

Til at notere rytmiske forhold bruger vi primært følgende nøgler:

`\dur`
: Tidsinterval mellem på hinanden følgende anslag, målt i taktslag (ikke sekunder).

`\legato`
: Hvor længe en tone klinger, målt relativt i forhold til `\dur`. Værdier større end 1 giver legato og værdier mindre end 1 giver staccato.



Figur 2.8: Sammenhængen mellem \dur og \legato i Pbind

```
1  // tidsinterval 1 taktslag, dvs. en 4.-del
2  Pbind(\dur, 1).play;
3
4  // tidsinterval 0.25 taktslag, dvs. en 16.-del
5  Pbind(\dur, 0.25).play;
6
7  // staccato
8  Pbind(\legato, 0.1).play;
9  // legato
10 Pbind(\legato, 1.5).play;
```

Kodeboks 2.25: Rytmik og frasering med dur og legato

For at justere timing relativt til den underliggende puls kan man bruge nøglen `\lag`, hvilket fx kan bruges til at „humanisere“ en ellers meget maskinel timing.

```
(
~almindelig = Pbind().play;
~forsinket = Pbind(
    // højere tone, så vi kan høre forskel
    \degree, 4,
    // skiftevis lidt forsinket og lidt tidlig
    \lag, Pseq([0.1, -0.1], inf)
).play;
)
~almindelig.stop; ~forsinket.stop;
```

Kodeboks 2.26: Rytmisk forskydning med lag

For at forskyde timingen af den enkelte tone relativt til andre ellers samtidigt klingende toner kan vi bruge `\strum`-nøglen, som forskyder toneanslag i akkorder ligesom ved et guitar-strum, hvor der kan være et kort tidsinterval mellem anslaget af de forskellige strenge. For at få denne effekt, skal vi notere en liste eller et pattern, som genererer en liste (fx med method'en `.clump`).

```
Pbind(
    \degree, [0, 2, 4, 6],
    \strum, 0.05
).play;
```

Kodeboks 2.27: Strumming med strum

2.3.2.1 Tempoangivelse med TempoClock

For at angive tempo kunne vi sådan set blot justere på de værdier, vi anvender til `\dur`-nøglen. Dobbelt så lange varigheder halverer tempoet, og halvt så lange varigheder fordobler tempoet. Men det ville betyde, at vi skulle ændre på alle vores Pbinds, hvis vi ville sætte tempoet bare en smule op eller ned, og det ville ikke være praktisk. Heldigvis er der en bedre måde at styre tempoet på, nemlig ved hjælp af klassen **`TempoClock`**.

`TempoClock` måler tempoet i *taktslag pr. sekund* (BPS). Dette står i modsætning til mange andre former for musiksoftware, hvor tempoet måles i *taktslag pr. minut* (BPM). Spørger vi **`TempoClock`** om den aktuelle tempoindstilling med method'en `.tempo`, vil vi derfor få en lidt usædvanlig tempoangivelse retur, nemlig tallet `1.0` (medmindre vi har ændret tempoet). Det betyder ét taktslag pr. sekund, hvilket svarer til 60 taktslag pr. minut. Hvis vi vil fordoble tempoet, kan vi dermed sætte det til `2`, hvilket svarer til 120 BPM.

```
TempoClock.tempo;
// -> 1.0

TempoClock.tempo = 2;
TempoClock.tempo;
// -> 2.0
```

Kodeboks 2.28: Tempo med TempoClock

Men hvad så, hvis vi gerne vil sætte tempoet til fx 87,3 BPM? Det kan vi let gøre ved at omregne fra BPM til BPS med en simpel formel: $BPS = BPM/60$

```
// Tempo sættes til 87,3 BPM
TempoClock.tempo = 87.3 / 60;

TempoClock.tempo; // Vis det aktuelle tempo, målt i BPS
// -> 1.455
```

Kodeboks 2.29: Tempoangivelse i BPM

Hvis vi *midlertidigt* vil skifte tempo, kan vi godt ændre på `TempoClock.tempo`, mens vores Pbind spiller. Men det kan her være nyttigt at bruge patterns til at sekvensere skiftet, så det sker præcist der, hvor vi ønsker det. Hvis vi vil skalere tidsintervallerne angivet med `\dur` uden at ændre disse direkte, kan vi tilføje nøglen `\stretch`, der som navnet antyder strækker eller skalerer varighederne med den faktor, vi angiver. Her er det altså ikke en direkte tempoangivelse som med `TempoClock.tempo`, men en relativ angivelse, som tager udgangspunkt i det eksisterende tempo.

```
Pbind(\dur, 1, \stretch, 1).play; // normalt tempo
Pbind(\dur, 1, \stretch, 2).play; // dobbelt tempo
Pbind(\dur, 1, \stretch, 3).play; // tredobbelt tempo
Pbind(\dur, 1, \stretch, 0.5).play; // halvt tempo
```

Kodeboks 2.30: Tempovariation med stretch

2.3.2.2 Traditionel rytmenotation og \dur-nøglen

For at notere nodeværdier som vi kender dem fra almindelig musikteori, skal vi tage højde for, at SuperCollider tæller varigheder i *taktslag*. Når vi almindeligvis taler om et „taktslag“, taler vi typisk om en fjerdedel (1/4). Men en fjerdedel af hvad? Rent

teknisk taler vi om en *fjerdedel af en (fire fjerdedels)takt*. I SuperCollider tæller vi med `\dur`-nøglen ikke i takter, men i *taktslag*. Derfor skriver vi 1 og ikke 1/4, når vi ønsker en varighed på ét taktslag.

Hvad nu, hvis vi gerne vil bruge almindelige rytmeangivelser i SuperCollider? Jo, vi kan for det meste ganske enkelt gange med 4 for at omregne fra „varighed målt i andele af takt“ til „varighed målt i taktslag“. For at notere sekstendedele, ottendedele, fjerdele, halvnoder og helnoder kan vi således skrive den ønskede nodeværdi ganget med 4.

```
1 Pbind(\dur, 1/16 * 4).play;
2 Pbind(\dur, 1/8 * 4).play;
3 Pbind(\dur, 1/4 * 4).play;
4 Pbind(\dur, 1/2 * 4).play;
5 Pbind(\dur, 1/1 * 4).play;
```

Kodeboks 2.31: Notation af traditionelle nodeværdier i Pbind 

2.3.3 Lydstyrke

Til at notere volumen kan man vælge mellem nøglerne `\db` eller `\amp`. Med `\db` omregner SuperCollider automatisk fra decibel, hvor `0` er den maksimale værdi. Med nøglen `\amp` angiver vi i stedet amplituden direkte, typisk som en værdi mellem 0 og 1.

```
1 Pbind(\db, -30).play;
2 Pbind(\amp, 0.2).play;
```

Kodeboks 2.32: Lydstyrke med amp og db 

2.4 Cheat sheet: Patterns

2.4.1 13 primære patterns

```
// Pseq - en fleksibel sequencer
Pbind(\degree, Pseq([0, -3, 1], 2)).play;

// Pser - endnu en fleksibel sequencer, med begrænsning af antal events
Pbind(\degree, Pser([0, -3, 1], 4)).play;

// Prand - vælger tilfældige elementer fra en liste
Pbind(\degree, Prand([0, 1, 2], inf)).play;

// Pxrand - vælger tilfældige elementer fra en liste, dog uden gentagelser
Pbind(\degree, Pxrand([0, 3, 4, 7], inf)).play;

// Pwrand - vælger tilfældige elementer fra en liste, med vægtede sandsynligheder
Pbind(\degree, Pwrand([0, -1, 1, 4], [0.6, 0.1, 0.1, 0.2], inf), \dur, 0.5).play;

// Pshuf - en fætter/kusine til Pseq, gentager en sekvens i tilfældig rækkefølge
Pbind(\degree, Pshuf([0, 1, 4, 6], 2), \dur, 0.5).play;
```

Kodeboks 2.33: 6 listebaserede patterns

```
// Pwhite - tilfældige tal, ligeligt fordelt mellem min og max
Pbind(\degree, Pwhite(-7, 7)).play;

// Pexprand - tilfældige tal, eksponentielt fordelt mellem min
↪  og max
Pbind(\freq, Pexprand(400, 800)).play;

// Pgauss - tilfældige tal, normalfordelt omkring en middelværdi
Pbind(\pan, Pgauss(0, 0.5)).play; // brug evt. hovedtelefoner
↪  for at høre panorering

// Pbrown - en "fordrukken" stifinder
Pbind(\degree, Pbrown(-7, 7, 2), \dur, 0.2).play;

// Pseries - en trinvis udvikling med startværdi, interval og
↪  antal
Pbind(\degree, Pseries(7, -2, 8)).play;

// Pgeom - gradvis udvikling med startværdi, vækstfaktor og
↪  antal
Pbind(\dur, Pgeom(1, 0.9, 8)).play;
```

Kodeboks 2.34: 5 tilfældighedsgeneratorer 

```
// Pfunc - når vi vil bruge en funktion til at udregne værdier
Pbind(\freq, Pfunc({ 220 + (220 * [0, 1.5, 2, 2.5, 3].choose)
↪  }), \dur, 0.2).play;
```

Kodeboks 2.35: 1 catch-all pattern 

2.4.2 3 vigtige Pattern-methods

```
// .repeat - gentager hele sekvenser
Pbind(\degree, Pshuf([0, 1, 3, 4], 2).repeat(2), \dur,
↪  0.5).play; // bemærk ændring efter 1. gentagelse

// .stutter - gentager de enkelte genererede værdier
Pbind(\degree, Pwhite(0, 9).stutter(3), \dur, 0.5).play;

// .clump - samler enkelte værdier til "akkorder"
Pbind(\degree, Pshuf([0, 1, 2, 4, 6, 8, 9], inf).clump(2)).play;
```

Kodeboks 2.36: 3 vigtige Pattern-methods 

2.4.3 5 nyttige meta-Patterns

```
// Pbind - knytter nøgler og patterns sammen i strømme af begivenheder (Events)
Pbind(\degree, Pseq([4, 2, 0])).play;

// Pmono og PmonoArtic - afspiller en kontinuerlig tone, skifter løbende mellem værdier
Pmono(\default, \degree, Pwhite(-10, 10), \dur, Pexprand(0.1, 0.5)).play;

// Pdef - yderst handy til synkronisering og dynamisk udskiftning af patterns
Pdef(\melodi, Pbind(\degree, Pshuf([0, 1, 3, 4], inf), \dur, 0.5)).play; // kør linjen flere gange

// Pfin - begrænser antallet af events fra en Pbind
Pfin(3, Pbind(\degree, Pwhite(0, 7), \dur, Pwhite(2, 4))).play;

// Pfindur - begrænser varigheden af en Pbind
Pfindur(3, Pbind(\degree, Pwhite(0, 7), \dur, Prand([0.25, 0.5, 0.125], inf))).play;
```

Kodeboks 2.37: 5 nyttige meta-Patterns

2.5 Øvelse: Grundlæggende brug af Patterns

Med denne øvelse får du grundlæggende erfaring med brug af patterns. Opgaverne er simple i forhold til potentialet i patterns, men en god forståelse af de grundlæggende forhold er afgørende for at man kan arbejde med de mere komplicerede teknikker senere hen.

2.5.1 Find fire fejl

Find og ret fejlene i de fire eksempler herunder.

1. Læs fejlmeddelelsen, før du retter fejlen.
2. Forklar i en kommentar for hvert eksempel hvad du har rettet og hvad problemet bestod i.
3. Ryd SuperColliders post window (Ctrl/Cmd-Shift-P) inden du starter med et nyt eksempel.

```
Pbind(
    Pwhite(0, 7), \degree,
).play;

Pbind(
    \degree, Pseq(3, 7),
).play;

Pbind(
    \degree, Pwhite(0, 7);
    \dur, 0.5;
).play;

Pbind(
    \dur, 0.25,
    \octave, Pseq([3. 4. 5]),
).play;
```

Kodeboks 2.38: Find fire fejl

2.5.2 Skabelon til opgaverne herunder

I opgaverne herunder skal du skrive dine egne Pbind-kompositioner. Du kan kopiere nedenstående skabelon for at gå i gang.

```
Pbind(

).play;
```

Kodeboks 2.39: Skabelon til øvelser i patterns

2.5.3 Pbind og tonehøjde

Denne opgave fokuserer på brug af nøgler til angivelse af tonehøjde i Pbind. Der skal ikke anvendes patterns (ud over Pbind).

Spil følgende toner i uendelig gentagelse:

1. Tonen c (brug `\degree`).
2. Tonen d (brug `\degree` eller `\root`).
3. Tonen d ved oktav 3 (brug `\degree` og `\octave`).
4. Tredje trin på en c-mol skala (brug `\scale` og `\degree`).
5. En A-dur-akkord (brug `\root` og `\degree`).
6. En f-mol-akkord (vælg selv passende nøgler).

2.5.4 Rytmik i Pbind

Denne opgave fokuserer på brug af nøgler til angivelse af rytmik og frasering i Pbind samt tempoangivelse med **`TempoClock`**. Der skal ikke anvendes patterns (ud over Pbind).

Spil tonen c i uendelig gentagelse med følgende rytmik og frasering:

1. Fjerdedele (én tone pr. taktslag).
2. Ottendedele (to toner pr. taktslag).
3. Sekstendedele (4 toner pr. taktslag).
4. Ottendedele med legato-frasering.
5. Ottendedele med staccato-frasering.
6. Fjerdedele ved 40 BPM.
7. Ottendedele ved 150 BPM.

2.5.5 Sekvenser med Pseq

Denne opgave fokuserer på brug af **`Pseq`** til at angive sekvenser.

Spil følgende ved hjælp af **Pseq**:

1. Første frase i melodien til 'Mester Jakob' (brug nøglen `\degree`).
2. Første frase i melodien til 'Mester Jakob' i D#-dur (brug `\root` til at angive grundtonen).
3. Første frase i melodien til 'Mester Jakob', spillet i en frygisk skala i stedet for en dur-skala (brug `\scale`).
4. Et c, der veksler mellem oktav 3 og 4.
5. En akkordbrydning (Dm7).
6. En selvkomponeret rytme, som indeholder ottendele, fjerdedele og halvnoder (brug `\dur`).
7. En sekvens, hvor alle toner spilles legato på nær hver 4. tone i sekvensen, som spilles staccato (brug nøglen `\legato`).

2.5.6 Tilfældighed med Pwhite

Afspil følgende ved hjælp af **Pwhite**:

1. 10 toner, valgt tilfældigt inden for en C-dur-skala.
2. Tilfældige frekvenser mellem 500 og 1000 hz (brug nøglen `\freq`).
3. Spil en uendelig række af akkordbrydninger med uregelmæssig rytmik og frasering.
4. Spil tilfældige skalatrin inden for en F-mol-skala, hvor alle toner gentages én gang (brug `.stutter`).

2.5.7 Tilfældighed med lister

1. Afspil med **Prand** 10 tilfældigt valgte elementer fra listen `~skalatrin`.
2. Afspil med **Pshuf** listen `~skalatrin` 2 gange i en tilfældig rækkefølge.
3. Afspil listen `~skalatrin` i en tilfældig rækkefølge 4 gange, gentag herefter dette med en ny tilfældig rækkefølge (dette kan gøres ved at kombinere **Pshuf** og `.repeat`).

```
1 ~skalatrin = [-2, 0, 1, 3, 4, 6];
```

Kodeboks 2.40: En række skalatrin

2.6 Øvelse: Generativ komposition

Denne øvelse handler om at bruge patterns til generativ komposition.

2.6.1 Aleatorik ud over det hele

Skriv en komposition med **Pbind**, hvor alle parametre genereres tilfældigt, dvs. de faste værdier skal erstattes med patterns, der genererer tilfældige værdier (se 2.2).

```
Pbind(
    // Tonehøjde
    \degree, 0,
    \octave, 4,
    \mtranspose, 0,

    \dur, 1,
    \legato, 1,

    \db, -20,
).play;
```

Kodeboks 2.41: Aleatorik ud over det hele

2.6.2 Sekvens og generativitet

1. Erstat teksten **Lunken** `kaffe` herunder med et udtryk efter eget valg og kør derefter den første kodeblok. Du har nu den `~sekvens`, som skal være grundlag for din komposition.
2. Skriv ved hjælp af **Pbind** og listebaserede patterns som **Pseq**, **Pshuf**, **Prand**, **Pxrand**, **Pwrand** og **Pser** en generativ komposition, der lever op til følgende krav:
   a) Kompositoriske parametre (tonehøjde, rytmik/frasering, lydstyrke, panorering osv.) skal så vidt muligt baseres på listen, som er gemt under variablen `~sekvens`.
   b) Forsøg at tilstræbe en balance mellem gentagelse og variation. Hertil kan det være en god ide at bruge pattern-methods som `.stutter`, `.repeat` og `.clump`.

```
~sekvens = "Lunken kaffe".ascii % 10;
~sekvens.postln;

Pbind(

).play;
```

Kodeboks 2.42: Sekvens og generativitet

KAPITEL 3

Komposition med patterns

Dette kapitel introducerer til nogle mere avancerede teknikker inden for pattern-baseret komposition. Dernæst ser vi på, hvordan man indlejrer patterns som input til andre patterns, altså en form for „pattern-inception“. Vi kigger også på, hvordan man kan sammensætte patterns og skabe variationer over de mønstre, vi definerer med `Pbind`. Derefter introduceres brugen af SuperColliders patterns til komposition via MIDI-output, således at vi i stedet for SuperColliders lydserver bruger en ekstern synthesizer eller sampler som lydgenerator til at skabe mere interessant klingende kompositioner. Med disse mere avancerede anvendelser af patterns kan vi skabe komplekse og varierede kompositionsmønstre. Som det sidste introducerende materiale i kapitlet indgår nogle tips og tricks, som kan gøre arbejdet med (at lære) patterns lettere og sjovere. Til sidst ser vi i en øvelse på, hvordan man i sammenhæng kan bruge de introducerede teknikker til at arbejde med minimalistisk inspireret komposition.

3.1 Indlejrede patterns

Det er relativt let at lave en simpel, algoritmisk komposition ved hjælp af patterns. Men det kan være mere vanskeligt at bevæge sig videre fra det meget simple eller meget tilfældighedsprægede udtryk. Her kan såkaldt indlejring af patterns være med til at give et mere nuanceret og subtilt udtryk.

Generativ eller algoritmisk komposition indebærer, at man i et vist omfang overlader dele af det kompositoriske arbejde til et system eller en algoritme. Her spiller tilfældighedsgeneratorer (se 2.2) ofte en betydelig rolle. Total tilfældighed er imidlertid sjældent specielt interessant. Derfor kan man med fordel indlejre tilfældighed som et begrænset element i en ellers fastlagt struktur, eller filtrere/afgrænse/gentage tilfældigt genererede data, så der skabes orden ud af en ellers kaotisk strøm af output.

3.1.1 Sekvenser af patterns

Vi har tidligere set, hvordan vi kan generere sekvenser af værdier (se 2.1.2). Men **Pseq** er fleksibel og kan lige så vel bruges til sekvenser bestående af patterns. Det betyder, at vi som elementer i vores sekvens kan angive patterns i stedet for værdier. Når **Pseq** når til et pattern, gennemløber den nemlig alle de værdier, som det pågældende pattern genererer, før den går videre. Herunder ses eksempelvis en sekvens med en blanding af faste og tilfældigt genererede skalatrin.

```
Pbind(
    \degree, Pseq([
        // først en fast starttone, c
        0,
        // derefter to tilfældige toner
        Pwhite(0, 7, 2),
        // derefter g og h i tilfældig rækkefølge
        Pshuf([-1, -3]),
        // og til sidst et dybt g
        -3
    // sekvensen forløber tre gange
    ], 3),
).play;
```

Kodeboks 3.1: Patterns som undersekvenser

For overskuelighedens skyld kan vi opnå præcis det samme som ovenfor med variabler til de enkelte underpatterns. Nedenstående er umiddelbart lettere at læse.

```
// To Tilfældige toner
~fritValg = Pwhite(0, 7, 2);
// g og h i tilfældig rækkefølge
~dybBlanding = Pshuf([-1, -3]);

Pbind(
    \degree, Pseq([0, ~fritValg, ~dybBlanding, -3], 3),
).play;
```

Kodeboks 3.2: Omskrivning med variabler 

3.1.2 Sammenflettede sekvenser og patterns

Place er et særligt pattern, som er velegnet til at flette sekvenser sammen. Her sætter man to eller flere sekvenser eller enkeltværdier sammen i en liste, og **Place** veksler så mellem de forskellige kilder. Listen gennemløbes det antal gange, man angiver (herunder 4 gennemløb).

```
(
Pbind(
    \degree, Place([
        [4, 3, 5, 4],
        [2, 1],
        -3,
        0
    ], 4).trace
).play;
)
// -> 4, 2, -3, 0, 3, 1, -3, 0, 5, 2, -3, 0, 4, 1, -3, 0
```

Kodeboks 3.3: Sammenflettede sekvenser med Place 

Der findes også en variant, som er endnu mere relevant i forhold til sammensætning af patterns, fordi den tillader, at man erstatter listerne i **Place** med patterns. **Patlace**, som denne variant hedder, er meget oplagt som ramme for indlejrede patterns og filtreret tilfældighed.

```
Pbind(
    \degree, Ppatlace([
        Pshuf([2, 3, 4, -1], inf),
        Pwhite(4, 7).stutter(4),
    ], inf).trace,
    \dur, 0.25
).play;
```

Kodeboks 3.4: Sammenflettede patterns med Ppatlace

3.1.3 Patterns som variererende argumenter

Man kan skabe interessante variationer ved at erstatte de faste værdier, vi ofte angiver som argumenter til patterns, med noget, som varierer. Som eksempel kan vi tage **Pseries**, der normalt giver lineære sekvenser ud fra tre argumenter - en startværdi, en trinstørrelse, og et antal.

```
Pseries(0, 1, 4)
// -> 0, 1, 2, 3
Pseries(6, -2, 5)
// -> 6, 4, 2, 0, -2
Pseries(5, 0.5, inf)
// -> 5, 5.5, 6, 6.5, 7, 7.5 ...
```

Kodeboks 3.5: Pseries med faste argumenter

Hvis vi ønsker at varierere trinstørrelsen i **Pseries**, kan vi gøre det ved at *indlejre* noget, der varierer - fx et andet pattern! Det kræver, at vi konverterer det indlejrede pattern til en *stream*. Det lyder måske lidt teknisk, men bare rolig, streams er meget nært beslægtede med patterns - en stream er blot et objekt, der kan levere en strøm af værdier. Når patterns skal generere værdier, bliver de faktisk automatisk konverteret til streams - det sker blot under motorhjelmen. Det kan vi heldigvis også gøre manuelt med method'en `.asStream`.

```
// Først gemmer vi en Pseq som en stream
~stream = Pseq([1, 2, 3], inf).asStream;

// Derefter kan vi bede om en ny værdi fra strømmen gang efter gang med .next-method'en
~stream.next.postln;
// -> 1, 2, 3, 1, 2, 3, 1, 2, 3 ...
```

Kodeboks 3.6: Pattern konverteret til stream

Det vil føre for vidt at uddybe den tekniske forskel mellem patterns og streams yderligere[1], men for nuværende kan du blot huske på, at indlejrede patterns, der anvendes som argumenter til andre patterns, ofte skal konverteres til streams med den handy method `.asStream`.

For at vende tilbage til vores **`Pseries`** ovenfor: Hvis vi ønsker at variere trinstørrelsen fra tone til tone, kan vi eksempelvis vælge et tilfældigt tal mellem -1 og 1 med en indlejret **`Pwhite`**.

```
Pseries(0, Pwhite(-1, 1).asStream, 10);
// -> 0, -2, -1, 0, 1, -1, 1, 3, 2, 2
```

Kodeboks 3.7: Varierende spring med indlejret Pwhite

Her er et andet musikalsk eksempel, hvor vi bruger **`Pseries`** til at styre hvor mange tilfældige toner fra **`Pwhite`**, der bliver flettet ind i sekvensen - med flere og flere for hvert gennemløb.

```
Pbind(
    \degree, Pseq([
        -7,
        -7,
        Pwhite(0, 4, length: Pseries(0, 1).asStream)
    ], 8),
    \dur, 0.25,
).play;
```

Kodeboks 3.8: Varieret fraselængde med indlejret Pseries

[1] Teknisk nysgerrige læsere kan konsultere Fieldsteels (2021) diskussion, som på ganske udmærket vis demonstrerer forbindelsen mellem patterns og streams.

3.2 Sammensætning af event patterns

Hidtil har vi betragtet patterns som opskrifter på strømme af værdier. Men hvis vi ser på **Pbind**, er det tydeligt, at den ikke producerer strømme af enkeltværdier, men i stedet producerer såkaldte *events* ud fra de nøgler og værdier/patterns, den knytter sammen. I denne terminologi svarer én event til én tone. Pbind er dermed et *event pattern*, i modsætning til *value patterns* som **Pwhite** og **Pbrown**, der producerer *værdier*. Man kan bruge event patterns som moduler, der kan sættes sammen og varieres på forskellig vis, hvilket vi skal kigge nærmere på herunder.

3.2.1 Sekvenser af Pbinds

Den mest åbenlyse måde at sammensætte Pbinds på er at afvikle dem sekventielt, altså den ene efter den anden. Lad os eksempelvis definere tre Pbinds, med forskellige melodiske motiver. Bemærk, at vi ikke afspiller disse Pbinds med `.play` men i stedet gemmer dem under deskriptive variabelnavne.

```
1  ~op = Pbind(
2      \degree, Pseries(0, 2, 4),
3      \db, Pseries(-20, 2, 4)
4  );
5
6  ~ned = Pbind(
7      \degree, Pseries(1, -2, 4),
8      \db, Pseries(-14, -2, 4)
9  );
10
11 ~rundt = Pbind(
12     \degree, Pseq([0, 1, 0, -3]),
13     \db, Pgauss(-16, 2)
14 );
```

Kodeboks 3.9: Tre simple Pbinds 

Med ovenstående tre Pbinds gemt under variabelnavne kan vi gemme deres navne i en liste (se 1.6) og afspille dem med velkendte, listebaserede patterns som **Pseq**, **Prand** m.fl.

```
~sekvenser = [~op, ~ned, ~rundt];
TempoClock.tempo = 130 / 60;

Pseq(~sekvenser).play;
Prand(~sekvenser, 4).play;
Pxrand(~sekvenser, 4).play;
Pshuf(~sekvenser, 2).play;
```

Kodeboks 3.10: Sekvensering af Pbinds

3.2.2 Begrænsning af Pbind-output med Pfin og Pfindur

Nogle gange er det mere relevant at definere Pbinds, som ikke har en begrænset varighed (dvs. hvor ingen af de anvendte value patterns har et endeligt antal output), og derefter begrænse, hvor mange events, de producerer. På den måde kan den samme Pbind anvendes i og tilpasses forskellige sammenhænge. Lad os eksempelvis definere to forskellige Pbinds, hvor den ene er lidt rodet i sin timing, dynamik og rytmik, mens den anden er mere regulært opbygget.

```
~drunk = Pbind(
    \degree, Pbrown(-3, 7, 7),
    \ctranspose, Pwrand([0, -1], [0.9, 0.1], inf),
    \scale, Scale.minor,
    \dur, Pexprand(0.2, 0.8),
    \legato, 0.25,
    \db, Pwhite(-25, -10),
);
~sober = Pbind(
    \degree, Pbrown(-3, 6, 3),
    \scale, Scale.minorPentatonic,
    \dur, Prand([0.5, 1, 1.5], inf),
    \legato, Pexprand(1.3, 1.5),
    \db, Pgauss(-20, 2),
);
```

Kodeboks 3.11: To forskellige Pbinds

Hvis vi kun ønsker 10 events fra en af ovenstående Pbinds, kan vi bruge et pattern, der hedder **Pfin**. Man kan også sekvensere **Pfin** med fx **Pseq**, så man kan sammensætte sekvenser fra forskellige Pbinds.

```
1 TempoClock.tempo = 115/60;
2
3 Pfin(10, ~drunk).play;
4 Pfin(10, ~sober).play;
5 Pseq([ Pfin(8, ~drunk), Pfin(8, ~sober) ], 4).play;
```

Kodeboks 3.12: Begrænsning af event-antal med Pfin

Da disse to pbinds har varierende `\dur`-værdier, er det umuligt at forudsige hvor mange events vi skal vælge med **`Pfin`** for eksempelvis at spille én takt med den ene Pbind, én takt med den anden, og så fremdeles. Hertil kan vi i stedet bruge et nært beslægtet pattern, der hedder **`Pfindur`**`(varighedsgrænse, pattern, toleran⌋ce)`, som kan begrænse *varigheden* af en Pbind.

```
1 TempoClock.tempo = 115/60;
2
3 Pseq([
4     Pfindur(4.01, ~drunk, 0.01),
5     Pfindur(4.01, ~sober, 0.01),
6 ], 4).play;
```

Kodeboks 3.13: Begrænsning af Pbind-varighed med Pfindur

3.2.3 At kombinere Pbinds med Pbindf

Som tidligere nævnt (se 1.4) kan det i programmering være fornuftigt med en vis portion strategisk dovenhed. Det skal forstås således, at det ofte er ønskeligt at undgå at skulle skrive den samme kildekode flere gange. Dette princip kan også bruges i forbindelse med patterns, og det viser sig at være meget nyttigt til at skabe variationer og sammensætninger, hvis vi som nævnt ovenfor bygger vores patterns op i mindre moduler, der så kan sammensættes på nye måder.

Hertil kan vi bruge **`Pbindf`** (bemærk f'et til sidst i klassenavnet), som skaber en nye Pbind baseret på en anden, foruddefineret Pbind. Det betyder, at vi kan definere én Pbind, som kan varieres og lægges til grund for en række andre Pbinds, hvilket åbner mange muligheder for at arbejde med variation og musikalsk elaborering.

Vi bruger **`Pbindf`** ved at angive en eksisterende Pbind som det første argument. De efterfølgende argumenter fungerer præcis som ved Pbind. Her kan vi eksempelvis definere et enkelt melodisk motiv i én Pbind og i en anden lave en „overstemme" gennem modal transponering, dvs. parallelføring inden for skala.

```
~melodi = Pbind(
    \degree, Pseq([2, 1, 3, 2], 4),
    \dur, 0.5
);
~overstemme = Pbindf(~melodi,
    // Læg enten en terts eller en sekst til
    \mtranspose, Prand([2, 5], inf)
);

~melodi.play; ~overstemme.play;
```

Kodeboks 3.14: Overstemme med Pbindf

Bemærk her, at vi i stedet for at omdefinere `\degree`-nøglen i Pbindf'en anvender `\mtranspose`. Havde vi brugt `\degree`, ville den eksisterende information om skalatrin fra `~melodi`-Pbind'en blive overskrevet. I stedet fungerer den oprindelige **`Pseq`** og den nye **`Prand`** sammen, så „stemmerne" kan følges ad.

3.2.4 En minimalistisk kompositionside

Et mere interessant eksempel kan være en algoritme inspireret af det minimalistiske værk *Piano Phase* fra 1967 af komponisten Steve Reich. I dette værk for to klaverer spiller hver pianist den samme melodiske frase igen og igen. Værkets særkende er, at den ene pianist spiller motivet i en lille smule højere tempo end den anden. Resultatet er, at frasens forskellige tonehøjder og rytmer konstant går ind og ud af „fase" med hinanden. Dette kan vi på simpel vis simulere med Pbindf, hvor vi sætter `\dur`-nøglen i to forskellige versioner af en `~basis`-Pbind til næsten ens nodeværdier.

```
TempoClock.tempo = 115/60;   // tempo 115 BPM

// Trinsekvenser
~trin = [0, 1, 4, 5, 6, 1, 0, 5, 4, 1, 6, 5];

~basis = Pbind(
    \root, 4,
    \octave, 4,
    \scale, Scale.dorian,
    \degree, Pseq(~trin, inf),
    \legato, 0.5,
);
~left = Pbindf(~basis, \dur, 0.250, \pan, -1).play;
~right = Pbindf(~basis, \dur, 0.252, \pan, 1).play;

~left.stop; ~right.stop;
```

Kodeboks 3.15: Forenklet udgave af Steve Reichs Piano Phase

Dette er blot en forsimplet illustration af, hvordan *Piano Phase* fungerer. I slutningen af dette kapitel (se 3.6) kan man øve sig i at skrive egne minimalistiske kompositioner, hvor lignende processer med gradvis udvikling gør sig gældende.

3.3 Pattern-komposition med MIDI-output

Når vi lærer at arbejde med patterns som kompositionsredskab, er det oplagt at anvende en mere interessant lydkilde end den indbyggede standardlyd i SuperColliders lydserver. Vi kommer senere til at designe vores egne lyde med SuperColliders lydserver (se 5.3), men for nuværende kan det være mere inspirerende at bruge vores patterns til MIDI-komposition. Det gøres i grove træk på følgende måde:

1. Opret en virtuel MIDI-port til kommunikation mellem SuperCollider og DAW.
2. Indstil DAW til at modtage MIDI-meddelelser på et spor med et virtuelt instrument.
3. Indstil SuperCollider til at sende MIDI-meddelelser med `MIDIClient.init` og `MIDIOut.newByName`.
4. Spil på instrument-plugin'et i DAW'en ved hjælp af `Pbind`.

Herunder kigger vi nærmere på disse trin, med særligt fokus på SuperCollider-delen. De øvrige trin er velbeskrevne andetsteds.

3.3.1 Opsætning og test af MIDI-kommunikation

3.3.1.1 Virtuel MIDI-port og opsætning af DAW

For at sende MIDI-signaler fra SuperCollider til en DAW på samme computer, skal man sætte en virtuel MIDI-port op til at skabe forbindelse mellem de to programmer. Dette er heldigvis enkelt: På Windows kan man bruge programmet *loopMIDI*, og på Mac kan man opsætte en virtuel port med den såkaldte *IAC Driver*. Dette forklares ganske fint i en guide fra Ableton.

Man sætter derefter DAW'en op, så den modtager MIDI-meddelelser fra den virtuelle MIDI-port på et spor, hvor man har sat et instrument-plugin op. Det vil føre for vidt at introducere til dette for alle tænkelige DAW-programmer her. Men denne funktionalitet er heldigvis dokumenteret grundigt andre steder, da vores foretagende på DAW-siden svarer til, hvordan man tilslutter et MIDI-keyboard. Følg instrukserne for din DAW:

- Reaper
- Ableton Live
- Logic Pro
- Ardour
- Cubase
- Studio One

- FL Studio
- Pro Tools
- Bitwig Studio

Det gælder blot om at sørge for, at DAW'en er indstillet til at modtage MIDI-meddelelser via den føromtalte, virtuelle MIDI-port. Alternativt kan man sende MIDI-meddelelser fra SuperCollider til et stykke hardware som er tilsluttet computeren via MIDI, fx en ekstern synthesizer. For enkelhedens skyld omtaler vi dog her modtageren af MIDI-meddelelser som en DAW. Jeg forudsætter herunder, at den virtuelle eller fysiske forbindelse er etableret, og at modtageren af vores MIDI-signal er klar til at modtage meddelelser.

3.3.1.2 Opsætning af MIDI-output fra SuperCollider

Når DAW er klar til at modtage MIDI-signal via en virtuel MIDI-port, starter man i SuperCollider med at køre linjen `MIDIClient.init`. Dette får SuperCollider til at kontakte computerens MIDI-system og vise i post window hvilke MIDI-porte der er tilgængelige. Derfra noterer man navnet på den ønskede port og bus. Dem angiver man så, når man med klassen `MIDIOut` og method'en `.newByName` opretter en forbindelse til output og gemmer dette under en global variabel.

```
1  // Start MIDI-kommunikation
2  MIDIClient.init;
3
4  // I post window ses nu to nyttige lister
5  // På Windows kan det fx se sådan ud:
6  MIDI Sources:
7      MIDIEndPoint("loopMIDI Port", "loopMIDI Port")
8  MIDI Destinations:
9      MIDIEndPoint("Microsoft GS Wavetable Synth", "Microsoft GS
       ↪ Wavetable Synth")
10     MIDIEndPoint("loopMIDI Port", "loopMIDI Port")
```

Kodeboks 3.16: Klargøring af MIDI med MIDIClient

Da vi skal sende MIDI ud af SuperCollider, kigger vi på listen over `MIDI Destinations` i post window for at identificere den virtuelle MIDI-port. Vi noterer os her hvilket `MIDIEndPoint`, vi ønsker at kommunikere med. På Windows vil det typisk være `MIDIEndPoint("loopMIDI Port", "loopMIDI Port")`, når vi bruger *loopMIDI*. Her kan vi kopiere den tekst, der står mellem parenteserne efter MIDIEndPoint og sætte den ind som argumenter til `MIDIOut.newByName`.

```
// Typisk portnavn på Windows
~daw = MIDIOut.newByName("loopMIDI Port", "loopMIDI Port");

// Typisk portnavn på Mac
~daw = MIDIOut.newByName("IAC Driver", "Bus 1");
```

Kodeboks 3.17: Opret MIDIOut til DAW/synthesizer

3.3.1.3 Test forbindelsen

Når vi har oprettet en **MIDIOut**, er vi klar til at sende MIDI-meddelelser fra SuperCollider. For at teste forbindelsen, kan vi sende en Note On- og en Note Off-meddelelse, hvorved vi kan se eller høre i vores DAW, at der spilles en tone.

```
~daw.noteOn(chan: 0, note: 64, veloc: 80);
~daw.noteOff(chan: 0, note: 64);
```

Kodeboks 3.18: Test MIDI-forbindelsen med toneanslag og -afslag

Hvis der ikke spiller en tone, kan du tjekke i SuperColliders post window om der skulle være nogen fejlmeddelelser, samt i den DAW/synthesizer, der skal modtage meddelelserne.

3.3.2 Patterns og MIDI-begivenheder

3.3.2.1 Særlige nøgler til MIDI-kommunikation

For at bruge **Pbind** som MIDI-generator skal vi angive et par oplysninger under nogle særlige nøgler:

`\type`
: Med denne nøgle angiver vi hvilken event-type, vores Pbind skal generererere. Vi har hidtil arbejdet med den event-type, som er default, nemlig den såkaldte `\note`-event. Når vi angiver `\midi` her, fortæller vi Pbind, at der skal produceres MIDI-meddelelser i stedet for meddelelser til lydserveren.

`\midiout`
: Her angiver vi den **MIDIOut**, vi oprettede ovenfor, så Pbind ved, hvor de genererede MIDI-meddelelser skal sendes hen.

`\chan`
: Her kan vi angive hvilken MIDI-kanal, der skal sendes på. Der er 16 tilgængelige kanaler, nummereret i SuperCollider fra 0 til 15.

Samler vi disse oplysninger, vil vi med nedenstående kildekode kunne høre en række toner blive spillet i DAW/synthesizer.

```
(
~komposition = Pbind(
    \type, \midi,
    \midiout, ~daw,
    \chan, 0,
).play;
)
~komposition.stop;
```

Kodeboks 3.19: Pbind med MIDI-output 

3.3.2.2 Automatisk Note On og Note Off

Når vi bruger Pbind til at sende MIDI-output, sendes der automatisk Note On- og Note Off-meddelelser i overensstemmelse med de værdier vi angiver med `\dur` og `\legato`-nøglerne, præcis som hvis vi brugte SuperColliders lydserver. Nedenstående eksempel demonstrerer dette ved at veksle fra legato- til staccatofrasering.

```
(
~komposition = Pbind(
    \type, \midi,
    \midiout, ~daw,
    \chan, 0,

    \degree, Pseries(0, 1, 8).repeat(2),
    \dur, 0.5,
    \legato, Pseq([1.5, 0.1]).stutter(8)
).play;
)
~komposition.stop;
```

Kodeboks 3.20: Automatisk Note On og Note Off 

Når man arbejder med patterns som MIDI-generator til at spille på et andet stykke software eller hardware, vil det hurtigt blive tydeligt, hvis vi stopper kompositionen

i utide med Ctrl/Cmd-Punktum. Derved stopper vi nemlig SuperCollider i at afsende den sidste Note Off-meddelelse, hvilket betyder, at den sidste tonen vil blive hængende i DAW/synth (medmindre der er tale om et instrument, som ignorerer Note Off-meddelelser). For at undgå dette, gemmer vi `Pbind().play` under en variabel, og vi kan så efterfølgende stoppe forløbet med `.stop`, som vist ovenfor.

3.3.2.3 Automatisk opsætning af MIDI-nøgler med Pbindf

Som vi har set tidligere (se 3.2.3), kan vi bruge `Pbindf` til at genbruge en tidligere defineret `Pbind`. På den måde kan vi definere vores MIDI-nøgler én gang for alle og derefter undgå at skrive dem igen.

```
~midi = Pbind(
    \type, \midi,
    \midiout, ~daw,
    \chan, 0,
);

~klaver = Pbindf(~midi,
    \degree, Pbrown(-2, 4, 3),
    \dur, Prand([0.5, 1], inf),
).play;
~bas = Pbindf(~midi,
    // Vi sender her på en anden MIDI-kanal
    // ved at overskrive \chan-nøglen
    \chan, 1,
    \octave, 3,
    \degree, Pseq([1, -3], inf),
).play;
```

Kodeboks 3.21: Genbrug af MIDI-nøgler med Pbindf

3.3.3 Begrænsninger ved MIDI-protokollen

Når vi komponerer med MIDI, er det væsentligt at notere sig dennes begrænsninger. Almindelige *Note On* og *Note Off*-meddelelser i MIDI indeholder blot information om hvilken MIDI-tone, det drejer sig om, samt en parameter, der kaldes *velocity*, hvilket ofte kobles med lydstyrke[2]. Begge disse oplysninger befinder sig i intervallet 0-127, så hvis vi manuelt skal sende disse beskeder (fx med `~daw.noteOn`), skal

[2] MIDI, der står for *Musical Instrument Digital Interface* blev oprindeligt udviklet af en sammenslutning af instrumentproducenter, som ønskede en fælles protokol for hvordan forskelligt musikudstyr kunne kommunikere med hinanden. Den prototypiske anvendelse af MIDI er således når man kobler et MIDI-keyboard sammen med en synthesizer for at kunne spille på denne via keyboardets tangenter.

værdierne befinde sig inden for dette interval. Bruger vi Pbind, kan vi heldigvis anvende de fleste af de nøgler, vi kender, som `\degree`, `\octave`, `\db` osv. Samtidig kan vi styre, *hvornår* meddelelserne afsendes, og med `\legato`-nøglen hvor længe bestemte toner skal „holdes“. Dermed har vi altså kontrol over følgende musikalske parametre:

- Tonehøjde via MIDI-tonetal
- Lydstyrke via MIDI-velocity
- Rytmik og frasering, i form af meddelelsernes timing

Vi har *ikke* kontrol over parametre som klang, envelope-parametre eller lignende. Dertil kan man evt. anvende de såkaldte *Control Change*-meddelelser, som nysgerrige læsere kan undersøge nærmere i SuperColliders dokumentation.

Deraf kom terminologien *velocity*, som oprindeligt vedrørte hvor hurtigt en tangent blev trykket ned og i dag stadig bruges, uagtet at der ikke altid er tale om en fysisk tangent, som trykkes ned.

3.4 Tips og tricks til patterns

Der er meget at lære, når man først starter med patterns. Der følger en helt ny, musikalsk logik med dette komplekse system af redskaber, der kan anvendes på kryds og tværs. Herunder kommer et par yderst nyttige tips og tricks, som du vil få glæde af, når du eksperimenterer med patterns.

3.4.1 Mød Pbinds bedste ven, Pdef

Når man skriver kompositioner med **Pbind** og foretager mange små ændringer, kan det være en lidt klodset proces konstant at skulle slukke for lyden først og derefter eksekvere **Pbind**().play igen. Her kan det være en stor fordel at bruge **Pdef**. Den giver os nemlig mulighed for at eksekvere den samme blok igen og igen, og så vil instrukserne i den nyligt eksekverede kildekode erstatte de hidtidigt kørende instrukser. Første argument til **Pdef**() er et unikt navn, som vi vil give vores komposition. Næste argument er det pattern, vi vil anvende - typisk en **Pbind**. Kildekoden herunder eksekveres flere gange uden Ctrl/Cmd-Punktum.

```
1 TempoClock.tempo = 250 / 60;
2 Pdef(\minKomposition,
3     Pbind(
4         \degree, Pshuf([-3, -2, 0, 4], inf)
5     )
6 ).play;
```

Kodeboks 3.22: Let kompositionsproces med Pdef

Kører vi kildekoden ovenfor, vil **Pshuf** vælge en tilfældig rækkefølge og gentage den, indtil vi stopper kompositionen igen. Men kører vi blokken igen uden at stoppe den først, bliver vores **Pdef**, **Pbind** og **Pshuf** genoprettet på ny, og **Pshuf** vil derfor vælge en ny rækkefølge. Vi fortsætter endda i takt med det, vi havde gang i før eksekveringen.

Når vi starter en **Pdef**, vil den automatisk blive registreret under det navn, vi angiver. Vi kan senere bruge navnet til fx at stoppe kompositionen igen. Bemærk her, at der ikke er behov for globale variabler, da **Pdef**-klassen selv holder styr på alle dens instances.

```
Pdef(\minKomposition,
    Pbind(
        \freq, Pexprand(55, 1500).round(55),
        \dur, Pexprand(0.1, 0.2),
        \legato, 2,
    )
).play;

Pdef(\minKomposition).stop;
```

Kodeboks 3.23: Start og stop af Pdef

`Pdef` har i øvrigt en slægtning kaldet `Pbindef`, som også er handy, men som jeg ikke anbefaler at bruge, da den kan give uventede resultater, hvis man tilføjer og fjerner nøgler og patterns løbende.

3.4.2 Hvad giver dit pattern egentlig af resultater?

Hvis man eksperimenterer og forsøger sig frem med forskellige patterns og værdier, kan det godt være vanskeligt at forstå hvad outputtet er. Derfor kan det være nyttigt med et overblik over de værdier, der bliver genereret.

3.4.2.1 Følg med i strømmen af data med .trace

I den situation er method'en `.trace` en stor hjælp, da den viser outputtet fra et pattern i SuperColliders post window.

```
Pbind(
    \freq, Pexprand(220, 1500, 10).round(220).trace,
    \dur, 0.25,
).play;
```

Kodeboks 3.24: Visning af pattern-output med .trace

Hvis vi har gang i flere patterns, som vi gerne vil følge, kan vi bruge argumentet `prefix` til at kende forskel ved hjælp af en angivet tekst.

```
Pbind(
    \octave, Pbrown(3, 5, 1).trace(prefix: "Oktav: "),
    \degree, Pseries(0, 1, 10).trace(prefix: "Skalatrin: "),
).play;
```

Kodeboks 3.25: Visning af pattern-output med .trace 

3.4.2.2 Generér en masse data på én gang og plot dem

Et sidste trick, der kan være nyttigt, er at generere alle værdierne fra et pattern på én gang, for at få et overblik over dem. Dette gør vi i et par trin:

1. Først skriver vi det pattern, vi vil undersøge - inklusiv de argumenter, vi vil eksperimentere med. Her er det meget vigtigt, at det er et pattern med et begrænset antal output. Ellers vil SuperCollider få det umådeligt svært, fordi det skal beregne samtlige værdier i en uendelig strøm.
2. Dernæst konverterer vi vores pattern til en såkaldt stream med method'en `.asStream`.
3. Vi kan derefter bruge method'en `.all` til at få alle de genererede værdier udregnet.

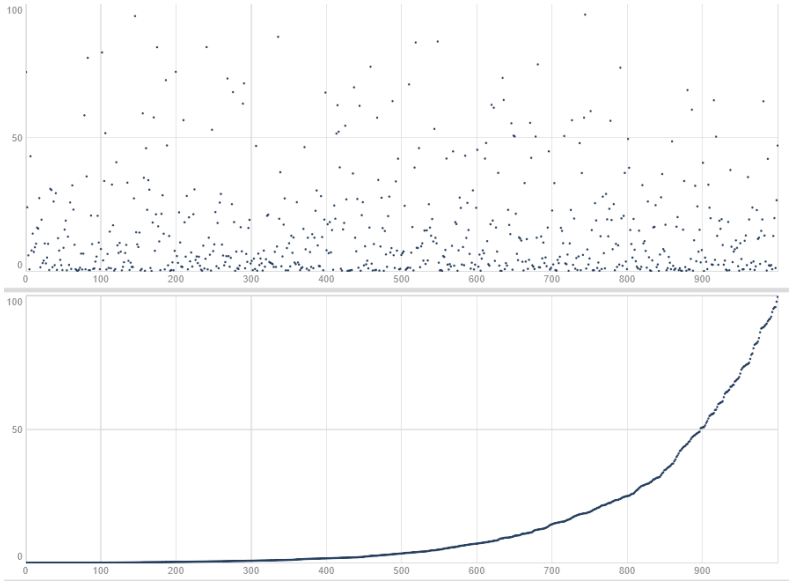
```
Pwhite(0, 7, 5).asStream.all;
// -> fx [4, 7, 2, 7, 0]
```

Kodeboks 3.26: Find alle genererede værdier fra et pattern 

Vi kan også sortere, visualisere og analysere disse data med forskellige methods (se 1.5), der knytter sig til lister (se 1.6).

```
~data = Pexprand(0.1, 100, 1000).asStream.all;
// -> [ 44.267960075825, 0.33396883358304, 6.4274511302122,
↪  1.1416103740564  ...

~data.mean; // Gennemsnit
// -> 14.903230351712

// Plot
[~data, ~data.copy.sort].plot(discrete: true);
```

Kodeboks 3.27: Analyse af pattern-output med liste-methods 



Figur 3.1: Plot af outputtet fra Pexprand

Vi bruger her method'en `.copy`, fordi `.sort` ellers vil forandre den oprindelige liste, som er gemt under variablen `~data` (og dermed give os to ens grafer). Når vi sætter argumentet `discrete: true`, vil datapunkterne i plottet blive vist med prikker i stedet for forbundne linjer.

3.5 Øvelse: Analyse og videreudvikling af kompositionseksempler

I denne øvelse skal du analysere udvalgte aspekter af fem eksempler på pattern-baseret komposition. Øvelsen skal gøre dig fortrolig med de teknikker, der er omtalt tidligere i bogen, da vi her anvender de introducerede teknikker i en kompositorisk sammenhæng. Du skal nemlig også videreudvikle kompositionerne med dit eget kreative bidrag.

Redskaber til analyse: Når du undersøger, hvordan teknikkerne fungerer, kan disse tricks være en god hjælp til at forstå hvad der foregår:

- Brug method'en `.trace` til at vise outputtet fra forskellige patterns i post window, fx `Pwhite(0, 5).trace`.
- Brug SuperColliders dokumentation - sæt cursoren ved et pattern-navn og tast Ctrl/Cmd-D. Scroll herefter ned til bunden af den pågældende dokumentationsside for at se eksempler på hvordan det pågældende pattern fungerer.
- Eksperimentér med at ændre på nogle af værdierne for at få en fornemmelse af, hvordan teknikkerne fungerer.

Lydeksempler: Lydeksemplerne herunder er realiseret ved at sende MIDI-meddelelser til en DAW og generere lyden med instrumentplugins, som vist i det foregående afsnit om pattern-baseret MIDI-komposition (se 3.3). Det fremgår under hvert eksempel hvilket konkret plugin, der er anvendt.

3.5.1 Sammensætning af nøgler og patterns

1. Analyse
   a) Hvordan fungerer de enkelte nøgler (`\octave`, `\root`, `\mtranspose`, `\lag` osv.)? Slå evt. op i forrige kapitel (se 2.3).
   b) Hvordan kombineres tilfældighed og repetition under nøglerne `\mtranspose` og `\db`? Se evt. afsnittet om sekvenser af patterns (se 3.1.1).
2. Kreativ opgave
   a) Eksperimentér med alternative skalaer, nodeværdier og trinsekvenser

```
TempoClock.tempo = 120 / 60;

~trin = [0, 4, 3, 1, 2];
~varigheder = [1/8, 1/8, 1/8, 1/16, 1/16];
Pbind(
    \octave, 5,
    \root, 2,
    \scale, Scale.lydian,
    \degree, Pseq(~trin, 16),
    \mtranspose, Pwhite(-3, 3).stutter(~trin.size + 2),

    \dur, Pseq(~varigheder, inf) * 4,
    \legato, 1.2,
    \lag, Pgauss(0, 0.005),

    \db, Pseq([-10, Pgauss(-15, 2, 4)], inf),
).play;
```

Kodeboks 3.28: Sammensætning af nøgler og patterns

Lydeksemplet er realiseret med instrument-plugin'et Helm og preset'et *Old Factory Presets CM Pluck Time* med portamento slået til.

3.5.2 Skala-udforskning med Pbrown

1. Analyse
    a) Hvilken funktion har nøglen `\ctranspose`?
    b) Hvad sker der, hvis du ændrer nøglen `\ctranspose` til `\mtranspose`?
    c) Hvad er forskellen på **`Pbrown`** og **`Pwhite`**?
2. Kreativ opgave
    a) Skab en mere interessant rytmik ved at erstatte den faste værdi 0.2 ved `\dur`-nøglen med et pattern efter eget valg

```
TempoClock.tempo = 85 / 60;

Pbind(
    \degree, Pbrown(0, 21, 3),
    \octave, 4,
    \ctranspose, Pbrown(-5, 4, 1, 4).stutter(32),
    \db, Pbrown(-15, -5, 2).trace,
    \dur, 0.25,
).play;
```

Kodeboks 3.29: Skala-udforskning med Pbrown

Lydeksemplet er realiseret med instrument-plugin'et sforzando og sfz-instrumentet *Marimba* fra Versilian Studios Chamber Orchestra 2 Community Edition.

3.5.3 Pentatone mønstre

1. Analyse
   a) Hvilke teknikker anvendes her til at opnå en balance mellem struktur/repetition og tilfældighed?
   b) Hvilket pattern styrer kompositionens storform, altså hvor mange toner vi samlet hører?
   c) Hvilken funktion har `Array.interpolation`? Se evt. afsnittet om lister (se 1.6.4.1).
2. Kreativ opgave
   a) Skriv to variationer af kompositionen:
      i. Én version, som har en højere grad af tilfældighed
      ii. Én version, som har en højere grad af struktur og gentagelse

```
TempoClock.tempo = 130 / 60;

Pbind(
    \scale, Scale.minorPentatonic,
    \octave, Pwhite(4, 5).stutter(4),
    \degree, Pshuf([0, 1, 2, 3, 4, 5], 4).repeat,
    \root, Pxrand([0, 2, 3]).repeat.stutter(24),

    \dur, 0.25,
    \legato, Pseq(Array.interpolation(24, 0.1, 3), 8),

    \db, Pbrown(-20, -12, 0.5)
).play;
```

Kodeboks 3.30: Pentatone mønstre 

Lydeksemplet er realiseret med instrument-plugin'et Vital med en let justeret udgave af preset'et *Super Pluck*.

3.5.4 Rytmiserede og dynamiske akkorder

1. Analyse
   a) Hvilken effekt har kombinationerne af `.stutter` og `.repeat` på outputtet fra de forskellige patterns?
   b) Hvad betyder `Array.interpolation(16, -20, -10)`?
2. Kreativ opgave
   a) Tilføj mindst én akkord til **`Pwrand`** (husk, at sandsynlighederne `[0.9, 0.1]` skal svare til antallet af valgmuligheder og tilsammen skal give 1).
   b) Erstat **`Pxrand`** med et andet listebaseret tilfældighedspattern (se 2.2.2) efter eget valg, og notér hvilken forskel dette gør.

```
1  TempoClock.tempo = 120 / 60;
2
3  Pbind(
4      \degree, Pwrand([
5          [-14, 0, 2, 4, 6],
6          [-12, -1, 1, 4, 5]
7      ], [0.9, 0.1]).stutter(16).repeat,
8
9      \mtranspose, Pxrand((-5..5)).repeat.stutter(16),
10
11     \dur, 1/16 * 4,
12     \legato, 0.7,
13
14     \db, Pseq(Array.interpolation(16, -20, -10), inf),
15 ).play;
```

Kodeboks 3.31: Rytmiserede og dynamiske akkorder

Lydeksemplet er realiseret med instrument-plugin'et sforzando og sfz-instrumentet *Upright Piano* fra Versilian Studios Chamber Orchestra 2 Community Edition.

3.5.5 Rytmiske motiver

1. Analyse
   a) Undersøg hvilke teknikker, der i dette tilfælde skaber balance mellem det tilfældige og det genkendelige.
   b) Undersøg hvad method'en `.normalizeSum` gør.
2. Kreativ opgave
   a) Skriv en ny komposition, som er inspireret af kildekoden herunder samt opgaverne ovenfor.

En sidebemærkning: Pattern'et **Pn** på linje 12 herunder er en eksplicit udgave af `.repeat`. **Pn** gentager den værdi eller det pattern, der angives som første argument, et angivet antal gange.

```
(
TempoClock.tempo = 85 / 60;

~melodi = Pbind(
    \scale, Scale.dorian,
    \degree, Pshuf((0..7), 4).repeat,

    \legato, 1.3,
    \dur, Pwrand([
        Pseq( [1/4, 1/4] ),
        Pseq( [1/16, 1/16, 1/8] ),
        Pseq( [1/16, 1/8, 1/16] ),
        Pseq( [Pn(1/24, 6), 1/4] ), // 16.-dels-trioler
        Pseq( [1/2, Rest(1/2)] ),   // Rest angiver pause
    ], [40, 40, 30, 5, 5].normalizeSum
    ).repeat * 4,
);
~komp = ~melodi.play;
)
// Flerstemmig variation med Ppar (parallelle Pbinds):
~komp = Ppar(~melodi ! 3).play;

~komp.stop;
```

Kodeboks 3.32: Korte, rytmiske sekvenser

Lydeksemplet er realiseret med instrument-plugin'et Spitfire LABS og sample pack'en *Charango - Charango Ensemble*.

3.6 Øvelse: Minimalistisk komposition

Denne øvelse går ud på at anvende patterns til at skabe en enkel, minimalistisk komposition.

På dette trin i kurset kan øvelsen oplagt udføres som en komposition, der bliver realiseret over MIDI. På den måde kan man fx lade en standalone-synthesizer eller en DAW udføre klangdannelsen og koncentrere sig om de kompositionsmæssige aspekter som tonehøjde, rytmik og dynamik i SuperCollider. Se hertil guiden til MIDI-output fra SuperCollider (se 3.3). Alternativt kan øvelsen udføres med standardlyden i SuperCollider. Senere i kurset designer vi egne lyde med SynthDefs (se 5.3), som kan erstatte den lidt kedelige standardlyd.

Skriv en komposition ved hjælp af **`Pbind`**. Kompositionen skal:

- Udvikle sig langsomt og gradvist, jævnfør minimalismen.
- Tage udgangspunkt i et yderst begrænset rytmisk og tonalt materiale.
- Anvende mindst to forskellige listebaserede patterns samt mindst ét tilfældighedsgenererende pattern.

Toner og nodeværdier vælges først og noteres i lister under `~toner` og `~varigheder`.

Der kan eksempelvis gøres brug af følgende virkemidler:

- Gentagelser med subtil variation.
- Gradvis dynamisk udvikling og variation (lydstyrke).
- Skiftende oktavering.
- Modaltransponering.
- Minimal rytmisk udvikling.
- Små forskelle på samtidigt klingende sekvenser.

Mulighed 1: Med MIDI-output til DAW

Følg først vejledningen til MIDI-output fra SuperCollider (se 3.3), så du har en korrekt konfigureret **`MIDIOut`** gemt under variablen `~daw`.

```
// Vælg nogle få toner og nodeværdier
~toner = [   ];
~varigheder = [   ];

Pbind(
    // MIDI-konfiguration
    \type, \midi,
    \midiout, ~daw,
    \chan, 0,

    // Tilføj egne patterns herunder
    \degree,   ,
    \octave,   ,
    \mtranspose,   ,

    \db,   ,

    \dur,   ,
    \legato,   ,
).play;
```

Kodeboks 3.33: En minimalistisk kompositionsopgave

Mulighed 2: Med SuperColliders standardlyd

```
// Vælg nogle få toner og nodeværdier
~toner = [   ];
~varigheder = [   ];

Pbind(
    // Tilføj egne patterns herunder
    \degree,   ,
    \octave,   ,
    \mtranspose,   ,

    \db,   ,

    \dur,   ,
    \legato,   ,
).play;
```

Kodeboks 3.34: En minimalistisk kompositionsopgave

KAPITEL 4

Oscillatorer og modulation

Dannelse og transformation af lyd er en central del af musik- og lydprogrammering. Redskaber som SuperCollider tillader os at arbejde meget fleksibelt med lyd på et detaljeret niveau. Dette kan give os unikke lyddesign til brug i musikalsk komposition, interaktive systemer, musikinstrumenter, lydkunst mm. Samtidig giver arbejdet med lyddesign på dette niveau en glimrende forståelse af principperne bag digital musik- og lydteknologi. I dette og de efterfølgende kapitler tager vi hul på dannelse og transformation af lyd ved hjælp af oscillatorer, filtre, envelopes, sample-afspillere mm. Disse signalbehandlingsredskaber kaldes UGens, og vi kigger i dette kapitel nærmere på, hvordan UGens fungerer, og hvordan vi anvender og sammensætter dem i såkaldte UGen-funktioner.

4.1 UGens og signalflow

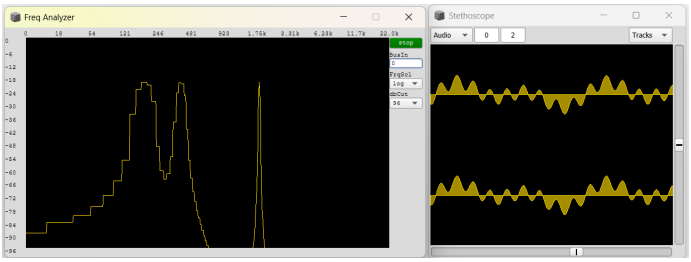
Det grundlæggende redskab for musikalsk lyddesign i SuperCollider og lignende platforme som puredata, Max/MSP, Csound m.fl. er såkaldte UGens, Unit Generators. Eksempler på UGens er oscillatorer, filtre, lyttemaskiner, envelope-generatorer og mange enheder til at generere og transformere lydsignaler. De kan sammenlignes med komponenterne i et elektronisk kredsløb eller modulerne i en modulær synthesizer. SuperCollider indeholder mere end 250 forskellige UGens og kan udvides med særlige UGen-plugins. I kildekode kan vi kombinere UGens på mange forskellige måder med et fleksibelt signalflow og på den måde skabe unikke lyddesign.

Når du arbejder med UGen-funktioner og eksperimenterer med klangdannelse, kan du med fordel starte et oscilloskop og et indbygget redskab til spektralanalyse, så du kan se en grafisk repræsentation af serverens lydlige output. Flyt evt. vinduerne på din skærm, så du kan se både bølgeform og frekvensspektrum på én gang.

```
1 s.boot;
2 (
3 // Når serveren er bootet
4 s.scope;
5 s.freqscope;
6 )
```

Kodeboks 4.1: Start lydserver og visuelle redskaber

I spektralanalyse-vinduet viser den vandrette akse frekvenser, målt i hertz, mens den lodrette akser viser lydstyrke ved den givne frekvens, målt i decibel. På oscilloskopet viser den vandrette akse tid, mens den lodrette akse viser lydsignalets udsving.



Figur 4.1: Spektrogram og oscilloskop

4.1.1 Oscillator-UGens

Lad os skabe vores første UGen-funktion. Den noteres ligesom andre funktioner (se 1.4) med tuborg-parenteser.

Den mest enkle UGen vi kan vælge til vores UGen-funktion er nok **SinOsc**, en sinustone-generator. Ønsker vi savtakkede eller firkantede bølgeformer, kan vi i stedet bruge **Saw** eller **Pulse**. Der er også mulighed for at afspille forskellige former for støj med fx **WhiteNoise** eller **PinkNoise**, samples med **PlayBuf** og **BufRd** eller live-lyd fra en mikrofon med UGen'en **SoundIn**. Du kan se en oversigt over de vigtigste UGens i det dertilhørende cheat sheet (se 4.4), og brug af samples i SuperCollider dykker vi ned i senere i bogen (se 8.1).

For at oprette en simpel oscillator bruger vi den relevante UGen's klassenavn sammen med method'en `.ar`. Den omsluttende UGen-funktion startes med method'en `.play`.

```
1 {SinOsc.ar}.play;
2 {Saw.ar}.play;
3 {Pulse.ar}.play;
```

Kodeboks 4.2: Tre forskellige oscillatorer

4.1.1.1 Oscillatorfrekvens og amplitude

For at styre oscillatorernes frekvens, kan vi angive den ønskede frekvens som det første argument til method'en `.ar`. Ønsker vi at justere oscillatorens amplitude, kan vi gange med den faktor, vi ønsker at skalere op eller ned.

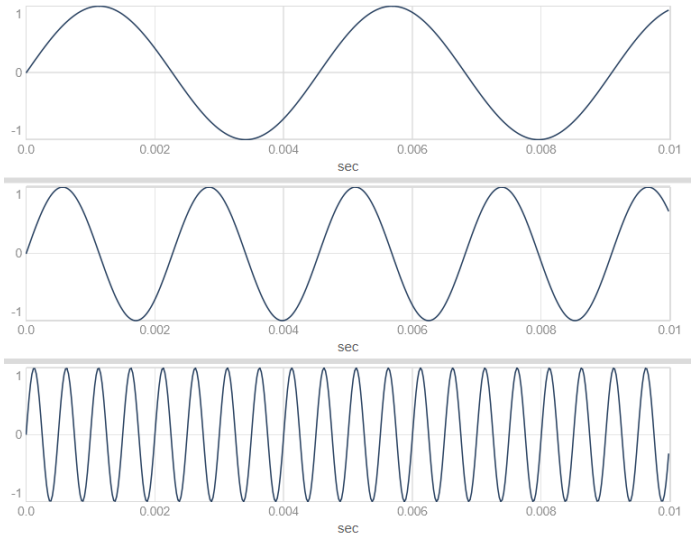
```
1 // Vi angive oscillatorens frekvens som det første argument til
  ↪  .ar-method'en
2 {SinOsc.ar(220)}.play;
3
4 // Vi skruer ned for lydstyrken ved at gange med 0.1
5 {SinOsc.ar(220) * 0.1}.play;
```

Kodeboks 4.3: Amplitude og frekvens

Hvis vi ønsker at se et fastfrosset billede af signalet fra en UGen-funktion, kan vi bruge method'en `.plot` i stedet for `.play`. Herunder plotter vi eksempelvis tre forskellige sinusbølger[1] på én gang.

```
// Her frekvenser på 220 Hz, 440 Hz og 2000 Hz
{SinOsc.ar([220, 440, 2000])}.plot;
```

Kodeboks 4.4: Plot af UGen-funktioner



Figur 4.2: Sinusbølger ved 220 Hz, 440 Hz og 2 kHz

4.1.2 Modulation

Sinustonen ovenfor bliver hurtigt lidt monoton, så lad os skabe lidt bevægelse i lyden ved at modulere nogle af de lydlige parametre. Der findes grundlæggende to parametre, man kan manipulere ved en oscillator: *Tonehøjde* (der bestemmes af oscillatorens frekvens) og *lydstyrke* (der bestemmes af oscillatorens amplitude). Der findes også en tredje parameter, nemlig (initial)fase, men den udelader vi her for enkelhedens skyld.

[1] Når vi noterer en liste (se 1.6) som argument til en UGen, udfører SuperCollider typisk en såkaldt „multichannel expansion" til at oprette en særskilt UGen for hvert af listens elementer. Dette kigger vi nærmere på senere (se 7.1).

4.1.2.1 Amplitude

Lad os først modulere sinustonens amplitude (lydstyrke). Det gør vi ganske enkelt ved *at gange med en anden UGen*. I dette eksempel bruger vi UGen'en **LFPulse**, som blot bevæger sig mellem 0 og 1 og dermed regelmæssigt tænder og slukker for lyden.

```
1 {SinOsc.ar(440) * LFPulse.kr(2) * 0.1}.play;
```

Kodeboks 4.5: Amplitudemodulation

Dette ligger til grund for de klangdannelsesteknikker, som kaldes amplitude modulation (AM) og ring modulation (RM).

4.1.2.2 Frekvens

Vi kan også modulere frekvensen. Her erstatter vi den fast angivne frekvens på 440 Hz med en anden SinOsc. Det er her nødvendigt at skalere outputtet fra den anden SinOsc, så vi får hørbare frekvenser (over 20 Hz) - det gør vi med .range, her fra 200 Hz til 400 Hz.

```
1 {SinOsc.ar( SinOsc.kr(5).range(200, 400) ) * 0.1}.play;
```

Kodeboks 4.6: Frekvensmodulation

Dette ligger til grund for den klangdannelsesteknik som kaldes frequency modulation (FM).

4.1.3 Signalflow med lokale variabler

Koden begynder nu at blive for kompliceret til at stå på én linje. For at gøre signalflowet mellem de forskellige UGens mere overskueligt og fleksibelt, kan vi derfor dele koden op, så den står på flere forskellige linjer. Dette indebærer, at vi indfører lokale variabler, så vi kan henvise til de forskellige signaler i vores UGen-funktion.

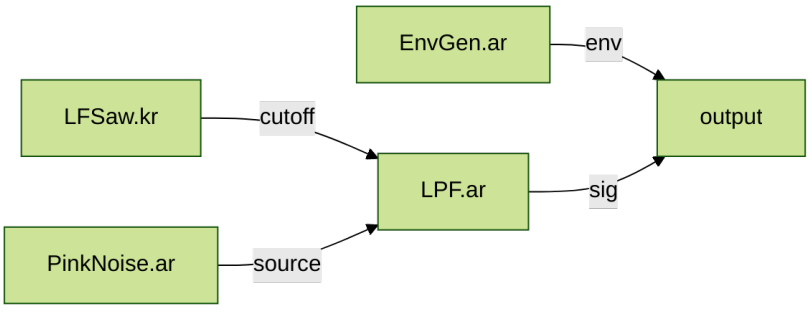
```
// Samme lyd som ovenfor, men kildekoden er lettere at læse og
↪ justere
{
    var modulator = SinOsc.kr(5).range(200, 400);
    var sig = SinOsc.ar(modulator);
    sig * 0.1;
}.play;
```

Kodeboks 4.7: Mere logisk og læsbar kildekode med lokale variabler

Vi kan oprette lige så mange lokale variabler, som vi har lyst til, de skal blot erklæres i begyndelsen af funktionen. Her er et eksempel med LFO-modulation, hvor en del forskellige UGens er på spil. Heldigvis kan vi gøre koden overskuelig, når vi bruger lokale variabler. Gæt selv, evt. med hjælp fra nedenstående graf, hvordan signalflowet fungerer i dette eksempel.

```
{
    var source = PinkNoise.ar;
    var env = EnvGen.ar(Env.triangle(5));
    var cutoff = LFSaw.kr(5).exprange(220, 880);
    var sig = LPF.ar(source, cutoff);
    sig = sig * env;
    sig * 0.1;
}.play;
```

Kodeboks 4.8: Eksempel på signalflow med lokale variabler



Figur 4.3: Signalflow mellem de anvendte UGens og variabler

4.1.4 Hvad er .ar og .kr?

Vi har hidtil primært arbejdet med SuperColliders patterns. Patterns kører i SuperColliders *fortolker* - det program, som fortolker den kildekode, vi eksekverer. I modsætning hertil kører UGens på SuperColliders *lydserver*, som er et andet program end fortolkeren (se 1.1.1). Rammen for vores arbejde med UGens er UGen-funktioner, der rent syntaktisk noteres ligesom de funktioner (se 1.4), vi tidligere har set. UGen-funktioner udskiller sig dog ved, at de beskriver noget, man med et teknisk udtryk kan kalde *signalgrafer*. Det betyder blot, at vores UGen-funktioner ikke bliver kaldt med `.value`, som vi tidligere har set med almindelige funktioner. Da de skal repræsentere lyd- og kontrolsignaler, udregnes de mange gange pr. sekund.

Hvor ofte „kører" vores UGen-funktion så? Jo, du har nok bemærket i eksemplerne ovenfor, at vi altid bruger en af to forskellige methods til at oprette UGens - `.ar` og `.kr`[2]. Disse methods henviser til de to forskellige frekvenser hvormed forskellige signalers værdi kalkuleres i lydserveren. *Audio rate* (`.ar`) svarer til serverens samplerate. Denne kan ligesom i anden musiksoftware justeres, men typisk vil sampleraten være 44,1 kHz eller 48 kHz. Det betyder, at UGens, der oprettes med `.ar` får beregnet deres konkrete værdi 48.000 gange pr. sekund (ved en samplerate på 48 kHz). Med et teknisk udtryk siger man, at audio rate UGens er „sample accurate". *Control rate* (`.kr`) er ikke lige så præcist som audio rate, da den for at spare på computerens regnekraft beregnes med en lavere frekvens[3], som dog er tilstrækkeligt præcis til eksempelvis at modulere en indstilling for en audio rate UGen.

Hvornår skal man så bruge disse to methods? Som tommelfingerregel kan vi følge nedenstående råd:

`.ar` Vi bruger typisk `.ar` til at oprette UGens, som indeholder et direkte hørbart signal, såsom oscillatorer, samples og digitale filtre.

`.kr` Vi bruger typisk `.kr` til at oprette UGens, som modulerer andre UGens, fx LFO'er, envelope-generatorer og triggere.

En af følgerne af, at UGens så at sige „lever" på lydserveren, er at man desværre ikke kan bruge patterns inde i UGen-funktioner. Men senere i kurset (se 5.3) kommer vi til at kombinere patterns og UGens ved at registrere vores UGen-funktioner som såkaldte **`SynthDef`**s. Så kan vi spille på UGen-funktioner ved hjælp af patterns. Forholdet mellem patterns og UGens er nemlig lidt ligesom forholdet mellem en musiker (patterns) og et instrument (UGens); Man kan godt komponere med patterns uden

[2] Der findes også en tredje „rate" i SuperCollider - `.ir`, som står for initialization rate. Den er kun relevant i nogle enkelte ydertilfælde og dækkes derfor ikke yderligere her.

[3] Når vi noterer en liste (se 1.6) som argument til en UGen, udfører SuperCollider typisk en såkaldt „multichannel expansion" til at oprette en særskilt UGen for hvert af listens elementer. Dette kigger vi nærmere på senere (se 7.1).

at bruge UGens (fx ved at spille på et andet instrument via MIDI). Man kan også godt komponere udelukkende ved hjælp af UGens (ligesom en selvkørende, modulær synthesizer). Men den særlige fordel ved platforme som SuperCollider er kombinationen af de to niveauer: Når vi bruger det righoldige pattern (se 2.1)-bibliotek sammen med vores egne UGen-lyddesign, får vi utroligt mange kompositionsmuligheder.

4.2 Skalering af signaler

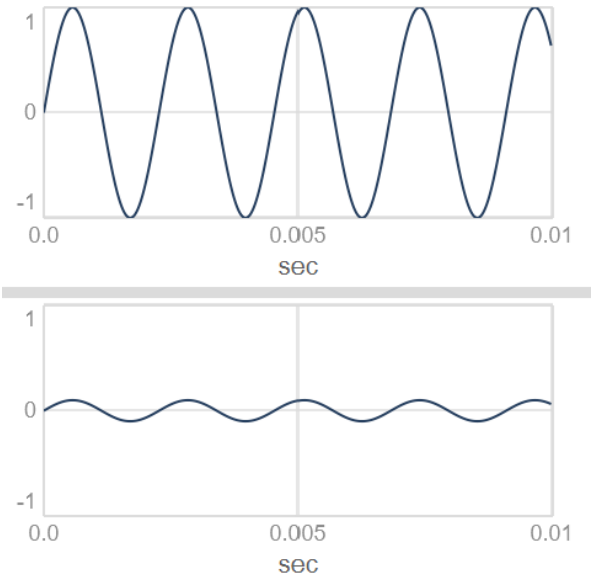
Ofte ønsker vi, at forskellige parametre ved UGens forandrer sig over tid. Som vist ovenfor kan vi gøre dette hjælp af modulation (se 4.1.2). Vi kan altid modulere outputtet fra UGens, og i mange tilfælde kan input/argumenter til UGens også moduleres på forskellig vis. I den forbindelse er det vigtigt at skalere signalerne korrekt og holde tungen lige i munden.

4.2.1 Skalering med henblik på styring af amplitude

Vi kan behandle outputtet fra en UGen på forskellige måder, blandt andet med filtre, delay-effekter, distortion med mere. I første omgang fokuserer vi her på at styre amplituden for outputtet fra en oscillator-UGen. Når vi ønsker at justere outputtet fra en UGen på denne måde, kan vi ganske enkelt gange outputtet med modulatoren. Hvis vi eksempelvis ganger outputtet fra en **SinOsc** med 0.1, nedskalerer vi amplituden.

```
{[
    SinOsc.ar(440),
    SinOsc.ar(440) * 0.1
]}.plot;
```

Kodeboks 4.9: Modulation af UGen-output



Figur 4.4: En umodificeret sinusbølge og en sinusbølge med nedskaleret amplitude

For et hørbart lydsignals amplitude er det væsentligt at bemærke, at vi bevæger os mellem -1 og 1. Udsving uden for dette interval vil skabe en uønsket, skrattende lyd. Når vi modulerer amplitude for hørbare UGens, skalerer vi derfor oftest amplituden **ned**. Det gør man ved at gange med en faktor mellem 0 og 1:

Tommelfingerregel: Outputtet fra en LFO eller envelope, der *modulerer et hørbart signals amplitude*, bør som udgangspunkt bevæge sig i intervallet 0-1.

4.2.1.1 Skalering fra 0 til maksimum med .unipolar

Hvis vi ønsker at modulere amplituden for outputtet fra en oscillator-UGen, kan vi bruge en anden UGen til dette. Vi kalder denne anden UGen for en modulator, som modulerer amplituden. Hvis vi tager udgangspunkt i tommelfingerreglen ovenfor, skal outputtet fra denne modulator bevæge sig i intervallet mellem 0 og 1. Hertil har vi en handy UGen-method, der hedder `.unipolar(maksimum)`. Den skalerer nemlig outputtet fra en UGen, så det ligger i intervallet fra 0 til et givet maksimum (default-værdien er 1).

Som eksempel kan vi tage UGen'en **`LFTri`**, der giver en trekantet bølgeform og er velegnet som lavfrekvent oscillator.

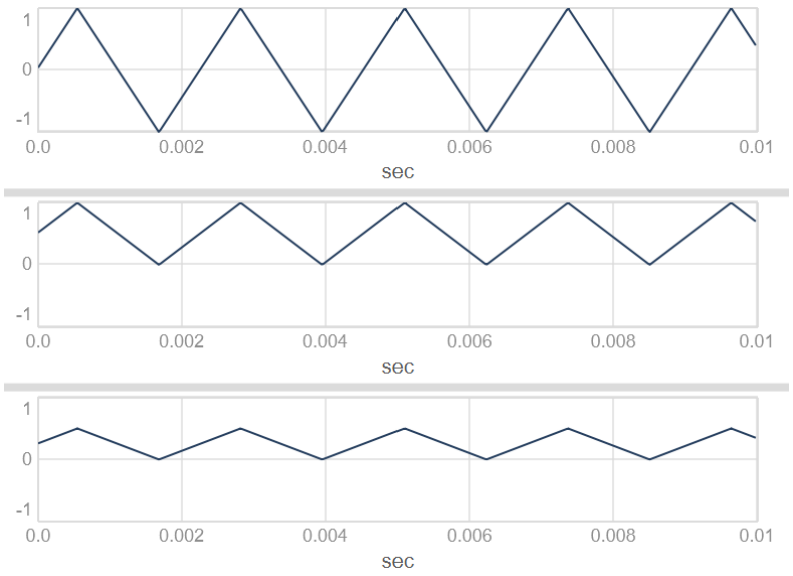
- Uden skalering bevæger **`LFTri`**`.ar` sig mellem -1 og 1, ligesom **`SinOsc`** og de fleste andre oscillatorer.

- Med `LFTri.ar.unipolar` uden argumenter bevæger vi os i stedet mellem 0 og 1.
- Med `LFTri.ar.unipolar(0.5)` bevæger vi os mellem 0 og ½.

Bemærk hvor outputtet befinder sig på Y-aksen.

```
{[
    LFTri.ar,
    LFTri.ar.unipolar
    LFTri.ar.unipolar(0.5)
]}.plot;
```

Kodeboks 4.10: En trekantet bølgeform udsat for .unipolar



Figur 4.5: Brug af UGen-method'en .unipolar

Hvis vi eksempelvis gerne vil modulere amplituden for et signal med pink støj med en trekantformet modulator, kan det gøres på følgende vis.

```
{ PinkNoise.ar * LFTri.kr(0.5).unipolar }.play;
```

Kodeboks 4.11: Pulserende, pink støj

Udforsk ved hjælp af SuperColliders dokumentation på egen hånd hvad den tilsvarende method `.bipolar` gør.

4.2.2 Skalering med henblik på justering af tonehøjde

Ofte kan det være interessant at modulere inputs til en UGen, altså at argumenter vi anvender til method'en `.ar` består af signaler fra andre UGens og dermed forandrer sig over tid. Her bør man overveje hvilke input, der giver mening for den UGen, man modulerer. Som eksempel kigger vi her på tonehøjde.

Ved en UGen, der generer en tone, skal frekvens-argumentet typisk ligge inden for den menneskelige hørelse, dvs. et sted mellem 20 Hz og 20 kHz. Noget lignende gælder cutoff-frekvenser til filtre. Her skal det signal, der modulerer inputtet, bevæge sig i et interval, der ligger inden for dette spænd.

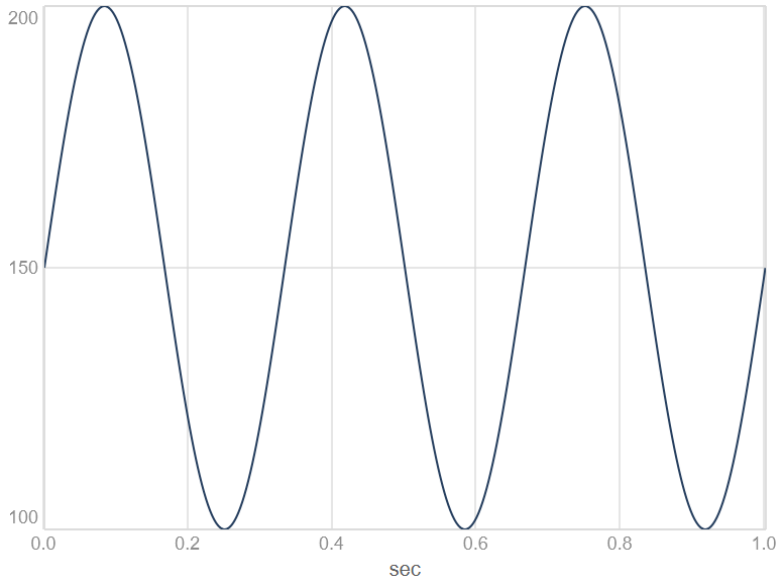
Tommelfingerregel:: Et signal, der *modulerer en oscillators frekvens/tonehøjde*, bør typisk bevæge sig inden for intervallet 20 til 20000. Det samme gælder typisk for modulation af cutoff-frekvenser ved filter-UGens.

4.2.2.1 Skalering fra minimum til maksimum med .range og .exprange

Her kan vi med fordel bruge to UGen-methods, som hedder `.range` og `.exprange`. I begge tilfælde angiver man et minimum og et maksimum, og oscillatorens output skaleres så henholdsvis lineært og eksponentielt til at bevæge sig mellem disse to værdier.

```
1 { SinOsc.ar(3).range(100, 200) }.plot(1);
```

Kodeboks 4.12: Skalering af output med .range - bemærk Y-aksen



Figur 4.6: Brug af UGen-method'en `.range`

Det er vigtigt, at vi bruger disse methods på frekvensmodulatoren (i modsætning til lydkildens amplitude). Hertil kan vi med fordel anvende lokale variabler (se 1.3.2) til at organisere kildekoden, så det er tydeligt, hvad der modulerer hvad.

```
{
    var freq = SinOsc.ar(3).exprange(100, 400);
    Pulse.ar(freq);
}.play;
```

Kodeboks 4.13: Skalering af output med .exprange

4.2.2.2 Intervaltransponering med .midiratio

Sitationer hvor man ønsker at modulere en tonefrekvens, så den bevæger sig op og ned på en tonalt velklingende måde, kan være lidt tricky. Vi tænker nemlig normalt toner i intervaller som sekunder, tertser, kvarter, kvinter, oktaver osv. Skal vi flytte et a, der klinger ved 440 Hz, en oktav op, skal vi gange med 2 - så får vi a'et ved 880 Hz. Men hvad hvis vi skal flytte en terts op? En kvarttone ned? Eller et antal cent) op eller ned?

Her kommer omregnings-method'en `.midiratio` os til undsætning. Med den kan vi omregne et interval, målt i halvtoner, til den faktor, vi skal gange en frekvens med, for at modulere præcis det ønskede antal halvtoner op eller ned.

```
// Prim, ingen skalering
0.midiratio;      // --> 1

// Oktav op
12.midiratio;     // --> cirka 2

// Oktav ned
(-12).midiratio; // --> cirka 0.5

// Lille terts op
3.midiratio;      // --> 1.189207 ...

// Kvint op
7.midiratio;      // --> cirka 1.5

// Kvarttone op
0.5.midiratio;    // --> 1.02902 ...

// 15 cent op
0.15.midiratio;   // --> 1.008702 ...
```

Kodeboks 4.14: Intervaltransponering med .midiratio 

Her er et eksempel, hvor vi anvender `.midiratio` sammen med `.unipolar` til at modulere en lille terts op.

```
{
    var freq = 440;
    var modulator = LFSaw.kr(1).unipolar(3).midiratio;
    Pulse.ar(freq * modulator);
}.play;
```

Kodeboks 4.15: .midiratio kombineret med .unipolar 

4.3 Tips og tricks til UGens

Med UGens åbner der sig en verden af muligheder for at skabe, forme og analysere lyd. Herunder finder du mine tips og tricks, der gør det lettere at eksperimentere og forstå, hvad der egentlig sker i dine signalveje.

4.3.1 Ndef – UGens' bedste ven

Når du eksperimenterer med forskellige UGens og deres indstillinger, kan det være besværligt hele tiden at skulle stoppe og starte synths manuelt. Her kommer **Ndef** ind i billedet – den fungerer på samme måde for UGens, som **Pdef** gør for patterns (se 3.4.1). Med **Ndef** kan du hurtigt ændre og genstarte din lyd uden at skulle rydde op i gamle synths eller bruge globale variabler.

```
1 Ndef(\minSynth, { SinOsc.ar(440) * 0.1 }).play;
2
3 // Prøv nu at ændre UGens eller parametre
4 Ndef(\minSynth, { Saw.ar(220) * 0.1 });
5 Ndef(\minSynth, { PinkNoise.ar * 0.1 });
```

Kodeboks 4.16: Let eksperimentering med Ndef

Når du kører koden igen med nye parametre, opdateres lyden med et kort crossfade. **Ndef** kan mange andre tricks, såsom automatisk etablering af grafiske brugerflader til test. Det overlades til nysgerrige læsere at opsøge dette emne på egen hånd.

4.3.2 Visualisér output med scope og freqscope

SuperCollider har indbyggede værktøjer til at visualisere både tids- og frekvensdomænet:

- Brug `s.scope` til at åbne et oscilloskop, der viser signalets bølgeform.
- Brug `s.freqscope` til at se signalets spektrale indhold.

```
1 { LFSaw.ar(220) * 0.1 }.play;
2 s.scope;      // Oscilloskop
3 s.freqscope;  // Frekvensanalyse
```

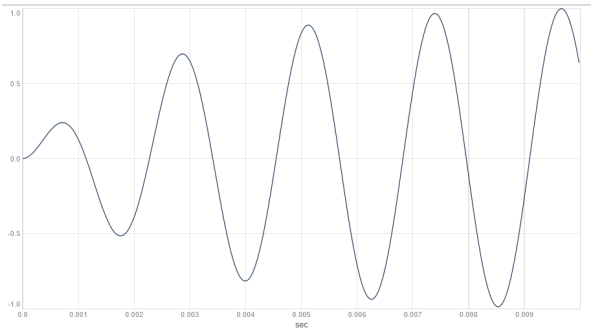
Kodeboks 4.17: Visualisering af signal

4.3.3 Plot outputtet med .plot

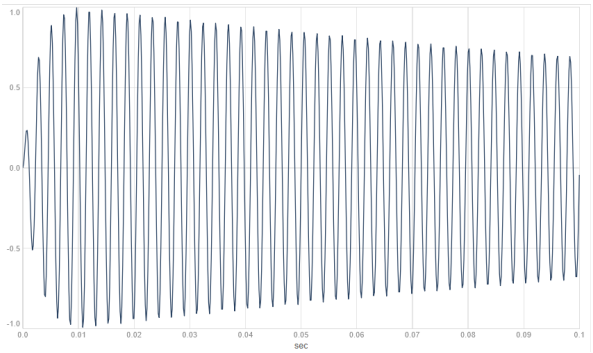
Hvis du vil se præcis, hvordan et signal eller en **UGen** udvikler sig over tid, kan du bruge `.plot`-method'en på en UGen-funktion. Dette er nyttigt til at inspicere signaler fra LFO'er, oscillatorer, og andre UGens.

```
{ SinOsc.ar * Env.perc.ar(2) }.plot;

// Plot over et andet tidsinterval med et argument
{ SinOsc.ar * Env.perc.ar(2) }.plot(0.100);
```

Kodeboks 4.18: Plot af UGen-funktioner



Figur 4.7: Plot: En sinustones amplitude moduleres af en perkussiv envelope, 10 ms



Figur 4.8: Plot: En sinustones amplitude moduleres af en perkussiv envelope, 100 ms

4.3.4 Følg værdierne med .poll

Når du vil følge med i, hvilke værdier en UGen genererer i realtid, kan du bruge UGen-method'en `.poll`. Den måler et signal og skriver dets værdi i SuperColliders post window, så du kan se, hvad der sker undervejs.

```
{ LFNoise1.kr(1).poll }.play;
// UGen(LFNoise1): -0.86944
// UGen(LFNoise1): -0.695801
// UGen(LFNoise1): -0.522162
// UGen(LFNoise1): -0.348523
```

Kodeboks 4.19: Overvågning af signal med .poll

Du kan også give `.poll` en label, så du lettere kan skelne mellem forskellige signaler.

```
{ SinOsc.kr(0.5).poll(label: "LFO:") }.play;
```

Kodeboks 4.20: Poll med label

Endelig kan du ændre opdateringsfrekvensen, hvis der er behov for at justere yderligere.

```
// Gør outputtet genkendeligt
{ SinOsc.kr(0.5).poll(label: "LFO: ") }.play;

// Opdater 5 gange pr. sekund
{ SinOsc.kr(0.5).poll(trig: 5) }.play;
```

Kodeboks 4.21: .poll-argumenter

4.4 Cheat sheet: Centrale UGens

Du kommer sandsynligvis primært til at bruge UGens fra dette cheat sheet i det første lange stykke tid af dine SuperCollider-studier. Det er en god ide at bruge SuperColliders dokumentation samt oscilloskop og spektrogram (se 4.1) til at nærstudere de forskellige UGens. Derefter kan du med fordel også orientere dig i dokumentet *Tour of UGens* fra SuperColliders dokumentation.

4.4.1 Almindelige oscillatorer og støjgeneratorer

Fire almindelige bølgeformer: Sinusbølge, savtakket, firkantet og trekantet. **SinOsc**, **Saw** og **Pulse** er i modsætning til **LFTri** „band limited". Det betyder, at de ikke skaber „aliasing", som er en form for digital støj, når de anvendes ved høje frekvenser.

```
// Sinusbølger, savtakkede og firkantede bølger:
{SinOsc.ar(440) * 0.1}.play;
{Saw.ar(440) * 0.1}.play;
{Pulse.ar(440) * 0.1}.play;
{LFTri.ar(440) * 0.1}.play;
```

Kodeboks 4.22: Oscillator-UGens til basale bølgeformer

Pulse og den beslægtede **VarSaw** kan begge indstilles med hensyn til duty cycle, dvs. symmetri/assymmetri i bølgeformen.

```
{VarSaw.ar(440, 0.1) * 0.1}.play;
{VarSaw.ar(440, 0.5) * 0.1}.play;
{Pulse.ar(440, 0.1) * 0.1}.play;
{Pulse.ar(440, 0.5) * 0.1}.play;
```

Kodeboks 4.23: Oscillator-UGens til bølgeformer med variabel duty cycle

Der findes også forskellige støjgeneratorer, hvoraf de mest almindeligt anvendte nok er hvid og pink støj.

```
{WhiteNoise.ar * 0.01}.play;
{PinkNoise.ar * 0.1}.play;
```

Kodeboks 4.24: UGens til hvid og pink støj 

4.4.2 LFO-egnede UGens

Disse UGens (samt **SinOsc**) er velegnede som LFO'er. Men de kan sagtens benyttes over 20 Hz.

```
// Savtakket, trekantet/pyramideformet og firkantede bølgeformer.
{LFSaw.kr(10)}.plot(0.2);
{LFTri.kr(10)}.plot(0.2);
{LFPulse.kr(10)}.plot(0.2);

// Lavfrekvent støj, velegnede som tilfældighedsgeneratorer.
{LFNoise0.kr(10)}.plot(1);
{LFNoise1.kr(10)}.plot(1);
{LFNoise2.kr(10)}.plot(1);
```

Kodeboks 4.25: LFO-egnede UGens 

4.4.3 Envelope-generatorer

Disse UGens gennemgås i kapitlet om envelopes (se 5.1).

```
// Line og XLine er nok de mest enkle envelopes, bestående af ét segment, lige eller eksponentiel linje.
{Line.kr(0, 1, 4, doneAction: 2) * PinkNoise.ar * 0.1}.play;
{XLine.kr(0.01, 1, 4, doneAction: 2) * PinkNoise.ar * 0.1}.play;

// EnvGen er en envelope-generator-UGen, som kan bruges sammen med forskellige envelopes:
{EnvGen.kr(Env.perc, doneAction: 2) * PinkNoise.ar * 0.1}.play;
{EnvGen.kr(Env.sine, doneAction: 2) * PinkNoise.ar * 0.1}.play;
{EnvGen.kr(Env.triangle, doneAction: 2) * PinkNoise.ar * 0.1}.play;
```

Kodeboks 4.26: Envelope-generator-UGens 

4.4.4 Filtre

Læs nærmere om anvendelsen i kapitlet om filterbaseret klangdannelse (se 6.1).

```
// Low-, high-, og band pass-filtre.
{LPF.ar(PinkNoise.ar, 440) * 0.1}.play;
{HPF.ar(PinkNoise.ar, 440) * 0.1}.play;
{BPF.ar(PinkNoise.ar, 440) * 0.1}.play;

// Low- og high pass-filtre med resonans.
{RLPF.ar(PinkNoise.ar, 440, 0.01) * 0.1}.play;
{RHPF.ar(PinkNoise.ar, 440, 0.01) * 0.1}.play;

// Shelf-filtre.
{BLowShelf.ar(PinkNoise.ar, 440, db: 20) * 0.05}.play;
{BHiShelf.ar(PinkNoise.ar, 440, db: 20) * 0.05}.play;

// Low pass-filter inspireret af filtrene i de klassiske
↪  Moog-synthesizere
{MoogFF.ar(PinkNoise.ar, 440) * 0.1}.play;
```

Kodeboks 4.27: Filter-UGens

4.4.5 Triggere og delays

Triggere genererer impulser. En impuls bruges typisk til at udløse noget, fx igangsætning af en envelope-generator, spring i afspilningsposition eller lignende. Triggere anvendes blandt andet i forbindelse med granular syntese (se 9.1.3).

```
// Impulse og Dust genererer impulser, regelmæssigt eller
↪  tilfældigt fordelt i tid.
{Impulse.ar(10) * 0.1}.play;
{Dust.ar(10) * 0.1}.play;

// DelayN, DelayL, DelayC er delay uden feedback, dvs. de
↪  forsinker blot lydsignalet.
{DelayL.ar(Impulse.ar(0), 1, 1) * 0.1}.play;

// CombN, CombL, CombC - såkaldt "comb filter", fungerer som
↪  delay med feedback.
{CombL.ar(Impulse.ar(0), decaytime: 10) * 0.1}.play;
```

Kodeboks 4.28: Trigger- og delay-UGens

4.4.6 Samples

Disse UGens gennemgås i kapitlet om samplebaseret komposition (se 8.1.2).

```
// Kør først denne linje for at indlæse et sample i en buffer:
~sample = Buffer.read(s, Platform.resourceDir +/+
↪  "sounds/a11wlk01.wav");

// PlayBuf og BufRd - afspiller indholdet af en buffer:
{PlayBuf.ar(1, ~sample, doneAction: 2)}.play;
{BufRd.ar(1, ~sample, SinOsc.ar(0.1).range(0,
↪  BufFrames.kr(~sample)))}.play;
```

Kodeboks 4.29: UGens til afspilning af samples 

4.4.7 Routing og stereofoni

Læs nærmere herom i kapitlet om klangdannelse med oscillatorbanke (se 7.1).

```
// Pan2 - placerer et signal i et stereofelt.
{Pan2.ar(SinOsc.ar, 0.5) * 0.1}.play;

// Balance2 - justerer forholdet mellem to kanaler (venstre og
↪  højre).
{Balance2.ar(SinOsc.ar(440), SinOsc.ar(443)) * 0.1}.play;

// Splay - fordeler et vilkårligt antal kanaler ligeligt i et
↪  stereofelt
{Splay.ar(SinOsc.ar([220, 440, 660, 880])) * 0.1}.play;

// Out - sender signal ud af en Synth.
{Out.ar(1, SinOsc.ar * 0.1)}.play;
```

Kodeboks 4.30: UGens til routing og stereofoni 

4.5 Cheat sheet: UGen-methods

Når vi arbejder med UGens og eksempelvis bruger signalet fra én UGen til at modulere en anden, er det nyttigt at kende til disse methods til oprettelse og justering af UGens.

```
// .ar - generer lydsignal (audio rate), anvendes typisk til lydkilder, filtre og routing-UGens
{SinOsc.ar * 0.1}.play;

// .kr - generer kontrolsignal (control rate), anvendes typisk til LFO'er og envelopes.
{SinOsc.kr(1)}.plot(1);

// .range og .exprange - justerer outputtet fra en UGen, så det ligger mellem et nyt min og max
{SinOsc.ar.range(50, 1000)}.plot;     // bemærk Y-aksen
{SinOsc.ar.exprange(50, 1000)}.plot;  // bemærk Y-aksen og bølgeform

// .unipolar og .bipolar - genveje til at skrive .range(0, x) og .range(-x, x)
{SinOsc.ar.unipolar(100)}.plot;       // bemærk Y-aksen, signalet går fra 0 til 100
{SinOsc.ar.bipolar(100)}.plot;        // bemærk Y-aksen, signalet går fra -100 til 100

// .dup - kopierer et signal, så det bliver til et multikanals-signal
{SinOsc.ar(440).dup * 0.1}.play;      // to kanaler
{SinOsc.ar(440).dup(10) * 0.1}.play;  // 10 kanaler
```

Kodeboks 4.31: Essentielle UGen-methods

4.6 Øvelse: Brug af UGens

I denne øvelse øver man sig i at anvende, visualisere og modulere UGens.

4.6.1 Blip båt

Afspil følgende lyde med peak-amplitude på 0.1:

1. Sinustone med frekvens på 220 Hz.
2. Savtakket bølgeform med frekvens på 100 Hz.
3. En firkantet bølgeform med frekvens på 350 Hz.

```
{SinOsc.ar(   ) * 0.1}.play;
{     * 0.1}.play;
{     * 0.1}.play;
```

Kodeboks 4.32: Oscillatorfrekvenser

4.6.2 Visualisering af bølgeform

SuperCollider kan plotte lyd-outputtet i en graf. Dette viser eksempelvis outputtet fra en `SinOsc` og en `LFTri`, målt over et stardard-tidsrum på 10 millisekunder.

```
{[
    SinOsc.ar(440),
    LFTri.ar(440)
]}.plot(0.010);
```

Kodeboks 4.33: Visualisering af bølgeform

Brug `{}.plot`-teknikken ligesom ovenfor til at overveje følgende spørgsmål:

1. Hvad er forskellen på `Pulse.ar(width: 0.5)` og `Pulse.ar(width: 0.1)`?
2. Hvad er forskellen på `LFNoise0.kr(2)` og `LFNoise1.kr(2)`?

Tip: Når vi skal visualisere LFO-UGens, der som her svinger ved 2 Hz, får vi ikke meget syn for sagen ved at plotte over de 10 millisekunder, der som udgangspunkt anvendes med `.plot`. Plot i stedet over fx 3 sekunder med `{}.plot(3)`.

4.6.3 Visualisering af frekvensspektrum

Vi kan vise SuperColliders lydlige output i frekvensdomænet med `s.freqscope;`

Brug `s.freqscope` til at besvare følgende spørgsmål:

1. Hvad kendetegner overtonespektrene for de forskellige bølgeformer fra opgaven *Blip båt* ovenfor?
2. Hvad styrer bevægelse af musen fra venstre til højre i nedenstående eksempel? Samt: Hvad kan vi herudfra konkludere om UG'enen **`Blip`**?

```
1 { Blip.ar(220, MouseX.kr(1,50)) * 0.1 }.play;
```

Kodeboks 4.34: Undersøgelse af UGen'en Blip

Har du mange ekstra vinduer åbne (fx fra ovenstående øvelse med plots), kan de lukkes på én gang med **`Window`**`.closeAll`.

4.6.4 Modulation af lydstyrke (amplitude)

Anvend følgende UGens, [skaleret med method'en `.unipolar`](a-skalering.md), til at modulere amplituden i intervallet 0-1 for en savtakket bølgeform. Modulatorens frekvens vælges frit i intervallet 0-20 Hz.

1. **`LFSaw`** (savtakket bølgeform)
2. **`LFPulse`** (firkantet bølgeform)
3. **`SinOsc`** (sinusbølge)

```
1 {Saw.ar *   }.play;
```

Kodeboks 4.35: Amplitudemodulation med LFO

4.6.5 Modulation af tonehøjde (frekvens)

Når vi modulerer frekvens, er det typisk nødvendigt at justere modulatorens frekvens og outputrækkevidde. `.range` og `.exprange` er oplagte redskaber at styre

en absolut modulation, og `.midiratio` (evt. kombineret med `.unipolar` eller `.bipolar`) er oplagt til at styre en relativ modulation af tonehøjde. Læs evt. nærmere herom i afsnittet om skalering (se 4.2).

Modulér frekvensen for en savtakket oscillator med følgende UGens og på følgende måder:

1. Rutsjebane
   a) Brug **`LFTri`** som modulator.
   b) Tonens frekvens skal bevæge sig mellem 440 Hz og 880 Hz.
   c) Valgfri modulatorfrekvens under 20 Hz.
2. Tonespring
   a) Brug **`LFPulse`** som modulator.
   b) Tonens frekvens skal bevæge sig mellem 220 Hz og 330 Hz.
   c) Valgfri modulatorfrekvens under 20 Hz.
3. Vibrato
   a) Brug **`SinOsc`** som modulator.
   b) Tonens frekvens på 660 Hz skal moduleres 15 cent op og ned.
   c) Modulatorfrekvensen skal være 15 Hz.
4. Tilfældige frekvenser
   a) Brug **`LFNoise0`** som modulator.
   b) Tonens frekvens på 440 Hz skal moduleres i halvtonetrin op til en oktav op og ned.
   c) Modulatorfrekvensen skal være 8 Hz.

Vælg selv en passende frekvens mellem 0 Hz og 20 Hz til modulatoren.

```
{
    var modulator =   ;
    Saw.ar(modulator) * 0.1;
}.play;
```

Kodeboks 4.36: Frekvensmodulation

4.6.6 Bonusopgave: FM, AM, RM

Har du mod på at gå yderligere på opdagelse i modulationens potentialer, kan du med fordel dykke ned i nogle mere avancerede teknikker, hvor modulation på forskellig vis udgør kernen i klangdannelsen. Når modulation bevæger sig fra det

lavfrekvente (under 20 Hz) til det hørbare frekvensbånd, sker der ganske interessante klanglige variationer.

1. Eksperimentér med eksemplerne på AM (Amplitude Modulation), RM (Ring Modulation) og FM (Frequency Modulation) fra kapitel 7 i Thor Magnussons *Scoring Sound* (2021).
2. Eksperimentér med eksemplerne på FM fra Eli Fieldsteels glimrende video om emnet. Kildekoden fra videoen findes på github.

KAPITEL 5

Envelopes og SynthDefs

Lydens forandring over tid er en vigtig del af lyddesign. Et af de vigtigste redskaber til at arbejde med lydlig forandring over tid er envelopes. Envelopes anvendes i elektronisk klangdannelse typisk til at styre en tone eller en lyds volumen over tid, men envelopes kan med fordel bruges på mange andre måder til at skabe forandring i lyden over tid. Dette kapitel introducerer til brug og design af envelopes i SuperCollider. Der introduceres også til SynthDefs, som er et centralt redskab til at lave UGen-funktioner til fleksible opskrifter på lyddesign, der kan anvendes sammen med patterns.

5.1 Brug af envelopes

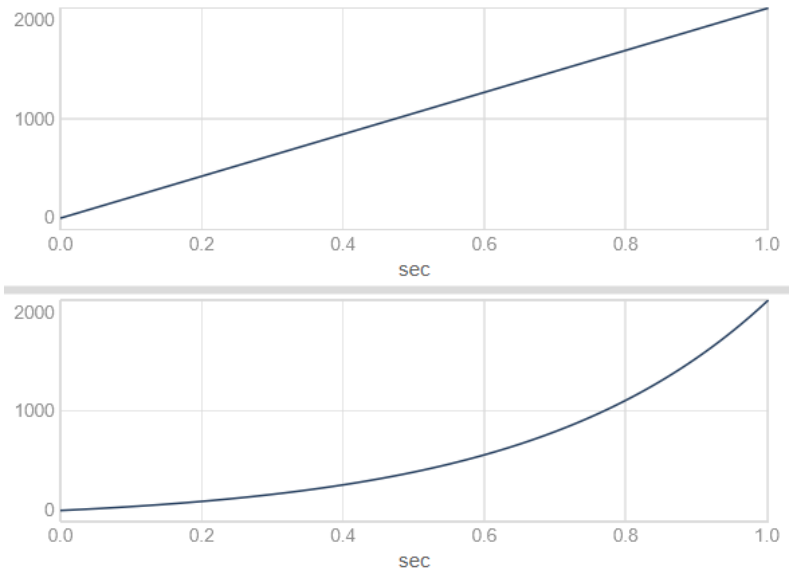
Hvor mange synthesizere kun har én envelope, typisk af typen ADSR (attack-decay-sustain-release), har SuperCollider en række forskellige, indbyggede envelopes. Man kan også definere sine egne envelopes. Det er endda muligt at loope envelopes, så de kommer til at udgøre LFO'er. Dermed kan envelopes potentielt være et særdeles kreativt virkemiddel.

5.1.1 Simple linjer

De mest enkle envelope-generatorer er **`Line`** og **`XLine`** - UGens, som genererer en henholdsvis lineær og eksponentiel udvikling fra ét punkt til et andet over et specificeret tidsrum. Herunder er et eksempel, hvor envelopen bevæger sig fra 100 til 2000 i løbet af 1 sekund.

```
1 Line.kr(100, 2000, 1)
2 XLine.kr(100, 2000, 1)
```

Kodeboks 5.1: Line og XLine



Figur 5.1: Line og XLine

Vi bruger **`Line`** og **`XLine`** ligesom andre UGens, fx til at styre frekvensen for en oscillator.

```
// lineær udvikling over 1 sekund
{SinOsc.ar(Line.kr(100, 800, 1)) * 0.1}.play;

// eksponentiel udvikling over 5 sekunder
{SinOsc.ar(XLine.kr(100, 800, 5)) * 0.1}.play;

// eksponentiel udvikling over 50 millisekunder
{SinOsc.ar(XLine.kr(100, 800, 0.050)) * 0.1}.play;
```

Kodeboks 5.2: Line og XLine over forskellige tidsintervaller

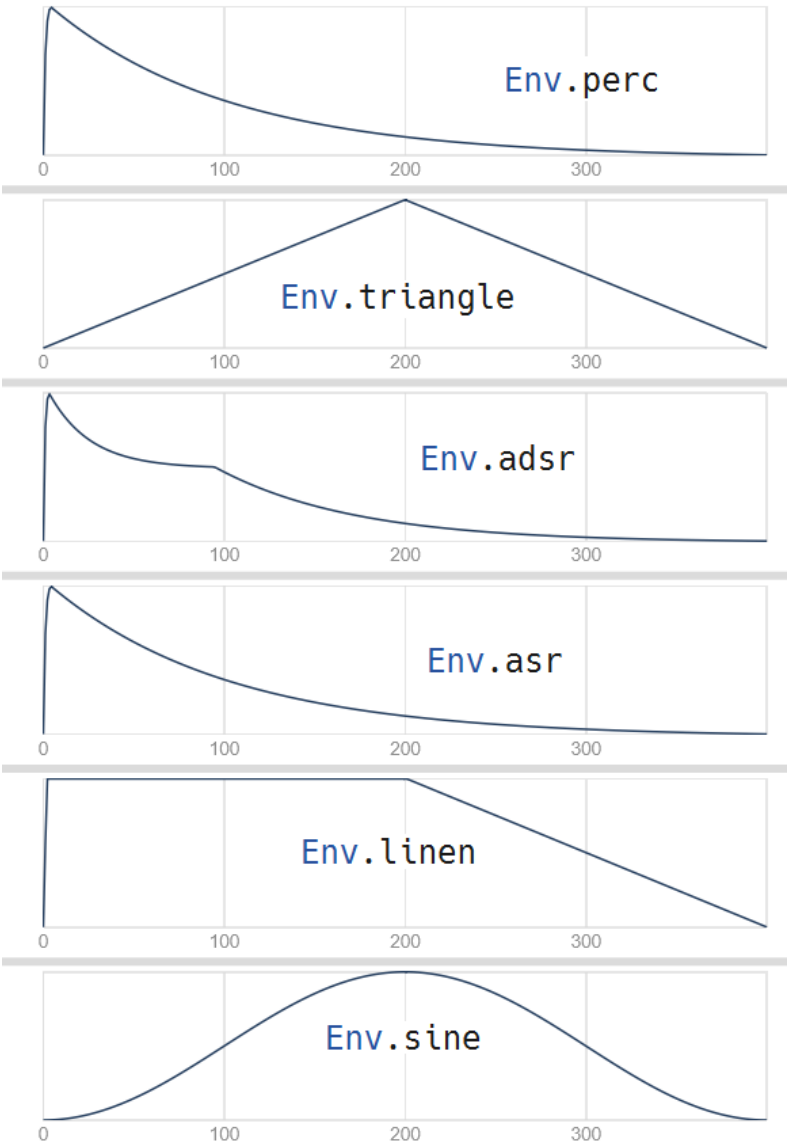
5.1.2 Envelopes for enhver smag

`Line` og `XLine` genererer envelopes med ét segment (dvs. ét tidsinterval med ét start- og ét slutpunkt). Envelopes har imidlertid typisk mere end ét segment. Og de forskellige segmenter kan have meget forskellige former/„krumninger“.Vi bruger `Env`-klassen til at definere disse envelopes. Herunder ses fx nogle forskellige indbyggede envelopes.

```
Env.perc;
Env.triangle;
Env.adsr;
Env.asr;
Env.linen;
Env.sine;
```

Kodeboks 5.3: Indbyggede envelopes

Vi kan vise en grafisk repræsentation med `.plot` - fx `Env.perc.plot`. Herunder ses et plot af de ovennævnte envelopes.



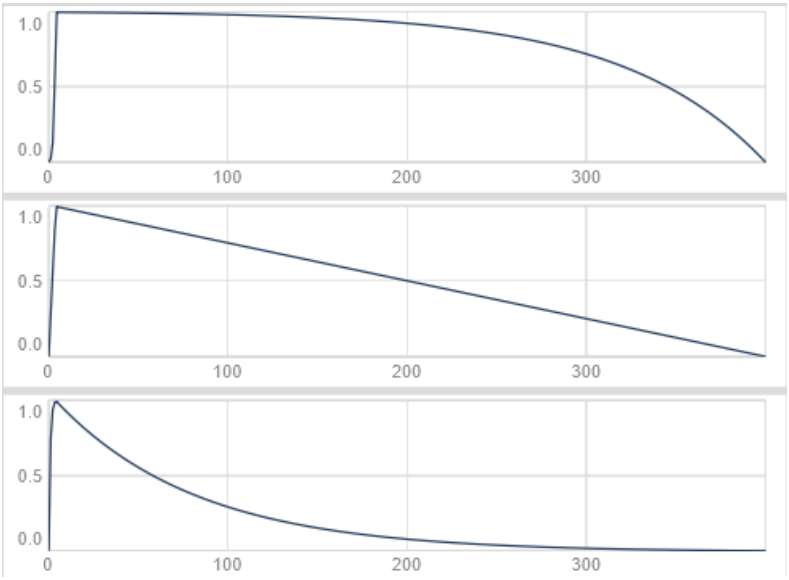
Figur 5.2: Forskellige standardenvelopes

5.1.2.1 Specifikation af segmentvarigheder og krumning

Lad os kigge nærmere på et eksempel - den simple envelope `Env.perc`.Den har to segmenter kaldet attack og release, og vi kan specificere deres varighed med argumenter. Herunder angiver vi en attack-tid på 1 sekund og en release-tid på 4 sekunder.

```
Env.perc(1, 4);

// Vi kan også vælge blot at justere fx release-segmentet:
Env.perc(releaseTime: 10);

// Segmenternes krumning kan justeres med argumentet curve:
[Env.perc(curve: 5), Env.perc(curve: 0), Env.perc(curve:
↪ -5)].plot;
```

Kodeboks 5.4: En perkussiv envelope



Figur 5.3: Forskellige krumningsværdier til envelopesegmenter: 5, 0 og -5

5.1.2.2 Brug af envelope

Når vi skal bruge en envelope som `Env.perc` i vores lyddesign, skal vi tage højde for, at `Env` blot specificerer en envelope-form - en `Env` er ikke en UGen. For at bruge `Env`-baserede envelopes skal vi anvende `EnvGen`, som er en envelope-generator-UGen. Vi fortæller `EnvGen`, at vi ønsker en `Env.perc` ved at angive den som første argument: `EnvGen.kr(Env.perc)`.

Vi kan nu bruge `EnvGen` ligesom `Line` og `XLine` ovenfor - fx til at modulere lydstyrken for en oscillator. Som nævnt i forbindelse med skalering (se 4.2) gør vi dette ved ganske enkelt at gange outputtet fra oscillatoren med outputtet fra envelope-generatoren.

```
{PinkNoise.ar * EnvGen.kr(Env.perc) * 0.1}.play;
```

Kodeboks 5.5: EnvGen og Env.perc

Inden vi går videre, er det vigtigt at skrive sig bag øret, at `EnvGen` ofte noteres implicit (skjult). De to nedenstående linjer har præcis samme resultat, og man kan selv vælge hvilken form man foretrækker.

```
{PinkNoise.ar * EnvGen.kr(Env.perc)}.play;
{PinkNoise.ar * Env.perc.kr}.play;
```

Kodeboks 5.6: Implicit EnvGen

Det er ofte nyttigt at skille disse elementer ad på forskellige linjer og bruge lokale variabler.

```
{
    var env = EnvGen.kr(Env.perc);
    var sig = PinkNoise.ar;
    sig * env;
}.play;
```

Kodeboks 5.7: Envelope som lokal variabel

Når vi gemmer envelope-generatoren under en lokal variabel, kan vi efterfølgende bruge envelope-signalet til flere forskellige formål. Fx kan vi styre både tonehøjde og lydstyrke med den samme envelope, således at flere parametre udvikler sig over tid i takt med hinanden. Dette kan give anledning til yderst interessante lyddesign, alt efter hvilke parametre man modulerer med envelopen. Lad os tage et eksempel.

```
{
    // Envelopen oprettes og gemmes under den lokale variabel env
    var env = EnvGen.kr(Env.perc(0.1, 5));
    // Envelopesignalet skaleres med .exprange til en mere passende rækkevidde for tonehøjde
    // Resultatet gemmes under variablen freq, som bruges til oscillatoren Pulse
    var freq = env.exprange(440, 880);
    var sig = Pulse.ar(freq);
    // Det umodificerede envelopesignal anvendes til at styre lydstyrken
    sig * env * 0.1;
}.play;
```

Kodeboks 5.8: Brug af envelope til modulation af flere parametre

5.1.3 Standardenvelopes

Ud over `Env.perc` og `Env.new`/`Env.circle` har vi adgang til en række standardenvelopes. Her kan vi skelne mellem to slags envelopes: *Selvafsluttende envelopes*, hvor vi kender eller angiver varigheden på forhånd (som ved `Env.perc`), og *vedvarende envelopes*, hvor varigheden ikke kendes på forhånd men fx afgøres af hvor længe, man holder en tangent nede på et MIDI-keyboard (her er `Env.adsr`, der svarer til den gængse ADSR-envelope, et oplagt eksempel).

5.1.3.1 Selvafsluttende envelopes (uden gate)

Selvafsluttende envelopes som `Env.perc` varer præcis den tid det tager at gennemløbe alle envelopens segmenter. Envelopens varighed er fast og afhænger ikke af en såkaldt gate. Det gælder blandt andet disse standardenvelopes:

- `Env.perc`
- `Env.triangle`
- `Env.linen`
- `Env.sine`

5.1.3.2 Vedvarende envelopes (med gate)

Vedvarende (sustaining) envelopes bliver hængende på et bestemt punkt imellem to segmenter, indtil de bliver bedt om at gå videre. Dette kender vi fra keyboards,

hvor tonen begynder at klinge, når vi trykker tangenten ned, og fortsætter indtil vi slipper tangenten igen. Måden hvorpå vi beder envelopen om at fortsætte til næste segment er ved at bruge en såkaldt gate. Centrale eksempler er de følgende envelopes:

- `Env.asr`
- `Env.adsr`
- `Env.cutoff`
- `Env.dadsr`

Når vi bruger **Pbind** og patterns til at styre artikulationen af toner, styres åbning og lukning af gates til envelopes automatisk. Vi kan dog sagtens anvende vedvarende envelopes med gates manuelt ved at indføre et gate-argument til vores UGen-funktion og angive det som argument nr. 2 til **EnvGen**. Derefter kan vi bruge method'en `.set` til at justere på indstillingen på et vilkårligt tidspunkt.

```
1 // Start en tone med åben gate (gate = 1)
2 ~tone = {arg gate = 1; SinOsc.ar * EnvGen.kr(Env.asr,
  ↪  gate);}.play;
3 // Vent lidt, før vi går videre til release-segmentet
4 ~tone.set(\gate, 0);
```

Kodeboks 5.9: Simpel ASR-envelope med gate

Du kan læse nærmere om brug af gate i vedvarende envelopes sammen med patterns, når vi kommer til dette emne i forbindelse med gennemgang af SynthDefs (se 5.3.4).

5.1.4 Automatisk oprydning med doneAction

Envelopes er forbundet med noget, der hedder `doneAction`, som angår hvad SuperColliders lydserver skal gøre med vores UGen-funktion, når envelope-generatoren har gennemløbet alle envelopens segmenter. I eksemplerne ovenfor har vi ikke bedt SuperCollider om at gøre noget særligt, når envelopen er slut. Men hvis vores UGen-funktion stopper med at producere lyd, når envelopegeneratoren har gennemløbet alle envelopens segmenter, bør vi faktisk fjerne klyngen af UGens i vores UGen-funktion (der betegnes en **Synth**) fra serveren igen.

Vi kan fjerne alle kørende **Synth**s fra lydserveren ved at taste Ctrl/Cmd-Punktum. Men det er ikke praktisk at gøre manuelt, så hvordan indretter vi en UGen-funktion,

så den kan fjerne sig selv automatisk, når en envelope er færdiggjort? Det gør vi ved hjælp af envelope-generatorens såkaldte `doneAction`-argument.

Vi kan få vist aktive **`Synth`**s på lydserveren ved at åbne et vindue, der viser serverens „Node Tree". Det kan vi gøre ved at køre `s.plotTree;` eller ved at klikke på de grønne tal nederst til højre i SuperColliders IDE og vælge „Show Node Tree". Sammenlign med et vågent øje på Node Tree-vinduet disse to eksempler og bemærk hvilken forskel `doneAction: Done.freeSelf` gør.

```
// Åben først Node Tree-vinduet
s.plotTree;

{PinkNoise.ar * EnvGen.kr(Env.perc) * 0.1}.play;
{PinkNoise.ar * EnvGen.kr(Env.perc, doneAction: Done.freeSelf) *
↪  0.1}.play;
```

Kodeboks 5.10: Visning af Synths på lydserveren

Hvornår skal man så bruge `doneAction: Done.freeSelf`? Jo, hvis man har gang i flere forskellige envelopes inden for den samme UGen-funktion (hvilket man sagtens kan have i SuperCollider), er det som tommelfingerregel en god ide *at bruge* `doneAction: Done.freeSelf` *til den envelope, som styrer tonens lydstyrke over tid*. Så undgår vi at få ophobet gamle **`Synth`**-nodes på lydserveren.

Hvis du er forvirret over forholdet mellem UGens, **`Synth`** osv., så er det helt i orden på nuværende tidspunkt. Det væsentlige her er blot at du forstår hvad `doneAction` betyder. Resten uddybes senere i dette kapitel i forbindelse med interfacet mellem Synth, SynthDef og Pbind (se 5.3.3).

5.1.4.1 Hvad er doneAction?

`doneAction: 2` og `doneAction: Done.freeSelf` betyder det samme - at Synth'en skal fjernes fra lydserveren, når envelopen er slut. Når vi noterer **`EnvGen`** er `doneAction` argument nr. 2, så man behøver ikke ekspicitere argumentets navn. Derfor giver alle fire kodelinjer herunder præcis samme resultat.

```
1 {PinkNoise.ar * EnvGen.kr(Env.perc, doneAction: Done.freeSelf) *
  ↪  0.1}.play;
2 {PinkNoise.ar * EnvGen.kr(Env.perc, Done.freeSelf) * 0.1}.play;
3 {PinkNoise.ar * EnvGen.kr(Env.perc, doneAction: 2) * 0.1}.play;
4 {PinkNoise.ar * EnvGen.kr(Env.perc, 2) * 0.1}.play;
```

Kodeboks 5.11: Varianter over doneAction: 2

Om man bruger `doneAction: 2` eller `doneAction: Done.freeSelf` er helt valgfrit. Førstnævnte er kortest at skrive, men sidstnævnte er umiddelbart lettest at forstå, når man læser koden.

5.2 Specialdesignede envelopes og LFO'er

Ud over de indbyggede envelopes tilbyder SuperCollider rig mulighed for at opfinde nye envelopes. Hvis vi looper en envelope, kan vi endda defineres vores helt egne former til specialdesignede LFO'er!

5.2.1 Design dine egne envelopes med Env.new

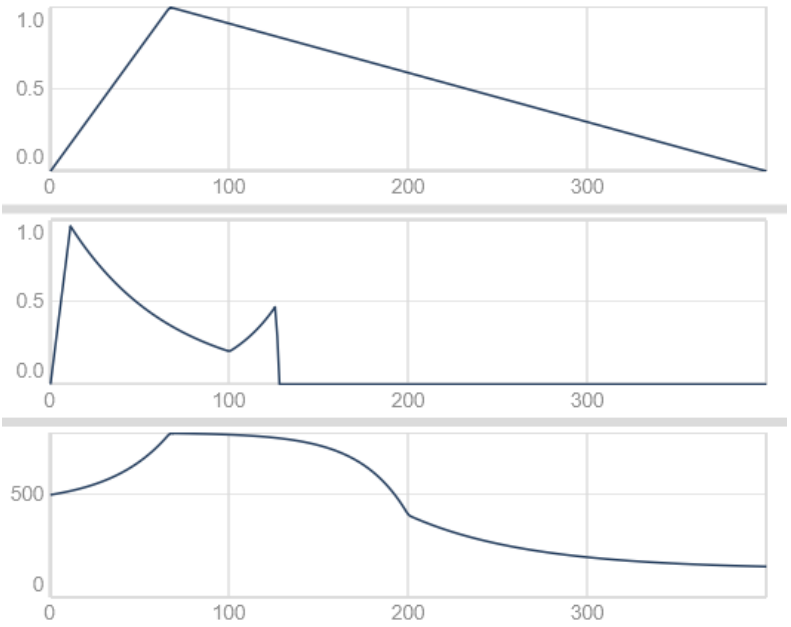
Vi definerer vores egne envelopes med `Env.new`. Argumenterne er som følger:

- Først angiver vi en liste med start- og slutniveauer for de enkelte segmenter.
- Dernæst angiver vi en liste med varigheder af de enkelte segmenter.
- Valgfrit: Til sidst kan vi bestemme segmenternes krumning ved at angive en eller flere såkaldte curve-værdier.

Her er et par eksempler.

```
1 Env.new([0, 1, 0], [0.2, 1]);
2 Env.new([0, 1, 0.2, 0.5, 0], [0.1, 1, 0.3, 3], \exp);
3 Env.new([500, 800, 400, 150], [1, 2, 3], [2, 5, -3]);
```

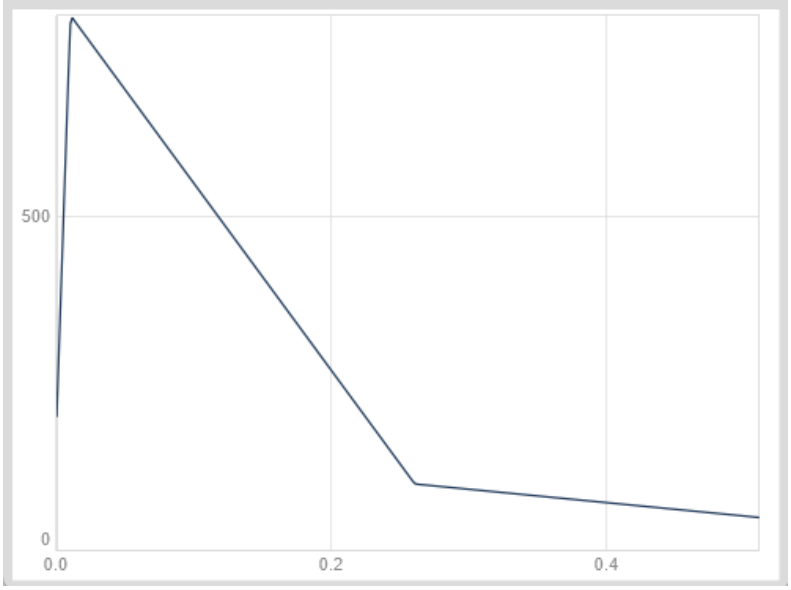
Kodeboks 5.12: Hjemmestrikkede envelopes 



Figur 5.4: Tre hjemmestrikkede envelopes

Lad os definere en ny envelope med tre segmenter, som vi vil bruge til at styre frekvensen for en oscillator:

- Først går vi fra 200 til 800, derefter til 100, og til sidst til 50.
- Første segment varer 10 millisekunder, de sidste to segmenter varer 250 millisekunder hver.



Figur 5.5: Envelope til styring af oscillatorfrekvens

```
( // Definér envelopen og gem den under en variabel
~frekvensEnvelope = Env.new(
    [200, 800, 100, 50], // niveauer
    [0.010, 0.250, 0.250]   // segment-varigheder
);
// Vis en grafisk repræsentation af envelopen
~frekvensEnvelope.plot;
)

( // Brug envelopen til at styre frekvens for en oscillator
{
    var env = EnvGen.kr(~frekvensEnvelope);
    SinOsc.ar(env) * 0.1;
}.play;
)
```

Kodeboks 5.13: En envelope til at modulere oscillatorfrekvens

Lydeksemplet her fungerer bedst med højttalere eller hovedtelefoner, som kan gengive dybe frekvenser med en vis styrke (dvs. ikke laptophøjttalere eller ear buds).

5.2.2 Envelope som LFO

En af de spændende muligheder med envelopes er, at de kan anvendes som LFO'er! Med `Env.circle`, har vi mulighed for at loope envelopes, så de fungerer som en slags oscillatorer.

`Env.circle` har næsten præcis samme syntaks som `Env.new`, vi skal blot tilføje en enkelt varighed til listen med segmentvarigheder - nemlig den tid det tager at vende tilbage til begyndelsen af envelopen, når vi looper.

```
{ // Eksemplet fra ovenfor, tilpasset Env.circle med et ekstra tidsinterval (0.24)
    var env = EnvGen.kr(
        Env.circle(
            [200, 800, 100, 50],
            [0.01, 0.25, 0.25, 0.24]
        )
    );
    SinOsc.ar(env) * 0.1;
}.play;
```

Kodeboks 5.14: Envelope som LFO

Vi kan endda modulere segment-varighederne med en LFO eller en anden envelope ved hjælp af timeScale-argumentet i `EnvGen`. Som eksempel kan vi bruge en simpel `Line` til at skalere varighederne over 10 sekunder fra en faktor 6 til en faktor 0.2.

```
{
    var env = EnvGen.kr(
        Env.circle(
            [200, 800, 100, 50],
            [0.01, 0.25, 0.25, 0.24]
        ),
        timeScale: Line.kr(6, 0.2, 10));
    SinOsc.ar(env) * 0.1;
}.play;
```

Kodeboks 5.15: Modulation af segmentvarigheder

5.3 Konstruktion af SynthDefs

Hidtil har vores lyddesign været enkle og midlertidige - med `{}.play` kan vi hurtigt teste UGens og ideer. Men vi kan gøre vores lyddesign meget mere fleksibelt og anvendeligt til længere patternbaserede kompositionsforløb ved at samle vores UGens i en såkaldt **SynthDef**.

5.3.1 Hvad er en Synth?

Bemærk hvad der bliver vist i SuperColliders post window, når vi kører nedenstående linje.

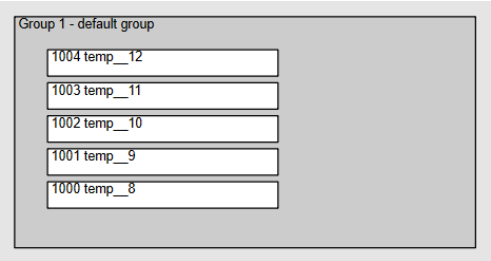
```
1 {SinOsc.ar * 0.1}.play;
```

Kodeboks 5.16: En simpel Synth

Vi får at vide, at vi har startet en såkaldt **Synth**. Teknisk set er en Synth en klynge af sammenhængende UGens, som defineres af en UGen-funktion (se 4.1) og kører på lydserveren som en „node“. Hver gang vi producerer lyd på lydserveren, har vi altså gang i en eller flere Synths. Når vi fx afspiller toner med en **Pbind**, genererer vi en ny Synth for hver tone, der spiller. Kør fx nedenstående blok og se listen over Synths i Node Tree-vinduet.

```
1 s.plotTree;
2 Pbind(
3     \degree, Pwhite(0, 4),
4     \legato, Pwhite(2.0, 4.0)
5 ).play;
```

Kodeboks 5.17: Pbind producerer Synths



Figur 5.6: Node Tree-vinduet viser serverens aktive nodes

Vi kan også starte Synths direkte ved at bruge **Synth**.new.

```
Synth.new(\default);
Synth(\default);
// Samme resultat: .new er implicit
```

Kodeboks 5.18: Start en Synth med Synth.new

Men hov, hvad er mon \default? Jo, det er navnet på en bestemt **SynthDef**.

5.3.2 Hvad er en SynthDef?

Ordet **SynthDef** står, ikke overraskende, for *Synth-definition*. Med **SynthDef**s kan vi lave lyddesign-opskrifter og generere **Synth**s på meget fleksibel vis.

En SynthDef adskiller sig fra den mere primitive form {}.play på følgende måder:

- En ny SynthDef (**SynthDef**.new) skal registreres på lydserveren med .add;[1].
- En SynthDef skal have et navn, så vi kan henvise til den senere, fx \minSynthDef.
- Vores UGen-funktion angives lige efter navnet, som argument nr. 2 til **SynthDef**.new().
- For at høre lyd-outputtet skal vi inde i UGen-funktionen route det ønskede signal ud med den særlige UGen **Out**.

```
(
// Vi skriver først SynthDef'en og registrerer den på
↪  lydserveren
SynthDef(\minSynthDef, {
    Out.ar(0, SinOsc.ar(440) * 0.1);
}).add;
)

// Start en Synth baseret på SynthDef'en
Synth(\minSynthDef);
```

Kodeboks 5.19: En simpel SynthDef

[1]Teknisk set kan SynthDefs også gemmes i et særligt filformat, hvilket dog kun er nødvendigt i særlige tilfælde. Læs nærmere herom i dokumentationen for SynthDefs.

5.3.3 Interfacet mellem Synth, SynthDef og Pbind

Sammenhængen mellem **Synth**, **SynthDef** og **Pbind** er helt central i SuperCollider.

- **SynthDef** kan forstås som et instrument, der bestemmer mulighederne for tonernes klangdannelse over tid (spektromorfologi).
- **Synth**s kan forstås som de konkrete lyde, man skaber med instrumentet.
- **Pbind** udgør således kompositionen eller partituret.

Hvis man i denne analogi synes, at der mangler en musiker til at udføre kompositionen, kan man tænke på den **EventStreamPlayer**, som opstår, når vi bruger **Pbind**`().play` - det er dette objekt, som faktisk udfører partituret og starter/stopper Synths på lydserveren ud fra de instrukser, vi har angivet med **Pbind**. Vi bruger sjældent denne klasse direkte, bortset fra når vi gemmer resultatet af **Pbind**`.play();` under en variabel og efterfølgende interagerer med denne variabel.

5.3.3.1 SynthDef og Synth

Lad os se på hvordan samspillet mellem **Synth** og **SynthDef** fungerer i praksis.

SynthDef'en `\minSynthDef` ovenfor fungerer, men er ikke særligt fleksibel. Vi kan fx kun spille én tone med den, og vi er nødt til at stoppe den manuelt med Ctrl/Cmd-Punktum. For at kunne spille forskellige toner kan vi indføre et argument. Herunder vælger jeg at indføre argumentet `freq` ved hjælp af nøgleordet **arg**, og jeg angiver også en standardværdi på `440`.

```
1 SynthDef(\tone, {
2     arg freq = 440;  // <--- her erklærer vi argumentet og
      ↪  angiver standardværdien
3     Out.ar(0, SinOsc.ar(freq) * 0.1); // <--- her bruger vi
      ↪  argumentet, ligesom en variabel
4 }).add;
```

Kodeboks 5.20: SynthDef og Synth

Når vi har tilføjet vores SynthDef til lydserveren, kan vi starte nye Synths baseret på den.

```
Synth(\tone);
Synth(\tone, [\freq, 220]);
Synth(\tone, [\freq, 1000]);

// Vi kan også justere frekvensen efterfølgende ved at gemme
↪  Synth'en under en variabel og bruge .set
~minLyd = Synth(\tone, [\freq, 330]);
~minLyd.set(\freq, 220);
~minLyd.set(\freq, rrand(100, 800));
```

Kodeboks 5.21: Synth baseret på SynthDef 

I mange SynthDefs er det nyttigt at bruge en envelope (se 5.1) til at styre volumen over tid. Med `doneAction` angiver vi, at Synth'en skal fjernes, når tonen er klinget ud (læs nærmere herom i afsnittet om automatisk oprydning med doneAction (se 5.1.4)).

5.3.3.2 SynthDef og Pbind

Vi behøver imidlertid ikke starte vores Synths manuelt. Når SynthDef er indlæst korrekt på lydserveren, kan vi bruge **Pbind** til at generere sekvenser af Synths helt automatisk. Vi fortæller Pbind, hvilken SynthDef, der skal anvendes, ved at bruge nøglen `\instrument`. Det betyder, at **Pbind**`(\instrument, \simpel)` vil starte Synths baseret på en SynthDef med navnet `\simpel`. Hvis ikke en sådan SynthDef er registreret på lydserveren, vil vi få en fejlmeddelelse.

```
// Først registrerer vi SynthDef'en på lydserveren
SynthDef(\simpel, {
    arg freq = 440;
    Out.ar(0, SinOsc.ar(freq) * 0.1
        * EnvGen.kr(Env.perc, doneAction: Done.freeSelf)
    );
}).add;
```

Kodeboks 5.22: En simpel SynthDef 

Nu kan vi bruge Pbind til at „spille på“ vores SynthDef, som om den er et instrument.

```
Pbind(
    // SynthDef-navnet angives under nøglen \instrument
    \instrument, \simpel,
    \degree, Pwhite(-7, 7).trace,
).play;
// -> 6, 0, -4, 6, 1, 1, -7, -4, -5, 4, 1, -7, -7 , 5, 4, 7
```

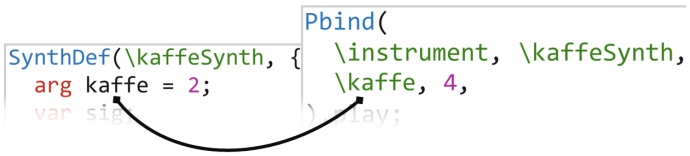
Kodeboks 5.23: Pbind bruger den simple SynthDef

SynthDefs bliver i øvrigt meget lettere at læse, hvis vi bruger lokale variabler (se 1.3.2). Koden herunder fungerer præcis som ovenfor, men er markant mere læsbar, da vi kan følge signelvejen gennem de lokale variabler (hvis ellers de variabelnavne vi har valgt er tilstrækkeligt deskriptive).

```
SynthDef(\simpel, {
    arg freq = 440;
    // 'sig' er variabelnavnet for vores hovedsignal
    var sig = SinOsc.ar(freq);
    // 'env' er variabelnavnet for vores envelope
    var env = EnvGen.kr(Env.perc, doneAction: Done.freeSelf);
    sig = sig * env * 0.1;
    Out.ar(0, sig);
}).add;
```

Kodeboks 5.24: Signalflow i SynthDef med lokale variabler

Vi kan indføre lige så mange argumenter i en **SynthDef**, som vi har lyst. Når vi vil bruge argumenterne i en **Pbind**, skal nøglerne blot svare til argumenterne.



Figur 5.7: Sammenhæng mellem SynthDef-argumenter og Pbind-nøgler

Med et par ekstra argumenter og en **Pan2**-UGen kan vi styre parametre som lydstyrke, release-tid og stereo-panorering. De argumenter, vi angiver i begyndelsen af en SynthDef, kan vi nemlig anvende som nøgler i **Pbind**. Dette interface mellem **SynthDef**/**Synth** og **Pbind**, danner grundlag for de utroligt righoldige, generative muligheder i SuperCollider.

```
SynthDef(\fleksibel, {
    arg freq = 440, pan = 0, amp = 0.1, release = 1;
    var sig = SinOsc.ar(freq);
    var env = EnvGen.kr(Env.perc(releaseTime: release),
    ↪  Done.freeSelf);
    sig = sig * env;
    sig = Pan2.ar(sig, pan, amp);
    Out.ar(0, sig);
}).add;
```

Kodeboks 5.25: SynthDef med variabel tonehøjde, panorering og volumen

Nu kan vi bruge vores viden om patterns til at skabe en algoritmisk komposition, hvor også klanglige forhold styres af patterns.

```
Pbind(
    \instrument, \fleksibel,
    \degree, Pwhite(-7, 7).stutter(4),
    \pan, Pbrown(-1.0, 1.0, 0.2),
    \amp, Pexprand(0.1, 0.3),
    \dur, Pseries(0.100, 0.010, 35),
    \release, Prand([0.100, 0.300, 2], inf)
).play;
```

Kodeboks 5.26: Pattern-komposition med SynthDef-argumenter

Hvis ovenstående pattern-komposition er vanskelig at følge, bør du genopfriske din viden om pattern-baseret komposition fra tidligere kapitler på grundlæggende (se 2.1) og lidt mere avanceret niveau (se 3.1).

5.3.4 Legato og staccato med vedvarende envelopes

Pbind kan også afslutte vedvarende envelopes (se 5.1.3.2) for os. Vi kan dermed få adgang til at komponere med legato- og staccatofrasering. Det kræver dog, at **SynthDef**'en indrettes på følgende måde:

- Vi indfører et argument kaldet `gate` med standardværdi 1.
- Vi bruger en vedvarende envelope, fx **Env**`.asr`.
- Vi angiver `gate` som argument nr. 2 til **EnvGen**.

```
SynthDef(\vedvarende, {
    arg freq = 440, pan = 0, amp = 0.1, gate = 1;
    var sig = SinOsc.ar(freq);
    var env = EnvGen.kr(Env.asr, gate, doneAction:
    ↪ Done.freeSelf);
    sig = sig * env;
    sig = Pan2.ar(sig, pan, amp);
    Out.ar(0, sig);
}).add;
```

Kodeboks 5.27: SynthDef med vedvarende envelope

Med ovenstående SynthDef indlæst på lydserveren kan vi nu også spille legato og staccato. Det gør vi ved hjælp af nøglerne `legato` og `\sustain` i **`Pbind`**. Til `\sustain`-nøglen kan vi angive en *absolut* nodeværdi, dvs. hvor længe tonerne skal holdes. Dette er et alternativ til `\legato`, som automatisk udregner `\sustain` *relativt* til `\dur`-nøglen.

```
( // Legato-frasering
Pbind(
    \instrument, \vedvarende,
    \degree, Pwhite(-7, 7, 5),
    \sustain, 3,
).play;
)

( // Staccato-frasering
Pbind(
    \instrument, \vedvarende,
    \degree, Pwhite(-7, 7, 5),
    \sustain, 0.1,
).play;
)
```

Kodeboks 5.28: Legato og staccato

5.3.4.1 Vedvarende Synth med Pmono og PmonoArtic

Som vi har set, er Pbind god til sekvenser, der starter mange ynths. Men til gengæld kan Pbind ikke ændre på indstillingerne for Synths, fx tonehøjde eller panorering, når de er startet. Det kan vi gøre manuelt ved hjælp af `.set`-metoden.

```
~minLyd = Synth(\default);

~minLyd.set(\freq, 1000);
~minLyd.set(\freq, 500);
```

Kodeboks 5.29: Synth og method'en .set

Hvis vi med vores egne SynthDefs vil skabe glissandi frem for spring i tonehøjde, kan vi bruge method'en `.lag`[2] på frekvens-argumentet, hvilket vil interpolere mellem værdier frem for at springe imellem dem. Dette er ganske nyttigt, hvis man vil blødgøre overgangen mellem skiftende værdier. Vi kan med et argument til `.lag` styre, hvor lang tid det tager at glide hen til den nye værdi.

```
SynthDef(\glissando, {
    arg freq = 440, pan = 0, amp = 0.1, gate = 1;
    // .lag giver en glidende overgang mellem skiftende værdier
    ↪ (her glissando)
    var sig = SinOsc.ar(freq.lag(0.01));
    var env = EnvGen.kr(Env.asr, gate, doneAction:
    ↪ Done.freeSelf);
    sig = sig * env;
    sig = Pan2.ar(sig, pan, amp);
    Out.ar(0, sig);
}).add;

~minLyd = Synth(\glissando);
~minLyd.set(\freq, exprand(200, 2000));
```

Kodeboks 5.30: Glissandi med .lag

Hvis vi vil skabe en komposition med patterns, hvor vi bruger denne glissando-egenskab ved vores SynthDef, kan vi i stedet for Pbind bruge **Pmono** eller **Pmono Artic**. Begge disse kusiner til Pbind starter blot én Synth ad gangen og justerer efterfølgende parametrene ved hjælp af de sædvanlige koblinger af nøgler og patterns, som vi kender fra Pbind. SynthDef-navnet angives som første argument, dvs. uden `\instrument`-nøglen.

[2] Teknisk set kan SynthDefs også gemmes i et særligt filformat, hvilket dog kun er nødvendigt i særlige tilfælde. Læs nærmere herom i dokumentationen for SynthDefs.

```
Pmono(\glissando,
    \degree, Pseq([0, 2, 4, 6], inf),
    \mtranspose, Pwhite(0, 7).stutter(4),
    \dur, 0.15,
).play;

// vis Synths på serveren - Pmono starter kun én Synth
s.plotTree;
```

Kodeboks 5.31: Pmono og glissando

Hvis vi har brug for at lave pauser i lyden mellem de strømme af værdier, der bliver skiftet imellem ved hjælp af **Pmono**, kan vi bruge **PmonoArtic**.

På pattern-siden styres artikulationen med `\sustain`- eller `\legato`-nøglen; når `\sustain` er mindre end `\dur`, afsluttes Synth'en, og der startes en ny ved næste event. Den mest praktiske tilgang er at bruge nøglen `legato`, hvor vi angiver en værdi, som er mindre end 1, når vi ønsker at lave ophold, og en værdi der er højere end eller lig med 1, når vi ønsker en sammenhængende tone.

```
PmonoArtic(\glissando,
    \degree, Pseq([0, 2, 4, 6], inf),
    \mtranspose, Pwhite(0, 7).stutter(4),
    \dur, 0.4,
    \legato, Pseq([1, 1, 1, 0.1], inf),
).play;
```

Kodeboks 5.32: Glidende arpeggio med luft mellem fraserne - med PmonoArtic

5.3.5 Argumentnavne i SynthDef til kombination med Pbind

Argumenterne i vores SynthDefs kan i princippet have de navne vi gerne vil give dem (dog skal de starte med små bogstaver, ligesom variabler). Argumentnavne kunne fx være `kaffe`, `the`, `mario`, `luke` eller `leia`. Men sædvanligvis kan det være en god ide at give argumenterne nogle *deskriptive* navne som fx `cutoffFreq`, `release`, `drive`, `delayTime` eller lignende. På den måde kan man regne ud hvad de betyder, når man vender tilbage til koden efter noget tid.

Når man vælger argumentnavne til sin SynthDef, findes der dog nogle få specielle navne, som er værd at kende og rette sig efter, hvis man gerne vil bruge sin SynthDef sammen med Pbind og patterns/events (se 2.3). Derudover findes der nogle bredt

anvendte, konventionelle argumentnavne, som det er nyttigt at kende til, samt nogle bestemte argumentnavne, som det er fornuftigt at undgå.

5.3.5.1 Vigtige argumentnavne

Disse argumentnavne er nødvendige for at fungere med centrale dele af SuperCollider såsam patterns, events, live coding-klasser mm.:

`freq`
: Anvendes til at angive tonefrekvens i SynthDefs, hvor det giver mening at forstå frekvensen som en tonehøjde, fx et højt c eller et lavt gis. Dette giver os mulighed for at bruge nøgler som `\degree`, `\midinote`, `\octave`, `\scale` osv. og automatisk få disse informationer omregnet til oscillatorfrekvens (se 2.3.1), når vi arbejder med **`Pbind`**. Standardværdien er ofte 440 (kammertonen).

`gate`
: Anvendes typisk til at afslutte vedvarende envelopes og arbejde med legato og staccato-frasering, som vist ovenfor. Standardværdien er typisk 1.

`amp`
: Bruges til at angive lydstyrke. Giver mulighed for at omregne automatisk fra `\db`, når vi arbejder med **`Pbind`**. Standardværdien er ofte 0.1, som svarer til -20 dB.

`out`
: Anvendes til at route et signal fra en Synth til en bestemt **`Bus`**. Dette er fx relevant, hvis man fx skal arbejde med mange højttalere på én gang i en lydinstallation, eller hvis man udforsker live coding med JITLib. Standardværdien er ofte 0, som er den første hardware-outputkanal.

5.3.5.2 Anbefalede argumentnavne

Disse argumentnavne er ikke så vidt vides strengt nødvendige. Men der er tale om meget udbredte konventioner, som gør kildekoden sammenlignelig og kompatibel med andres kildekode.

`pan`
: Bruges til at angive panorering, ofte i et stereofelt mellem -1 og 1. Standardværdien er ofte 0, i midten af et stereofelt.

`buf`
: Anvendes til at angive en **`Buffer`** på lydserveren. Dette er fx relevant, når der arbejdes med samples (se 8.1), wavetables og lignende.

5.3.5.3 Argumentnavne, som næsten altid bør undgås

`dur, scale, sustain, stretch, midinote, detune` MED FLERE
: Disse navne bruges til automatiske omregninger, når vi komponerer med Pbind (se 2.3). Hvis man bruger dem som SynthDef-argumentnavne uden at være klar over dette, kan der hurtigt opstå mærkelige konsekvenser, som kan være svære at gennemskue. Man kan se en samlet liste over termer, der bruges til automatisk udredning af tonehøjde, timing og amplitude i patterns og events i James Harkins' oversigt (2009).
Min anbefaling er derfor:

- Brug navnene `freq`, `amp` og `gate` til SynthDef-argumentnavne, som vist ovenfor.
- Brug `\scale`, `\degree`, `\mtranspose`, `\sustain`, `\legato`, `\db` osv. som nøgler i forbindelse med patterns og events (se 2.3), og ikke som argumentnavne i SynthDefs.

5.4 Syntetisk stortrommelyd

Èt område inden for elektronisk klangdannelse, hvor vi gør rig brug af envelopes, er dannelsen af trommelyde. Uanset om man ønsker at simulere en bestemt tromme for at opnå en „realistisk“ klang eller målet er en tydeligt syntetisk klang, er trommelyde karakteriseret ved, at forskellige aspekter af lydene forandrer sig dramatisk over korte tidsspænd. Herunder kigger vi på hvordan man kan skabe en stortrommelyd ved hjælp af en oscillator og en støjkilde, der moduleres af korte envelopes. Tilgangen her er løseligt inspireret af Gordon Reids gennemgang af analoge kredsløb i tidlige trommemaskiner (Reid, 2002a) samt et par stortromme-SynthDefs af Nathan Ho (2017).

5.4.1 Forskellige aspekter af en stortrommelyd

Trommelyde er klangligt komplekse, og syntetisering af en stortrommelyd er et problem, der kan angribes på mange måder. For enkelhedens skyld deler vi her stortrommelyden op i to elementer: Den dybe tone, som vi kan kalde for kroppen, og en lysere, kort støjimpuls, som på en akustisk tromme opstår, når et objekt rammer på trommeskindet. Sidstnævnte kalder vi for klikket.

5.4.1.1 Krop

Stortrommens krop kan vi simulere med en simpel sinustone hvis frekvens moduleres af en enkel envelopegenerator, `XLine`. Amplituden moduleres af en `Env.perc` med yderst kort attack, releasetid på 200 ms og 5 ms forsinkelse (dette for at skabe plads til klik-lyden i begyndelsen af stortrommens klanglige forløb).

```
{
    var body = SinOsc.ar(XLine.ar(350, 50, 0.045));
    body * Env.perc(0.0001, 0.2, 1, \lin).delay(0.005).ar(2);
}.play;
```

Kodeboks 5.33: Stortrommens krop

5.4.1.2 Klik

For at simulere en kliklyd kan vi bruge en støjgenerator, `WhiteNoise.ar`. Fuldspektret hvid støj vil dog blive for dominerende, og vi anvender derfor et båndpasfilter til

at fjerne en væsentlig mængde støj og bevare et begrænset frekvensbånd centreret omkring 500 Hz. Filtre introduceres i næste kapitel, så for nuværende kan du blot betragte variablen `click` som indeholdende støj med primær intensitet omkring 500 Hz. Envelopen til klikket er kort, 21 ms i alt. Dette naturligvis fordi kliklyden hurtigt dør hen igen. I nogle genrer er kliklyden meget eftertragtet, mens den fylder mindre i andre genrer.

```
{
    var click = BPF.ar(WhiteNoise.ar, 500, 0.4);
    click * Env.perc(0.001, 0.02, 0.5).ar(2);
}.play
```

Kodeboks 5.34: Stortrommens klik

5.4.2 Sammensætning til SynthDef

For at demonstrere interfacet mellem Pbind og SynthDef (se 5.3.3) sammensætter vi de to ovennævnte elementer i en SynthDef, hvor de fleste af indstillingerne kan justeres ved hjælp af SynthDef-argumenter og dermed varieres med patterns i Pbind. I forhold til `doneAction` og de to envelopes, så lader vi envelopen til trommens krop afslutte synth'en, da den varer længst.

```
SynthDef(\kick, {
    arg pan = 0, amp = 0.1, out = 0, release = 0.2, cutoff =
    ↪ 8000,
    clickFreq = 500, clickRq = 0.4, clickAmp = 0.5, bodyDelay =
    ↪ 0.005,
    bodyStart = 350, bodyEnd = 50, bodyAmp = 1, bodyTime =
    ↪ 0.045;

    var clickEnv = Env.perc(0.001, 0.02, clickAmp).ar;
    var click = BPF.ar(WhiteNoise.ar, clickFreq, clickRq) *
    ↪ clickEnv;

    var bodyEnv = Env.perc(0.0001, release, bodyAmp,
    ↪ \lin).delay(bodyDelay).ar(doneAction: 2);
    var body = SinOsc.ar(XLine.ar(bodyStart, bodyEnd, bodyTime))
    ↪ * bodyEnv;

    var sig = (body + click).tanh;

    sig = LPF.ar(sig, cutoff);
    sig = Pan2.ar(sig, pan, amp);
    Out.ar(out, sig);
}).add;
```

Kodeboks 5.35: SynthDef til stortromme

Vi kan teste SynthDef'en med at starte et par toner med forskellige indstillinger.

```
Synth(\kick);
Synth(\kick, [\clickAmp, 0.2, \bodyStart, 1000]);
```

Kodeboks 5.36: Test af stortromme-SynthDef

Husk at lytte til disse lyde på gode hovedtelefoner eller højttalere, som kan gengive de dybe frekvenser.

5.4.2.1 En demonstration af de klanglige muligheder

For at illustrere hvor mange klanglige muligheder, der faktisk findes blot med denne enkle SynthDef, kan vi skrive en Pbind, hvor alle SynthDef-argumenter varieres, så vi konstant bliver præsenteret for nye udgaver af denne stortrommelyd. Der er tilføjet `.trace` til alle de anvendte patterns, så man kan stoppe strømmen og notere sig de forskellige værdier, hvis man hører en lyd, man godt kan lide.

```
TempoClock.tempo = 110 / 60;
p = Pbind(
    \instrument, \kick, \db, -10,
    \release, Pexprand(0.1, 0.8, 10).trace(prefix: 'release: '),
    \bodyStart, Pexprand(200, 1200).trace(prefix: 'bodyStart:
    ↪ '),
    \bodyEnd, Pexprand(35, 90).trace(prefix: 'bodyEnd: '),
    \bodyAmp, Pgauss(1, 0.1).trace(prefix: 'bodyAmp: '),
    \bodyTime, Pexprand(0.025, 0.100).trace(prefix: 'bodyTime:
    ↪ '),
    \bodyDelay, Pexprand(0.002, 0.020).trace(prefix: 'bodyDelay:
    ↪ '),
    \clickRq, Pexprand(0.5, 0.9).trace(prefix: 'clickRq: '),
    \clickFreq, Pexprand(300, 2000).trace(prefix: 'clickFreq:
    ↪ '),
    \clickAmp, Pexprand(0.3, 0.8).trace(prefix: 'clickAmp: '),
);
p.stutter(4).play;
```

Kodeboks 5.37: En demonstration

5.5 Cheat sheet: SynthDef

SynthDefs kan indrettes på mange forskellige måder og til mange forskellige formål. Til mono- eller stereofone lyde med tonalt indhold, kan skabelonerne herunder være et godt udgangspunkt. Begge skabeloner er i sig selv ret kedelige med en simpel sinustone, men det kan du nemt udbygge med et mere interessant lyddesign. Du bruger skabelonen på følgende måde:

- `\navn` skal ændres til et mere deskriptivt navn for din SynthDef.
- Du kan tilføje argumenter efter behov ved at følge den syntaks, der er anvendt på linje 2 og 3.
- Du kan tilføje lokale variabler ligesom på linje 5 til at definere signalflowet i SynthDef'en.
- Du kan oplagt tilføje en envelope (se 5.1) for at styre lydens volumen over tid. Husk `doneAction` for at rydde op automatisk (se 5.1.4).
- Du kan ændre `SinOsc.ar(freq)` til noget andet, alt efter hvilken lydkilde, du ønsker at arbejde med.
- Du kan tilføje yderligere indhold og lydlig transformation til SynthDef'en mellem linje 4 og 9 (tilføj gerne flere linjer).
- Linje 9 med `Out.ar` er oftest den sidste linje i en SynthDef, som sørger for, at signalet panoreres og volumenjusteres som ønsket, inden det sendes til de ønskede output-kanaler.

5.5.1 Monofone lydkilder

Her er lydkilden en enkelt oscillator.

```
1 SynthDef(\navn, {
2     arg freq = 440, pan = 0,
3     amp = 0.1, out = 0;
4     var sig = SinOsc.ar(freq);
5
6     // sig er et monofont signal
7     Out.ar(out, Pan2.ar(sig, pan, amp));
8 }).add;
```

Kodeboks 5.38: SynthDef-skabelon til monofone lydkilder

5.5.2 Stereofone lydkilder

Her er lydkilden blot en duplikeret (se 7.1.1) sinustone-oscillator, som klinger i to kanaler. Med `Balance2.ar` balanceres der mellem dissee to kanaler.

```
SynthDef(\navn, {
    arg freq = 440, pan = 0,
    amp = 0.1, out = 0;
    var sig = SinOsc.ar(freq).dup;

    // sig er et stereofont signal
    Out.ar(out, Balance2.ar(sig[0], sig[1], pan, amp));
}).add;
```

Kodeboks 5.39: SynthDef-skabelon til stereofone lydkilder 

Du kan læse nærmere om stereofoni og forskellen på `Pan2` og `Balance2` senere i bogen (se 7.1.2).

5.6 Øvelse: Envelopes

I denne øvelse arbejder du med at anvende og designe dine egne envelopes og LFO'er.

5.6.1 Brug af indbyggede envelopes

1. Tag udgangspunkt i nedenstående kildekode, og brug følgende envelopes:
   a) Indbyggede envelopes med standardværdier for attack, release mm.
      i. `Env.perc`
      ii. `Env.linen`
      iii. `Env.sine`
      iv. `Env.triangle`
   b) Indbyggede envelopes med specifikke indstillinger
      i. `Env.perc` med attack-tid på 200 milisekunder og release-tid på 3 sekunder
      ii. `Env.linen` med sustain-**tid** på 2 sekunder
      iii. `Env.sine` med varighed på 0.1 sekund
2. Modificér dernæst koden, således at vi også modulerer `Pulse`-oscillatorens frekvens med envelope-generatoren, skaleret med `.exprange` til intervallet 55-440.

Justér kun på de markerede linjer i kodeblokken herunder.

```
1 {
2     var env = EnvGen.kr(      , doneAction: Done.freeSelf);
3     var freq = 440;
4     Pulse.ar(freq) * env * 0.1;
5 }.play;
```

Kodeboks 5.40: Øvelse med indbyggede envelopes 

Husk, at `.plot` kan være en nyttig hjælp: `Env.perc(1, 3).plot`

5.6.2 Simpel lilletrommelyd

Du skal her fremstille en simpel lilletrommelyd. De centrale lydlige egenskaber ved trommelyde varierer meget hurtigt over tid, hvilket du kan simulere ved hjælp af envelopes.

Skriv tre envelopes:

1. *body:* En `XLine`, som bevæger sig fra 220 til 110 over 275 ms.
2. *sweep:* En `XLine`, som bevæger sig fra 8000 til 2500 over 10 ms.
3. *vol:* En `Env.perc`, hvor attack-segmentet varer 0.5 ms(!) og release-segmentet varer 150 ms.

Prøv efterfølgende at justere på parametrene i de forskellige envelopes og bemærk hvordan selv mindre ændringer påvirker det lydlige resultat.

```
{
    var body =      ;
    var sweep =      ;
    var vol =      ;

    // justér ikke herunder
    var resonance = VarSaw.ar(body);
    var seiding = WhiteNoise.ar;
    var sig = resonance + seiding;
    sig = LPF.ar(sig, sweep);
    sig = sig * EnvGen.ar(vol, doneAction: Done.freeSelf);
    Out.ar(0, sig.dup);
}.play;
```

Kodeboks 5.41: Simpel lilletrommelyd 

Lilletrommen dannes på følgende måde (Pejrolo & Metcalfe, 2017):

- Resonansen i trommens krop, dvs. dens „tone“, simuleres med en trekantformet lydbølge. Vi kan således „stemme“ trommen ved at justere start- og slutværdier for envelopen.
- Trommens seiding (metaltråde monteret på undersiden) simuleres med hvid støj.
- Med et dynamisk low pass filter simulerer vi trommens klanglige forandring over tid (mere højfrekvent støj i starten).

I næste kapitel gennemgås et mere elaboreret bud på dannelse af lilletrommelyd (se 6.3).

5.6.3 En unik envelope til tonehøjde

Definér din egen envelope, som overholder følgende krav:

1. Envelopen skal bestå af mindst to segmenter.
2. Værdierne i envelopen (linje 3 i kodeblokken herunder) skal ligge mellem 220 og 880.
3. Tidsintervallerne (linje 4 i kodeblokken herunder) vælges frit (husk, at der skal være ét tidsinterval færre end antallet af trin).

Justér kun på de markerede linjer i kodeblokken herunder.

```
1  ~frekvensEnvelope = Env(
2      [   ],
3      [   ],
4      \exp
5  );
6
7  // Vis envelopen grafisk
8  ~frekvensEnvelope.plot;
9
10 // Test envelopen med lyd
11 {
12     var freq = EnvGen.kr(~frekvensEnvelope);
13     var vol = EnvGen.kr(
14         Env.sine(~frekvensEnvelope.duration),
15         doneAction: Done.freeSelf
16     );
17     SinOsc.ar(freq) * vol * 0.1;
18 }.play;
```

Kodeboks 5.42: Unikke envelopes

5.6.4 Hjemmestrikket LFO

1. Modificér SynthDef'en herunder, således at LFO'en modulerer *mindst to forskellige, lydlige parametre* (fx tonehøjde, panorering, lydstyrke, cutoff-frekvens etc.). Husk at skalere outputtet fra LFO'en (se 4.2), så det passer til modulationens formål.
2. Design din egen LFO ved hjælp af `Env.circle`. Tag eventuelt udgangspunkt i et tidligere præsenteret eksempel (se 5.2.2).
3. Skriv på egen hånd en komposition ved hjælp af `Pbind` eller `Pmono`, hvor du demonstrerer mulighederne i SynthDef'en.

```
SynthDef(\lfo, {
    arg freq = 440, pan = 0, amp = 0.1, out = 0, gate = 1;

    var lfo = EnvGen.kr(
        Env.circle(
            [   ],
            [   ],
            \exp
        )
    );

    var env = EnvGen.kr(Env.asr, gate, doneAction:
    ↪  Done.freeSelf);
    var sig = Saw.ar(freq);

    // Lavpas-filter med cutoff-frekvens to oktaver over
    ↪  grundtonen
    sig = LPF.ar(sig, freq * 4);
    sig = Pan2.ar(sig, pan, amp) * env;
    Out.ar(out, sig);
}).add;
```

Kodeboks 5.43: SynthDef med hjemmelavet LFO

5.7 Øvelse: SynthDefs

SynthDef-skrivning er måske et lidt tørt, teknisk emne. Men med lidt øvelse er det nøglen til at åbne de mange fantastiske variationsmuligheder, som interfacet mellem **SynthDef** og **Pbind** lægger op til (se 5.3.3). Du må i denne øvelse gerne skele til cheat sheet'et om SynthDefs (se 5.5) og andre relevante afsnit (se 5.3.5), men du skal i denne øvelse skrive SynthDef-kildekoden selv. Det får du nemlig mest ud af.

5.7.1 Min første SynthDef

1. Skriv en SynthDef, som overholder følgende krav:
   a) SynthDef'ens navn skal være `\hello`.
   b) Som lydkilde skal SynthDef'en indeholde en oscillator (se 4.1.1) efter eget valg.
   c) Oscillatorens frekvens kan styres med et SynthDef-argument kaldet `freq`, defaultværdi `440`.
   d) Oscillatorens lydstyrke over tid styres af en envelope kaldet **Env**`.linen`.
2. Test, at `freq`-argumentet i din SynthDef fungerer.
   a) Brug en kommando som **Synth**`(\hello, [\freq, 1000]);`, hvor du ændrer på frekvensen.
3. Justér SynthDef'en, så der kan ændres på lydstyrken.
   a) Indfør et nyt SynthDef-argument kaldet `amp` med defaultværdi `0.1`.
   b) Sørg for, at amplituden for lydsignalet bliver ganget med `amp`, før det sendes ud med **Out**.
4. Test din SynthDef med nedenstående Pbind.

```
Pbind(
    \instrument, \hello,
    \degree, Pwhite(0, 7),
    \octave, Pwhite(3, 6),
    \dur, 0.25,
).play;
```

Kodeboks 5.44: Test af SynthDef

5.7.2 En SynthDef med firkantede bølgeformer

1. Skriv en SynthDef, som overholder følgende krav:

a) SynthDef'ens navn skal være `\firkant`.
b) Lydkilden er en **`Pulse`**-UGen hvis frekvens kan styres med et SynthDef-argument kaldet `freq`, defaultværdi `220`.
c) Lydstyrken kan styres med et SynthDef-argument kaldet `amp`, defaultværdi `0.1`.
d) Lydstyrken moduleres af en slagtøjsagtig envelope (**`Env`**`.perc`), hvor attack og release kan styres med SynthDef-argumenter `attack` og `release`.
e) Bredden af firkanten i **`Pulse`**-oscillatoren (2. argument til **`Pulse`**`.ar`) kan styres med et SynthDef-argument kaldet `width`.

2. Test din SynthDef med nedenstående Pbind. Den bør give et resultat, der minder om lydeksemplet herunder.

```
TempoClock.tempo = 130 / 60;
Pbind(
    \instrument, \firkant,
    \degree, [0, 2, 4, 6],
    \scale, Scale.minor,
    \octave, 4,
    \mtranspose, Pwhite(-2, 2).stutter(Phprand(2, 20).asStream),
    \db, Pseries(-25, Pwhite(3, 4).asStream, 4).repeat,
    \attack, 0.001,
    \release, Pgeom(0.1, 1.03, 100),
    \dur, Pwrand([1/8, 1/4, Rest(1/8)], [0.8, 0.15, 0.05], inf)
    ↪  * 4,
    \strum, Pexprand(0.0001, 0.01),
    \width, Pseries(0.2, 0.005, 100)
).play;
```

Kodeboks 5.45: Test af SynthDef

5.7.3 SynthDef med vedvarende envelope

1. Justér nedenstående SynthDef, så den anvender en vedvarende envelope i stedet for en selv-afsluttende envelope. OBS: Dette kræver, at du tilføjer et gate-argument (se 5.1.3.2) til **`EnvGen`**.
2. Skriv en **`Pmono`**-baseret komposition, hvor du varierer `\degree`, `\lfoFreq` og `\lfoDepth` ved hjælp af patterns, fx tilfældighedsgeneratorer (se 2.2). Der findes en kodeblok herunder, som du kan tage udgangspunkt i.

Justér kun på de markerede linjer i kodeblokkene herunder.

```
SynthDef(\gliss, {
    arg freq = 440, pan = 0, amp = 0.1, out = 0,
    lfoFreq = 1, lfoDepth = 0.025;

    var env = EnvGen.kr(Env.perc, doneAction: Done.freeSelf);

    var sig = Pulse.ar(
        freq.lag(0.01) *
        LFTri.kr(lfoFreq).bipolar(lfoDepth).midiratio
    );

    sig = Pan2.ar(sig, pan, amp) * env;

    Out.ar(out, sig);
}).add;
```

Kodeboks 5.46: SynthDef med glissando 

Afsæt for komposition med **Pmono** ses herunder.

```
Pmono(\gliss,
    \degree,    ,
    \lfoFreq,    ,
    \lfoDepth,    ,
    \dur, 0.4,
).play;
```

Kodeboks 5.47: Glidende komposition med Pmono 

KAPITEL 6

Klangdannelse med filtre

Filteret er et allestedsnærværende redskab i elektronisk musikproduktion, lige fra de filtre, der udgør en equalizer, til de berømte filterkredsløb, som indgik i de tidlige Moog-synthesizere. Der findes en hel gruppe af klangdannelsesteknikker, som baseres på filtre, kaldet *subtraktiv klangdannelse*. I dette kapitel ser vi nærmere på brugen af filtre i SuperCollider, og vi arbejder med en række eksempler på subtraktiv klangdannelse fra simple blæser- og strygerlyde til lilletromme og strengsimulering med den såkaldte Karplus-Strong-algoritme.

6.1 Filterbaseret klangdannelse

SuperCollider har en række indbyggede filter-UGens, der implementerer forskellige typer af digital lydfiltrering. Du kan se en oversigt over disse i cheat sheet'et i slutningen af dette kapitel (se 6.5). De anvendes selvsagt med en anden lydkilde som input, der angives som første argument. Her er eksempelvis et par filtre, der dæmper forskellige dele af frekvensspektret for pink støj.

```
{LPF.ar(PinkNoise.ar) * 0.1}.play;
{HPF.ar(PinkNoise.ar) * 0.1}.play;
```

Kodeboks 6.1: Lav- og højpasfiltre

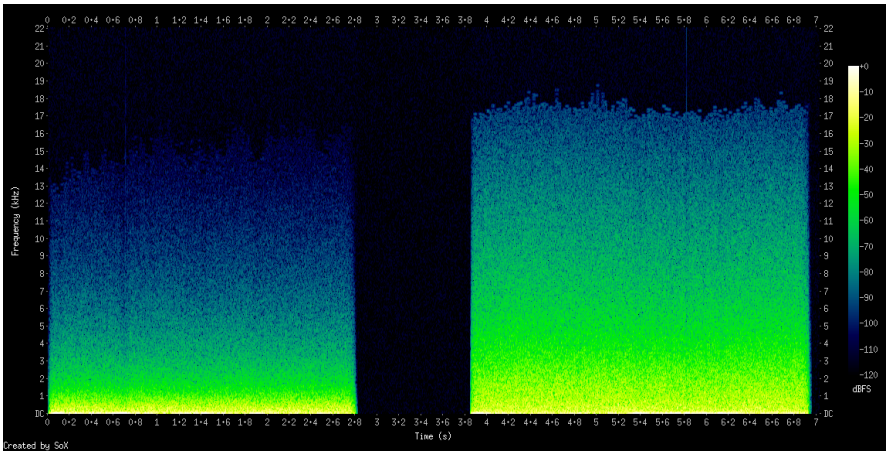
De mest almindelige filtre gennemgås herunder, og du kan finde en oversigt over filter-UGens i dette kapitels cheat sheet (se 6.5).

6.1.1 Cutoff-frekvens

Ved de fleste filter-UGens kan vi angive cutoff-frekvensen som argument nr. 2. For god ordens skyld bør det nævnes, at cutoff-frekvenser bør holdes mellem 20 Hz og 20 kHz.

```
// Cutoff ved 500 Hz
{LPF.ar(PinkNoise.ar, 500) * 0.1}.play;

// Cutoff ved 2000 Hz
{LPF.ar(PinkNoise.ar, 2000) * 0.1}.play;
```

Kodeboks 6.2: Specifikation af cutoff-frekvens

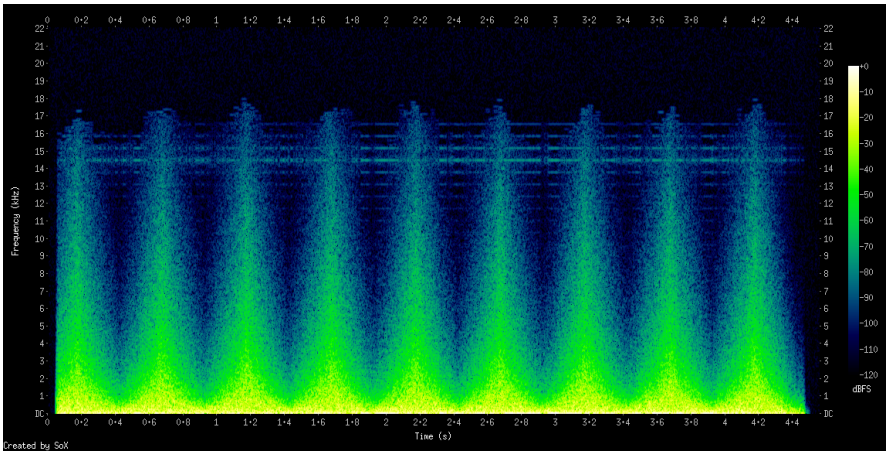


Figur 6.1: Spektrogram: Cutoff-frekvenser ved 500 Hz og 2000 Hz

Cutoff-frekvensen kan moduleres (se 4.1.2) af andre UGens, fx en LFO. Herunder skalerer (se 4.2) vi fx outputtet fra LFO'en **`LFTri`** med `.exprange`.

```
{
    var lfo = LFTri.kr(2).exprange(200, 2000);
    var source = PinkNoise.ar;
    LPF.ar(source, lfo);
}.play;
```

Kodeboks 6.3: Modulation af cutoff-frekvens



Figur 6.2: Spektrogram: Cutoff-frekvens for et lavpasfilter modulers af en LFO

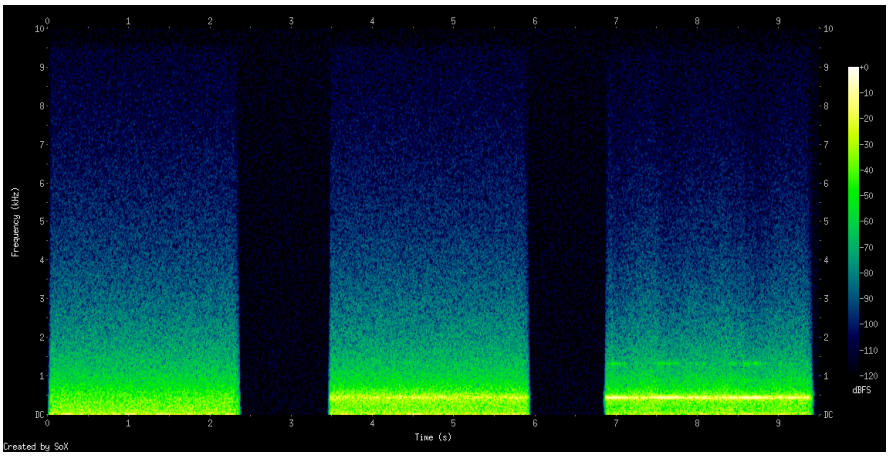
6.1.2 Filterets resonans og argumentet 'rq'

Ved filtre som **RLPF** og **RHPF** kan man angive et argument for at styre filterets såkaldte „kvalitet". Kvalitet er i denne sammenhæng et lidt mærkeligt begreb, da denne indstilling ikke har noget direkte at gøre med en æstetisk kvalitet. I praksis påvirker kvalitetsindstilling både hvor hårdt der dæmpes i det behandlede signal og hvor meget resonans, der opstår ved cutoff-frekvensen. I SuperCollider angives filterkvalitet typisk i en reciprok form, altså „1/q". Det lyder kompliceret men betyder blot, at når vi er på 1, er resonansen skruet helt ned, hvorimod den er skruet højt op, når vi nærmer os 0. Indstil aldrig `rq` til 0 (hvilket ville svare til en uendelig mængde resonans).

```
1 {RLPF.ar(PinkNoise.ar, rq: 1) * 0.1}.play;
2 {RLPF.ar(PinkNoise.ar, rq: 0.1) * 0.1}.play;
3 {RLPF.ar(PinkNoise.ar, rq: 0.01) * 0.1}.play;
```

Kodeboks 6.4: Variationer i rq

På spektrogrammet kan vi se, hvordan de tre forskellige `rq`-værdier påvirker den bredspektrede støj fra **PinkNoise**. Bemærk, hvordan intensiteten skabt af resonans ved cutoff-frekvensen (440 Hz) adskiller sig på de tre klip.



Figur 6.3: Variation i rq-argumentet til RLPF

6.1.3 Filtrering af lydkilde med tonehøjde

Ovenfor arbejder vi for enkelhedens skyld med filtreret støj, men i musikalsk sammenhæng bruger vi naturligvis også filtre til at forme klangen af andre lydkilder. Det

typiske eksempel er en savtakket eller firkantet bølgeform, som har et righoldigt overtonespektrum, der således med fordel kan formes for at opnå en ønsket klang. Herunder dæmper vi overtonerne for en savtakket bølgeform fra 1000 Hz.

```
1 {LPF.ar(Saw.ar(440), 1000) * 0.1}.play;
```

Kodeboks 6.5: Filtrering af Saw

Dette er dog ikke særligt fleksibelt, da vi ikke blot kan spille en anden tone (dvs. bruge en anden oscillatorfrekvens) og forvente, at klangen (altså tonens overtonespektrum) er det samme som før, blot tilsvarende højere eller lavere i frekvens. Ønsker vi, at cutoff-frekvens tilpasser sig oscillatorens frekvens automatisk, kan det heldigvis let gøres ved at regne cutoff-frekvensen ud særskilt. Uanset hvad vi angiver under `freq` herunder, vil filterets cutoff-frekvens ligger en oktav højere end oscillatorens tonehøjde.

```
1 {
2     var freq = 440;
3     var cutoff = freq * 2;
4     LPF.ar(Saw.ar(freq), cutoff) * 0.1;
5 }.play;
```

Kodeboks 6.6: Automatisk tilpasset cutoff-frekvens

Ønsker vi at kunne styre afstanden mellem cutoff-frekvens og oscillatorfrekvens med et argument (fx hvis vi ønsker at lave en SynthDef med subtraktiv klangdannelse), kan vi angive dette på forskellige måder - jeg foretrækker typisk at angive filterets cutoff målt i antal oktaver over oscillatorfrekvensen, hvilket kan udregnes let med en faktor `freq * 2.pow(oktav)` (hvor `2.pow(5)` betyder „to i femte“). Det betyder ganske enkelt, at hvis oscillatorens frekvens er 100 Hz, og vi ønsker en cutoff-frekvens 3 oktaver derover, ser regnestykket således ud: `100 Hz * 2.pow(3) = 100 Hz * 8 = 800 Hz`.

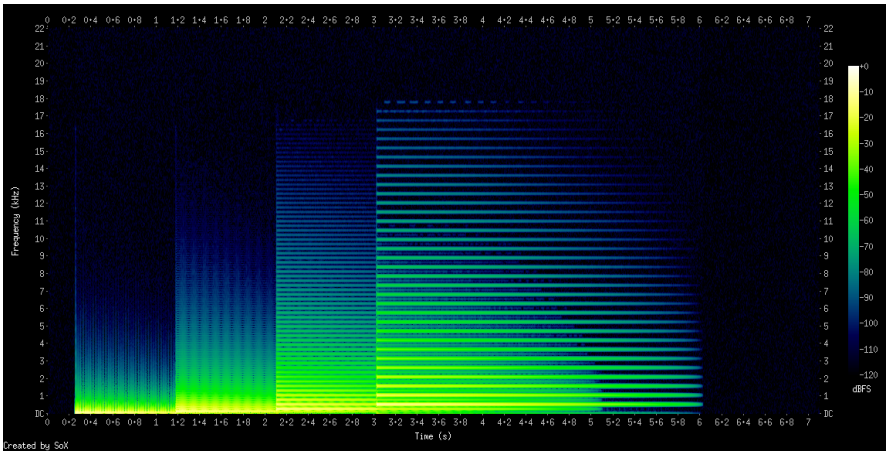
```
SynthDef(\autoCutoff, {
    arg freq = 440, pan = 0,
    amp = 0.1, out = 0, cutoffOktav = 2;
    var sig = Saw.ar(freq);
    var env = EnvGen.kr(Env.perc(0.005, 3), doneAction: 2);
    var cutoff = freq * 2.pow(cutoffOktav);
    sig = LPF.ar(sig, cutoff);
    sig = sig * env;
    Out.ar(out, Pan2.ar(sig, pan, amp));
}).add;
```

Kodeboks 6.7: Automatisk tilpasset cutoff-frekvens

Bemærk, at overtonespektret for de følgende toner er ens, blot flyttet relativt til tonehøjde.

```
Pbind(
    \instrument, \autoCutoff,
    \octave, Pseq([3, 4, 5, 6]),
    \dur, 2,
).play;
```

Kodeboks 6.8: Demonstration af automatisk udregnet cutoff

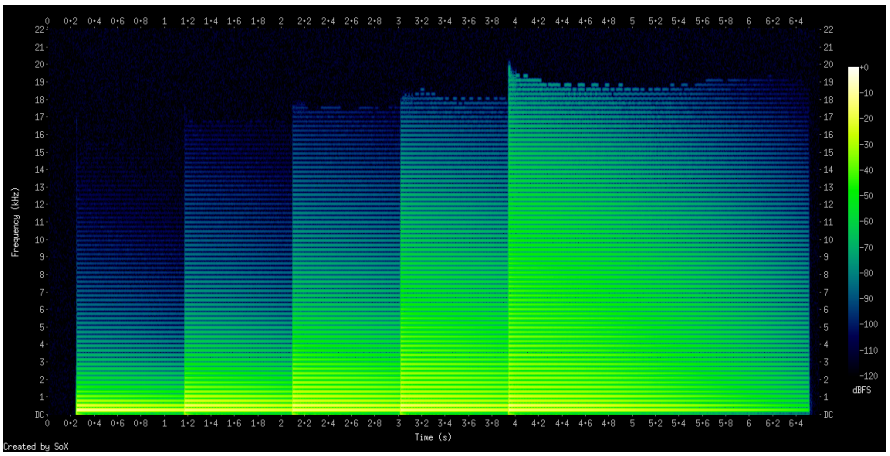


Figur 6.4: Spektrogram: Cutoff-frekvens to oktaver over oscillatorfrekvens

Vi kan også fremstille variable klange ved at indstille cutoffOktav-argument med dets tilhørende nøgle i Pbind.

```
Pbind(
    \instrument, \autoCutoff,
    \cutoffOktav, Pseq([1, 2, 3, 4, 5]),
    \dur, 2,
).play;
```

Kodeboks 6.9: Klanglig variation med filter-cutoff



Figur 6.5: Spektrogram: Cutoff-frekvenser i stigende oktaver over oscillatorfrekvens

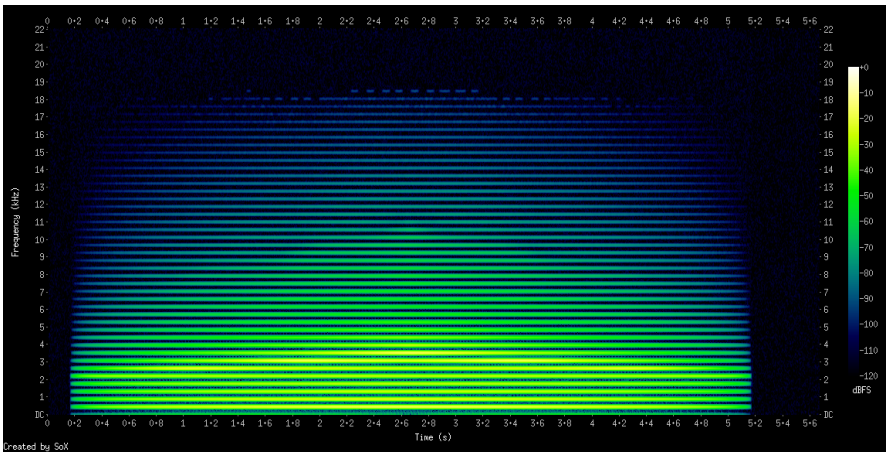
Her kan man bemærke, at den perciperede intensitet fremstår kraftigere, når cutoff-frekvensen øges, da øret rammes af et bredere spektrum af overtoner. Dette blot for at minde om de psykoakustiske forhold som spiller ind på opfattelsen af intensitet - amplitude er ikke nødvendigvis den afgørende faktor.

6.1.4 Cutoff moduleret af envelope

En meget udbredt klangdannelsesteknik angår en sammenkobling af envelope og cutoff-frekvens, således at cutoff-frekvensen moduleres af envelopen. Her kan vi på samme måde som ovenfor udregne en cutoff-frekvens baseret på envelope. Nedenstående SynthDef overdriver effekten med en `Env.triangle`, således at resultatet er tydeligt hørbart både i envelopens attack- og releasesegment.

```
(
SynthDef(\envCutoff, {
    arg freq = 440, pan = 0,
    amp = 0.1, out = 0, duration = 5;
    var sig = Saw.ar(freq);
    var env = EnvGen.kr(Env.triangle(duration), doneAction: 2);
    var cutoff = freq * env.range(2, 8);
    sig = RLPF.ar(sig, cutoff, 0.1);
    sig = sig * env;
    Out.ar(out, Pan2.ar(sig, pan, amp));
}).add;
)
Synth(\envCutoff);
```

Kodeboks 6.10: Cutoff-frekvens knyttet til envelope



Figur 6.6: Spektrogram: Cutoff-frekvens styret af trekantformet envelope

Denne teknik, hvor klangen (altså fordelingen af intensitet i overtonespektret) forandres over en enkelt tones levetid, kendetegner også lyden af akustiske instrumenter.

6.2 Enkle eksempler på subtraktiv syntese

Subtraktiv syntese er et omfattende emne. Udtrykket dækker ofte over forskellige teknikker, der anvendes til at emulere lyden af akustiske instrumenter. Dermed naturligvis ikke sagt, at resultatet altid klinger præcis som det akustiske forbillede; syntetiske blæser-, stryger- og trommelyde har fået deres egen genkendelige rolle inden for både den elektroniske musik og populærmusikken. Tænk blot på trommelydene fra en Roland *TR-808*-trommemaskine, der indgår i utallige hiphop-tracks.

Nedenstående klarinet- og strygerlyde er løseligt baseret på Pejrolo og Metcalfes (2017) gennemgang af subtraktive klangdannelsesteknikker.

6.2.1 Syntetisk klarinet

For at opnå en simpel, „klarinet“-lignende lyd, kan vi først og fremmest vælge en oscillator med en kvadratisk bølgeform (`Pulse`) som lydkilde. Dette skyldes, at den firkantede bølgeform udelukkende producerer „ulige“ overtoner, dvs. den første, tredje, og femte overtone (og så fremdeles). Dette skaber en „hul“ klang, som kan minde lidt om visse blæseinstrumenter. Vi kan blødgøre klangen ved at køre oscillatoren igennem et lavpas-filter med resonans, hvilket kan emulere den naturlige dæmpning af øvre frekvenser i et akustisk instrument (Pejrolo & Metcalfe, 2017, p. 119).

Bemærk i kildekoden herunder, hvordan cutoff-frekvensen beregnes ud fra oscillatorens frekvens: Cutoff-frekvens beregnes til at ligge 18 halvtoner (dvs. cirka 1½ oktav) over oscillatorens frekvens. Genlæs eventuelt afsnittet om skalering (se 4.2), hvis du er i tvivl om, hvordan dette fungerer.

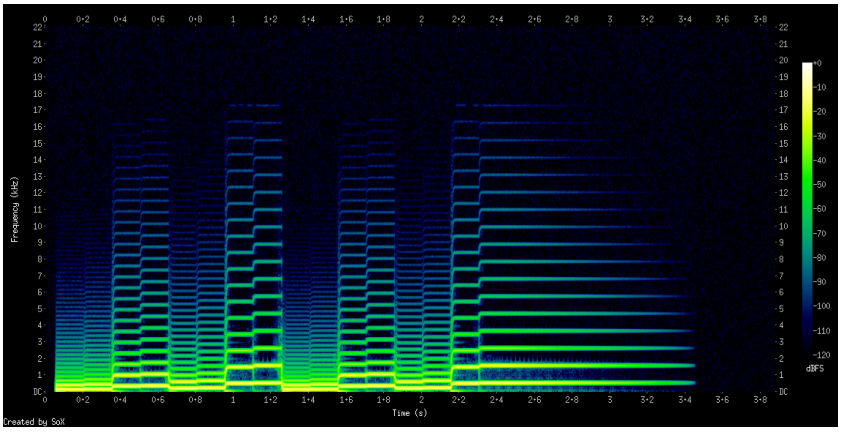
```
SynthDef(\clarinet, {
    arg freq = 440, gate = 1;

    // lydkilde: firkantet bølgeform med mulighed for glissando via .lag
    var oscillator = Pulse.ar(freq.lag(0.025));
    // resonant low pass-filter anvendes til klanglig justering
    var sig = RLPF.ar(oscillator, freq * 18.midiratio, 0.5);

    // lydstyrke styres med en envelope
    sig = sig * EnvGen.kr(Env.asr, gate, doneAction: Done.freeSelf) * 0.1;
    Out.ar(0, sig.dup);
}).add;
```

Kodeboks 6.11: SynthDef til syntetisk klarinet

Hvis vi skal „spille på“ denne SynthDef og ønsker at fastholde ideen om en klarinetlyd, er det værd at bemærke det måske indlysende faktum, at klarinetten er et *monofont* instrument. Det betyder, at der ikke kan være tidsligt overlappende toner som ved akkordinstrumenter som klaveret eller guitaren. Vi kan derfor oplagt bruge `Pmono` eller `PmonoArtic` til fx at spille et par skalaløb.

```
Pmono(\clarinet,
    \degree, Pseq([0, 1, 2, 3, 4, 5, 6, 7], 2),
    \octave, Pseq([4, 5], inf).stutter(2),
    \dur, 0.15,
).play;
```

Kodeboks 6.12: Et par smidige skalaløb



Figur 6.7: Spektrogram: Syntetisk klarinet med hyppige oktavspring

6.2.2 1980'er-strings

Når man emulerer lyden af strygerinstrument, kan man bruge oscillatorer, der genererer savtakkede bølgeformer. Bortset fra den meget skarpe klang af savtakkede bølgeformer, som opstår på grund af høj intensitet i overtonespektret, giver de savtakkede bølgeformer en passende klang. Her kan vi også kombinere to oscillatorer, der er stemt meget tæt på hinanden, hvilket giver en slags chorus-effekt - som om der er to akustiske instrumenter, som spiller på samme tid og varierer i frekvens i meget små intervaller, som kendetegner akustiske instrumenter, sangstemmer osv. Til sammen kan dette skabe en klang, der minder om 1980'er-strings (Pejrolo & Metcalfe, 2017, p. 120).

```
SynthDef(\stringz, {
    arg freq = 440, gate = 1;

    // lydkilde: to savtakkede bølgeformer, den ene med en smule detuning
    var oscillator = Saw.ar(freq) + Saw.ar(freq * 0.9971);

    // cutoff styres af oscillatorens frekvens samt af en envelope
    var cutoff = freq * EnvGen.kr(
        Env.adsr(0.2, 0.7),
        gate,
        levelBias: 1
    );
    var sig = RLPF.ar(oscillator, cutoff, 0.4);

    // lydstyrke styres med en separat ADSR-envelope
    sig = sig * EnvGen.kr(Env.adsr(0.2, 0.4), gate, doneAction: Done.freeSelf) * 0.1;
    Out.ar(0, sig.dup);
}).add;
```

Kodeboks 6.13: SynthDef til firser-strings

For at få et indtryk af de fyldige lydmuligheder kan vi afspille et par akkorder med vores syntetiske strygerlyd.

```
Pbind(
    \instrument, \stringz,
    \degree, [-7, 0, 2, 4, 6, 8],
    \mtranspose, Pseq([3, 1, 0]),
    \dur, 2,
).play;
```

Kodeboks 6.14: En akkordrække med firser-strings

6.3 Syntetisk lilletromme

Her udvikler vi en lilletrommelyd, som er inspireret af Gordon Reids artikel fra magasinet *Sound on Sound* (Reid, 2002b) om syntetisk dannelse af lilletrommelyd. Lad os med lydproduktionen her simulere forskellige klangelementer i en „realistisk" lilletrommelyd, fra trommens „krop" og resonans til seiding og anden støj. Strategien går ud på at bruge støj og simple oscillatorer moduleret af envelopes, sendt gennem forskellige filtre (se 6.5), der også moduleres af envelopes (se 5.1). Dette skal samlet set simulere lilletrommelydens klanglige forandring over (kort) tid - hvad Denis Smalley har kaldt lydens *spektromorfologi* (Smalley, 1997).

6.3.1 Elementerne i en lilletrommelyd

Resonansrummet inde i en lilletromme kan vi simulere med to sinustoner ved hhv. 180 Hz og 330 Hz, der klinger ud efter godt 200 millisekunder.

```
{
    SinOsc.ar([180, 330]).sum
    * EnvGen.ar(Env.perc(0.03, 0.2));
}.play;
```

Kodeboks 6.15: Lilletrommelyd: Primært resonansrum

Der vil typisk også være en række overtoner, hvis frekvens bevæger sig op og ned over de første 10 millisekunder. Disse simuleres herunder af et par `LFTri`-UGens, hvor både amplitude og frekvens moduleres af envelopes. Når man spiller på en fysisk tromme, vil der altid være en smule variation i klangen, og dette simuleres blandt andet ved, at varigheden af overtonernes volumen-envelope gøre tilfældig (100-120 ms).

```
{
    LFTri.ar(
      [286, 335] *
      EnvGen.ar(Env.new([1, 1.5, 1], [0.01, 0.09], \sine))
    ).sum * EnvGen.ar(
      Env.perc(0.01, Rand(0.09, 0.11))
    );
}.play;
```

Kodeboks 6.16: Lilletrommelyd: Overtoner

Når en trommestik rammer trommeskindet, genereres en støjimpuls, som både sætter hele trommen i svingninger og genererer en kort, støjende lyd, som ofte omtales som et „klik“.

```
{
    LPF.ar(
      HPF.ar(WhiteNoise.ar, 300),
      8000
    ) * EnvGen.ar(
      Env.linen(0.001, 0.01, 0.001)
    );
}.play;
```

Kodeboks 6.17: Lilletrommelyd: Klik

Derudover er der på undersiden af lilletrommen monteret en såkaldt „seiding“ - et metalbånd med små tråde, som vibrerer og skaber en karakteristisk raslende lyd, som er essentiel i lilletrommens klang. Her filtrerer vi den spektralt righoldige hvide støj, og volumen-envelopen er den længste, som derfor styrer, hvornår Synth'en fjernes fra lydserveren igen. Dette simulerer, at lyden fra seidingen er den sidste, der klinger ud i det korte forløb, en lilletrommelyd er.

```
{
HPF.ar(
    BPeakEQ.ar(WhiteNoise.ar, 4000, 0.5, 3),
    300
    ) * EnvGen.ar(
      Env.perc(0.05, Rand(0.16, 0.19)).delay(0.01),
      doneAction: Done.freeSelf
    );
}.play;
```

Kodeboks 6.18: Lilletrommelyd: Seiding

Trommen kan også have en dybere støjlyd, som simuleres på en lignende måde, blot med dybere cutoff-frekvenser.

```
{
    LPF.ar(
      HPF.ar(WhiteNoise.ar, 230),
      500
    ) * EnvGen.ar(
      Env.perc(0.1, Rand(0.09, 0.11))
    );
}.play;
```

Kodeboks 6.19: Lilletrommelyd: Dyb støj

6.3.2 En lilletromme-SynthDef

Herunder sættes de ovenfor omtalte komponenter sammen i en liste, som summeres til sidst. Vi indfører desuden nogle argumenter, så vi kan styre lydstyrken for de enkelte komponenter og på den måde fintune vores lilletromme. Hertil anvendes method'en `.dbamp`, som omregner fra dB-skalaen til en skaleringsfaktor, vi kan bruge til at styre lydstyrke. For at have en balance i lydniveauerne mellem de enkelte lydkilder indeholder SynthDef'en nogle forudprogrammerede lydniveauer, som sammen med argumenterne styrer lydstyrkerne. Sidst men ikke mindst foretages der lidt distortion/komprimering med `.tanh`, som er en form for *waveshaping*[1].

[1] Waveshaping er en digital form for klanglig manipulation med distortion, som det vil føre for vidt at introducere her. Se evt. Curtis Roads' udmærkede introduktion til emnet (2023, p. 273).

```
SynthDef(\snare,{
    arg pan = 0, amp = 0.1, out = 0,
    body = 0, harmonics = 0, click = 0,
    highNoise = 0, lowNoise = 0;

    var sig = [
        // Resonansrum
        SinOsc.ar([180, 330]).sum
        * EnvGen.ar( Env.perc(0.03, 0.2))
        * (body - 3).dbamp,

        // Overtoner
        LFTri.ar([286, 335] *
          EnvGen.ar(Env.new([1, 1.5, 1], [0.01, 0.09], \sine))
        ).sum * EnvGen.ar(Env.perc(0.01, Rand(0.09, 0.11)))
         * (harmonics + 2).dbamp,

        // Klik
        LPF.ar(HPF.ar(WhiteNoise.ar, 300), 8000)
        * EnvGen.ar(Env.linen(0.001, 0.01, 0.001))
        * click.dbamp,

        // Seiding
        HPF.ar(BPeakEQ.ar(WhiteNoise.ar, 4000, 0.5, 3),300)
        * EnvGen.ar(
          Env.perc(0.05, Rand(0.16, 0.19)).delay(0.01),
          doneAction: Done.freeSelf)
        * (highNoise - 8).dbamp,

        // Dyb støj
        LPF.ar(HPF.ar(WhiteNoise.ar, 230),500)
        * EnvGen.ar(Env.perc(0.1, Rand(0.09, 0.11)))
        * (lowNoise-5).dbamp
      ].sum;

    sig = (sig * 1.5).tanh;  // Distortion/komprimering
  Out.ar(out, Pan2.ar(sig, pan, amp));
}).add;
```

Kodeboks 6.20: En SynthDef til syntetisk emuleret lilletrommelyd

Med denne SynthDef kan vi producere forskelligt klingende lyde. Herunder afspilles standardudgaven samt en version, hvor seidingen er slået fra.

```
// Standardindstillinger
Synth(\snare);

(
// Mindre seiding/støj
Synth(\snare, [
  \body, 3,
  \harmonics, -2,
  \click, -15,
  \highNoise, -40,
  \lowNoise, -40,
]);
)
```

Kodeboks 6.21: To forskellige lilletrommelyde

6.4 Simulering af strenge

En kerne af teknikker inden for subtraktiv syntese går ud på at sende en støjimpuls igennem en feedbackløkke med delay og filtrering. Dette simulerer på sin vis den fysiske filtrering af lydbølger, som dæmpes, når de bevæger sig igennem et materiale.

6.4.1 Karplus-Strong

Et centralt eksempel er her den berømte Karplus-Strong-algoritme, som har til formål at generere en klang, der minder om lyden af en streng, der bliver slået an (Karplus & Strong, 1983). Der findes i SuperCollider en særlig UGen, der implementerer Karplus-Strong kaldet **Pluck**, som vi kigger på om lidt. Men det kan være interessant at se, hvordan algoritmen fungerer, hvorfor et mere håndholdt eksempel er inkluderet herunder. Hertil bruger vi et par UGens, som vi ikke har set hidtil, blandt andet **LocalIn** og **LocalOut**, som definerer en feedbackløkke, samt **DelayC**, som forsinker lydsignalet (med feedback). Decay-tiden og **OnePole**-filterets koefficient er afgørende for tonehøjde og klang.

```
{
    var freq = 440, coef = 0.2;
    var sig = LocalIn.ar(1); // Feedback-løkke starter
    var noise = Impulse.ar(0); // Støjimpuls
    var sound = DelayC.ar(
        noise + (sig * 0.99), // 0.99 styrer decay-tiden
        20.reciprocal,
        freq.reciprocal - ControlRate.ir.reciprocal
    );
    // Filter
    sound = OnePole.ar(sound, coef);

    LocalOut.ar(sound); // Feedback-løkke slutter
    sig;
}.play
```

Kodeboks 6.22: Manuel Karplus-Strong

En tilsvarende lyd kan opnås mere effektivt med **Pluck**, som vi derfor anvender herefter.

```
{
    var freq = 440, coef = 0.2;
    Pluck.ar(
        in: WhiteNoise.ar,
        trig: Impulse.ar(0),
        maxdelaytime: 0.1,
        delaytime: freq.reciprocal,
        decaytime: 0.99,
        coef: coef
    );
}.play;
```

Kodeboks 6.23: Karplus-Strong med Pluck

6.4.2 Inkorporering i SynthDef

Vi kan inkorporere ovenstående i en SynthDef med argumenter for frekvens, decay-tid og filter-koefficient. Herunder er der også inkluderet et argument til simulering af vibrato. Derudover anvendes UGen'en `DetectSilence` til at fjerne Synth'en fra lydserveren, når lyden har klinget ud (hvilket er nødvendigt, da vi ikke styrer lydstyrken med en envelope, som ellers ville kunne udføre denne funktion).

```
SynthDef(\karplus, {
    arg freq = 440, decay = 1, coef = 0.1,
    pan = 0, amp = 0.1, out = 0, vibrato = 0.05;

    var freqScale = SinOsc.ar(
        XLine.kr(7, 1, decay)
    ).bipolar(vibrato).midiratio;

    var sig = Pluck.ar(
        in: WhiteNoise.ar,
        trig: Impulse.kr(0),
        maxdelaytime: freq.reciprocal * vibrato.midiratio,
        delaytime: freq.reciprocal * freqScale,
        decaytime: decay,
        coef: coef
    );

    DetectSilence.ar(sig, doneAction: 2);
    sig = Pan2.ar(sig, pan, amp);
    Out.ar(out, sig);
}).add;
```

Kodeboks 6.24: Karplus-Strong SynthDef

Med ovenstående SynthDef indlæst kan vi prøve klangmulighederne af.

```
Synth(\karplus, [\coef, 0.1, \decay, 1])
Synth(\karplus, [\coef, 0.4, \decay, 1])
Synth(\karplus, [\coef, 0.7, \decay, 1])
Synth(\karplus, [\freq, 110, \coef, 0.005, \decay, 10, \vibrato,
↪  0.25])
```

Kodeboks 6.25: Klanglige variationsmuligheder

6.4.3 Kompositionseksempel

Vi kan selvfølgelig også skrive en lille komposition med patterns, hvor ovenstående SynthDef benyttes.

```
TempoClock.tempo = 110/60;
Pbind(
    \instrument, \karplus,

    // Streng-indstillinger
    \decay, Pgauss(5, 1).clip(0, 10),
    \coef, Pgauss(0.55, 0.05),
    \vibrato, Pexprand(0.1, 0.02),

    // Tonehøjde
    \degree, Pseries(
        start: 0,
        step: Pwhite(1, 4).asStream
        * Prand([-1, 1], inf).asStream,
        length: 4
    ).repeat,
    \mtranspose, Pwhite(-4, 4).stutter(3),
    \root, -3,
    \scale, Scale.minorPentatonic,

    // Rytmik
    \dur, Pseq([
        Prand([1/16, 1/32, 1/8, 3/16] * 4, Pwhite(3,
        ↪  12).asStream),
        Prand([Rest(2.5), Rest(2), Rest(3)])
    ], inf),
    \lag, Pgauss(0, 0.003),

    // Panorering og lydstyrke
    \pan, Pbrown(-0.3, 0.3, 0.2),
    \db, Pgauss(-20, 1),
).play;
```

Kodeboks 6.26: Komposition med Karplus-Strong

6.5 Cheat sheet: Filter-UGens

OBS: Pas på din hørelse!

- Undgå cutoff-frekvenser tæt på 0 - brug værdier mellem 20 Hz og 20 kHz.
- Sæt ikke `rq` til 0 (ved filter med resonans).

Filter-UGens kan bruges på mange forskellige lydkilder, men hvid støj (`WhiteNoise.ar`) er særligt nyttigt til at visualisere, hvordan filtre påvirker lydsignalet, da hvid støj teoretisk set har samme lydstyrke i hele frekvensspektret. Cutoff- eller centerfrekvensen er i næsten alle eksempler herunder 440 Hz, så filtrenes respons kan sammenlignes.

```
// Low/high pass filter, 2. orden
{LPF.ar(WhiteNoise.ar * 0.5, 440)}.play;
{HPF.ar(WhiteNoise.ar * 0.5, 440)}.play;

// Low/high pass filter med resonans, 2. orden, 1/Q = 0.1
{RLPF.ar(WhiteNoise.ar * 0.5, 440, 0.1)}.play;
{RHPF.ar(WhiteNoise.ar * 0.25, 440, 0.1)}.play;

// Low/high pass filter med resonans, 4. orden, 1/Q = 0.1
{BLowPass4.ar(WhiteNoise.ar * 0.5, 440, 0.1)}.play;
{BHiPass4.ar(WhiteNoise.ar * 0.5, 440, 0.1)}.play;

// Moog-emulerende low pass filter, "resonans" = 3
{MoogFF.ar(WhiteNoise.ar * 0.5, 440, 3)}.play;
```

Kodeboks 6.27: Low/high pass filtre

```
// Low/high shelf filter, gain -20 dB
{BLowShelf.ar(WhiteNoise.ar * 0.5, 440, 1, -20)}.play;
{BHiShelf.ar(WhiteNoise.ar * 0.5, 440, 1, -20)}.play;

// Peak EQ filter, 1/Q = 0.8, gain = +30 dB
{BPeakEQ.ar(WhiteNoise.ar(0.05), 440, 0.8, 30)}.play;

// Band pass filter, 1/Q = 0.5
{BPF.ar(WhiteNoise.ar * 0.5, 440, 0.5)}.play;
{Resonz.ar(WhiteNoise.ar * 0.5, 440, 0.5)}.play;

// Band reject (notch) filter, 1/Q = 0.9 - bred profil
{BRF.ar(WhiteNoise.ar * 0.5, 440, 0.9)}.play;

// Band reject (notch) filter, båndbredde 4 oktaver
{BBandStop.ar(WhiteNoise.ar * 0.5, 440, 4)}.play;
```

Kodeboks 6.28: Øvrige EQ-lignende filtre

```
// Band pass filter med feedback-resonans, decaytid på 1 sekund
{Ringz.ar(WhiteNoise.ar(0.005), 440, 1)}.play;

// Bank af band pass filtre med feedback-resonans og decaytid på
↪  1 sekund
{Klank.ar(`[[440, 923, 1153, 1723], nil, [1, 1, 1, 1]],
↪  Dust.ar(5, 0.5))}.play;
```

Kodeboks 6.29: Specialfiltre

Der findes mange andre filtre, fx **Comb** og **Allpass**, men de anvendes typisk til specielle formål som rumklangsdesign og feedbackbaserede effekter.

6.6 Øvelse: Anvendelse af filtre

Denne øvelse går ud på at anvende og modulere gængse filter-UGens.

6.6.1 Valg og indstilling af filtre

Brug følgende filtre til at modificere klangen af hvid støj:

1. Et low pass-filter med cutoff-frekvens på 1000 Hz.
2. Et high pass-filter med cutoff-frekvens på 800 Hz.
3. Et band pass-filter med centerfrekvens på 500 Hz.
4. Et resonerende low pass-filter med cutoff-frekvens på 800 Hz og rq-værdi på 0.1.

Du kan finde hjælp og eksempler i cheat sheetet om filtre (se 6.5).

```
{
    var source = WhiteNoise.ar;
    var sig =   ;
    sig * 0.1;
}.play;
```

Kodeboks 6.30: Filtreret støj

6.6.2 Modulation af filtre

Brug følgende kilder til at modulere cutoff-frekvensen for et low pass-filter, så cutoff-frekvensen bevæger sig mellem 500 Hz og 1000 Hz (se evt. afsnittet om modulation af UGens (se 4.2) for relevante teknikker).

1. Den allerede noterede envelope.
2. En `XLine`-envelope, vælg selv tidsinterval.
3. En LFO-UGen - vælg selv bølgeform og passende frekvens.

```
{
    var source = PinkNoise.ar;
    var env = EnvGen.ar(Env.linen(0.5, 0.5, 2), doneAction:
    ↪ Done.freeSelf);
    var cutoff =     ;
    var sig = LPF.ar(source, cutoff);
    sig * env;
}.play;
```

Kodeboks 6.31: Modulation af cutoff-frekvens

6.6.3 Subtraktiv SynthDef

Omskriv kildekoden fra ovenstående opgave til en SynthDef, som gør brug af subtraktiv syntese. Du kan evt. gøre brug af en SynthDef-skabelon (se 5.5) som ramme for dit arbejde. Se også hvordan en SynthDef adskiller sig fra UGen-funktioner (se 5.3.2).

SynthDef'en skal overholde følgende krav:

1. Lydkilden ændres fra pink støj til en oscillator med en periodisk bølgeform og et righoldigt spektrum (fx en savtakket eller firkantet bølge).
2. Oscillatorens frekvens styres af et argument kaldet `freq` med default-værdi 440 Hz.
3. Filteret ændres til typen **RLPF**.
4. Filterets cutoff-frekvens bevæger sig mellem 2 og 4 oktaver over oscillatorfrekvensen i takt med envelope-generatoren.
5. Filterets rq-værdi styres af et SynthDef-argument med default-værdi 0.5.
6. Følgende envelope-parametre kan styres ved hjælp af SynthDef-argumenter:
    a) `attackTime`, default-værdi 0.1.
    b) `curve`, default-værdi 0.

Skriv en Pbind-komposition, som demonstrerer SynthDef'ens forskellige klangmuligheder, dvs. hvor nøglerne i kodeblokken herunder varieres ved hjælp af pattterns. Husk at erstatte `\navnetPåMinFantastiskeSynthDef` med det navn, du selv har valgt.

```
Pbind(
    // Valg af SynthDef
    \instrument, \navnetPåMinFantastiskeSynthDef,

    // Tonehøjde
    \degree,     ,

    // Envelope
    \attackTime,        ,
    \curve,         ,

    // Filter
    \rq,    ,
).play;
```

Kodeboks 6.32: Komposition for subktraktiv SynthDef

KAPITEL 7

Klangdannelse med oscillatorbanke

Dette kapitel introducerer til multikanalslyd i SuperCollider. Dette er et potentielt meget vidtrækkende emne, da SuperCollider tillader os at arbejde med banke af oscillatorer og andre UGens meget fleksibelt. I dette kapitel kigger vi blandt andet på additiv syntese, der netop baseres på banke af lydsignaler, som tilsammen kan skabe et overtonespektrum, præcist som vi ønsker det. Ved hjælp af additiv syntese skaber vi i dette kapitel standardbølgeformer, emulering af elektrisk orgel samt dannelse af forskellige klokkelyde. Banke af UGens kan dog også bruges til andet end additiv syntese, og vi skal også omkring brug af UGen-banke til fremstilling af en klassisk vocoder.

7.1 Oscillatorbanke og multikanalslyd

En af de store styrker ved SuperColliders UGen-system er evnen til fleksibelt at arbejde med store banke af oscillatorer og lydkanaler. Men netop dette område kan samtidig være et lidt subtilt system at forstå og navigere, da det notationsmæssigt kan være vanskeligt at få øje på. Det er værd at dykke ned i, fordi det er et oplagt redskab til at arbejde med additiv klangdannelse, detuning, chorus-effekter, multikanalslyd og lignende.

7.1.1 Duplikering

Den mest simple måde at skabe et lydsignal med flere kanaler er ved at duplikere en UGen. Det kan vi gøre med method'en `.dup`, der også gælder for duplikering af andre objekter end UGens. Som argument til `.dup` kan vi angive, hvor mange kopier, vi ønsker - defaultværdien er 2.

```
10.dup; // -> [10, 10]
10.dup(5); // -> [10, 10, 10, 10, 10]
SinOsc.ar.dup; // -> [ a SinOsc, a SinOsc ]
SinOsc.ar.dup(7); // -> [ a SinOsc, a SinOsc, a SinOsc, a
↪  SinOsc, a SinOsc, a SinOsc, a SinOsc ]
```

Kodeboks 7.1: Duplikering med .dup

Der findes en syntaktisk genvej, som kan være nyttig at kende, nemlig operatoren `!`, der blot gør det samme som `.dup` med argument.

```
\kaffe ! 3; // -> [ kaffe, kaffe, kaffe ]
Pulse.ar ! 4; // -> [ a Pulse, a Pulse, a Pulse, a Pulse ]
```

Kodeboks 7.2: Operatoren ! fungerer ligesom .dup

Når vi kører ovenstående kodelinjer, kan vi i post window se, at SuperCollider skaber *lister*, som indeholder kopier af det duplikerede objekt. Hertil kan man eventuelt genlæse afsnittet om lister (se 1.6). Men når vi bruger `.dup` på oscillatorer i en UGen-funktion, kan man sige, at *listen udgør et lydsignal med mere end én kanal*. Duplikering er altså én simpel teknik til at skabe multikanalslyd. Her kan man dog sige, at der egentlig ikke er tale om stereofoni, hvis signalet i de to kanaler er helt identisk - så er det nærmere en slags fordoblet monofoni.

```
{ SinOsc.ar.dup * 0.1; }.play;
```

Kodeboks 7.3: Dobbelt monofoni

7.1.2 Panorering og stereofoni

Vi har hidtil typisk arbejdet med monofone (enstemmige) signaler. Ved hjælp af UGen'en **Pan2** kan vi placere et monofont signal i et stereofelt. UGen'en har tre argumenter: **Pan2**.ar(sig, pan, amp), hvor sig indeholder det monofone signal, som placeres i et stereofelt ud fra argumentet pan (hvor -1 er helt til venstre og 1 helt til højre) med lydstyrken amp. **Pan2** indgår ofte som det sidste trin i en UGen-funktion, fordi den både inkorporerer panorering og justering af lydstyrke.

```
// Pink støj i midten af stereofeltet
{ Pan2.ar(PinkNoise.ar, 0, 0.1) }.play;

// Pink støj helt til venstre af stereofeltet
{ Pan2.ar(PinkNoise.ar, -1, 0.1) }.play;

// Pink støj i midten af stereofeltet
{ Pan2.ar(PinkNoise.ar, 1, 0.1) }.play;
```

Kodeboks 7.4: Panorering med Pan2

Panoreringensargumentet kan selvfølgelig moduleres af en anden UGen. Her er det væsentligt at sikre sig, at modulatoren er korrekt skaleret (se 4.2), da panoreringsværdier skal ligge mellem -1 og 1. **LFTri** bevæger sig, som de fleste andre oscillatorer, netop i dette interval.

```
{
    var pan = LFTri.kr(0.5);
    var sig = Pulse.ar;
    Pan2.ar(sig, pan, 0.1)
}.play;
```

Kodeboks 7.5: Modulation af panorering

Der findes **Pan4** og **PanAz**, som på tilsvarende vis panorerer et monofont signal mellem henholdsvis 4 eller et vilkårligt antal højttalere. Skal man derimod panorere et signal, som allerede er i stereo, er **Pan2** ikke det rette valg - i stedet kan man

anvende **Balance2**, som i stedet for at lave et monofont signal til stereo justerer balancen mellem to input-kanaler.

7.1.3 Multikanalslyd med multichannel expansion

Som vi så ovenfor med duplikering, har vi faktisk ikke brug for specielle UGens som **Pan2** for at kunne arbejde med multikanalslyd. På et grundlæggende niveau understøtter de fleste UGens, at de på enkel vis kan „ekspanderes“ med en teknik, der kaldes *multichannel expansion*. Dette er et lidt tricky emne at forstå, men det er egentlig ganske enkelt. Giver vi fx **SinOsc**.ar en liste med frekvenser i stedet for blot en enkelt frekvens, kan vi høre, at der oprettes flere forskellige oscillatorer.

```
1 {SinOsc.ar([100, 250, 719, 1682]) * 0.1}.play;
2 {SinOsc.ar([100, 250, 719, 1682]).sum * 0.1}.play;
3 {Splay.ar( SinOsc.ar([100, 250, 719, 1682]) ) * 0.1}.play;
```

Kodeboks 7.6: Multichannel expansion

Hvad hører vi i dette eksempel? På et stereosystem eller hovedtelefoner kan vi kun høre to kanaler ad gangen:

- På første linje hører vi en sinustone ved 100 Hz i venstre side og én ved 250 Hz i højre. De højeste toner kan vi ikke høre[1].
- På 2. linje summerer vi signalet med method'en .sum, og her er alle fire sinustoner hørbare, fordi de bliver lagt sammen og således kun udgør én kanal - den venstre.
- På 3. linje fordeler vi et vilkårligt antal kanaler jævnt i et stereofelt med UGen'en **Splay**.

Det tricky her er den subtile notation: Selvom vi kun noterer én **SinOsc**, bliver der automatisk oprettet 3 ekstra **SinOsc**-UGens, svarende til det antal frekvenser, som fremgår af listen.

[1] Vi kan forvisse os om, at der faktisk produceres mere end to kanaler, ved at indstille SuperColliders lydserver til at arbejde med fx 8 outputkanaler og vise lydstyrken på disse. Det gør vi med `s.options.numOutputBusChannels = 8;`. Efter `s.reboot;` kan vi så køre `s.meter;`, som viser signalets amplitude på de 8 kanaler (hvoraf vi dog stadig kun kan høre to, hvis vi lytter på et almindeligt stereosetup med højttalere eller hovedtelefoner).

7.1.3.1 Videreførelse af ekspansion

Ekspansionen videreføres til UGens, som ligger senere i signalkæden. I eksemplet herunder opretter vi først to tilfældighedsgeneratorer, den ene genererer 5 tilfældige værdier pr. sekund, den anden 10. Når **SinOsc** på næste linje modtager dette signal med to kanaler, ekspanderes der videre, så vi får to tilsvarende sinusbølge-oscillatorer.

```
{
    var freqs = LFNoise0.kr([5, 10]).exprange(200, 800);
    SinOsc.ar(freqs); // skaber to sinusbølge-oscillatorer
}.play;
```

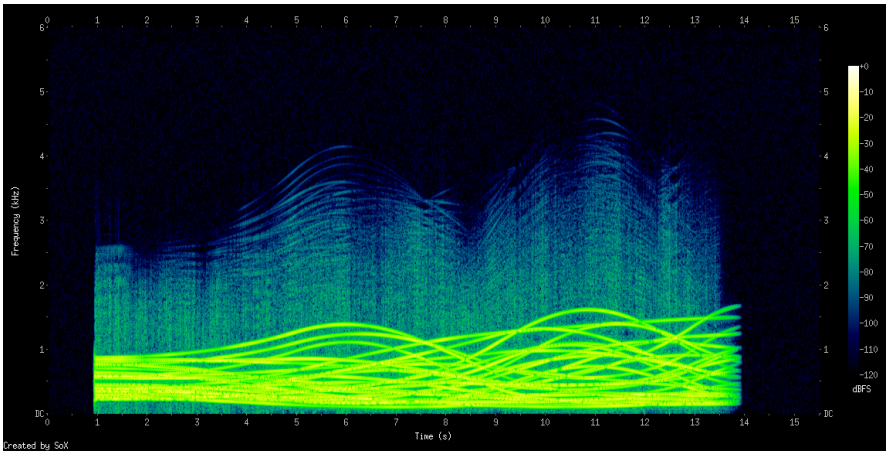
Kodeboks 7.7: Stereosignal med arrays

Dette betyder også, at vi med en ganske begrænset mængde kildekode kan oprette et væld af oscillatorer og, fx ved hjælp af **Splay**, samle dem igen til et meget fyldigt lydbillede. Herunder kan vi som eksempel bruge duplikering som vist ovenfor (se 7.1.1) til at generere et signal med 30 oscillatorer af hver slags (**LFNoise2**, **LFNoise1**, **SinOsc**).

```
{
    var freqs = LFNoise2.kr(0.25.dup(30)).exprange(110, 1760);
    var amps = LFNoise1.kr(2.dup(30)).exprange(0.01, 0.1);
    var sig = SinOsc.ar(freqs) * amps;
    Splay.ar(sig);
}.play;
```

Kodeboks 7.8: Mange oscillatorer med duplikering og multichannel expansion

Dette kan vi også kalde en *oscillatorbank*. Her kan vi forhåbentlig begynde at ane, hvordan brug af multikanalslyd kan være en effektiv teknik inden for lyddesign.



Figur 7.1: Spektrogram: 30 sinusbølge-oscillatorer styres af lavfrekvente støjgeneratorer

7.1.4 Detuning

Oscillatorbanke er fx særdeles velegnede til at skabe „tykkere“ klange gennem detuning. Her opretter vi med multichannel expansion fire forskellige oscillatorer, der klinger med næsten samme frekvens, lægger dem sammen med `.sum` og panorerer til sidst med **Pan2**.

```
{
    var freq = 220;
    var transposeFactors = [0, 0.10, -0.07, -0.08].midiratio;
    var sig = Saw.ar(freq * transposeFactors);
    sig = sig.sum * 0.1;
    Pan2.ar(sig);
}.play;
```

Kodeboks 7.9: Detuning med .midiratio

7.1.5 Algoritmisk oprettelse af oscillatorbanke

Computere er glimrende redskaber til at gentage de samme operationer mange gange, så længe vi beskriver præcist hvad der skal gøres. Vi kan danne oscillatorbanke ved at beskrive hvad der skal skabes med en funktion (se 1.4), der så udføres og skaber oscillatorbanken ved hjælp af iteration (se 1.6.3). Genlæs evt. de relevante afsnit herom, før du læsere videre.

7.1.5.1 Algoritmisk dannede oscillatorbanke

Hvis vi eksempelvis vil skabe en klang med de første toner i den naturlige overtonerække, kan vi oprette en liste med grundtonen og frekvenserne for de 4 oktaver, som ligger over grundtonen. Her lægger vi 1 til, fordi vi som bekendt tæller fra 0 (se 1.6.1).

```
5.collect({
    arg num;
    var octave = num + 1;
    var freq = 110;
    freq * octave;
}); // -> [ 110, 220, 330, 440, 550 ]
```

Kodeboks 7.10: Grundtone og overtoner

Vi kan omsætte dette til et klingende eksempel ved at lade tonerne fremstille sinusbølge-oscillatorer og blive fordelt jævnt i et stereofelt med **Splay**.

```
{
    var freq = 110;
    var sig = 5.collect({
        arg num;
        var octave = num + 1;
        SinOsc.ar(freq * octave);
    });
    Splay.ar(sig);
}.play;
```

Kodeboks 7.11: Fordeling af overtoner i stereofeltet

Dette princip kan udfoldes og danne rammen for mange forskellige lyddesign, heriblandt først og fremmest additiv klangdannelse.

7.2 Additiv syntese

Klassisk additiv syntese baseres på ideen om, at komplekse bølgeformer og dermed interessante klange kan fremstilles (approksimativt) ved ganske enkelt at sammenlægge en række sinusbølger, som svinger ved forskellige frekvenser og amplituder. Den matematiske ide bag dette findes i teorien om Fourierrækker.

7.2.1 Standardbølgeformer

Som eksempel på additiv syntese kan vi starte med at generere tre standardbølgeformer. Kilden til algoritmerne er Thor Magnussons (2021) udmærkede introduktion til additiv syntese. Sørg for at visualisere lydserverens output (se 4.1).

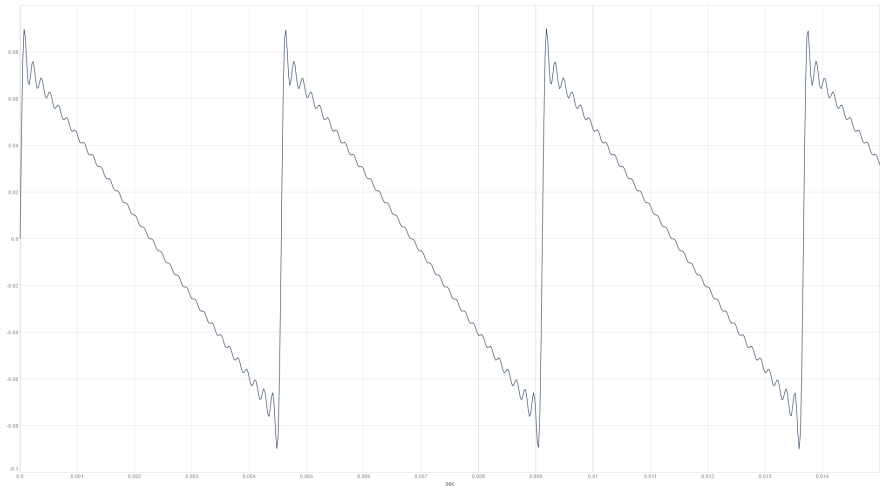
Disse bølgeformer udnytter den såkaldt naturlige overtonerække, som består af en række overtoner med frekvenser, der ligger et helt antal gange over en grundtones frekvens. Hvis grundtonen svinger ved 100 Hz, vil den første tone i den naturlige overtonerække svinge ved 200 Hz, den næste ved 300 Hz, derefter 400 Hz og så fremdeles.

7.2.1.1 Savtakket bølgeform

Den savtakkede bølgeform indeholder alle overtoner i overtonerækken. Her blot de første 30 overtoner.

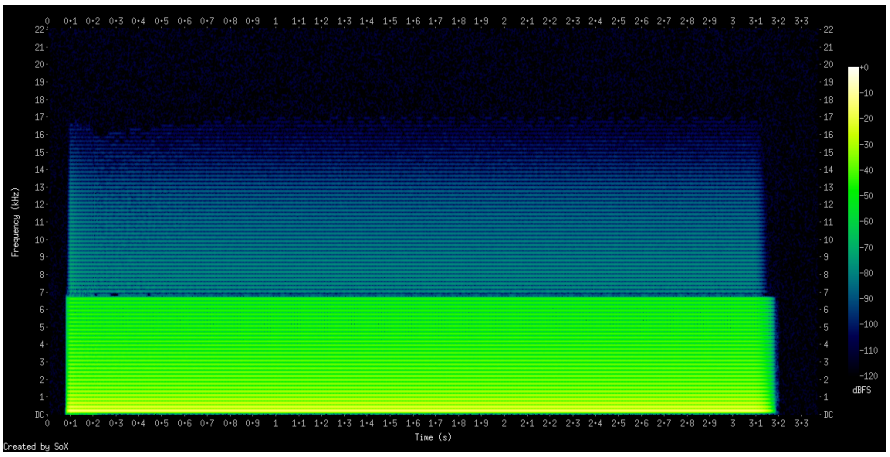
```
{
    var sig = 30.collect({
        arg num;
        var overtone = num + 1;
        SinOsc.ar(220 * overtone) * 0.5 / overtone;
    });
    sig.sum * 0.1;
}.play;
```

Kodeboks 7.12: Fremstilling af en savtakket bølgeform ved addition af sinusbølger



Figur 7.2: Bølgeform: Savtakket lydbølge genereret ved additiv syntese

Bemærk på spektrogrammet, at overtonerne stopper brat ved cirka 30 * 220 Hz = 6,6 kHz.



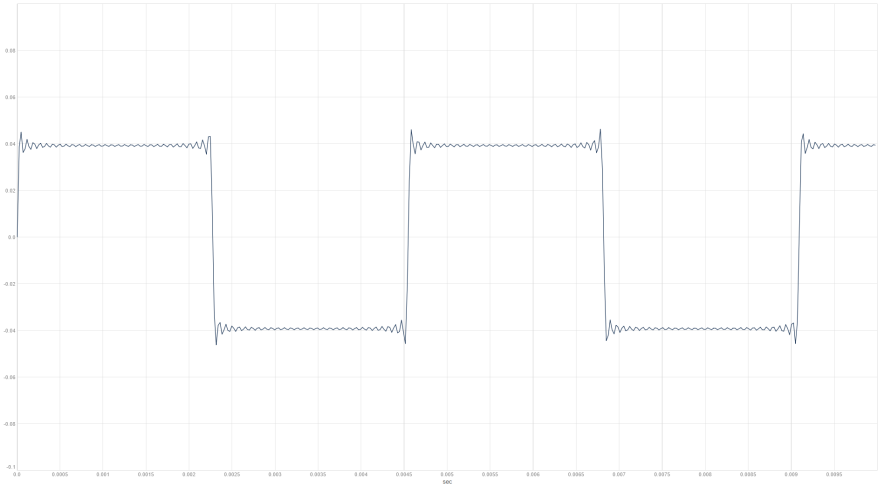
Figur 7.3: Spektrogram: Savtakket lydbølge genereret ved additiv syntese

7.2.1.2 Firkantet bølgeform

Den firkantede bølgeform kan skabes med overtonerne med ulige numre, dvs. nr. 1 (grundtonen), nr. 3, nr. 5, nr. 7 osv. Overtonerne falder i amplitude, jo højere vi kommer op. Her blot de første 30 overtoner.

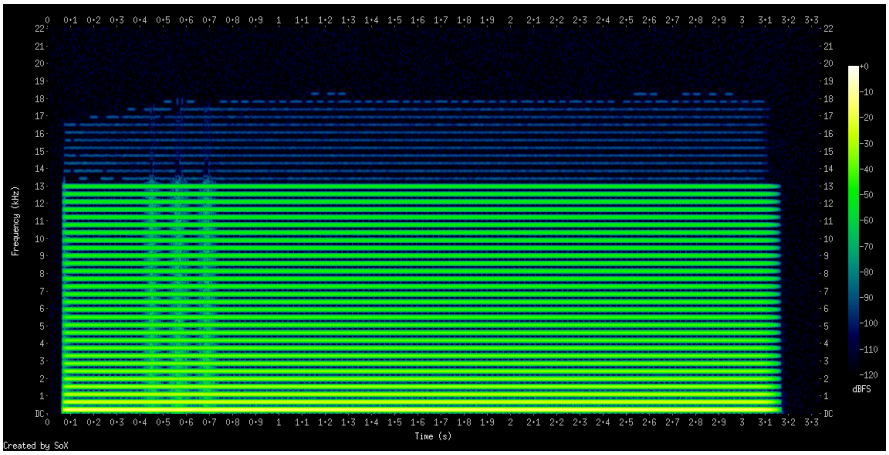
```
{
    var sig = 30.collect({
        arg num;
        var overtone = num * 2 + 1;
        SinOsc.ar(220 * overtone) * 0.5 / overtone;
    });
    sig.sum * 0.1;
}.play;
```

Kodeboks 7.13: Fremstilling af en firkantet bølgeform ved addition af sinusbølger



Figur 7.4: Bølgeform: Firkantet lydbølge genereret ved additiv syntese

Bemærk på spektrogrammet, hvordan der er større afstand mellem overtonerne end ved den savtakkede bølgeform (se 7.2.1.1) ovenfor.



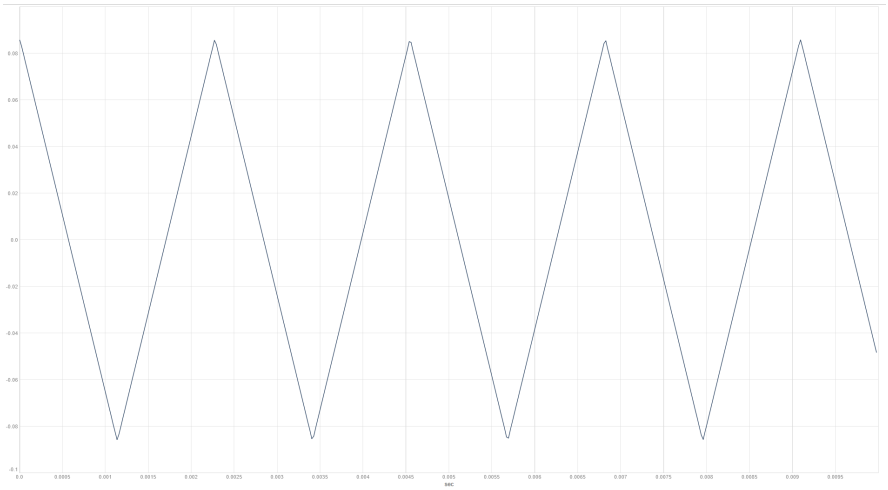
Figur 7.5: Spektrogram: Firkantet lydbølge genereret ved additiv syntese

7.2.1.3 Trekantet bølgeform

Den trekantede bølgeform kan ligesom den firkantede skabes med overtonerne med ulige numre, dvs. nr. 1 grundtonen, nr. 3, nr. 5, nr. 7 osv. Overtonerne falder i amplitude, jo højere vi kommer op, men med en lidt anden formel end ved den firkantede. Her blot de første 30 overtoner.

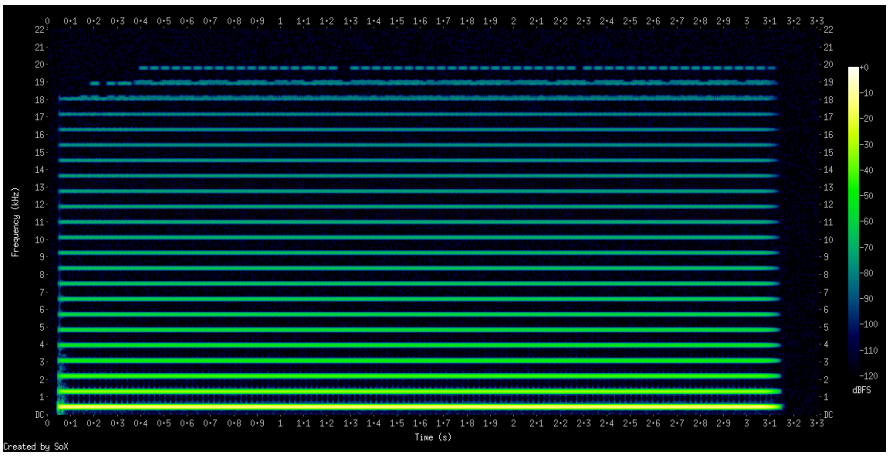
```
{
    var sig = 30.collect({
        arg num;
        var overtone = num * 2 + 1;
        SinOsc.ar(440 * overtone, pi/2) * 0.7 / overtone.pow(2);
    });
    sig.sum * 0.1;
}.play;
```

Kodeboks 7.14: Fremstilling af en trekantet bølgeform ved addition af sinusbølger



Figur 7.6: Bølgeform: Trekantet lydbølge genereret ved additiv syntese

Bemærk på spektrogrammet, hvordan der er endnu større afstand mellem overtonerne end ved den savtakkede (se 7.2.1.1) og den firkantede (se 7.2.1.2) bølgeform ovenfor.



Figur 7.7: Spektrogram: Trekantet lydbølge genereret ved additiv syntese

7.2.2 Andre eksempler

Man behøver naturligvis ikke begrænse sig til standardbølgeformer eller for den sags skyld den naturlige overtonerække.

7.2.2.1 En ikke-standard bølgeform

Her er eksempelvis en ikke-standard bølgeform, som består af hver tredje overtone.

```
{
    var sig = 15.collect({
        arg num;
        var overtone = num * 3 + 1;
        SinOsc.ar(220 * overtone) * 0.6 / overtone
    });
    sig * 0.1;
}.play;
```

Kodeboks 7.15: Fremstilling af en bølgeform med hver 3. overtone fra overtonerækken

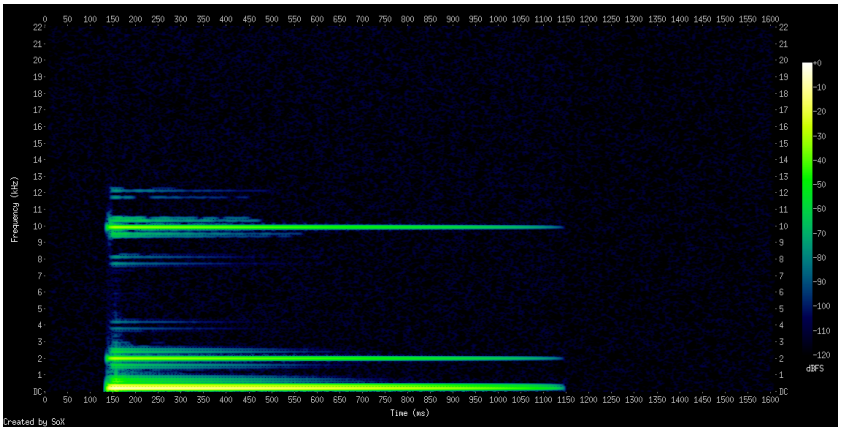
7.2.2.2 En klokkelyd

Dette eksempel på en klokkelyd er løseligt baseret på et eksempel fra Curtis Roads (2023, p. 123). Lyden indeholder følgende sinustoner:

- En *grundtone*, som klinger ved 200 Hz.
- En *overtone*, som klinger ved 2000 Hz.
- To *partialtoner* (dvs. ikke-harmoniske overtoner), som klinger ved henholdsvis 347,5 Hz og 9921,8 Hz.

```
{
    var freqs = [200, 347.5, 2000, 9921.8];
    var amplitudes = [0.73, 0.18, 0.05, 0.04];
    var sig = SinOsc.ar(freqs) * amplitudes;
    var env = EnvGen.kr(Env.perc, doneAction: Done.freeSelf);
    sig = sig.sum;
    sig.dup * env * 0.1;
}.play;
```

Kodeboks 7.16: Klokkelyd beskrevet af Curtis Roads



Figur 7.8: Spektrogram: Klokkelyd beskrevet af Curtis Roads, genereret ved additiv syntese

7.2.2.3 En mere kaotisk klokkelyd

Klokkelyde er generelt kendetegnet ved et højt indhold af partialtoner. For at skabe unikke lyde kan vi give hver partialtone en tilfældig frekvens, amplitude og envelope. Dertil udnytter vi de to generatorer **Rand** og **ExpRand**, som genererer en tilfældig værdi mellem et minimum og et maksimum, når Synth'en startes. Da vi ikke har én envelope, som styrer lydstyrken, kan vi heller ikke vide hvilken af de 16 envelopes, der skal afslutte Synth'en med `doneAction`. Derfor anvender vi en særlig UGen kalder **DetectSilence**, som kan stoppe Synth'en, når den har detekteret stilhed i et givet tidsrum.

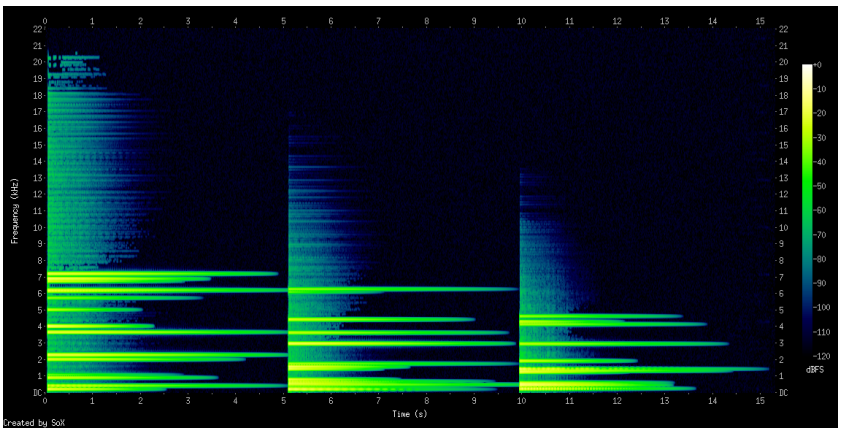
Kør blokken flere gange for at høre variationsmulighederne. Bemærk hvor længe de forskellige partialtoner klinger på grund af forskellige længder af release-segmenter.

Se i øvrigt også afsnittet om Rissets klokke (se 7.4).

```
{
    var sig = 16.collect({
        var freq = ExpRand(200, 8000);
        var amp = ExpRand(0.05, 0.8);
        var release = Rand(2, 6);
        var env = EnvGen.kr(Env.perc(0.01, release));
        SinOsc.ar(freq) * amp * env;
    });
    sig = sig.sum;
    DetectSilence.ar(sig, doneAction: Done.freeSelf);
    sig.dup * 0.1;
}.play;
```

Kodeboks 7.17: Kaotisk klokkelyd

Bemærk på spektrogrammet, hvordan de forskellige partialtoner klinger ud på forskellige tidspunkter.



Figur 7.9: Spektrogram: Kaotiske klokkelyde genereret ved additiv syntese

7.3 Additive orgelklange

Orgelet - elektrisk eller akustisk - er et glimrende eksempel på additiv klangdannelse. Orgelets klang bestemmes af registertræk, som er fysiske lister, man trækker ud eller skubber ind (på engelsk kaldet *drawbars*). Registertrækkene styrer mængden og karakteren af overtonerne, hvilket vi kan simulere med simpel addition af sinusbølger.

7.3.1 Emulering af et elektrisk orgel

Herunder fremgår en meget simpel emulering af et elektrisk orgel, løseligt baseret på data om Hammond's B3-orgel fra Electric Druid. Bortset fra de 8 overtoner, hvis amplitude bestemmes af registerudtræk, er særligt Hammond-orgler også kendt for den lyd, som frembringes af en tilknyttet Leslie-højttaler med roterende horn, der kan justeres i rotationshastighed. Den kan vi implementere med et lavpasfilter, hvor cutoff-frekvensen moduleres af en LFO (se 6.1.1).

```
SynthDef(\organ, {
    arg freq = 440, amp = 0.1, pan = 0, gate = 1,
    rotationSpeed = 0, drawbars = #[0.2, 0.4, 1, 0.2, 0.1, 0.1,
    ↪  0, 0];
    var sig, env;

    env = EnvGen.kr(Env.adsr(0.05, 0.1, 0.8, 0.05), gate,
    ↪  doneAction: Done.freeSelf);

    sig = SinOsc.ar(freq * [0.5, 1.498823530, 1, 2, 2.997647060,
    ↪  4, 5.040941178, 5.995294120, 8]);

    sig = sig * drawbars.lag(0.1);

    sig = sig.sum * 0.5;

    // Roterende horn i Leslie-højttaler
    rotationSpeed = rotationSpeed.clip(0, 1).round * 7;
    sig = LPF.ar(sig, SinOsc.kr(rotationSpeed).range(1600,
    ↪  2000));

    sig = sig * env;

    sig = Pan2.ar(sig, pan, amp);

    Out.ar(0, sig);
}).add;
```

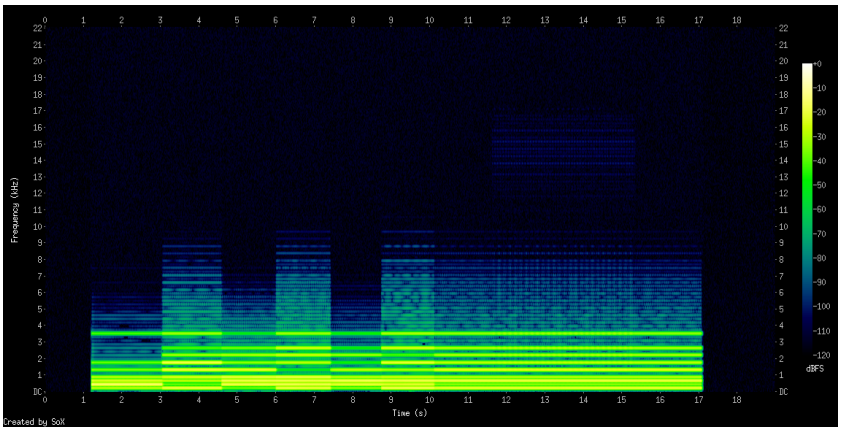
Kodeboks 7.18: SynthDef til simpel emulering af Hammond-orgel

Med SynthDef'en kan vi indstille orgelets registertræk og hornets rotationshastighed.

```
x = Synth(\organ);

// Tilfældige drawbar-positioner
x.set(\drawbars, Array.rand(8, 0.0, 1.0).postln);

// Hurtig/langsom rotation
x.set(\rotationSpeed, 1);
x.set(\rotationSpeed, 0);

x.set(\gate, 0);
```

Kodeboks 7.19: Test lyden af Hammond-orgel-SynthDef'en

Bemærk på spektrogrammet, hvordan sammensætningen af overtoner skifter, hver gang vi indstiller registertrækkene tilfældigt med `x.set(\drawbars, Array.rand(8, 0.0, 1.0).postln);`.



Figur 7.10: Spektrogram: Tilfældige indstillinger af registertræk for emuleret elektrisk orgel

For at demonstrere én af de klanglige muligheder ved denne orgel-SynthDef kan vi skrive en lille algoritmisk komposition til en (eventuelt lettere beruset) baseball-organist.

```
Pbind(
    \instrument, \organ,
    \drawbars, [[0.5, 0.2, 1, 0.2, 0.6, 0.1, 0.2, 0.9]],
    \rotationSpeed, 1,
    \degree, Pseq([0, -3, -2, -1], 10),
    \ctranspose, Pseries(0, Prand([-2, 0, 1, 2],
    ↪  inf).asStream).stutter(4),
    \dur, Pgeom(0.5, 0.85).stutter(4),
    \legato, Pexprand(1.3, 1.7),
).play;
```

Kodeboks 7.20: Algoritmisk komposition for baseball-orgel

7.4 Rissets klokke

Jean-Claude Risset var en fransk komponist, der blandt andet arbejdede med additiv syntese og som en pioner inden for computermusikken var en af de første, der eksperimenterede med de store klanglige muligheder man kan opnå med mikrovariationer i overtoners amplitude over tid (Roads, 2023, p. 123).

Nedenstående SynthDef er en SuperCollider-implementering af en berømt klokkelyd, som Risset skabte. Værdierne er baseret på Miller Puckettes (2007, pp. 107-108) gennemgang af teknikken. Der gøres i denne SuperCollider-implementering flittig brug af oscillatorbanke skabt med multichannel expansion (se 7.1.3).

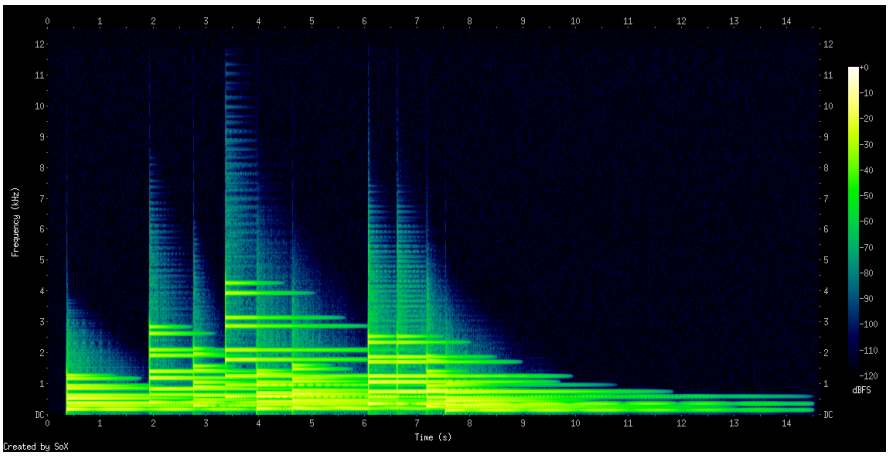
```
SynthDef(\risset,{
    arg freq = 440, atk = 0.01, rel = 3, pan = 0, amp = 0.1, out
    ↪  = 0;
    var ampRatios = [1, 0.67, 1, 1.8, 2.67, 1.67, 1.46, 1.33,
    ↪  1.33, 1, 1.33];
    var durRatios = [1, 0.9, 0.65, 0.55, 0.325, 0.35, 0.25, 0.2,
    ↪  0.15, 0.1, 0.075];
    var freqRatios = [0.56, 0.56, 0.92, 1.19, 1.7, 2, 2.74, 3,
    ↪  3.76, 4.07];
    var detune = [0, 1, 0, 1.7, 0, 0, 0, 0, 0, 0, 0];

    var sig = SinOsc.ar((freq * freqRatios) + detune);

    sig = sig * ampRatios * 0.05;

    sig = sig * EnvGen.kr(
        Env.perc(atk, rel),
        timeScale: durRatios
    );

    sig = sig.sum;

    DetectSilence.ar(sig, doneAction: Done.freeSelf);

    sig = Pan2.ar(sig, pan, amp);

    Out.ar(out, sig);
}).add;
```

Kodeboks 7.21: SynthDef til Rissets klokke

Med patterns kan vi blandt andet variere på envelopesegmenterne og strække de varigheder, som oprindeligt er angivet.

```
Pbind(
    \instrument, \risset,
    \scale, Scale.minorPentatonic,
    \degree, Pwhite(0, 10, 10),
    \db, Pgauss(-20, 2),
    \rel, Pexprand(4, 12),
    \dur, Pexprand(0.15, 2.5),
    \pan, Pgauss(0, 0.3),
).play;
```

Kodeboks 7.22: En generativ komposition for Rissets klokke



Figur 7.11: Spektrogram: En lille generativ komposition for Rissets klokke

7.5 Old school vocoder

En vocoder er en lydeffekt, der oftest bliver brugt til at skabe den efterhånden velkendte „robotstemme“-lyd. Vocoderen fungerer således, at man analyserer frekvensspektret for ét lydsignal og anvender resultatet af analysen til at justere på frekvensspektret for et andet lydsignal. Det analyserede lydsignal er ofte en menneskestemme, da menneskelig tale og sang er karakteriseret ved, at overtonespektret indeholder grupper af fremtrædende overtoner kaldet *formanter*, hvis konstante forskydning og afveksling danner stavelser, ord og sætninger (Berg, 2025). Vocoderen overfører variationerne i overtonespektret fra lydkilden til en ofte syntetisk dannet klang, eksempelvis en savtakket bølgeform, der indeholder et righoldigt overtonespektrum. Dette kan implementeres på forskellige måder, men her kigger vi på en digital emulering af den tidlige analoge vocoder fra 1930'erne, som beskrevet af Curtis Roads (2023, pp. 207-208).

7.5.1 Den grundlæggende vocoder-teknik

Vi starter med den lydkilde, som skal udsættes for analyse. Den kalder vi for **kilde A**. Det er som sagt ofte den menneskelige stemme, som er rig på formanter, men alle signaler med et dynamisk spektralt indhold kan anvendes. Her anvender vi blot en støjgenerator, som vi senere erstatter med et vokalsample. For at analysere et lydsignals frekvensspektrum, kan man dele dette op i en række „bånd“ og måle på lydstyrken inden for hvert af disse bånd. Lad os illustrere, hvordan man kan gøre dette for ét bånd med centerfrekvens på 200 Hz ved at isolere båndet med et band pass-filter (`BPF.ar`) og måle amplituden med `Amplitude.ar`.

```
{
    var kildeA = PinkNoise.ar;
    var amp = Amplitude.ar(BPF.ar(kilde, 200, 0.15));
    // amp indeholder amplituden for frekvensbåndet
}
```

Kodeboks 7.23: Analyse af amplitudeenvelope i frekvensbånd

Hvis vi ønsker at arbejde med en række bånd frem for blot ét, kan vi bruge en algoritme til at danne en bank af signaler for os (se 7.1.5). Herunder analyserer vi amplituderne ved 500 Hz, 1000 Hz, 2000 Hz og 4000 Hz.

```
{
    var kilde = PinkNoise.ar;
    var amps = [500, 1000, 2000, 4000].collect({
        arg centerFreq;
        Amplitude.ar(BPF.ar(kilde, centerFreq, 0.15));
    });
    amps.poll;
}.play;
```

Kodeboks 7.24: Spektral envelopeanalyse

Når vi har fundet disse amplituder, kan vi bruge dem til at styre amplituden af tilsvarende frekvensbånd i et andet lydsignal, som vi kan kalde for **kilde B**. Her kan vi tage en savtakket bølgeform som eksempel. Den er velegnet, fordi den har et righoldigt overtonespektrum, hvor manipulationen med den spektrale envelope fra kilde A tydeligt vil kunne høres. Med en sinusbølge, som ikke har nogen overtoner, vil der ikke være nogen hørbar effekt.

```
{
    var kildeA = PinkNoise.ar;
    var kildeB = Saw.ar;
    var sig = [500, 1000, 2000, 4000].collect({
        arg centerFreq;
        var amp = Amplitude.ar(BPF.ar(kildeA, centerFreq,
        ↪  0.15));
        amp * BPF.ar(kildeB, centerFreq, 0.15);
    }).sum;
    // sig indeholder den modificerede version af kildeB
}
```

Kodeboks 7.25: Basal vocoder-teknik

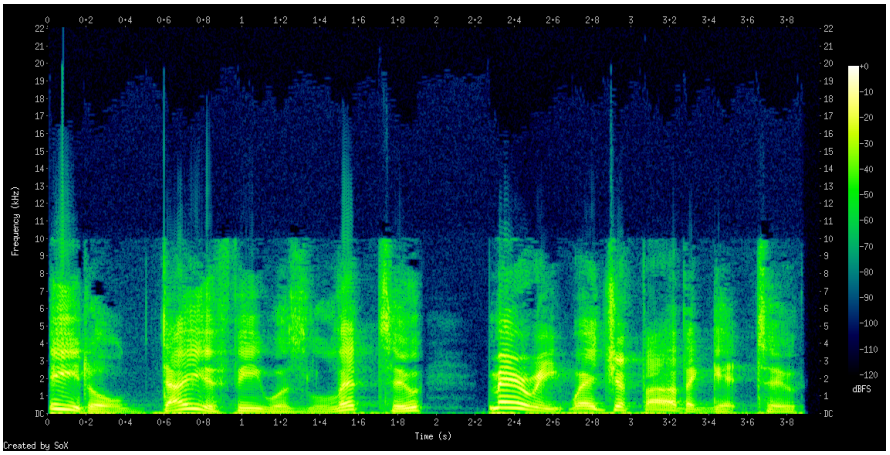
7.5.2 Sammensætning i SynthDef

For at udnytte ovenstående teknik til at fremstille den klassiske vocoderlyd, kan vi anvende et vokalsample som kilde A. Brug af samples gennemgås senere (se 8.1), hvorfor vi for nuværende blot kan betragte UGen'en **PlayBuf** som en lydkilde på linje med oscillatorer og støjgeneratorer. Vi bruger her et sample med et udklip fra en oplæsning af digtet *'Urry*[2] af C. J. Dennis (2015).

[2] Både digtet af C. J. Dennis og oplæsningen af Son of the Exiles tilhører det offentlige domæne. Oplæsningen kan findes i fuld længde via platformen LibriVox.

```
~speak = Buffer.readChannel(s,
↪  "C:/lydfiler/urry-eat-all-day.ogg");
```

Kodeboks 7.26: Indlæsning af sample til vocoder



Figur 7.12: Spektrogram: Udsnit fra oplæsning af digtet 'Urry af C. J. Dennis

Derudover skal vi fremstille en liste med centerfrekvenser til vores frekvensbånd, hvilket vi nemt kan gøre ved hjælp af iteration (se 1.6.3). Her har jeg valgt at inddele spektret mellem 50 Hz og 12 kHz i 12 bånd.

```
~bandFrequencies = 12.collect({
    arg num;
    num.linexp(0, 11, 50, 12000);
});
// -> [ 50, 82.291095610801, 135.43648833652, 222.90434021783,
↪  366.86084745856, 603.78762148144, 993.72689775894,
↪  1635.4975030901, 2691.7376279603, 4430.1207700334,
↪  7291.1898370843, 12000 ]
```

Kodeboks 7.27: Beregning af centerfrekvenser

Når ovenstående frekvenser er beregnet og gemt under variablen `~bandFrequencies`, kan vi registrere nedenstående SynthDef, som er baseret på teknikkerne ovenfor.

```
SynthDef(\vocoder, {
    arg freq = 440, gate = 1, pan = 0, amp = 0.5, rq = 0.15,
    ↪  noiseThreshold = 0.1;
    var kildeA, kildeB, sig;
    kildeA = PlayBuf.ar(1, ~speak, loop: 1);
    kildeB = Saw.ar(freq);

    sig = ~bandFrequencies.collect({
        arg centerFreq;
        Amplitude.ar(BPF.ar(kildeA, centerFreq, rq)).lag(0.01)
        * BPF.ar(kildeB, centerFreq, rq);
    }).sum * 2;

    sig = sig * Env.asr(0.01, 1, 0.2).kr(2, gate);
    // En simpel noise gate
    sig = sig * (DetectSilence.ar(kildeA + Impulse.ar(0), amp:
    ↪  noiseThreshold, time: 0.05) - 0.9).fold(0, 1).round;
    sig = Pan2.ar(sig, pan, amp);
    Out.ar(0, sig);
}).add;
```

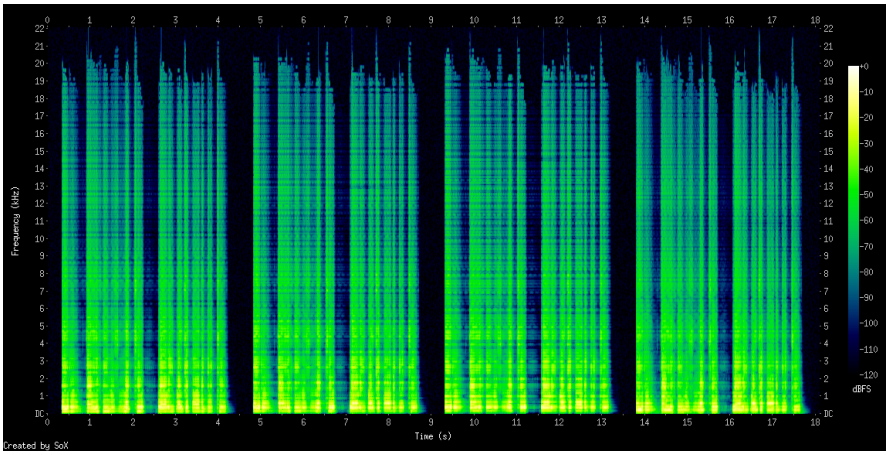
Kodeboks 7.28: En old school vocoder-SynthDef

Når SynthDef'en er indlæst, kan vi bruge den til at spille et robotstemme-mønster i skiftende akkorder ved hjælp af patterns.

```
Pbind(
    \instrument, \vocoder,
    \degree, [-7, 0, 2, 4],
    \octave, 5,
    \mtranspose, Pseq([0, 1, 2, -3]),
    \dur, ~speak.duration,
    \pan, Pgauss(0, 0.1),
    \db, 0,
    \legato, 1,
).play;
```

Kodeboks 7.29: Vi spiller på vocoderen med patterns

Sammenlign spektrogrammet med det oprindelige vokalsample ovenfor.



Figur 7.13: Spektrogram: Vocoder-genskabelse af udsnit fra digtet 'Urry af C. J. Dennis

7.5.2.1 Keyboardudgave?

Det overlades til den nysgerrige læser på egen hånd at implementere en „keyboard og mikrofon"-udgave af den vocoder, som er beskrevet ovenfor. Det er ikke vanskeligt, men det kræver nogle få justeringer:

- UGen'en `SoundIn` kan anvendes i stedet for `PlayBuf` for at bruge lyden fra en fysisk input-kanal som kilde A.
- Man starter og slukke for enkelte toner/`Synth`-instanser med `MIDIdef` baseret på input fra et MIDI-keyboard (i stedet for at spille med patterns).

I forhold til den sidstnævnte justering findes der et udmærket eksempel til inspiration i SuperColliders dokumentation.

7.6 Øvelse: Multikanalslyd og additiv syntese

I denne øvelse arbejdes der med multikanalslyd og additiv syntese. Det er i denne øvelse en fordel at visualisere serverens lydlige output (se 4.1).

7.6.1 Monofoni i to kanaler

1. Fremstil den samme sinustone i begge lydkanaler ved hjælp af duplikering (se 7.1.1).

```
{
    var sig = SinOsc.ar * 0.1;
    sig    ;
}.play;
```

Kodeboks 7.30: Monofoni i to kanaler

7.6.2 Stereofoni

Løs følgende opgaver ved hjælp af `pan`-argumentet til `Pan2.ar`:

1. Fremstil en sinustone i venstre side af stereofeltet.
2. Fremstil en sinustone i højre side af stereofeltet.
3. Fremstil en sinustone midt i stereofeltet.
4. Fremstil en sinustone, der ved hjælp af modulation bevæger sig mellem højre og venstre side af stereofeltet.

```
{
    var sig = SinOsc.ar * 0.1;
    Pan2.ar(sig,    );
}.play;
```

Kodeboks 7.31: Stereofoni med Pan2

7.6.3 Klangdannelse med additive elementer

1. Fremstil i nedenstående SynthDef en kompleks klang ved hjælp af additiv syntese med sinustoner (eksempler herpå kan findes i afsnittet om additiv syntese (se 7.2)). Klangen skal overholde følgende krav:

a) Klangen skal ud over grundtonen (`freq`) indeholde mindst tre forskellige *overtoner* (dvs. med et harmonisk forhold til `freq`).
b) Klangen skal derudover indeholde mindst to forskellige *partialtoner* (dvs. med et inharmonisk forhold til `freq`).
c) Hver tone har en unik *amplitude*, eventuelt genereret tilfældigt (se 7.2.2.3).
d) Kanalerne med de forskellige toner skal summeres før linje 10 herunder (som starter med `env = EnvGen ..`).

2. Test din SynthDef med nedenstående Pbind.

```
1  SynthDef(\additivo, {
2      arg freq = 440, pan = 0, amp = 0.1,
3      attack = 0.01, release = 1, gate = 1;
4      var env, sig;
5      // udfyld herunder
6      sig =
7
8      // udfyld ikke herunder
9      env = EnvGen.kr(Env.asr(attack, 1, release), gate,
       ↪  doneAction: 2);
10     sig = Pan2.ar(sig * env, pan, amp);
11     Out.ar(0, sig);
12 }).add;
```

Kodeboks 7.32: Additiv syntese

SynthDef'en kan testes med nedenstående enkle komposition.

```
1  TempoClock.tempo = 125/60;
2  Pbind(
3      \instrument, \additivo,
4      \degree, Pshuf([0, 1, 3, 4], 4).repeat(4),
5      \mtranspose, Pbrown(-2, 2, 1).stutter(16),
6      \dur, Pseq([1/4, 1/8, 1/16, 1/16], inf) * 4,
7      \attack, Pexprand(0.001, 0.02),
8      \release, Pexprand(0.5, 1.5),
9      \legato, Pgauss(1, 0.2),
10 ).play;
```

Kodeboks 7.33: Komposition til additiv SynthDef

7.6.3.1 Bonusudfordring

Arbejd videre med SynthDef'en fra forrige opgave.

1. Tilføj et element af filtreret støj med egen amplitude-envelope, helt i begyndelsen af anslaget.
2. Tilføj en sub-oktav, dvs. en tone, som klinger en oktav under freq.
3. Gør mængden (lydstyrke) af anslagsstøj og sub-oktav justerbar ved at tilføje argumenter og varier disse ved at tilføje patterns og nøgler til Pbind-kompositionen.

7.6.4 Duplikering af kaotiske UGens kan føre til unikke resultater - men hvordan?

Dette er en analytisk opgave, men du er som altid velkommen til at eksperimentere videre med kildekoden.

1. Hvordan udnytter nedenstående kildekode multichannel expansion? Kig efter `.dup`-method'en og gennemgå, hvordan de forskellige UGens i signalkæden ekspanderer.
2. Hvilke envelopes (se 5.1) anvendes her, og hvordan skaleres (se 4.2) deres output?

```
~numVoices = 5;
{
    var trigger = Dust.kr(XLine.kr(0.2, 100, 15, doneAction:
    ↪  2).dup(~numVoices));
    var env = Env.perc(releaseTime: 2.5).kr(gate: trigger +
    ↪  Impulse.kr(0));
    var freq = TExpRand.kr(220.dup(~numVoices), 880, trigger);
    freq = freq * env.range(24, 0).midiratio;
    Splay.ar(Pulse.ar(freq.lag(0.01)) * env);
}.play;
```

Kodeboks 7.34: Duplikering af kaotiske UGens

KAPITEL 8

Samplebaseret komposition

Vi har indtil nu arbejdet med rent syntetisk dannede klange i SuperCollider. Men der er også gode muligheder for at arbejde med præindspillet lyd eller for den sags skyld live-lyd fra en mikrofon eller andet musikudstyr. Her koncentrerer vi os i første omgang om at arbejde med samples i form af præeksisterende lydfiler, som vi først indlæser og derefter kan bearbejde i SuperCollider. Sample-manipulation er ganske fleksibelt i SuperCollider, men det kræver en lille smule teknisk forberedelse, som gennemgås herunder. Derefter introducerer kapitlet til tre kompositionsstrategier, hvor vi komponerer med samples ved hjælp af pattern-teknikker: Algoritmisk produktion af dynamiske trommebeats, beatslicing, hvor vi klipper beats op i metrisk organiserede enheder, samt manipulation og sammensætning af samples til abstrakte lydcollager. Kaptitlet indeholder som altid øvelser inden for disse områder, hvor man kan bruge egne samples for at opnå særegne resultater.

8.1 Samples i SuperCollider

SuperCollider kan fungere som en yderst fleksibel sample-maskine, hvor vi indlæser eksisterende lydfiler og arbejder med dem på forskellige måder. SuperCollider kan også skabe lydfiler, hvilket dog først gennemgås senere (se 10.1). Når vi arbejder med samples i SuperCollider, skal vi indlæse disse i en såkaldt **`Buffer`** på lydserveren. Derfra kan vi afspille samples med forskellige UGens: Herunder kigger vi først og fremmest på, hvordan man bruger **`PlayBuf`** samt hvordan den lidt mere fleksible **`BufRd`** fungerer.

Inden vi går i gang, kan det imidlertid være nyttigt at have de skiftende betydninger af ordet „sample“ for øje. Det forudsættes herunder, at du er fortrolig med disse tre betydninger af ordet og kan forstå ud fra konteksten, hvad der menes.

- Den éne betydning af samplebegrebet er meget teknisk og angår hvordan lyd repræsenteres og bearbejdes i digitale systemer (Roads, 2023, p. 29). Når man optager lyd via et lydkort, bliver lyden omdannet fra det analoge domæne (fx hvis vi optager lydsignalet fra en mikrofon) til det digitale domæne ved hjælp af en såkaldt analog-til-digital converter (ADC), hvor lydsignalet „måles“ („samples“) mange gange pr. sekund. Det er her, udtrykket *samplerate* eller *samplingfrekvens* kommer i spil: Sampleraten for en lydfil definerer, hvor ofte lydsignalet oprindeligt blev målt, og dermed hvor hurtigt vi skal læse de enkelte målinger efter hinanden, når vi skal afspille filen. Almindelige samplerates inden for musik i dag er 44,1 kHz (CD-kvalitet), 48 kHz, 88,2 kHz, 96 kHz og 192 kHz. Målingens „opløsning“, altså hvor fintfølende vores sampling af det analoge signal måler, kaldes lydoptagelsens *bitdybde* (på engelsk: *bit rate* eller *bit depth*). Man kan som en øvelse selv teste i SuperCollider, hvor mange valgmuligheder samplingen har ved bitdybder på hhv. 8, 16, 24 og 32 bit (med `2.pow(8)`, `2.pow(16)` osv.).
- En anden betydning af samplebegrebet er meget udbredt på nettet og knytter sig til de såkaldte sampler-musikinstrumenter (Kvifte, 2007, p. 107). Her er der tale om korte lydoptagelser, der ofte emulerer akustiske musikinstrumenter. Det kan fx være lyden af et lilletromme-hit, et toneanslag på klaver eller guitar osv. Disse er ofte knyttet til interfaces som keyboards, elektroniske trommesæt/-maskiner, MIDI-instrumenter i DAW-programmer osv.
- En tredje betydning af sampling kan skelnes fra de første, og den vedrører den meget udbredte praksis, at komponister og producere inkorporerer større eller mindre dele af andre værker i deres egne værker (Kvifte, 2007, p. 107). Dette har været en meget udbredt kunstnerisk strategi siden før ophavsretslovgivningens fremmarch og er fortsat særdeles udbredt i blandt andet populærmusikalske genrer, hvor særligt hiphoppen traditionelt har indebåret sampling i meget vid udstrækning. Der findes en del historisk og aktuel forskning om de musikformer og den såkaldte remixkultur, som baserer sig på sampling i denne betydning

af ordet (Dyndahl, 2005; Kvifte, 2007; Miller, 2008; Navas, 2012; Rodgers, 2003; Sewell, 2014; Tillet, 2014).

8.1.1 Indlæsning af lydfil i Buffer

For at arbejde med et sample/en lydfil, skal samplet indlæses i en såkaldt **Buffer**. En **Buffer** er et afgrænset område i lydserverens hukommelse, der fungerer lidt ligesom en variabel (se 1.3)[1]. Derfor skal lydserveren også være bootet, før vi indlæser vores sample.

Vi foretager indlæsning af lydfiler med **Buffer**.read(s, 'C:/lydfiler/minlydfil.wav'), hvor første argument er lydserveren s, og andet argument er stien til den lydfil, vi ønsker at indlæse. Vi gemmer ofte resultatet af denne instruks i en variabel, så vi ved hjælp af variablen kan henvise til bufferen.

Som et eksempel kan man bruge et sample, der følger med SuperCollider (linje 2 herunder). Man kan også nemt arbejde med egne samples (se linje 5 herunder) - erstat blot C:/lydfiler/galivanting.wav med *stien* til din egen lydfil. En sti til en fil er blot en tekst-streng, som viser hvor i computerens mappehierarki, filen befinder sig. Her har jeg indlæst et udklip fra en oplæsning af digtet *'Urry*[2] af C. J. Dennis (Dennis, 2015).

```
1 // Et indbygget sample indlæses i buffer
2 ~sample = Buffer.read(s, Platform.resourceDir +/+
  ↪  "sounds/a11wlk01.wav");
3
4 // En ekstern lydfil indlæses i buffer
5 ~sample = Buffer.read(s, "C:/lydfiler/galivanting.wav");
```

Kodeboks 8.1: Indlæsning af lydfil i Buffer

Et tip til den dovne programmør: Stier til filer kan genereres ganske let i SuperCollider! Man kan trække filen ind i SuperColliders tekstfelt med musen, eller man kan copy-paste filen fra en mappe på din computer. Når du slipper en fil med musen eller taster Ctrl/Cmd-V på tastaturet for at sætte filen ind, vil SuperColliders IDE

[1] I modsætning til de variabler, vi tidligere har arbejdet med, som hører til fortolkeren (se 1.1.1), er buffere nogle objekter, der tilhører lydserveren. Lydserveren har en liste med buffere og identificerer disse ved hjælp af deres unikke indekstal, som kan findes med method'en .bufnum. I praksis har vi sjældent brug for at arbejde direkte med bufferes nummer, da vi i stedet bruger almindelige variabler på fortolkeren til at holde styr på bufferne, og SuperCollider internt håndterer nummereringen af bufferne automatisk for os.

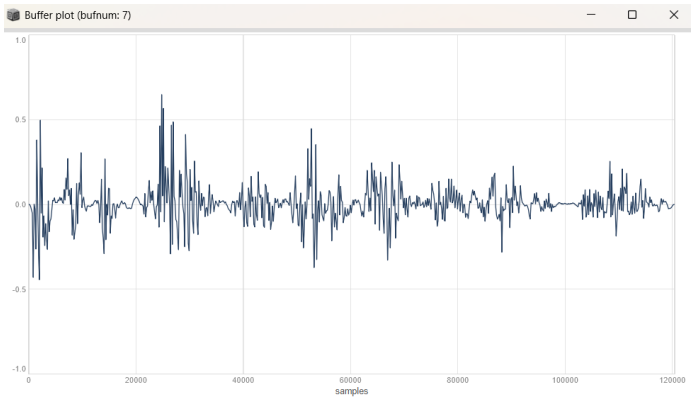
[2] Både digtet af C. J. Dennis og oplæsningen af Son of the Exiles tilhører det offentlige domæne. Oplæsningen kan findes i fuld længde via platformen LibriVox.

skrive filens sti, der hvor musemarkøren eller cursoren befinder sig. Smart, ikke sandt?

Når lydfilen er indlæst i en **`Buffer`** under variabelnavnet `~sample`, som vist ovenfor, kan vi bruge forskellige instance methods (se 1.5.3), til at vise grundlæggende oplysninger om samplet.

```
~sample.path;      // samplets oprindelige sti i filsystemet
// -> C:/lydfiler/galivanting.wav

~sample.duration;  // varighed, målt i sekunder
// -> 2.7284126984127

~sample.query;     // teknisk information om bufferen
// bufnum: 7
// numFrames: 120323
// numChannels: 1
// sampleRate: 44100.0

~sample.plot;      // visuel repræsentation
```

Kodeboks 8.2: Nyttige info-methods til buffere



Figur 8.1: Plot af sample indlæst i buffer

Hvis du arbejder med mange lydfiler og bliver træt af at indlæse dem én ad gangen, kan du automatisere indlæsningen ved at skrive en funktion (se 1.4), der itererer over (se 1.6.3) filerne i en mappe og indlæser dem en efter en. Alternativt kan du installere og anvende udvidelsen *PolyBuf* af Mads Kjeldgaard, som gør netop dette.

8.1.1.1 Indlæsning af enkeltkanaler fra lydfiler

Ønsker vi blot at indlæse én af flere kanaler i en lydfil, kan vi i stedet for method'en `Buffer.read` bruge `Buffer.readChannel` og under argumentet `channels` angive en liste med et tal for hver af de kanaler, vi ønsker at indlæse. Her er 0 typisk den venstre og 1 typisk den højre kanal.

```
1 // Venstre kanal fra en ekstern lydfil indlæses i mono-buffer
2 ~sampleV = Buffer.readChannel(s, "C:/lydfiler/minlydfil.wav",
  ↪  channels: [0]);
3
4 // Højre kanal fra en ekstern lydfil indlæses i mono-buffer
5 ~sampleH = Buffer.readChannel(s, "C:/lydfiler/minlydfil.wav",
  ↪  channels: [1]);
```

Kodeboks 8.3: Indlæsning af lydfil i mono-buffer

8.1.2 Sample-afspilning med PlayBuf

Den mest enkle måde at afspille samples på er at bruge UGen'en `PlayBuf`. Når vi gør det, er der to tekniske forhold, som er værd at huske på:

- **Sampleraten** i en given lydfil matcher ikke nødvendigvis lydserverens samplerate. Hvis lydserverens samplerate er 44,1 kHz, men lydfilen er produceret ved 48 kHz, vil afspilningen ske for langsomt. Det byder blandt andet, at tonehøjder og tempo ikke vil passe til det, vi forventer. Derfor har vi en særlig UGen, der kan tage højde for sådanne uoverensstemmelser, nemlig `BufRateScale`.
- **Antal samples:** Et lydsample eller en lydfil består som beskrevet ovenfor af et antal målinger („samples"). Når vi bruger `PlayBuf`, arbejder vi med samplets varighed ud fra antallet af disse samples. Sammenholdt med sampleraten, dvs. hvor hurtigt vi skal læse disse samples, afspiller `PlayBuf` det sample, som er indlæst i bufferen. Vi kan imidlertid ikke bare gå ud fra antallet af enkeltsamples i en Buffer, da en buffer kan indeholde data fra lydfiler med et variabelt antal kanaler. En stereo-lydfil vil have dobbelt så mange samples som en mono-lydfil af samme varighed og med samme samplerate. De fleste filformater understøtter nu om dage mange kanaler, så hvordan tager vi højde for det? Jo, for at finde varigheden af en indlæst lydfil, målt i samples, kan vi bruge den dertil beregnede UGen `BufFrames`. Den giver os antallet af såkaldte *sample frames*, som er det samme, uanset hvor mange kanaler, bufferen indeholder.

Herunder fremgår, hvordan man indstiller argumenterne til **PlayBuf** og bruger **BufRateScale** samt **BufFrames** til at tage højde for det indlæste samples varighed og samplerate.

```
{
    PlayBuf.ar(
        // antal kanaler
        numChannels: 1,

        // variabel, der henviser til den ønskede Buffer
        bufnum: ~sample,

        // afspilningshastighed 1
        // BufRateScale tager højde for evt. mismatch mellem serverens og lydfilens respektive samplerates
        rate: 1 * BufRateScale.kr(~sample),

        // reset-trigger, kan bruges til at angive spring til startposition
        trigger: 0,

        // startposition måles i sample frames
        // BufFrames.kr giver det samplede antal sample frames i bufferen/lydfilen
        startPos: 0 * BufFrames.kr(~sample),

        // loop - start forfra, når vi rammer slutningen af bufferen (0 = nej, 1 = ja)
        loop: 1,

        // doneAction - hvad sker der, når vi rammer den sidste sample frame i bufferen?
        doneAction: Done.freeSelf
    )
}.play
```

Kodeboks 8.4: PlayBuf-argumenter

Vi kan modulere flere af **PlayBuf**s parametre ved hjælp af andre UGens. Herunder moduleres afspilningshastighed af henholdsvis **SinOsc** og **LFNoise1**.

```
{
    PlayBuf.ar(1, ~sample,
        SinOsc.kr(2) * BufRateScale.kr(~sample)
    )
}.play;

{
    PlayBuf.ar(1, ~sample,
        LFNoise1.kr(1).exprange(0.5, 2) *
        ↪  BufRateScale.kr(~sample),
        loop: 1
    )
}.play;
```

Kodeboks 8.5: Modulation af sampleafspilning med LFO

Med et triggersignal (se 4.4.5), her skabt af UGen'en **Impulse**, kan vi springe hen til den position i bufferen, som er angivet med argumentet `startPos`.

```
// Fast startposition - midt i bufferen
{
    PlayBuf.ar(1, ~sample,
        trigger: Impulse.kr(4),
        startPos: 0.4 * BufFrames.kr(~sample),
    )
}.play;

// Dynamisk startposition, moduleret af en LFO
{
    PlayBuf.ar(1, ~sample,
        trigger: Impulse.kr(4),
        startPos: BufFrames.kr(~sample) *
        ↪  LFTri.kr(0.05).unipolar,
    )
}.play
```

Kodeboks 8.6: Spring til position i sample

8.1.3 Ekstra fleksibilitet med BufRd

BufRd er mere fleksibel end **PlayBuf**, fordi den tillader, at vi styrer læsningen af data fra bufferen direkte med en anden UGen. Det svarer lidt til, at en pladespillers pickupnål aflæser den lyd, som er indpræget i en vinylplade. Men til forskel fra pladespillerens stabile rotationsbevægelse kan vi med et bredt udvalg af UGens kan

flytte nålen rundt på meget forskellig vis. Til at styre afspilningspositionen angiver vi hertil en UGen under **BufRd**.ar's argument phase.

Ofte anvendes UGen'en **Phasor** hertil, fordi den skaber en lineær rampe fra en start- til en slutværdi. Til disse to værdier kan vi angive henholdsvis 0 og bufferens samlede antal sample frames ved hjælp af **BufFrames**. Dette vil resultere i en afspilning af samplet præcist som det er indlæst i bufferen, når vi samtidig skalerer afspilningshastigheden med **BufRateScale** for at tage højde for eventuelt samplerate-mismatch, akkurat ligesom ved **PlayBuf**.

```
{
    BufRd.ar(
        numChannels: 1,
        bufnum: ~sample,
        // phase-argumentet er afspilningspositionen, målt i
        ↪  sample frames
        phase: Phasor.ar(
            trig: 0, // ligesom PlayBuf's trigger-argument
            rate: BufRateScale.kr(~sample),
            start: 0,
            end: BufFrames.kr(~sample)
        ),
        loop: 1
    )
}.play;
```

Kodeboks 8.7: Sample-afspilning med BufRd og Phasor

Det interessante ved **BufRd** er imidlertid, at man frit kan anvende andre UGens end **Phasor**. Herunder moduleres afspilningspositionen i **BufRd** af henholdsvis en slagtøjsagtig envelope og en lavfrekvent støjgenerator, der bevæger sig tilfældigt.

```
{
    var position = EnvGen.ar(Env.perc(0.1, 2)) *
    ↪  BufFrames.kr(~sample);
    BufRd.ar(1, ~sample, position, 1);
}.play;

{
    var position = LFNoise2.ar(2).range(0,
    ↪  BufFrames.kr(~sample));
    BufRd.ar(1, ~sample, position, 1);
}.play;
```

Kodeboks 8.8: Envelope og tilfældighedsgenerator som pickupnål

8.2 Algoritmiske trommebeats

Beatproduktion spiller en central rolle i megen nyere populærmusik. Vi kigger her primært på trommebeats, idet vi udforsker forskellige teknikker inden for algoritmisk beatproduktion. De fleste musikproducere, der selv laver musik på computere, kender nok den situation, hvor laver et beat ved at komponere et MIDI-klip, som så afspiller samples via et trommesampler-plugin i en DAW. Når man har lavet et fedt beat, kan man så kopiere det, så vi hører det samme beat i alle tre vers. Dette risikerer dog hurtigt at blive noget monotont og maskinelt, og derfor findes der i nogle DAW-programmer forskellige teknikker til at indarbejde forskellige små afvigelser i timing og dynamik.

Men i SuperCollider har vi adgang til et righoldigt bibliotek af patterns, som vi kan bruge til at skabe både fremtrædende og mere subtile variationer i beats, fra rytmik til klang, mikrotiming og dynamik. Når vi afspiller samples med en specialdesignet SynthDef på SuperColliders lydserver, kan vi samtidig bruge vores patterns til at skabe detaljeret, klanglig variation, som kun vanskeligt kan lade sig gøre med den relativt begrænsede MIDI-protokol i en traditionel DAW. Dette betyder selvfølgelig ikke, at man ikke må bruge en DAW - lydeksemplerne i dette afsnit er faktisk efterbehandlet med rumklang, da det giver en bedre fornemmelse af de ellers noget tørre trommelyde.

8.2.1 Forberedelse til beatfremstilling

Inden vi kan gå i gang med at bruge patterns til at fremstille trommebeats, skal vi forbered nogle redskaber og materialer. Først og fremmest skal vi bruge en SynthDef, der kan afspille vores samples med de klanglige og afspilningsmæssige justeringer, vi ønsker. Derudover skal vi selvfølgelig vælge og indlæse nogle passende samples til brug i vores beat.

8.2.1.1 SynthDef til oneshot-samples

Når vi skaber vores egne med trommebeats ud af eksisterende samples, er det nyttigt at bruge såkaldte *oneshot*-samples. Der er med dette udtryk blot tale om korte samples, som indeholder lyden af ét anslag af fx en tromme, en klavertangent, eller andet instrument. Derfor indretter vi en SynthDef på følgende måde:

- Vi afspiller den valgte buffer fra start til slut og afslutter derefter Synth'en med `PlayBuf`s `doneAction`-argument. Brug af SynthDef'en beror således på, at vores samples er velorganiserede og egnede til en sådan oneshot-anvendelse.

- For at kunne styre afspilningshastigheden og fx afspille nogle samples baglæns indfører vi et SynthDef-argument kaldet `rate`.
- På nogle akustiske trommer vil der opstår små variationer i de dominerende partialtoners frekvenser (se 7.2.2.2), alt efter hvor på trommeskindet, man slår, og hvorvidt skindet er udsat for varierende spændthed. For at kunne simulere disse små variationer indfører vi et SynthDef-argument kaldet `cent`, hvor vi kan „stemme“ trommen et antal cent op eller ned.
- For yderligere at kunne variere de valgte samples klangligt anvender vi to teknikker til ændring af klang:
  - Lav- og højpasfiltre med tilhørende argumenter `lpf_cutoff` og `hpf_cutoff`.
  - En simpel form for klanglig manipulation kaldet waveshaping[3] med method'en `.htan` som kan styres med argumentet `drive` i intervallet 0-1. Dette kan skabe en mere forvrænget klang, hvis vi ønsker det.

```
SynthDef(\oneshot, {
    arg amp = 0.1, out = 0, pan = 0,
    buf, cent = 0, rate = 1,
    drive = 0, lpf_cutoff = 20000, hpf_cutoff = 20;
    var freqScale = (cent*0.01).midiratio;
    var dur = BufDur.kr(buf) / rate.abs;
    var sig = PlayBuf.ar(
        numChannels: 2,
        bufnum: buf,
        rate: rate * BufRateScale.kr(buf) * freqScale,
        loop: 1
    );
    sig = (sig* drive.linexp(0, 1, 1, 25)).tanh; //
    ↪ drive/distortion
    sig = sig * (1 - (drive * 0.5)).pow(2.5);

    sig = sig * Env.linen(0.001, dur - 0.002,
    ↪ 0.001).kr(doneAction: 2);

    sig = LPF.ar(sig, lpf_cutoff);
    sig = HPF.ar(sig, hpf_cutoff);
    sig = Balance2.ar(sig[0], sig[1], pan, amp);
    Out.ar(out, sig);
}).add;
```

Kodeboks 8.9: SynthDef til oneshot-samples

[3] Waveshaping er en digital form for distortion. Kort fortalt fungerer den typisk ved at tilføje overtoner ved hjælp af en såkaldt transfer-funktion. Jo kraftigere et signal, der fødes ind i transfer-funktionen, desto mere udtalte bliver overtonerne/forvrængningen (frem for at overstyre). Derfor skaleres drive-argumentet her til en faktor mellem 1 og 25.

SynthDef'en ovenfor baseres på stereo-samples. Hvis du i stedet ønsker at anvende mono-samples, kan du justere linje 7, så `PlayBuf` læser én kanal i stedet for to fra den angivne buffer, samt ændre `Balance2.ar(sig[0], sig[1], pan, amp)` til `Pan2.ar(sig, pan, amp)`, så vi panorerer korrekt.

8.2.1.2 Valg af oneshot-samples

Som eksempelmateriale bruger jeg herunder udvalgte samples fra en samling af samples kaldet *electro-drums*[4] af freesound.org-brugeren 'soneproject' (2013).

```
1 ~kick = Buffer.read(s,
  ↪ "C:/lydfiler/252821__soneproject__kik2.wav");
2 ~snare = Buffer.read(s,
  ↪ "C:/lydfiler/244354__soneproject__snr001.wav");
3 ~clap = Buffer.read(s,
  ↪ "C:/lydfiler/252823__soneproject__clp2.wav");
4
5 // Vi kan teste vores SynthDef med disse tre samples
6 Synth(\oneshot, [\buf, ~kick, \pan, -0.6]);
7 Synth(\oneshot, [\buf, ~snare, \pan, 0]);
8 Synth(\oneshot, [\buf, ~clap, \pan, 0.6]);
9 Synth(\oneshot, [\buf, ~clap, \rate, -1.5]);
```

Kodeboks 8.10: Indlæsning af tre tromme samples

8.2.1.3 Valg af beattype

En simpel indgangsvinkel til at danne et trommebeat er som følger:

1. Vi starter med at dele takten op i mindre segmenter.
2. Derefter fremstiller vi en række patterns, som kan udfylde disse segmenter.
3. Til sidst kombinerer vi disse patterns på forskellig vis.

Som eksempel kan vi tage et traditionelt, populærmusikalsk trommebeat i 4/4. Her kunne en helt enkel tilgang være at veksle mellem stortromme på de ulige taktslag, 1 og 3, og lilletromme på de lige taktslag, 2 og 4 („backbeatet").

[4] *electro-drums* er udgivet til det offentlige domæne under Creative Commons-licensen CC0 1.0 og kan findes via platformen freesound.

```
// Tilbagelænet tempo: 85 BPM
TempoClock.tempo = 85 / 60;

// ~kick og ~snare er defineret ovenfor
Pbind(\instrument, \oneshot, \buf, Pseq([~kick, ~snare],
↪  4)).play;
```

Kodeboks 8.11: Et meget simpelt beat

Dette simple beat udfylder måske nok de mest grundlæggende funktioner, men det bliver hurtigt monotont at lytte til. Lad os derfor skabe nogle variationsmuligheder, hvor forskellige segmenter varer præcis ét taktslag, så de kan indgå på henholdsvis de ulige og lige taktslag og erstatte det simple beat her. Her viser jeg nogle eksempler, men du kan selvsagt med stor fordel skrive dine egne patterns i stedet.

Her er det oplagt at navngive de forskellige variationsmuligheder, så vi kan anvende dem efterfølgende ved at angive deres navn. Dertil kunne vi bruge en række variabler med unikke navne, men vi bruger i stedet en datastruktur, som minder om en liste (se 1.6), nemlig en `Dictionary`. Som navnet antyder er en dictionary en samling af indhold, der kan tilgås ved hjælp af bestemte nøgler (små tekstbidder). Dette står i modsætning til lister, hvor vi tilgår elementerne ved hjælp af deres indeks (tal) (se 1.6.1). Vi opretter en dictionary med `Dictionary.new` og gemmer den under den globale variabel `b`, så vi let kan henvise til den senere. Vi kan derefter tilføje en Pbind med `b.put(\minPbind, Pbind())` og tilgå denne igen med `b[\minPbind]`.

```
b = Dictionary.new;
```

Kodeboks 8.12: En dictionary til variationsmuligheder

Da vi arbejder med variationsmuligheder, som skal vare præcist ét taktslag, vil jeg herunder notere nodeværdierne på SuperColliders måde (se 2.3.2.2), dvs. `\dur, 1` svarer til ét taktslag. `\dur, 0.5` svarer dermed til et halvt taktslag (hvad vi typisk omtaler som ottendedele), og så fremdeles.

8.2.2 Ulige taktslag

Som det første kan vi definere nogle grundlæggende nøgler og værdier/patterns, som segmenterne til de ulige taktslag kan have som udgangspunkt:

- Vi bruger SynthDef'en `\oneshot`, som vi definerede ovenfor (se 8.2.1.1).

- Vi vælger bufferen `~kick`, som indeholder vores trommesample
- Vi vælger ved hjælp af **Pgauss** at variere lydstyrken en lille smule, med middelværdi omkring -18 dB.
- Parameteren `\drive` sættes i udgangspunktet til 0,2.

Vi gemmer disse med en Pbind under den globale variabel `~uligeBasis`. Bemærk, at denne Pbind, afspillet med `~uligeBasis.play;` kan generere et ubegrænset antal events og derfor ikke overholder vores strategi om at begrænse sig til ét taktslag. Dette løser vi imidlertid herunder, når vi skaber nye Pbinds med afsæt i denne.

```
~uligeBasis = Pbind(
    \instrument, \oneshot,
    \buf, ~kick,
    \db, Pgauss(-18, 0.5),
    \drive, 0.2,
    \lag, Pgauss(0, 0.002),
);
```

Kodeboks 8.13: Grundlagspattern til ulige taktslag

Herefter kan vi bruge sammensætning af patterns (se 3.2) til at skabe en række variationsmuligheder. Her skal vi holde tungen lige i munden, når vi arbejder med `\ldur`-nøglen og de patterns, som definerer hvor mange events, variationsmuligheden skal bestå af, da vores sekvenser skal vare præcist ét taktslag.

8.2.2.1 Første mulighed: Enkelt stortrommeslag

Her har vi bare et enkelt anslag ved kraftig lydstyrke og dermed den simpleste variationsmulighed, som vi kan kalde for `\downbeat`.

```
b.put(\downbeat,
    Pbindf(~uligeBasis, \db, Pgauss(-15, 0.5, 1))
);
b[\downbeat].play;
```

Kodeboks 8.14: Enkelt stortrommeslag

8.2.2.2 Anden mulighed: Ottendedele

Her har vi ganske enkelt to halve taktslag (dvs. ottendedele), som vi kalder `\dobbelt`.

```
b.put(\dobbelt,
    Pbindf(~uligeBasis, \dur, Pseq([0.5, 0.5]))
);
b[\dobbelt].play;
```

Kodeboks 8.15: Stortromme-ottendele

8.2.2.3 Tredje mulighed: Trioliserede sekstendedele

Her har vi et mønster af trioliserede sekstendedele, distortion-præget klang og faldende afspilningshastighed/tonehøjde, som lyder lidt `\badass`.

```
b.put(\badass,
    Pbindf(~uligeBasis,
        \dur, Pseq([1/3, 1/6, 1/3, 1/6]),
        \lpf_cutoff, 2500,
        \drive, 0.6,
        \rate, Pseries(1, -0.2),
    )
);
b[\badass].play;
```

Kodeboks 8.16: Trioliserede sekstendedele

8.2.3 Lige taktslag

Til de lige taktslag definerer vi også først en Pbind med de grundlæggende indstillinger under den globale variabel `~ligeBasis`. Her lægger vi os med `\lag`-nøglen en lille smule lidt frem på beatet, dvs. med mulighed for, at anslaget ligger nogle få millisekunder før den maskinelt definerede, ideelle timing. Vi varierer afspilningshastigheden en lille smule med `\cent`-nøglen (jævnfør SynthDef'en ovenfor (se 8.2.1.1)).

```
~ligeBasis = Pbind(
    \instrument, \oneshot,
    \buf, ~snare,
    \lag, Pgauss(-0.003, 0.001),
    \db, Pgauss(-20, 0.5),
    \lpf_cutoff, 3000,
    \cent, Pgauss(0, 1),
    \drive, 0.2,
);
```

Kodeboks 8.17: Grundlagspattern til lige taktslag

8.2.3.1 Første mulighed: En langstrakt lilletrommelyd

Et enkelt anslag, men med nedsat afspilningshastighed, hvilket strækker samplet til at vare mere end dobbelt så lang tid som oprindeligt. Den distortion-prægede klang giver anledning til navnet `\cyberpunk`.

```
b.put(\cyberpunk,
    Pbindf(~ligeBasis,
        \dur, 1, \lpf_cutoff, 1800,
        \drive, Pwhite(0.6, 0.8),
        \rate, Pwhite(0.35, 0.48, 1),
    )
);
b[\cyberpunk].play;
```

Kodeboks 8.18: Cyberpunk-lilletromme

8.2.3.2 Anden mulighed: Sekstendedelstrioler

Her afspilles seks fordoblede lilletrommelyde med faldende lydstyrke, tilfældig transponering og skiftende stereo-placering. Vi kalder segmentet for `\stagger` på grund af den systematiske forskydning.

```
b.put(\stagger,
    Pbindf(~ligeBasis,
        \dur, 1/6, \db, Pseries(-25, -2, 6),
        \rate, Pseries([1.5, 3.7], Pwhite(-0.4,
        ↪  0.4)).stutter(2),
        \pan, Pxrand([-0.75, 0.75], inf).stutter(2).clump(2)
    ),
);
b[\stagger].play;
```

Kodeboks 8.19: Sekstendedelstrioler

8.2.3.3 Tredje mulighed: Varierende underdeling

Den tredje mulighed er mest rytmisk interessant, da vi her deler vores ene taktslag op i en tilfældig underdeling valgt med **Pwhite**. Hertil bruger vi **Pkey**, som henviser til den værdi, der er knyttet til en anden (forudgående) nøgle. Afspilningshastigheden og dermed tonehøjden varieres betydeligt, og mikrotimingen justeres en smule, så vi ikke i udgangspunktet ligger så langt fremme på beatet som ellers. Panoreringen skifter, så overraskelsesmomentet tydeliggøres, og vi kalder derfor segmentet for `\surprise`.

```
b.put(\surprise,
    Pbindf(~lige,
        \num, Pwhite(3, 9).stutter(inf),
        \rate, Pexprand(0.5, 4.0, Pkey(\num).asStream),
        \dur, 1 / Pkey(\num),
        \drive, Pexprand(0.4, 0.5),
        \pan, Pwhite(-0.8, 0.8),
        \lag, Pgauss(0, 0.004)
    )
);
b[\surprise].play;
```

Kodeboks 8.20: Varierende underdeling

8.2.3.4 Fjerde mulighed: Tynd ud

Toogtredivtedele, skiftende mellem `~snare` og `~clap` med „fortyndet“ klang og det passende navn `\tynd`, grundet højpasfilter med cutoff ved 2 kHz og øget afspilningshastighed. Panoreringen starter fra midt og bevæger sig gradvist ud mod skiftevis højre og venstre.

```
b.put(\tynd,
    Pbindf(~ligeBasis,
        \dur, 1/8, \hpf_cutoff, 2000,
        \rate, Pseries(1.5, 0.2).stutter(2),
        \buf, Pxrand([~snare, ~clap], 4).stutter(2),
        \pan, Pseries(0, 0.1) * Pseq([1, -1], inf),
        \drive, Pwhite(0.4, 0.7)
    )
);
b[\tynd].play;
```

Kodeboks 8.21: Klanglig udtynding

8.2.4 Sammensætning af variationsmuligheder i generative beat-opskrifter

Nu når vi endelig frem til sammensætning af vores variationsmuligheder i egentlige beats. Vi har en række valgmuligheder gemt under de to lister `~ulige` og `~lige`, som beskrevet ovenfor.

8.2.4.1 Enkel sekvensering

Som det første kan vi ganske enkelt vælge nogle af de mest enkle valgmuligheder og afspille dem i en valgt rækkefølge med **`Pseq`**. Her bruger vi en direkte notation for at tilgå de enkelte elementer i `b`.

```
Pseq([
    b[\downbeat],
    b[\cyberpunk],
    b[\dobbelt],
    b[\cyberpunk],
], 2).play;
```

Kodeboks 8.22: Simpel sekvensering

Men det bliver hurtigt trivielt at skrive `b[\navn]` frem for blot `\navn`. For at finde de forskellige Pbinds from fra en **`Dictionary`** inden for en pattern-kontekst, kan vi i stedet bruge pattern'et **`Psym`**, som bruger *symboler* (dvs. tekststrenge i formen `\tekst`) til at finde de ønskede patterns eller værdier fra dictonary'en.

Med andre ord: Hvis vi bruger **`Pseq`** til at sekvensere symboler som `\downbeat` og `\surprise`, kan **`Psym`** finde de relevante Pbinds frem fra vores dictionary `b`. Det

virker måske lidt overflødigt, men bliver hurtigt nyttigt, hvis vi skal notere navnene mange gange for at beskrive forskellige kombinationsmuligheder.

```
Psym(
    Pseq([
        \downbeat,
        \cyberpunk,
        \dobbelt,
        \stagger
    ], 2),
    b
).play;
```

Kodeboks 8.23: En simpel sekvens af variationsmuligheder

Bemærk, at hvis vi eksekverer ovenstående kildekode flere gange, vil den overordnede sekvens være ens, men der vil også være små variationer, da de valgte Pbinds indeholder patterns, som genererer tilfældige værdier inden for de angivne parametre.

Hvis vi vil spille to variationsmuligheder på samme tid, kan vi notere en liste med de ønskede variationer. Her lader vi fx stortrommen spille i ottendedele også fra 2- og 4-slaget i takten.

```
Psym(
    Pseq([
        \downbeat,
        [\dobbelt, \cyberpunk],
        \downbeat,
        [\dobbelt, \tynd],
    ], 2),
    b
).play;
```

Kodeboks 8.24: Flere segmenter på én gang

8.2.4.2 Varierede sekvenser med aleatorik

Vi kan selvfølgelig også i stedet for faste elementer i vores sekvens bruge patterns, der vælger mellem en række valgmuligheder, såsom **Prand**.

```
Psym(
    Pseq([
        Prand([\downbeat, \dobbelt]),
        \cyberpunk,
        Prand([\dobbelt, \badass]),
        Prand([\stagger, \surprise, \tynd])
    ], 4),
    b
).play;
```

Kodeboks 8.25: Tilfældige valgmuligheder

Er vi ligeglade med, om stortrommen rammer downbeatet og lilletrommen backbeatet, kan vi bede om en tilfældig rækkefølge med **Pshuf**. Interessant nok kan man på grund af den gentagne rækkefølge godt fornemme pulsen, selvom underdeling og klang skifter for hvert taktslag.

```
Psym(Pshuf([\surprise, \stagger, \badass, \tynd], 4), b).play;
```

Kodeboks 8.26: Tilfældig rækkefølge med Pshuf

8.2.4.3 Aleatoriske beats med konstant forandring

Vi kan få en automatisk liste med nøglerne til en dictionary med `.keys.asArray`, og kombineret med **Pxrand** og `.stutter` kan vi få få et lidt kaotisk men metrisk velfungerende beat, hvor de tilfældigt valgte segmenter gentages for at skabe genkendelighed i strømmen af skiftende segmenter.

```
Psym(Pxrand(b.keys.asArray, 4).stutter(2), b).play;
```

Kodeboks 8.27: Aleatorisk beat med genkendelighed gennem repetition

Dette bliver dog hurtigt lidt for tilfældigt til at kunne udgøre et stabilt beat. Vi kan i stedet lave en algoritme, der bruger alle de genererede variationsmuligheder, men hvor vi bruger **Pwrand** til at gøre nogle af mulighederne mere sandsynlige end andre. Dertil noterer vi først to lister med navnene på de forskellige segmenter i prioriteret rækkefølge.

```
~ulige = [\downbeat, \dobbelt, \badass];
~lige = [\cyberpunk, \stagger, \surprise, \tynd];
```

Kodeboks 8.28: Lister med segmenternes navne i prioriteret rækkefølge

Her genererer vi med `Array.rand` en liste med tilfældige tal mellem 0,01 og 1. Derefter bliver tallene skaleret, så de tilsammen giver 1 (med `.normalizeSum`), og derefter sorteret med `.sort`, så de mindste kommer først. Da vi har noteret de navne først, som vi ønsker er mest sandsynlige, vender vi listernes rækkefølge om med `.reverse`. På den måde bliver de første variationsmuligheder i listerne `~ulige` og `~lige` mere sandsynlige end de sidste.

```
~uligeSandsynligheder = Array.rand(~ulige.size, 0.01,
↪  1).normalizeSum.sort.reverse;
// -> [ 0.56223882208174, 0.34494655317443, 0.092814624743825 ]

~ligeSandsynligheder = Array.rand(~lige.size, 0.01,
↪  1).normalizeSum.sort.reverse;
// -> [ 0.46594787914676, 0.28348766391966, 0.19495731192664,
↪  0.055607145006942 ]
```

Kodeboks 8.29: At generere sandsynligheder

Derefter kan vi enkelt bruge `Pwrand` til at generere vores beat.

```
Psym(
    Pseq([
        Pwrand(~ulige, ~uligeSandsynligheder),
        Pwrand(~lige, ~ligeSandsynligheder),
    ], 16),
    b
).play;
```

Kodeboks 8.30: Vægtede sandsynligheder for segmenter med Pwrand

Der er her stadig tale om et beat i konstant forandring, hvilket måske er at strække definitionen af et beat. Men der er ingen der siger, at det er æstetisk forkert, eller at man skal bruge den genererede lyd præcist, som den er skabt. For at opnå et mere repetitivt præg kan man eksempelvis lade algoritmen køre og generere en række forskellige beats, som man så kan klippe i, repetere og viderebehandle i sin DAW efter forgodtbefindende. Eller man kan modificere ovenstående kildekode,

så der skabes et mere repetitivt udtryk. Under alle omstændigheder er det forhåbentlig klart, at de klanglige og kombinatoriske variationsmuligheder er et reelt og righoldigt supplement til mere strømlinede sampler-plugins. Som i resten af bogen anbefales det kraftigt, at du på egen hånd arbejder med ovenstående teknikker og udvikler dine egne beats.

8.3 Beatslicing med patterns

Beatslicing er en teknik, som kan udvide det kreative potentiale i loop-biblioteker. Her bruger vi altså ikke oneshot-samples, som vi gjorde ovenfor (se 8.2.1.1), men samlede beats/loops. Teknikken indebærer at skære et beat eller en lydsekvens op i mindre dele, som derefter kan arrangeres på nye måder. Ved at omarrangere slices, kan loops få et helt unikt udtryk og hjælpe med at skille din musik ud fra mængden.

Beatslicing kan implementeres på forskellige måder, blandt andet ved hjælp af såkaldt „onset detection“, som er en form for maskinlytning, der kan detektere anslag i et lydsignal ved at lede efter pludselig skift fra stilhed til væsentlige udsving i amplitude. Dette er dog lidt mere avanceret og gennemgås ikke her. I stedet fremstiller vi her en SynthDef, som ganske enkelt deler en buffer op i et givet antal lige store „slices“ og afspiller et af disse slices, baseret på SynthDef-argumenter. Her kan vi så bruger vi patterns til at afspille disse slices på forskellig vis. Lydeksemplerne i dette afsnit er efterbehandlet med rumklang, da det giver en bedre fornemmelse af de ellers noget tørre trommelyde.

8.3.1 Kildemateriale og slice-varighed

Med beatslicing er det afgørende, hvordan kildematerialet er organiseret. I denne artikel bruger vi som eksempel et sample, der indeholder et metrisk organiseret trommebeat af præcis én takts varighed. Der findes ganske mange af disse samples på nettet under kategorier som „drum loops“, „instrumental beats“, etc.

Til eksemplerne herunder bruger jeg et udsnit af lydfilen *funky drum loops.84 bpm.mp3*[5] af freesound.org-brugeren 'ajubamusic' (2015). Filen består af en række vellydende, akustisk indspillede trommebeats. Jeg har valgt netop dette udsnit af filen, fordi det indeholder et mindre „fill“ og dermed en række forskellige trommelyde, som vi kan klippe ud og sætte sammen på ny.

```
1 ~beat = Buffer.read(s, "C:/lydfiler/beat.wav");
2 ~beat.play;
```

Kodeboks 8.31: Indlæsning af det udvalgte beat

Trommebeatet er underdelt i sekstendedele, og det giver således mening at dele det op i 16 lige lange slices. Når vi afspiller et slice, kan vi derfor beregne en startposition

[5] *funky drum loops.84 bpm.mp3* er udgivet under Creative Commons-licensen CC BY 3.0 og kan findes via platformen freesound. Det fremgår af dette kapitel, hvordan samplet er anvendt og behandlet i denne bog.

i bufferen ud fra hvilket nummer det ønskede slice har (her fra 0 til 15). Når vi kender samplets samlede varighed, kan vi også fremstille en envelope med en fikseret varighed på en sekstendedel heraf. Vi kan indrette envelopen således, at den sidste halvdel af envelopens release-segment ganske kort overlapper med begyndelsen af det efterfølgende slice.

Med denne algoritme kan vi således afspille det første slice i en buffer med i alt 16 slices på følgende vis.

```
{
    var buf = ~beat, slice = 0, numSlices = 16,
    attack = 0.002, release 0.010;

    // Beregn startposition for det ønskede slice
    var startPos = (slice % numSlices) / numSlices;

    // Beregn varigheden af et slice og envelopens sustain-tid
    var duration = BufDur.kr(buf) / numSlices;
    var sustainTime = duration - attack - (release * 0.5);

    var env = EnvGen.kr(
        Env.linen(attack, sustainTime, release),
        doneAction: Done.freeSelf
    );

    PlayBuf.ar(
        numChannels: 2,
        bufnum: buf,
        rate: BufRateScale.kr(buf),
        startPos: BufFrames.kr(buf) * startPos
    ) * env;
}.play;
```

Kodeboks 8.32: Afspilning af et enkelt slice med PlayBuf

8.3.1.1 En SynthDef til beatslicing

Ovenstående kan omskrives til en SynthDef, hvor de første variabler laves om til argumenter, så vi kan styre dem med patterns. Vi kan også tilføje transponering og afspilningsretning, klanglig manipulation med drive og lavpasfilter samt panorering, amplitude og output-routing med tilhørende argumenter. Således opnår vi nedenstående SynthDef.

```
SynthDef(\slice, {
    arg buf, slice = 0, numSlices = 16,
    attack = 0.002, release = 0.010,
    transpose = 0, direction = 1,
    drive = 0, cutoff = 20000, rq = 1,
    amp = 0.1, out = 0, pan = 0;

    var startPos = (slice % numSlices) / numSlices;

    var duration = BufDur.kr(buf) / numSlices;
    var sustainTime = duration - attack - (release * 0.5);

    var env = EnvGen.kr(
        Env.linen(attack, sustainTime, release),
        doneAction: Done.freeSelf
    );

    var sig = PlayBuf.ar(
        numChannels: 2,
        bufnum: buf,
        rate: BufRateScale.kr(buf) * transpose.midiratio *
        ↪  direction.sign,
        startPos: BufFrames.kr(buf) * startPos
    );

    sig = (sig * drive.linexp(0, 1, 1, 100)).tanh;
    sig = sig * drive.lincurve(0, 1, 1, 0.1, -2);
    sig = RLPF.ar(sig, cutoff.clip(20, 20000), rq.clip(0.0001,
    ↪  1));
    sig = Compander.ar(sig, sig, 0.1, 1.0, 0.25, 0.01, 0.01) *
    ↪  10.dbamp;

    sig = Balance2.ar(sig[0], sig[1], pan, amp) * env;
    Out.ar(out, sig);
}).add;
```

Kodeboks 8.33: SynthDef til beatslicing

Vi kan teste SynthDef'en ved at afspille tilfældigt valgte slices.

```
Synth(\slice, [\buf, ~beat, \slice, rrand(0, 15)]);
```

Kodeboks 8.34: Tilfældigt valgte slices

8.3.2 Algoritmisk sammensætning af slices

Med ovenstående SynthDef kan vi fleksibelt sammensætte slices ved hjælp af patterns.

En enkelt overvejelse vi skal have med angår forholdet mellem to tempi: Originalsamplets tempo versus tempoet i vores nye beat. Der kan nemlig opstå „huller“ i strømmen af slices, hvis vores nye tempo er langsommere end tempoet i det oprindelige, da ét slice nødvendigvis vil vare kortere end den tid, det skal udfylde. Vil man omgå dette, må man modificere SynthDef'en, så envelopens varighed udregnes på anden vis.

```
TempoClock.tempo = 45 / 60;
TempoClock.tempo = 84 / 60;

Pbind(
    \instrument, \slice,
    \buf, ~beat,
    \dur, 1/16 * 4,
    \numSlices, 16,
    \slice, Pseries(length: 16),
).play;
```

Kodeboks 8.35: Vær opmærksom på tempoforskelle

Da mange af vores Pbinds vil tage udgangspunkt i nogle af de samme data, kan vi oprette en Pbind som basis for andre Pbinds (se 3.2.3) samt angive et tempo, som anvendes i resten af eksemplerne herunder, medmindre andet er angivet.

```
TempoClock.tempo = 95 / 60;

~base = Pbind(
    \instrument, \slice,
    \buf, ~beat,
    \dur, 1/16 * 4,
    \numSlices, 16
);
```

Kodeboks 8.36: Basisindstillinger til beatslicing

8.3.2.1 Semi-tilfældig rækkefølge af slices

Vi kan selvsagt sammensætte slices på kryds og tværs. Vi behøver ikke bruge alle slices, og det er kun fantasien (og SuperColliders bibliotek af patterns), der sætter grænser. Her er eksempelvis en sekvens, hvor nogle slices er faste og andre vælges tilfældigt blandt et udvalg af muligheder. I kompositionsprocessen er denne balance mellem det helt tilfældige og den genkendelige struktur ofte væsentlig (men også til tider vanskelig) at finde.

```
Pbindf(~base,
    \db, Pgauss(-18, 0.5),
    \slice, Pseq([
        2,
        Prand([1, 3, 7, 9], 1),
        15,
        Prand([5, 4, 14, Rest()], 1)
    ], 8).stutter(2),
).play;
```

Kodeboks 8.37: Faste og tilfældige slices

Til beatslicing kunne det være relevant at generere en tilfældig rækkefølge bestående af et vilkårligt antal elementer ud fra en liste med mulige valg. På den måde kan vi fx i forhold til beatslicing skabe korte sekvenser ud af de 16 mulige slices, hvor måske kun 4 slices bliver valgt og udgør en repeterende sekvens. Dette kan ikke lade sig gøre i ét pattern med de indbyggede patterns i SuperCollider. Men heldigvis kan man selv udvikle nye pattern-klasser, og med netop dette formål har jeg designet et pattern kaldet **Pshunc**. Man kan let installere det[6] som en udvidelse til SuperCollider med Quark'en *alea* (Eskildsen, 2022). **Pshunc** har tre argumenter: **Pshunc**`(list, length, repeats)`. `list` er den liste af muligheder, som vi kan vælge imellem. Den bliver sat i tilfældig rækkefølge ligesom ved **Pshuf**, men medtager kun det antal elementer, vi angiver med `length`. `repeats` fungerer som ved **Pseq** og **Pshuf**, idet den angiver hvor mange gange den genererede sekvens skal gentages.

[6] For at installere udvidelser til SuperCollider (såkaldte Quarks), skal man først installere programmet git på sin computer. Det bruger SuperCollider til at downloade og installere udvidelsen. Når git er installeret, kan man køre denne linje i SuperCollider for at installere *alea*: `Quarks.install("https://github.com/aeskildsen/alea")`.

```
TempoClock.tempo = 95 / 60;
Pbindf(~base,
    \slice, Pshunc((0..15), 4, 4)
).play;
```

Kodeboks 8.38: Beatslicing med Pshunc

8.3.2.2 Klangligt manipuleret beatslicing

Vi kan selvfølgelig også variere andet end slice-rækkefølge. Her følger en Pbind, som transponerer, skifter retning til baglæns en gang imellem, varierer filter-indstillinger og panorering samt drive, lydstyrke og mikrotiming.

```
TempoClock.tempo = 130 / 60;
Pbindf(~base,
    \slice, Pseq([
        0,
        Prand([
            Pwhite(1, 3, 3),
            Pbrown(0, 15, 1, 3),
            Prand((0..15)).stutter(3),
            Pseq(Rest().dup(3))
        ]),
    ], inf),

    // Transponering og retning
    \transpose, Prand([-12, -7, 0], inf) + Pwhite(-5,
    ↪  5).stutter(32),
    \direction, Pwrand([1, -1], [0.8, 0.2], inf),

    // Klanglig manipulation
    \drive, Pgauss(0.7, 0.05),
    \cutoff, Pexprand(3500, 8000),
    \rq, Phprand(0.1, 0.8),

    // Panorering og volumen
    \db, Pgauss(-15, 1),
    \lag, Pgauss(0, 0.005),
    \pan, Pgauss(0, 0.2),
).play;
```

Kodeboks 8.39: Pattern-baseret beatslicing

8.4 Lydcollage

En lydcollage bestående af samples, sammensat på kryds og tværs uden skelen til gængse konventioner for rytmik, kan være en interessant, abstrakt komposition-stilgang. Ved at fremstille en fleksibel SynthDef til sample-afspilning og anvende den til komposition med patterns, kan vi skabe interessante lydcollager, teksturer og klangflader.

8.4.1 Forberedelse

Som forberedelse skal vi indrette en passende SynthDef samt udvælge og indlæse samplemateriale.

8.4.1.1 SynthDef til samplebaseret lydcollage

Når vi skal skabe en lydcollage, er det nyttigt at indrette vores SynthDef, så at den passer til de teknikker, vi vil bruge i vores collage-komposition. Herunder findes en SynthDef, som minder om de SynthDefs, vi tidligere har udviklet til sampleafspilning. Men den skiller sig alligevel ud på en række egenskaber, fordi den er særligt indrettet til lydcollage:

- Det er en central del af lydcollage-teknikken at kunne styre, hvor mange samtidigt klingende samples, der afspilles. Med andre ord bør vi kunne styre, hvor mængden af *overlap*, hvilket vi som tidligere nævnt kan gøre med Pbind-nøglen (se 2.3.2) `\legato`. Det kræver imidlertid, at vi er nødt til at indrette vores SynthDef således, at afspilningen indhegnes af en vedvarende envelope (se 5.1.3.2). Afspilningens varighed styres dermed af en envelope (frem for samplets egen varighed). Som udgangspunkt indstiller vi derfor **`PlayBuf`** til at *loope* samplet, så det klinger i hele envelopens tidsrum.
- Vi indfører også et SynthDef-argument kaldet `startPos`, der angiver en position mellem 0 og 1, som angiver hvor i bufferen, læsningen af et sample skal starte. Dette betyder, at vi kan starte et vilkårligt stykke inde i et
- I stedet for at styre afspilningshastigheden direkte, er det mere relevant at kunne transponere et antal halvtoner op eller ned. Derfor bruger SynthDef'en `.midiratio` på argumentet `transpose` til dette formål. Når vi angiver en transponeringsafstand frem for afspilningshastighed, mister vi evnen til at afspille samplet *baglæns* (idet transponering blot går ud på at skalere afspilningshastigheden op eller ned).
- For at kunne afspille baglæns, opretter vi et andet argument, `direction`, hvor vi i SynthDef'en ved hjælp af method'en `.sign` registrerer, om der er tale om et

negativt eller et positivt tal. Negative tal fører til baglæns afspilning, positive tal giver forlæns afspilning, og tallets størrelse gør ingen forskel.

```
SynthDef(\collage, {
    arg amp = 0.1, out = 0, pan = 0,
    transpose = 0, startPos = 0, direction = 1,
    buf, loop = 1, t_reset = 1,
    drive = 0, cutoff = 20000, rq = 1,
    atk = 0.005, sus = 1, rel = 0.2, gate = 1;
    var env = EnvGen.kr(Env.asr(atk, sus, rel), gate,
    ↪ doneAction: 2);
    var sig = PlayBuf.ar(
        numChannels: 1,
        bufnum: buf,
        rate: transpose.midiratio * BufRateScale.kr(buf) *
        ↪ direction.sign,
        trigger: t_reset,
        startPos: startPos.linlin(0, 1, 0, BufFrames.kr(buf) -
        ↪ 2),
        loop: loop
    );
    sig = (sig * drive.linexp(0, 1, 1, 100)).tanh; //
    ↪ drive/distortion
    sig = RLPF.ar(sig, cutoff, rq) * env;
    sig = Pan2.ar(sig, pan, amp);
    Out.ar(out, sig);
}).add;
```

Kodeboks 8.40: SynthDef til sampleafspilning

Bemærk, at ovenstående SynthDef er beregnet til mono-samples. Hvis du i stedet ønsker at bruge samples, som er i stereoformat, har du to valgmuligheder:

1. Tilpas SynthDef'en, så `PlayBuf` læser 2 kanaler i stedet for 1, og der anvendes `Balance2` i stedet for `Pan2` til panorering (se hertil den relevante dokumentation).
2. Indlæs kun én af de to kanaler (se 8.1.1.1), når du indlæser dine samples.

8.4.1.2 Samplemateriale

Til vores lydcollage skal vi naturligvis bruge ét eller flere samples. For enkelhedens skyld nøjes vi her med ét sample, og jeg har valgt samplet *Emin 9 guitar chord*

interup.wav[7] af freesound.org-brugeren 'Sub-d' (2008). Samplet indeholder lyden af en akkordbrydning på en elguitar samt til sidst en telefon, der ringer.

Det valgte sample indeholder en stabil, tonal sekvens, klinger kontinuerligt uden lange pauser, og varer flere sekunder. Dette gør samplet relevant til æstetikken i en lydcollage, hvor vi kan lægge lag på lag.

```
1 ~guitar = Buffer.read(s, "C:/lydfiler/guitar.wav");
2 ~guitar.play;
```

Kodeboks 8.41: Indlæsning af det udvalgte beat

8.4.2 Collage med patterns

Der findes selvsagt mange forskellige muligheder for at fremstille en lydcollage med samples og patterns. Herunder udforsker vi to: Et udtryk, hvor lange, udstrakte udsnit af samplet væver sig ind og ud af hinanden, samt et udtryk, hvor kortere udklip støder sammen i et mere kaotisk collageunivers.

8.4.2.1 En kontinuerlig strøm af lyd

Undersøg selv effekten af de forskellige patterns herunder. Bemærk, at det samlede antal sampleafspilninger afhænger af den sekvensen, som defineres under `\drive`-nøglen.

[7] *Emin 9 guitar chord interup.wav* er udgivet til det offentlige domæne under Creative Commons-licensen CC0 1.0 og kan findes via platformen freesound.

```
TempoClock.tempo = 60/60;
Pbind(
    \instrument, \collage,

    // PlayBuf-indstillinger
    \buf, ~guitar,
    \transpose, Prand([12, 7, 0, -12], inf).clump(3),
    \direction, -1,
    \startPos, Pbrown(0.0, 1.0, 0.2),
    \loop, 1,

    // Rytmik/timing
    \strum, 0.25,
    \dur, Pwhite(1.0, 3.0),
    \legato, Pexprand(2.5, 3.5),

    // Klang
    \cutoff, Pexprand(500, 3000),
    \drive, Pseq(Array.interpolation(10, 0.1, 0.8).mirror),

    // Panorering, envelope, amplitude
    \pan, Pgauss(0, 0.4),
    \atk, 2, \rel, 3,
    \db, -30,
).play;
```

Kodeboks 8.42: Lydcollage med patterns

8.4.2.2 Tiltagende kaotisk collage

Hvis vi vil gå efter et mere abrupt udtryk, kan vi reducere mængden af overlap med `\legato`-nøglen. Genne Pbind giver desuden en klar, gradvis udvikling qua de mange indlejrede (se 3.1) **`Pseries`** og **`Pgeom`**. Undersøg selv kildekoden med `.trace`-method'en, hvis du er usikker på, hvordan de forskellige patterns bidrager til kompositionen.

```
TempoClock.tempo = 120 / 60;
Pbind(
    \instrument, \collage,

    \buf, ~guitar,
    \transpose, Pxrand([12, 24, 0, 7, -5, -12], inf) + Pgauss(0, Pseries(0, 0.020)).round,
    \direction, Prand([1, -1], inf),
    \startPos, Pgauss(0.4, 0.1),
    \loop, 1,

    \dur, Pgeom(1, 0.983),
    \lag, Pgauss(0, 0.01),
    \legato, Pgeom(1, 0.99),
    \atk, 0.002, \rel, 0.010,

    \cutoff, Pexprand(500, 3000),
    \rq, Pexprand(0.1, 0.5),
    \drive, Pexprand(0.01, 0.2) + Pseries(0, 0.003),

    \pan, Pgauss(0, Pseries(0, 0.005).asStream),
    \db, Pseries(-20, 0.1, 150),
).play;
```

Kodeboks 8.43: Tiltagende kaotisk lydcollage

8.5 Øvelse: Samples

I denne øvelse arbejder du med sample-afspilning ved hjælp af **`PlayBuf`**.

8.5.1 Klargøring af sample

Til brug i denne øvelse skal der indlæses et sample. Du kan bruge din egen lydfil, den skal blot have disse egenskaber:

- Samplet skal vare maksimalt 10 sekunder.
- Samplet skal være trimmet (brug hertil evt. Audacity), så der ikke er stilhed i begyndelsen eller slutningen af samplet.
- Samplet skal indlæses i mono.

Som forberedelse til de øvrige opgaver herunder:

1. Indlæs et sample i en buffer gemt under variablen `~buffer`.
2. Hvis din lydfil er i stereo, kan du indlæse den første kanal med `.readChannel` (se kodeblokken herunder).
3. Hvis du ikke har en lydfil klar selv, kan du bruge den nederste linje herunder i stedet med et indbygget sample fra SuperCollider.

```
// Udfyld her med dit eget sample
~buffer = Buffer.read(s,      );

// Brug .readChannel, hvis dit sample er i stereo
~buffer = Buffer.readChannel(s, channels: [0], path:   );

// Hvis du ikke har en lydfil klar, kan du bruge et indbygget sample i stedet:
~buffer = Buffer.read(s, Platform.resourceDir +/+ "sounds/a11wlk01.wav");
```

Kodeboks 8.44: Indlæsning af sample

8.5.2 Afspilning med PlayBuf

1. Afspil dit sample ved dobbelt hastighed.
2. Afspil dit sample ved halv hastighed.
3. Afspil den sidste halvdel af dit sample.

4. Afspil dit sample baglæns (bemærk, at dette kræver justering af enten `startPos` eller `loop` og `doneAction`).

```
{
    PlayBuf.ar(
        numChannels: 1,
        bufnum: ~buffer,
        rate: BufRateScale.kr(~buffer) * 1,
        trigger: 1,
        startPos: BufFrames.kr(~buffer) * 0,
        loop: 0,
        doneAction: Done.freeSelf
    )
}.play;
```

Kodeboks 8.45: Sampleafspilning med PlayBuf

8.5.3 Afspilningshastighed

1. Modulér afspilningshastigheden ved at fjerne `//`'erne for en række forskellige LFO'er herunder.
2. Notér i kommentarer, hvilken effekt de forskellige modulatorer har på den æstetiske oplevelse.

```
{
    var rate = 1; // afkommentér linjerne herunder for eksempler
    // rate = LFPulse.kr(1).range(0.5, 2);
    // rate = SinOsc.kr(1).range(-1, 1);
    // rate = SinOsc.kr(1).range(0.5, 2);
    // rate = SinOsc.kr([1, 1.02]).range(1, 1.5);
    // rate = LFNoise1.kr(2.dup).exprange(0.5, 4);
    // rate = Line.kr(-5, 5, 10, doneAction: 2);

    PlayBuf.ar(
        1,
        ~buffer,
        BufRateScale.kr(~buffer) * rate,
        loop: 1,
        doneAction: Done.none
    );
}.play;
```

Kodeboks 8.46: Modulation af afspilningshastighed

8.5.4 Lydcollage-komposition

Fremstil en abstrakt lydcollage. Kompositionen skal baseres på ét sample og realiseres ved at bruge patterns sammen med en udleveret SynthDef.

1. Indlæs SynthDef'en fra afsnittet om lydcollage (se 8.4.1.1).
2. Vælg og indlæs et sample, som...
   a) ... indeholder en vedvarende lyd (dvs. uden lange pauser i lyden).
   b) ... varer mellem 2 og 10 sekunder.
   c) ... tonalt set er relativt enkel og stabil.
   d) ... indeholder én monokanal (indlæs blot én kanal (se 8.1.1.1), hvis din ønskede fil er stereo-format).
3. Modificér **Pbind**'en herunder ved at erstatte faste værdier med patterns, således at vi hører en klangligt varieret lydcollage baseret på det valgte sample.

```
1 // Erstat stien med en sti til din egen lydfil
2 ~vedvarendeLyd = Buffer.readChannel(s,
  ↪  "C:/lydfiler/minSejeLydfil.wav", [0]);
```

Kodeboks 8.47: Indlæs et sample

```
TempoClock.tempo = 60/60;
Pbind(
    \instrument, \collage,

    // PlayBuf-indstillinger
    \buffer, ~vedvarendeLyd,
    \startPos, 0,
    \transpose, 0,
    \loop, 1,
    \direction, 1,

    // Timing og overlap
    \dur, 1,
    \legato, 1,

    // Klang
    \drive, 0.1,
    \cutoff, 1000,
    \rq, 0.5,

    // Panorering og lydstyrke
    \pan, 0,
    \db, -20,
    \atk, 0.1, \rel, 1,
).play;
```

Kodeboks 8.48: Samplecollage genereret med patterns 

8.6 Øvelse: Beatslicing

I denne øvelse producerer du beats ved hjælp af algoritmisk beatslicing.

8.6.1 Klargøring

1. Indlæs først et eller flere trommeloops under variablen `~sample`til brug i nedenstående øvelse. Det er (for denne øvelse) vigtigt, at trommeloopet er metrisk underdelt i sekstendedele og varer præcis én takt.
2. Indlæs SynthDef'en `\slice` fra afsnittet om beatslicing (se 8.3.1.1).

8.6.2 Algoritmiske beats

Skab med afsæt i klargjort SynthDef of sample et nyt beat ud af et eksisterende beat. Dit nye beat skal overholde følgende krav:

1. Slices skal vælges tilfældigt af patterns.
2. Klang skal varieres ved hjælp af `\drive` og de to filterparametre (`\cutoff` og `\rq`).
3. Mikrotimingen skal varieres ved hjælp af `\lag`-nøglen - vælg hertil selv et passende pattern og værdier.
4. Vælg selv yderligere parametre til justering .

```
TempoClock.tempo = 115 / 60;
Pdef(\beat,
    Pbind(
        \instrument, \slice,
        \buf, ~sample,

        \numSlices, 16,
        \slice, Pseries(0, 1, 16).repeat,
        \dur, 1/16 * 4,
        \lag, 0,

        \pan, 0,
        \direction, 1,

        \release, 0.01,

        \drive, 0,
        \cutoff, 16000,
        \rq, 1,
        \amp, 0.5,
    )
).play;
```

Kodeboks 8.49: Beatslicing med patterns

8.6.3 Synkretisme med to breakbeats

Tag afsæt i samme ressourcer som ovenfor plus mindst ét ekstra sample og fremstil et nyt beat. Opgaven her går ud på at få to breakbeats til at fungere sammen klangligt, rytmisk og evt. tonalt.

1. Ved nøglen `\buf` skiftes der vha. patterns mellem de indlæste samples.
2. Justér gerne på anvendte nøgler og på den SynthDef, som ligger til grund for kompositionen.
3. Fremstil mindst tre forskellige varianter af beatet, eksporter dem fra SuperCollider og indlæs i en DAW som trommebeat for en komposition.

KAPITEL 9

Klangdannelse med granular syntese

Dette kapitel introducerer til granular syntese som teknik og kompositionsredskab i SuperCollider. Vi ser på grundlæggende aspekter som lydkilde, forskellen på synkron og asynkron granular syntese, samt hvordan man automatisk kan udregne centrale parametre som grainvarighed, tæthed og triggerfrekvens. Vi ser også på hvordan man med granular syntese kan at transponere et sample uden ændring i tempo og vice versa, samt hvordan vi med disse redskaber også kan skabe unikke teksturer og klangflader på måder, som ikke er praktisk mulige med traditionelle samplere eller synthesizere.

9.1 Redskaber til granular syntese

Granular syntese er kort fortalt en teknik, der går ud på at danne komplekse klange og teksturer ved at sammensætte korte lydklip („grains“) til unikke strømme af lyd. Grains varer typisk mellem 1 og 100 ms (Roads, 2002, p. 87), hvilket er meget korte tidsrum at klippe og sammensætte med magnetbånd og andre analoge teknologier. Vi har derfor at gøre med en af de klangdannelsesteknikker, som hører til den digitale lydteknologi. Granular syntese gør arbejdet med lydmateriale utroligt fleksibelt, og SuperCollider indeholder heldigvis glimrende redskaber til at arbejde med teknikken.

På den ene side er granular en metode til at adskille de såkaldte frekvens- og tidsdomæner. Det betyder fx at man kan undgå, at et vokalsample bliver til en musestemme, selvom man sætter tempoet op (eller omvendt). Men på den anden side er granular oplagt til mere abstrakte former for lydproduktion, til at skabe unikke teksturer, klangflader og sågar abstrakte grooves, som ikke kan fremstilles på anden vis. Vi kan med granular syntese strække lyden, så de små detaljer vi ellers ikke bemærker, bliver forstærket og forlænget i det uendelige. Vi kan også sammensætte lyde fra meget forskellige sammenhænge til ét klangtæppe. Her gives en grundlæggende introduktion til disse sider af granular syntese, idet mere teknisk interesserede læsere henvises til Curtis Roads' uomgængelige grundbog om emnet, *Microsound* (Roads, 2002).

9.1.1 Kildemateriale

Man kan i princippet danne grains af hvilken som helst lydkilde. I SuperCollider findes der indbyggede UGens til at arbejde med syntetisk dannede grains (**GrainSin** og **GrainFm**), livelyd (**GrainIn**) og lydmateriale indlæst i en buffer (**GrainBuf** og **TGrains**). Man kan også foretage granular syntese „manuelt“, fx med **PlayBuf** eller **BufRd**, jævnfør forrige kapitel (se 8.1).

Det nok mest anvendte kildemateriale inden for granular syntese er samples. Når man vælger sit lydmateriale, skal man være klar over, at pauser eller stilhed i kildematerialet også bliver til stilhed, når det bliver afspillet i grain-form. Det er også væsentligt at gøre sig klart, om lydmaterialet primært indeholder tonale lyde (fx sang, akkord- eller blæser-/strygerinstrumenter) eller ikke-tonale lyd som feltoptagelser, støj, percussion osv.

Herunder tager vi udgangspunkt i et sample, der indeholder tonalt materiale i form af nogle toner spillet på en kalimba samt lidt baggrundstøj. **GrainBuf**, som vi primært skal arbejde med, kan kun læse fra én kanal ad gangen. Man kan sagtens modulere/ændre hvilken buffer, der læses fra, men vi kan kun læse fra én ad gangen.

Bufferne som indeholder lydmaterialet skal derfor indeholde præcis én kanal (dvs. mono), hvorfor vi herunder kun indlæser den ene kanal (se 8.1.1.1).

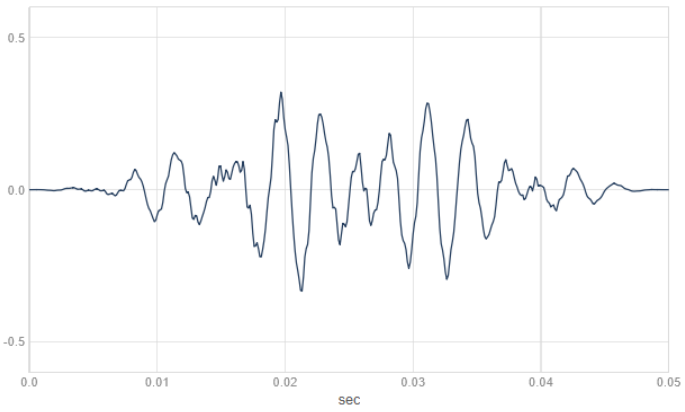
```
// Indlæs sample i mono - husk at erstatte stien til en lydfil på din egen computer
~kalimba = Buffer.readChannel(s, "C:/lydfiler/kalimba.wav", channels: [0]);
~kalimba.play;
```

Kodeboks 9.1: Indlæsning af sample til eksempler på granular syntese

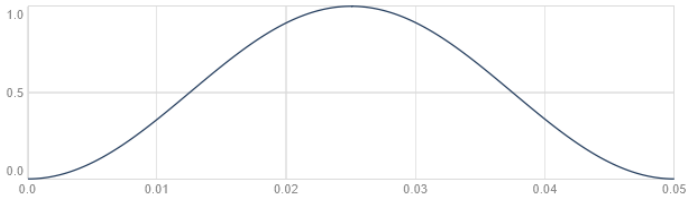
Vi kan fremstille et enkelt grain ved at afspille samplet med amplituden styret af en kort envelope. Der findes mange forskellige envelopes, som kan anvendes til grains, hvilket vi overlader til nysgerrige læser at nærstudere i litteraturen på området (Roads, 2002, p. 88). Her anvender vi for illustrationens skyld en sinusbølgelignende envelope med en varighed på 50 ms sammen med **PlayBuf**, der starter afspilningen en fjerdedel inde i samplet.

```
{
    PlayBuf.ar(1, ~kalimba, startPos: BufFrames.kr(~kalimba) * 0.25)
    * Env.sine(0.050).ar(2)
}.play;
```

Kodeboks 9.2: Afspilning af et enkelt grain



Figur 9.1: Bølgeform for et enkelt grain



Figur 9.2: Envelope for et enkelt grain

Selvom vi fint kunne fremstille grains på denne måde, er det mere oplagt at anvende specialindrettede redskaber som UGen'en **`GrainBuf`**.

9.1.2 Granular syntese med GrainBuf

Når vi arbejder med granulering af lydmateriale fra en **`Buffer`** er det oplagt at anvende **`GrainBuf`**, som netop læser grains fra en buffer. **`GrainBuf`**`.ar` har en række argumenter[1], som det er mest oplagt at demonstrere herunder. Her hører vi:

- 5 grains pr. sekund,
- hver med en varighed på 25 ms,
- læst fra midten af den buffer, hvor vores sample er indlæst,
- afspillet ved almindelig afspilningshastighed (her er **`BufRateScale`** *ikke* nødvendig, da **`GrainBuf`** tager højde for forskelle i samplerate),
- og placeret i midten af et stereofelt.

[1] Der findes yderligere argumenter til `GrainBuf`, men de er af mindre betydning for den grundlæggende anvendelse. Nysgerrige læsere henvises til SuperColliders dokumentation.

```
{
    GrainBuf.ar(
        // Antallet af output-kanaler
        numChannels: 2,

        // Et triggersignal, som udløser grains
        trigger: Impulse.ar(5),

        // Varigheden af grains
        dur: 0.025,

        // Den buffer, grains skal læses fra
        // Bufferen skal indeholde én lydkanal (mono)
        sndbuf: ~kalimba,

        // Afspilningshastighed for grains
        rate: 1,

        // Læseposition i bufferen
        // 0 er begyndelsen og 1 er slutningen
        pos: 0.5,

        // Position i stereofelt, ligesom ved Pan2 (når der arbejdes med 2 output-kanaler)
        pan: 0
    );
}.play;
```

Kodeboks 9.3: Brug af GrainBuf

Det lyder ikke af meget endnu, da vi har indstillet **GrainBuf** meget monotont og kedeligt for at demonstrere, hvordan den fungerer. Men det laver vi lige straks om på.

9.1.3 Synkron og asynkron granular syntese

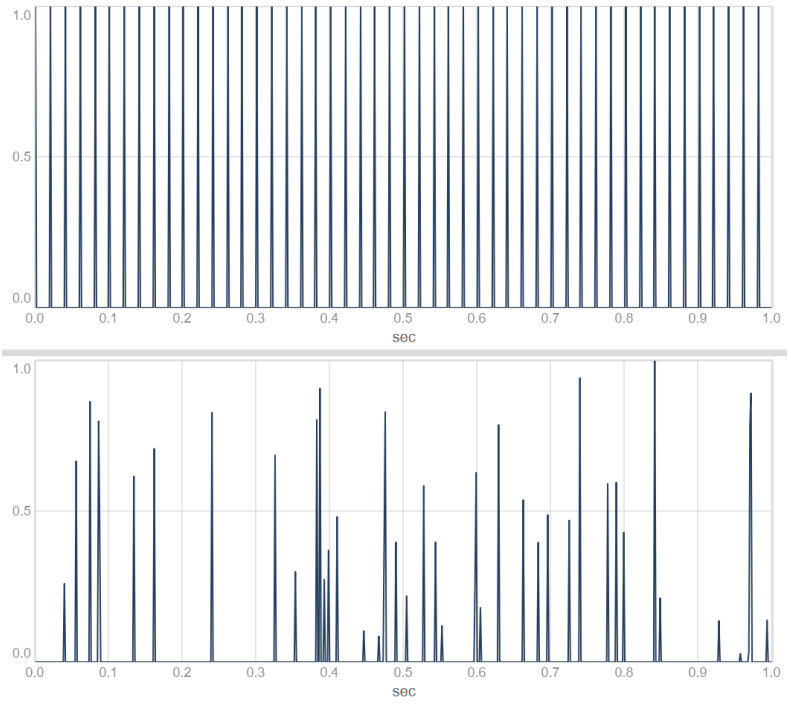
Et af argumenterne til **GrainBuf**.ar fortjener særlig uddybning, nemlig triggeren. Den fortæller nemlig **GrainBuf**, hvornår der skal udløses et nyt grain. Når vi vælger trigger, vælger vi samtidig mellem *synkron* og *asynkron granular syntese*.

Synkron granular syntese med **Impulse** som trigger

Med **Impulse** som trigger starter vi grains med ensartede tidsintervaller, der styres med den *triggerfrekvens*, vi angiver som første argument (freq) til **Impulse**.ar. Dette kaldes *synkron granular syntese* (Roads, 2002, p. 93).

Asynkron granular syntese med `Dust` som trigger

`Dust` genererer triggere med mere uregelmæssige tidsintervaller. Man angiver som første argument (`density`) til `Dust.ar` det ønskede, gennemsnitlige antal triggere pr. sekund, hvilket i vores lyddesign er sammenligneligt med triggerfrekvensen for `Impulse`. Med `Dust` som trigger producerer vi såkaldt *asynkron granular syntese* (Roads, 2002, p. 96).



Figur 9.3: To triggere: Impulse og Dust

Vi kan bruge multichannel expansion (se 7.1.3) til at demonstrere forskellen på `Impulse` (venstre kanal) og `Dust` (højre kanal) som trigger ved en lav triggerfrekvens/tæthed med (omtrent, ved brug af `Dust`) 5 grains pr. sekund.

```
{
    GrainBuf.ar(
        numChannels: 1,
        trigger: [Impulse.ar(5), Dust.ar(5)],
        dur: 0.100,
        sndbuf: ~kalimba,
        pos: 0.4
    );
}.play;
```

Kodeboks 9.4: Impulse og Dust som trigger til GrainBuf

Men hvorfor bruge den ene frem for den anden tilgang her? Man kan selvfølgelig have en æstetisk præference for det mere spredte eller kaotiske udtryk, som kendetegner asynkron granular syntese med `Dust` eller det mere rytmiske udtryk, der kendetegner synkron granular syntese med `Impulse`. Men for at forklare forskellen på de to tilgange er det nyttigt at dvæle ved perceptionen af granular syntese ved forskellige triggefrekvenser.

9.1.3.1 Effekter af forskellige triggerfrekvenser

Det er almenkendt, at den laveste tonefrekvens, det menneskelige øre kan opfatte, ligger omkring 20 Hz. Men hvad sker der under 20 Hz, og hvordan opfatter vi granular syntese, når vi bruger en triggerfrekvens, som ligger på over eller under dette skel?

Triggerfrekvenser under 20 Hz
: Når vi hører grains med hørbart anslag/attack produceret med en (trigger)frekvens på under 20 Hz, hører vi i udgangspunktet lyden som noget rytmisk (ved synkron granular syntese) eller noget arytmisk (ved asynkron granular syntese).

Triggerfrekvenser over 20 Hz
: Ved højere triggerfrekvenser smelter vores grains sammen i en kontinuerlig strøm. Med synkron granular syntese skaber triggeren sammen med amplitudeenvelopen en periodisk amplitude-modulation, hvor triggerfrekvensen træder frem som et hørbart artefakt, ofte med særskilt tonehøjde og hørbare „sidebånd“. Den har ofte en lidt „metallisk“ klang, som granular syntese i noget omfang er blevet kendt for. For at modvirke dette kan vi bruge asynkron granular syntese, der ikke resulterer i samme periodiske mønster.

Vi kan få en fornemmelse af forskellen ved at skifte mellem forskellige triggerfrekvenser, som ligger under og over 20 Hz. Herunder først den synkrone granular syntese,

hvor vi hører en tydelig tonehøjde og „metallisk“ klang, når triggerfrekvensen er over 20 Hz.

```
{
    var trigFreq = Env.new([1, 2, 5, 10, 15, 25, 50, 100, 200,
    ↪  300], 1, \step).kr.poll;
    GrainBuf.ar(
        numChannels: 1,
        trigger: Impulse.ar(trigFreq),
        dur: 0.05,
        sndbuf: ~kalimba,
        pos: 0.1
    ) * 0.1;
}.play;
```

Kodeboks 9.5: Synkron granular syntese med forskellige triggerfrekvenser

Med den asynkrone granular syntese hører vi ikke samme skift i tonehøjde som ovenfor, da tonehøjden i det oprindelige sample bevares.

```
{
    var trigFreq = Env.new([1, 2, 5, 10, 15, 25, 50, 100, 200,
    ↪  300], 1, \step).kr.poll;
    GrainBuf.ar(
        numChannels: 1,
        trigger: Dust.ar(trigFreq),
        dur: 0.05,
        sndbuf: ~kalimba,
        pos: 0.1
    ) * 0.1;
}.play;
```

Kodeboks 9.6: Asynkron granular syntese med forskellige triggerfrekvenser

9.1.4 Tæthed i strømmen af grains

En central parameter i granular syntese er tætheden mellem grains, dvs. om vi hører én kontinuerlig strøm af grains, eller om vi hører korte lydbidder med „luft“ imellem. Tæthed afhænger af forholdet mellem to af de parametre, vi angiver som argumenter til `GrainBuf`: Grainvarighed og triggerfrekvens. Man kan beregne mængden af tæthed med følgende simple formel:

$$\text{tæthed} = \text{triggerfrekvens} \times \text{grainvarighed}$$

Som eksempel kan vi tale om forskellige tætheder:

- Ved en triggerfrekvens på 100 Hz og en grainvarighed på 1/100 sekund vil der hverken være overlap eller luft mellem de enkelte grains: $100Hz \times 1/100s = 1$
- Var triggerfrekvensen på 50 Hz, ville tætheden være ½, dvs. der vil være stilhed 50% af tiden: $50Hz \times 1/100s = 0,5$
- Med en triggerfrekvens på 200 Hz, vil der konstant være 2 samtidigt klingende grains: $200Hz \times 1/100s = 2$

Rent kompositorisk er de mest interessante parametre at angive fast eller modulere med LFO'er nok tæthed og grainvarighed. Den konkrete triggerfrekvens kan vi udregne ved at isolere den i ovenstående ligning. Ønsker vi i stedet at angive triggerfrekvensen direkte for at opnå en bestemt effekt, kan vi også gøre det. Ønsker man at angive triggerfrekvens og mængde af tæthed for at lade SuperCollider udregne grainvarigheden, eller lade SuperCollider udregne mængden af tæthed ved specificerede triggerfrekvenser og grainvarigheder, kan man ligeledes isolere den ubekendte i ligningen:

$$\begin{aligned} \text{tæthed} &= \text{triggerfrekvens} \times \text{grainvarighed} \\ \iff \text{triggerfrekvens} &= \frac{\text{tæthed}}{\text{grainvarighed}} \\ \iff \text{grainvarighed} &= \frac{\text{tæthed}}{\text{triggerfrekvens}} \end{aligned}$$

Hvis fx grainvarighed og tæthed er styret af patterns via SynthDef-argumenter eller af en LFO, kan vi således beregne triggerfrekvensen automatisk. Herunder får vi eksempelvis fornemmelsen af, at lyden „strækkes", fordi der kommer luft i strømmen af grains, når mængden af tæthed, styret af LFO'en **`LFTri`**, dykker ned under 1.

```
{
    var grainDur = 0.050;
    var density = LFTri.kr(0.5).exprange(0.5, 2);
    var trigFreq = (density / grainDur);

    GrainBuf.ar(
        numChannels: 2,
        trigger: Impulse.ar(trigFreq),
        dur: grainDur,
        sndbuf: ~kalimba,
        pos: LFSaw.ar(1/~kalimba.duration, 1).unipolar
    );
}.play;
```

Kodeboks 9.7: Automatisk beregning af triggerfrekvens

9.2 Adskil tempo og tonehøjde

Én af de særligt nyttige egenskaber ved granular syntese er evnen til at adskille varighed og tonehøjde - eller mere teknisk - tids- og frekvensdomænet. Når vi normalvis skruer afspilningshastigheden for et sample op, bevæger tonehøjden sig også op. Men med granular syntese kan vi let skille disse dimensioner ad. Dette skyldes, at vi kan læse de enkelte grains med én hastighed, mens vi bevæger os igennem et sample med en anden hastighed.

Til nedenstående eksempler bruges samplet fra forrige afsnit (se 9.1.1), som er indlæst under variablen `~kalimba`.

9.2.1 En pointer til at styre tempo

Til at styre bevægelsen gennem en buffer kan vi genere et signal, som vi kalder en *pointer*. Pointerens funktion minder om en pickup-nål, som aflæser de informationer der er indprentet i en vinylplade ved at bevæge sig rundt i de dertil indrette riller. Pointeren angiver hvor i bufferen, grains skal læses fra.

Som pointer er det oplagt at bruge UGen'en **Phasor**, der genererer en lineær bevægelse fra én værdi til en anden. Når vi justerer den hastighed hvormed **Phasor** bevæger sig, justerer vi dermed *tempoet* i afspilningen.

```
~granulator = {
    arg moveRate = 1;

    var buf = ~kalimba;
    var numFrames = BufFrames.kr(buf);

    // Pointeren implementeres med Phasor
    var pointer = Phasor.ar(
        rate: moveRate,
        start: 0,
        end: numFrames
    ) / numFrames;

    GrainBuf.ar(
        numChannels: 2,
        trigger: Dust.kr(100),
        dur: 0.100,
        sndbuf: buf,
        rate: 1,
        pos: pointer,
        pan: 0
    ) * 0.1;
}.play;
```

Kodeboks 9.8: Fleksibelt tempo med Phasor som pointer 

Når vi har sat ovenstående i gang, kan vi styre tempoet med method'en `.set`, fordi vi har et argument i UGen-funktionen (se 5.3.3) kaldet `moveRate`.

```
// halvt tempo
~granulator.set(\moveRate, 0.5)

// tredobbelt tempo
~granulator.set(\moveRate, 3)

// ingen bevægelse
~granulator.set(\moveRate, 0)

// baglæns
~granulator.set(\moveRate, -1)
```

Kodeboks 9.9: Tempo påvirker ikke tonehøjde 

9.2.2 Fleksibel tonehøjde

Modificerer vi ovenstående ved at tilføje et argument til transponering, kan vi ændre afspilningshastigheden for de enkelte grains via **GrainBuf**'s argument `rate` og dermed ændre på *tonehøjden*.

```
~granulator = {
    arg moveRate = 1, transpose = 0;

    var buf = ~kalimba;
    var numFrames = BufFrames.kr(buf);

    // Pointeren implementeres med Phasor
    var pointer = Phasor.ar(
        rate: moveRate,
        start: 0,
        end: numFrames
    ) / numFrames;

    GrainBuf.ar(
        numChannels: 2,
        trigger: Dust.kr(100),
        dur: 0.1,
        sndbuf: buf,
        rate: transpose.midiratio,
        pos: pointer,
        pan: 0
    ) * 0.1;
}.play;
```

Kodeboks 9.10: Fleksibel tonehøjde

Vi kan igen justere indstillingerne, når ovenstående er sat i gang. Nu har vi foruden `moveRate` adgang til argumentet `transpose`.

```
// en kvint op
~granulator.set(\transpose, 7)

// en oktav ned
~granulator.set(\transpose, -12)

// tempo halveret to gange + en oktav op
~granulator.set(\moveRate, 0.25, \transpose, 12)
```

Kodeboks 9.11: Transponering påvirker ikke tempo

Bortset fra, at vi her kan høre tonehøjde og tempo som to helt adskilte parametre, er det interessant at høre en lille detalje ved kildematerialet: På en kalimba skabes lyden af metalstænger i forskellige længder, såkaldte lameller, der sættes i svingninger. Med et udtryk fra organologien (læren om musikinstrumenters karakteristika og historie) kan vi kalde et sådant instrument for en *idiofon* (Brown & Palmer, 2001). Man spiller på lamellerne med fingrene, og idet man rør ved den genstand som vibrerer og skaber lyden, skaber man helt kort en smule støj på grund af den friktion mellem fingeren og lamellen, som sætter lamellen i svingninger. I kvart tempo og oktaveret tonehøjde i slutningen af ovenstående lydeksempel bliver de støjfulde anslag pludselig ganske tydelige. Dette er en af de interessante egenskaber ved granular syntese: Teknikken tillader os fx at zoome helt ind på detaljer, som ellers ikke fremstår så tydeligt.

9.3 Klangflade og tekstur

Som vi så tidligere, giver granular syntese adgang til at strække eller transponere lyd på meget fleksible måder. Men teknik er også attraktiv på grund af dens evne til at skabe brede klangflader og abstrakte teksturer. Disse kompositionsmuligheder antyder vi herunder, men det er væsentligt at bemærke, at potentialet i denne teknik bedst kan udforskes gennem egne eksperimenter, da forskelligt lydmateriale vil give anledning til vidt forskellige klange, når det bearbejdes med granular syntese.

I det følgende bruger vi samme kalimba-sample som i de foregående afsnit (se 9.1.1), indlæst i buffer under variablen `~kalimba`.

9.3.1 Teksturdannelse i granular syntese

Ved at læse korte grains fra forskellige positioner i et sample, kan vi skabe en interessant, lydlig tekstur. For at skabe spredning og variation i teksturen kan vi fordele de læste grains i stereofeltet og transponere hvert enkelt grain et tilfældigt antal halvtoner op og ned.

Til dette formål kan vi bruge nogle specielle UGens, der genererer nyt output, hver gang de modtager en trigger. Her kan vi især bruge tilfældighedsgeneratorer som **TRand**`.kr(minimum, maksimum, trigger)` eller **TIRand**`.kr(minimum, maksimum, trigger)`, som henholdsvis genererer tilfældige decimaltal og heltal mellem et givet minimum og maksimum. Der findes også UGens som **TExpRand**, der i sin funktion minder om vores gode ven (se 2.2.1) **Pexprand**, og **TChoose**, der tilsvarende minder om **Prand**, idet den vælger tilfældigt mellem et antal givne muligheder (se 2.2.2).

Lad os skabe en lydlig tekstur med **GrainBuf** og nogle af disse redskaber:

- Her sætter vi tæthed til 10 for at få en kontinuerlig strøm af lyd.
- Grainvarigheden sættes til 25 ms.
- Triggerfrekvensen udregnes automatisk (se 9.1.4).
- Vi transponerer det enkelte grain op til to oktaver op eller ned med en kombination af **TIRand** og `.midiratio`.
- Vi læser med **TRand** det enkelte grain fra et tilfældigt sted i bufferen.
- Vi placerer med **TRand** det enkelte grain tilfældigt i hele stereofeltets bredde.

```
{
    var density = 10;
    var grainDur = 0.025;
    var trigFreq = (density / grainDur);
    var trigger = Dust.kr(trigFreq);

    GrainBuf.ar(
        numChannels: 2,
        trigger: trigger,
        dur: grainDur,
        sndbuf: ~kalimba,
        rate: TIRand.kr(-24, 24, trigger).midiratio,
        pos: TRand.kr(0, 1, trigger),
        pan: TRand.kr(-1, 1, trigger)
    );
}.play;
```

Kodeboks 9.12: Intens tekstur præget af aleatorik på mikroskala

Dette må siges at resultere i en intens, abstrakt og tæt tekstur, der ville egne sig til en tæt baggrundslyd eller et lydligt intermezzo. Men som jeg tidligere har bemærket, holder den konstante og totale tilfældighed hurtigt op med at være perceptuelt interessant, og øret mættes hurtigt af den tætte tekstur. Men kombinerer vi disse teknikker med en mere ordinær sample-afspilning, kan vi bevæge os ind på nogle ganske interessante områder.

9.3.1.1 Klangtekstur med jitter

Vi kan kombinere elementer af den kaotiske tekstur med den teknik, vi bruger til at styre aflæsningspositionen med en pointer (se 9.2.1). Det gør vi konkret ved at tilføje lidt „støj“ til pointeren. Man kan forestille sig en lynhurtig DJ, der dog ryster lidt på hånden og derfor springer lidt frem og tilbage i lyden. Støjen, som man med et engelsk udtryk kan kalde for *jitter*, kan vi generere med **TRand** og lignende UGens, som vist ovenfor. Vi indfører dertil et argument `jitter`, som angiver den maksimale afstand til pointeren, vi læsepositionen på springe til - målt i sekunder[2]. Vi laver også justerbar spredning i stereofeltet med et argument kaldet `spread`.

[2] Bemærk her, at vi både kan springe frem og tilbage. Det effektive „vindue“ vi læser i bufferen fra er derfor i praksis dobbelt så langt som jitter-argumentets værdi.

```
~shaky = {
    arg transpose = 0, moveRate = 1,
    jitter = 0.01, spread = 0.1;

    var buf = ~kalimba;
    var numFrames = BufFrames.kr(buf);
    var pointer = Phasor.ar(
        rate: moveRate,
        start: 0,
        end: numFrames
    ) / numFrames;

    var trigger = Dust.kr(200);

    var jit = TRand.kr(jitter.neg, jitter, trigger) /
    ↪  BufDur.kr(buf);

    var pan = TRand.kr(spread.neg, spread, trigger);

    GrainBuf.ar(
        numChannels: 2,
        trigger: trigger,
        dur: 0.1,
        sndbuf: buf,
        rate: transpose.midiratio,
        pos: pointer + jit,
        pan: pan
    ) * 0.1;
}.play;
```

Når ovenstående er startet, kan vi justere på indstillingerne og introducere jitter og stereo-spredning med `.set`-method'en. Vi kan også justere transponering og pointerens fart.

```
// Fordeling af grains i stereofelt (høres bedst i
↪  hovedtelefoner)
~shaky.set(\spread, 0.3)
~shaky.set(\spread, 1)

// Jitter
~shaky.set(\jitter, 0.1)
~shaky.set(\jitter, 0.5)

// Langsom eller stillestående pointer
~shaky.set(\moveRate, 0.5)
~shaky.set(\moveRate, 0)

// Tilfældig transponering
~shaky.set(\transpose, rrand(-12, 12))
```

Kodeboks 9.13: Variationsmuligheder med stereo-spredning, jitter mm.

Ønsker man at gå videre ud ad denne sti, kan man starte med at tilføje en tilsvarende form for jitter til transponering. Dette overlades til den nysgerrige læser at implementere på egen hånd.

9.3.2 Klangflader

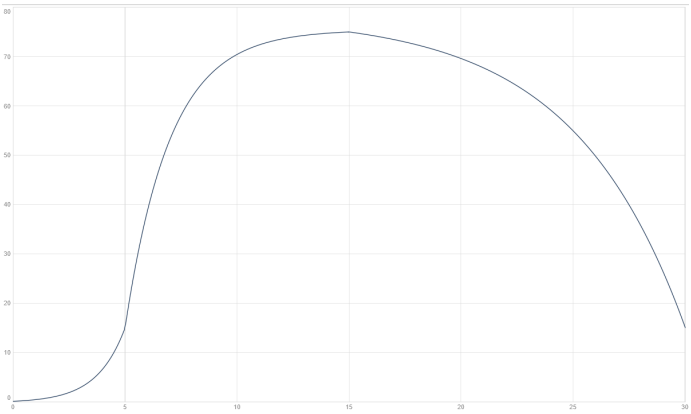
Hidtil har vi anvendt **Phasor** som pointer, hvilket resulterer i en lineær bevægelse frem eller tilbage. Men vi er på ingen måde tvunget til at tænke statisk eller lineært på denne eller andre parametre. Ved at modulere parametrene med envelopes og LFO'er kan vi fremmane andre lydlige forløb og måske endda abstrakte rytmer, som skaber klanglige mønstre over tid. Her handler det procesmæssigt om at vælge og indstille modulatorerne omhyggeligt, lytte til resultatet, og tune modulatorerne, lytte igen og så fremdeles.

9.3.2.1 En atmosfærisk rejse baglæns i kalimba-samplet

Når vi komponerer på dette abstraktionsniveau, dvs. uden brug af patterns som formgivende element, kan vi i stedet bruge envelopes (se 5.1). Som eksempel kan vi tage grain-tætheden som en central kompositorisk parameter. Til at styre denne opretter jeg en ny envelope med en varighed på i alt 30 sekunder, som kan ses på et plot herunder.

```
~densityEnv = Env.new(
    levels: [0.1, 15, 75, 15],
    times: [5, 10, 15],
    curve: [4, -5, 3]
);
~densityEnv.plot;
```

Kodeboks 9.14: En envelope til at styre grain-tætheden over tid



Figur 9.4: En envelope til at styre grain-tætheden over tid

Sammen med envelopen til styring af tæthed anvendes der en `Env.linen` til den overordnede volumen i kompositionen, og en simpel `Line` styrer sammen med en smule jitter læsepositionen i bufferen. Grains med en varighed på 200 ms kan måske knap nok kaldes grains, men det er en mindre detalje.

```
{
    var density = EnvGen.kr(~densityEnv);
    var grainDur = 0.200;
    var trigFreq = density / grainDur;
    var trigger = Dust.ar(trigFreq);
    var env = Env.linen(sustainTime: 30, releaseTime: 3, curve:
    ↪  \sin).kr(2);
    GrainBuf.ar(
        numChannels: 2,
        trigger: trigger,
        dur: grainDur,
        sndbuf: ~kalimba,
        rate: TRand.ar(-0.1, 0.1, trigger).midiratio,
        pos: Line.kr(0.95, 0.05, 30) + TRand.ar(-0.01, 0.01,
        ↪  trigger),
        pan: TRand.ar(-1, 1, trigger),
    ) * 0.1 * env;
}.play;
```

Kodeboks 9.15: Atmosfærisk klangflade

Bemærk, at der ikke er tilføjet rumklang til dette lydeksempel. Fornemmelsen af rumlighed stammer fra dels stereospredningen, dels fra en chorus-lignende effekt af de mange samtidigt klingende grains.

9.3.2.2 En transponeringssekvens, som går i stå og bevæger sig på samme tid

Hvis vi vil arbejde med lidt en lidt mere tonalt dynamisk komposition, kan vi bruge en envelope til at modulere **GrainBuf**s `rate`-argument. Herunder anvender jeg en **Env**`.circle` med en række transponeringstrin, der fungerer lidt ligesom en LFO (se 5.2.2) og omregnes til afspilningshastighed med `.midiratio`. Til envelopens `curve`-argument angives værdien `\step`, hvilket får envelopen til at bevæge sig i trin frem for i gradvise kurver.

Derudover kan vi under en lokal variabel `xline` definere en **XLine**, som bevæger sig i en eksponentiel bane fra 1 til 3 over 30 sekunder. Denne UGen spiller en central rolle, da den modulerer flere andre parametre:

- Den føromtalte envelope til modulation af tonehøjde bliver gradvist strakt, da segmenternes varigheder skaleres med `xline`. Tempoet i sekvensen bliver derfor langsommere og langsommere.
- Skiftene mellem modulationstrinnene sker pludseligt, men med `.lag` glider vi kort mellem værdierne. `xline` modulerer her hvor hurtigt dette slide sker.

- **GrainBuf** producerer i modsætning til tidligere eksempler ikke to kanaler men én. Til gengæld bliver den fordoblet på grund af multichannel expansion (se 7.1.3) i triggeren **Dust** samt den **LFTri**, der bestemmer aflæsningspositionen. I begyndelsen læser vi fra den samme position i bufferen (0.1), men over tid vokser **LFTri**s rækkevidde med `xline`, og vi hører som resultat større og større forskel på venstre og højre kanal.

```
{
    var density = 25;
    var grainDur = 0.100;
    var trigFreq = density / grainDur;
    var trigger = Dust.ar(trigFreq.dup(2));
    var xline = XLine.kr(1, 3, 30);
    var rate = EnvGen.kr(
        Env.circle(
            levels: [-12, -8, -5, -14],
            times: [1, 1, 2, 1],
            curve: \step
    ), timeScale: xline).lag(0.05 * xline.pow(2)).midiratio;
    var env = Env.linen(sustainTime: 30, releaseTime: 5, curve:
    ↪  \sin).kr(2);
    GrainBuf.ar(
        numChannels: 1,
        trigger: trigger,
        dur: grainDur,
        sndbuf: ~kalimba,
        rate: rate,
        pos: LFTri.kr([10, 10.1]).range(0.1, 0.1 * xline),
    ) * 0.1 * env;
}.play;
```

Kodeboks 9.16: Granular syntese med en XLine som central modulator

9.3.3 Videre perspektiver med granular syntese

Det er helt forståeligt, hvis eksempler som det foregående er vanskelige at gennemskue ved første øjekast. Men hvis man læser grundigt op på de grundlæggende emner som envelopes (se 5.1), skalering (se 4.2) og multichannel expansion (se 7.1.3), er det bestemt muligt at forstå og lade sig inspirere af kildekoden. Rent teknisk er granular syntese et komplekst emne med mange muligheder og parametre, og i dette kapitel er kun et par grundlæggende teknikker og kompositionsmuligheder antydet. For videregående emner såsom granulering af et live-lydsignal henvises nysgerrige læsere til Eli Fieldsteels *SuperCollider Tutorial: 26. Granular Synthesis, Part I/II* (Fieldsteel, 2020a, 2020b).

Da vi grundet de tekniske kompleksiteter ikke her har implementeret granular syntese i en SynthDef-form, er dette naturligvis en oplagt øvelse for læseren. Her gælder det blot om at læse godt op på SynthDef-konstruktion (se 5.3) og så ellers eksperimentere løs på egen hånd. God fornøjelse med det!

9.4 Øvelse: Granular syntese

I denne øvelse arbejder du med granulering af samples ved hjælp af UGen'en **GrainBuf**.

9.4.1 Valg og indlæsning af sample

Til brug i denne øvelse skal der indlæses et sample. Du kan bruge din egen lydfil, den skal blot:

- Vare maksimalt 10 sekunder.
- Være trimmet, så der ikke er stilhed i begyndelsen eller slutningen af samplet.
- Indlæses i mono.

Som forberedelse til de øvrige opgaver herunder:

1. Indlæs et sample i en buffer gemt under variablen `~buffer`.
2. Hvis din lydfil er i stereo, kan du indlæse den første kanal med `.readChannel` - se kodeblokken herunder.
3. Hvis du ikke har en lydfil klar selv, kan du bruge den nederste linje herunder, som indlæser et indbygget sample fra SuperCollider.

```
// Udfyld her med dit eget mono-sample
~buffer = Buffer.read(s,      );

// Brug .readChannel, hvis dit sample er i stereo
~buffer = Buffer.readChannel(s, channels: [0], path:   );

// Hvis du ikke har en lydfil klar, kan du bruge et indbygget
↪  sample i stedet:
~buffer = Buffer.read(s, Platform.resourceDir +/+
↪  "sounds/a11wlk01.wav");
```

Kodeboks 9.17: Indlæsning af sample

9.4.2 Grainvarighed, tæthed og triggerfrekvens

1. Undersøg ved hjælp af nedenstående kodeblok og din mus/touchpad (se hvilke parametre **MouseX** og **MouseY** styrer herunder), hvordan den automatiske udregning af triggerfrekvens fungerer ved forskellige kombinationer af grainvarighed og mængden af tæthed.

```
{
    var grainDur = MouseX.kr(0.01, 0.2, \linear).poll(label:
    ↪ \grainDur);
    var density = MouseY.kr(0.5, 20, \exponential).poll(label:
    ↪ \density);
    var trigFreq = (density / grainDur).poll(label: \trigFreq);

    GrainBuf.ar(
        numChannels: 2,
        trigger: Dust.ar(trigFreq),
        dur: grainDur,
        sndbuf: ~kalimba,
        rate: 1,
        pos: LFSaw.ar(0.5, 1).unipolar,
        pan: 0
    );
}.play;
```

Kodeboks 9.18: Grainvarighed, tæthed og triggerfrekvens

9.4.3 Modulation af granular syntese

Justér nedenstående kodeblok, således at:

1. Der kommer overlap mellem de enkelte grains, dvs. relativt høj tæthed.
2. Følgende parametre moduleres af LFO'er, fx **SinOsc**, **XLine**, **EnvGen**, **LFNoise**, **LFTri** etc.:
   a) `dur`
   b) `rate`
   c) `pos`
   d) `pan`
3. **Dust** anvendes som trigger i stedet for **Impulse**.
4. Overvej: Hvordan påvirker disse parametre og modulationer lyden? Hvilke æstetiske muligheder kan du se i denne form for sample-manipulation?

```
{
    GrainBuf.ar(
        numChannels: 2,
        trigger: Impulse.ar(10),
        dur: 0.025,
        sndbuf: ~buffer,
        rate: BufRateScale.kr(~buffer) * 1,
        pos: 0.5,
        pan: 0
    )
}.play;
```

Kodeboks 9.19: Modulation af GrainBuf-parametre

KAPITEL 10

Nye muligheder

Dette sidste kapitel indeholder nogle nyttige ressourcer og råd, som du kan bruge fremover. Først forklares det, hvordan man kan få lyden ud af SuperCollider og ind i andre programmer (typisk en DAW). Derefter præsenteres et par ideer til dit videre arbejde med komposition og andre projekter samt en liste med gode kilder til videre studier på egen hånd. Kapitlet kan dermed forhåbentlig udgøre et springbræt, som kan sende dig mod nye udfordringer og eventyr udi SuperCollider og computermusik.

10.1 Optag lyd fra SuperCollider

På et tidspunkt i din rejse med musik- og lydprogrammering ved hjælp af SuperCollider bliver det relevant at kunne bruge lyd fra SuperCollider i andre programmer. Det kan fx være vi vil eksportere en atmosfærisk lydtekstur skabt med granular syntese eller en melodi, vi har genereret ved hjælp af subtraktiv syntese og patterns. Hertil findes der i SuperCollider flere metoder, som passer til forskellige scenarier. Den interne optagelse er mest enkel og anbefales til begyndere, men at route signalet fra SuperCollider til en DAW er en fleksibel og nyttig metode.

10.1.1 Intern optagelse af SuperColliders lydserver

At optage outputtet fra lydserveren og gemme optagelsen i en lydfil er ganske enkelt.

```
1  // Start optagelsen
2  s.record;
3
4  // Spil noget lyd
5  { Pulse.ar([220, 222]) * Env.perc.kr(2) }.play;
6
7  // Afslut optagelsen
8  s.stopRecording;
9
10 // Tilgå mappen hvor filen med lydoptagelsen er gemt
11 Platform.recordingsDir.openOS;
```

Kodeboks 10.1: Enkel optagelse med s.record

Efter ovenstående linjer er kørt, vil SuperColliders post window vise hvor på din harddisk, lydfilen med optagelsen er gemt. Som udgangspunkt gemmes optagelser i en mappe, der kan findes ved at køre en linje med koden `Platform.recordingsDir;`. Filerne som indeholder optagelserne får tildelt et navn med et unikt timestamp, så flere optagelser foretaget efter hinanden ikke overskriver eksisterende filer.

I stedet for den automatiske navngivning er det muligt at angive filsti og -navn. Man kan også definere antal kanaler der skal optages samt en varighed for optagelsen mm.[1]. Her eksempelvis en fil, der hedder „optagelse.wav“ og gemmes under mappen „C:/lydfiler/“, har to lydkanaler (dvs. stereo) og varer tre sekunder:

[1] Du kan læse nærmere om argumenterne til `s.record()` i SuperColliders dokumentation.

```
s.record(
    path: "C:/lydfiler/optagelse.wav",
    numChannels: 2,
    duration: 3
);
// s.stopRecording er ikke nødvendigt her
```

Kodeboks 10.2: Argumenter til s.record

Laver du optagelser på denne måde, bør du være opmærksom på, at kører man koden ovenfor flere gange, vil den seneste eksekvering overskrive den tidligere optagelse. Man mister dermed de første „takes“. For at undgå dette kan man fx lave en algoritme, som tilføjer dato og tidspunkt til filnavnet for at gøre dette unikt. Det er i øvrigt netop hvad `s.record` af sig selv gør.

10.1.2 Optagelse med præcist begyndelsestidspunkt

Fordi der skal allokeres buffer-hukommelse til optagelsen starter `s.record` optagelsen et kort stykke tid efter, at kodelinjen er kørt. Det er lidt upraktisk, hvis man gerne vil starte optagelsen, præcist når man sætter en lyd i gang. Derfor kan man forberede optagelsen med `s.prepareForRecord`, så den kan startes på et præcist tidspunkt.

```
// Her kan path og numChannels evt. specificeres som argumenter
s.prepareForRecord();

(
// Optagelsen startes og stopper automatisk igen efter 1.1
↪  sekund
s.record(duration: 1.1);
// Vi starter også en stereo-lyd, som varer lidt mere end 1
↪  sekund
{ Pulse.ar([220, 222]) * Env.perc.kr(2) }.play;
)
```

Kodeboks 10.3: Forberedt optagelse

10.1.3 Superhurtig NRT-optagelse

Hvis man har lavet en algoritme, der kan fremstille lydoptagelser der er lange eller et stort antal, kan det være nyttigt at få SuperCollider til at generere lydfilerne i „non-

realtime“ - deraf tilnavnet NRT. Det er et lidt mere kompliceret emne, som læseren selv kan studere nærmere i SuperColliders dokumentation samt i en udmærket blog-post af Mads Kjeldgaard.

10.1.4 Routing fra SuperCollider til DAW

I mange tilfælde er det nyttigt at route lyden fra SuperCollider over til et andet program, fx en DAW. Det kan man gøre på tre overordnede måder, som nævnt i Abletons udmærkede vejledning:

- Analog loopback
- Digital loopback
- Virtuel audio routing

Analog og digital *loopback* beror på, at der sendes lyd ud af et lydkorts udgange og direkte tilbage via lydkortets indgange. Nogle lydkort-drivere understøtter, at dette kan gøres uden fysiske kabler. Disse tilgange kræver typisk et eksternt lydkort.

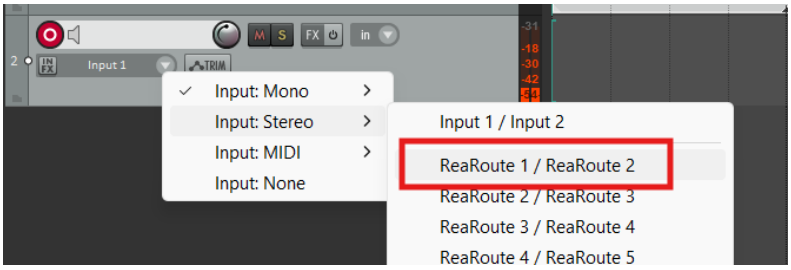
Virtuel audio routing kan udføres uden særlig hardware (eksternt lydkort) og er derfor en meget anvendt tilgang. Det udføres som regel med tredjepartssoftware, der figurerer i styresystemet som en ekstra lydkortdriver. Denne kan så både bruges som in- og output i programmer som SuperCollider og DAWs. I vejledningen fra Ableton fremgår en række redskaber til Mac og Windows. På Linux findes der udmærkede redskaber som jack og pipewire til avanceret, intern audio-routing.

10.1.4.1 Eksempel: SuperCollider til Reaper via ReaRoute

Som eksempel på virtuel audio routing kan vi tage det scenarie, at en Windows-bruger ønsker at sende lyd fra SuperColliders lydserver til DAW'en Reaper. Dette kan gøres ved hjælp af systemet ReaRoute, der følger med Reaper, hvis man vinger det af under installationen. Fremgangsmåden er som følger:

Forberedelse i Reaper

1. Start Reaper.
2. Aktivér derefter et spor til optagelse og vælg ReaRoute som audio-input.



Figur 10.1: Vælg ReaRoute som input i Reaper

1. Du er nu klar til at se og høre outputtet fra SuperCollider i Reaper.

Fremgangsmåde i SuperCollider

1. I SuperCollider kan vi indstille lydserveren til at sende lyden til et bestemt output ved hjælp af `s.options.device = "Mit lydkort";`. Men hvordan ved man, hvad man skal skrive i stedet for „Mit lydkort“?
   a) Når man booter lydserveren med `s.boot`, vises en liste med mulighederne under `Device options:`.
   b) Her vil der fremgå en linje, der minder om denne: `- ASIO : ReaRoute ASIO (x64) (device #9 with 16 ins 16 outs)`.
   c) ReaRoutes navn i SuperCollider er altså `"ASIO : ReaRoute ASIO (x64)"`.
   d) Vi angiver derfor `s.options.device = "ASIO : ReaRoute ASIO (x64)";`.
2. Valget af ReaRoute træder først i kraft, næste gang vi booter serveren. Derfor kører vi `s.reboot;` (eller `s.boot`, hvis serveren ikke er bootet allerede).
   a) Herefter kan vi se, hvilken port SuperCollider sender lyd til under `Booting with:` i post window.
3. Test at lyden går igennem til Reaper: `{PinkNoise.ar * Env.perc.kr(2)}.play;`.

Du kan finde en grunding introduktion til valg af lydkort/driver for lydserveren i artiklen *Audio device selection* fra SuperColliders dokumentation.

10.2 Dine næste skridt

Tillykke med, at du har læst denne bog og forhåbentlig har arbejdet aktivt med dens indhold på egen hånd. Du har i så fald gennemført et stort stykke arbejde, som jeg håber har været udbytte- og lærerigt. Men din rejse udi komposition og lydproduktion med SuperCollider behøver ikke slutte her. Herunder deler jeg et par ideer til hvad du kan arbejde videre med efter denne bog, samt hvilke andre læringsmaterialer, du med fordel kan opsøge. God fornøjelse med din videre færd udi (computer)musikkens forunderlige verden.

10.2.1 Ideer til projekter og kompositioner

Når du har fået styr på det grundlæggende, åbner der sig et væld af muligheder. Her er nogle ideer til, hvordan du kan arbejde videre med komposition og lydproduktion i SuperCollider, både i forhold til tekniske og kunstneriske projekter.

10.2.1.1 Kompositionsideer

Opstil dogmer og benspænd
: Sæt kreative benspænd for dig selv, fx at din kompositionen højst må vare ét minut og kun må bruge **`SinOsc`**, **`Env.sine`** og **`Pseries`**. Eller at et helt værk skal genereres ud fra tekststrenge fra dit yndlingsdigt, der omsættes til tonehøjder og rytmer (se 2.6.2). Eller at du fremstiller en komposition udelukkende i én UGen-funktion, hvor formen bestemmes af en eller flere envelopes og LFO'er.

Fortolk et eksisterende værk
: Prøv at implementere et kendt værk i SuperCollider, ligesom [vi arbejdede med en del af Steve Reichs *Piano Phase*](../03/a-sammensaetning.md#en-minimalistisk-kompositionside). Det kræver, at du oversætter og fortolker værkets instrukser til kildekode, hvilket både kan være en kreativ og lærerig proces. Du kan derefter skrive din egen komposition, inspireret af din indsigt i det andet værk.

Kombinér SuperCollider med andre teknologier
: Der er ingen grund til at at være SuperCollider-purist og udføre alt sit lydarbejde med ét redskab. Ofte kan det være inspirerende at kombinere forskellige teknologier og se, hvad den særlige kombination lægger op til. Brug fx SuperCollider som MIDI-generator til dit yndlings-plugin i en DAW. Eller indlæs VST plugins i SuperCollider med udvidelsen VSTPlugin og komponér løs med patterns og UGens.

Udforsk klangdannelsesteknikker
: Dyk ned i teknikker som FM/PM-syntese, granular-syntese, physical modelling osv. Brug litteratur om disse teknikker som inspiration (se hertil litteraturhenvisningerne i denne bog), og prøv at implementere teknikkerne i SuperCollider. Brug patterns til at udforske mulighederne i de SynthDefs, du kan konstruere. Det er både lærerigt og giver masser af nye ideer til egne kompositioner.

10.2.1.2 Projektideer

Skab dit eget unikke sample-bibliotek
: Lav dine egne samlinger af beats, lydeffekter, trommelyde, abstrakte lydtæpper, køkkenlyde eller hvad du nu kan finde på – alt sammen genereret eller manipuleret i SuperCollider. Husk, at SuperCollider også er et programmeringsredskab. Du kan derfor beskrive algoritmer, som genererer et væld af variationsmuligheder, som du kan modificere og selektere i.

Byg din egen synthesizer
: Design og programmér din egen synthesizer, evt. med et MIDI-keyboard som input. Prøv at køre det hele på en minicomputer som Raspberry Pi for at gøre det til et selvstændigt instrument. Konsultér Marije Baalmans glimrende bog om udvikling af sådanne interaktive systemer (Baalman, 2022), og se evt. Eli Fieldsteels guide til organisering af større SuperCollider-projekter (Fieldsteel, 2024a, p. 249).

GUI (graphical user interface)
: Skab dine egne grafiske brugerflader i SuperCollider og brug dem til at styre en algoritmisk komposition. Start med Eli Fieldsteels tutorial om GUI (Fieldsteel, 2015).

Udforsk nye klange med wavetables
: Dette emne kan du let sætte dig ind, da det beror på en grundlæggende forståelse af buffere og oscillatorer. Med wavetables kan man i princippet spille med enhver bølgeform, og der findes ganske mange gratis wavetables, fx pakken AKWF FREE, som allerede er konverteret til SuperColliders wavetable-format.

Udforsk udvidelser til SuperCollider
: Prøv kræfter med de mange udvidelser, der findes til SuperCollider, og se, hvordan de kan udvide dine muligheder. Start med at kigge på de mange forskellige Quarks og ekstra plugins. Kig også på den desværre ikke helt opdaterede men alligevel velkuraterede liste Awesome SuperCollider.

Deltag i open source-fællesskabet
: Engagér dig i online-fællesskaber som scsynth.org, studér andres ideer på sccode.org og følg med på sociale medier. Her kan du finde inspiration, få hjælp og dele dine egne projekter. Denne bog handler om open source software,

og hermed er opfordringen givet videre til at udbrede open source-kulturen yderligere - til alles fordel.

10.2.2 Ressourcer til videre studier

Selvom denne bog har givet dig en grundig introduktion til komposition og lydproduktion i SuperCollider, kan én tekst dog ikke gøre rede for alt. Der findes heldigvis glimrende læringsmaterialer om anvendelse af SuperCollider, som dog stort set alle er på engelsk. Men det er altid lærerigt og bestemt indsatsen værd at genstudere det samme emne gennem andre forfattere og underviseres perspektiv. Dette skyldes blandt andet, at programmering ofte - og i SuperCollider i særdeles - kan udføres på mange forskellige måder. Med afsæt i denne bog har du gode forudsætninger for at forstå den tekniske lingo, som måske i første omgang kan virke afskrækkende, når man dykker ned i litteraturen. Følgende ressourcer med fokus på SuperCollider henvender sig både til begyndere og til de, som har styr på det grundlæggende og er interesseret i mere avancerede teknikker. De får alle en klar anbefaling herfra, og med afsæt i denne bog kan de nedenstående ressourcer hjælpe dig langt videre.

10.2.2.1 Anbefalede materialer til begyndere

- E-bogen *A Gentle Introduction to SuperCollider* af Bruno Ruviaro (2015) er en glimrende, og som titlen indikerer, blid introduktion til SuperCollider. Den kan ikke anbefales nok.
- Videoserien *SuperCollider Tutorials* af Eli Fieldsteel (2024b) er ganske enkelt fremragende, ikke blot som formidling af grundlæggende programmering i SuperCollider, men (især i den sidste halvdel af serien) også som en god introduktion til kerneemner inden for computermusikken. Serien har udgjort en af de primære ressourcer i mine egne studier udi SuperCollider.
- Bogen *SuperCollider for the Creative Musician: A Practical Guide*, også af Eli Fieldsteel (2024a), er en rigtig god introduktion til SuperCollider for musikere.

10.2.2.2 Anbefalede materialer på videregående niveau

- E-bogen *Scoring Sound: Creative Music Coding with SuperCollider* af Thor Magnusson (2021) er en meget grundig og omfangsrig introduktion til ikke blot SuperCollider men også kerneemner som klangdannelse og elektronisk komposition inden for computermusik. Den har været et lærerigt bekendtskab i min egen rejse med SuperCollider.
- Bogen *Introduction to SuperCollider* er en engelsk oversættelse af Andrea Valles (2016) udmærkede grundbog om SuperCollider.

- Videoserien *Musical Sound Design In SuperCollider* af Alik Rustamoff (2022) introducerer til avancerede emner inden for klangdannelse og lyddesign, med fokus på FM, wavetable-syntese, rumklangsdesign, lofi-æstetik og meget mere.

Der findes mange andre materialer, som kunne have været nævnt her, men jeg har primært nævnt de ressourcer, jeg selv har et godt kendskab til og kan anbefale til denne bogs læsere. God fornøjelse med at udforske dem på egen hånd.

Figurer

Kodebokse

Referencer

ajubamusic. (2015 august). Funky drum loops.84 bpm.mp3. Hentet 5. juni 2025, fra https://freesound.org/people/ajubamusic/sounds/320803/

Artists' Statements I. (2017). I N. Collins & J. D'Escrivan (Red.), *The Cambridge Companion to Electronic Music* (s. 75–85). Cambridge University Press.

Berg, R. E. (2025 marts). Sound. Hentet 12. marts 2025, fra https://www.britannica.com/science/sound-physics/Steady-state-waves

Brown, H. M., & Palmer, F. (2001). Idiophone. Hentet 10. juni 2025, fra https://www.oxfordmusiconline.com/grovemusic/display/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000050024

Baalman, M. (2022). *Composing Interactions: An Artist's Guide to Building Expressive Interactive Systems*. V2_Publishing.

Dennis, C. J. (2015 marts). 'Urry. Hentet 6. juni 2025, fra https://librivox.org/short-poetry-collection-142-by-various/
Part of LibriVox' Short Poetry Collection 142.

Dyndahl, P. (2005). Kulturens Xerox-grad Eller Remixet Autentisitet? Gjenbruk Og Originalitet i Hiphop Og Samplingkultur. I P. Dyndahl & L. A. Kulbrandstad (Red.), *High Fidelity Eller Rein Jalla? Purisme Som Problem i Kultur, Språk Og Estetikk* (s. 201–228, Bd. 309). Oplandske Bokforlag.

Eskildsen, A. (2022 maj). Alea. Hentet 6. juni 2025, fra https://github.com/aeskildsen/alea

Eskildsen, A. (2024). *Soundpainting: Collaborative Creativity in Conducted Improvisation*. Springer Nature. https://doi.org/10.1007/978-981-96-1690-9

Eskildsen, A., & Horvath, A. S. (2022). Sonic Coexistence. *Proceedings of the 14th Conference on Creativity and Cognition*, 660–662. https://doi.org/10.1145/3527927.3531198

Eskildsen, A., & Walther-Hansen, M. (2020). Force Dynamics as a Design Framework for Mid-Air Musical Interfaces. *Proceedings of the International Conference on New Interfaces for Musical Expression*, 361–366.

Essl, K. (2017). Algorithmic Composition. I N. Collins & J. D'Escrivan (Red.), *The Cambridge Companion to Electronic Music* (s. 104–122). Cambridge University Press.

Fieldsteel, E. (2015 marts). SuperCollider Tutorial: 14. GUI. Hentet 11. juni 2025, fra https://www.youtube.com/watch?v=W2D_PzOVfT0

Fieldsteel, E. (2020a juni). SuperCollider Tutorial: 25. Granular Synthesis, Part I. Hentet 10. juni 2025, fra https://www.youtube.com/watch?v=WBqAM_94TW4

Fieldsteel, E. (2020b juni). SuperCollider Tutorial: 26. Granular Synthesis, Part II. Hentet 10. juni 2025, fra https://www.youtube.com/watch?v=MnD8stNB5tE

Fieldsteel, E. (2021 juli). SuperCollider Mini Tutorial: 4. Patterns vs Streams. Hentet 31. januar 2025, fra https://www.youtube.com/watch?v=17uMs9HpMgE

Fieldsteel, E. (2024a januar). *SuperCollider for the Creative Musician: A Practical Guide*. Oxford University Press. https://doi.org/10.1093/oso/9780197616994.003.0011

Fieldsteel, E. (2024b marts). SuperCollider Tutorials - YouTube. Hentet 16. april 2024, fra https://www.youtube.com/playlist?list=PLPYzvS8A_rTaNDweXe6PX4CXSGq4iEWYC

Gregersen, E. (2015 december). Ada Lovelace: The First Computer Programmer. Hentet 12. februar 2025, fra https://www.britannica.com/story/ada-lovelace-the-first-computer-programmer

Harkins, H. J. (2009). Pattern Guide 07: Value Conversions (A Practical Guide to Patterns). Hentet 17. juni 2025, fra http://doc.sccode.org/Tutorials/A-Practical-Guide/PG_07_Value_Conversions.html

Ho, N. (2017 juni). Three Kicks. Hentet 11. juni 2025, fra http://sccode.org/1-57g

Hunt, A. (2000). *The pragmatic programmer: From journeyman to master*. Reading, Mass. : Addison-Wesley. Hentet 9. april 2024, fra http://archive.org/details/isbn_9780201616224

Karplus, K., & Strong, A. (1983). Digital Synthesis of Plucked-String and Drum Timbres. *Computer Music Journal*, *7*(2), 43–55. https://doi.org/10.2307/3680062

Kvifte, T. (2007). Digital Sampling and Analogue Aesthetics. I A. Melberg (Red.), *Aesthetics at Work* (s. 105–128). Unipub.

Lovelace, A. A. (2015). 1842 Notes to the Translation of the Sketch of the Analytical Engine. *Ada User Journal*, *36*(3), 152–180.

Magnusson, T. (2021). Scoring Sound: Creative Music Coding with SuperCollider. Hentet 16. april 2024, fra https://leanpub.com/ScoringSound

McCartney, J. (2002). Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, *26*(4), 61–68. Hentet 14. februar 2025, fra https://www.jstor.org/stable/3681770

McManus, K. (Red.). (2024 februar). *Usage in Second Language Acquisition: Critical Reflections and Future Directions*. Routledge. https://doi.org/10.4324/9781032668475

Miller, P. D. (Red.). (2008). *Sound Unbound: Sampling Digital Music and Culture*. MIT Press.

Mozart, W. A. (1793). *Musikalische Würfelspiele, K.Anh.C.30.01*. N. Simrock. https://imslp.org/wiki/Special:ReverseLookup/798372

Navas, E. (2012). *Remix Theory: The Aesthetics of Sampling*. Springer.

Pejrolo, A., & Metcalfe, S. B. (2017). *Creating Sounds from Scratch: A Practical Guide to Music Synthesis for Producers and Composers*. Oxford University Press.

Puckette, M. (2007). *The Theory and Technique of Electronic Music*. World Scientific Publishing Co. Pte. Ltd. https://doi.org/10.1142/6277

Puckette, M. S. (1996). Pure Data. *Proceedings of the International Computer Music Conference*, 269–272.

Reid, G. (2002a februar). Practical Bass Drum Synthesis. Hentet 11. juni 2025, fra https://www.soundonsound.com/techniques/practical-bass-drum-synthesis

Reid, G. (2002b april). Practical Snare Drum Synthesis. Hentet 29. januar 2025, fra https://www.soundonsound.com/techniques/practical-snare-drum-synthesis

Roads, C. (2002). *Microsound*. MIT Press.

Roads, C. (2023). *The Computer Music Tutorial* (2.). MIT Press.

Rodgers, T. (2003). On the Process and Aesthetics of Sampling in Electronic Music Production. *Organised Sound*, *8*(3), 313–320. https://doi.org/10.1017/S1355771803000293

Rustamoff, A. (2022). Musical Sound Design In SuperCollider. Hentet 11. juni 2025, fra https://www.youtube.com/playlist?list=PLXCUkMwOEWQtB-leHHSexTizzcACdozp9

Ruviaro, B. (2015). *A Gentle Introduction to SuperCollider*. CCRMA.

Sewell, A. (2014). Paul's Boutique and Fear of a Black Planet: Digital Sampling and Musical Style in Hip Hop. *Journal of the Society for American Music*, *8*(1), 28–48. https://doi.org/10.1017/S175219631300059X

Smalley, D. (1997). Spectromorphology: Explaining Sound-Shapes. *Organised Sound*, *2*(2), 107–126.

soneproject. (2013 september). Pack: Electro-Drums. Hentet 5. juni 2025, fra https://freesound.org/people/soneproject/packs/12711/

Sub-d. (2008 januar). Emin 9 guitar chord interup.wav. Hentet 4. juni 2025, fra https://freesound.org/people/Sub-d/sounds/46992/

SuperCollider 3 documentation contributors. (u.d.). Event. Hentet 17. juni 2025, fra https://doc.sccode.org/Classes/Event.html

Tillet, S. (2014). Strange Sampling: Nina Simone and Her Hip-Hop Children. *American Quarterly*, *66*(1), 119–137.

Valle, A. (2016). *Introduction to SuperCollider*. Logos Verlag Berlin GmbH.

Walther-Hansen, M., & Eskildsen, A. (2024). Forceful Action and Interaction in Non-Haptic Music Interfaces. I J.-O. Gullö, R. Hepworth-Sawyer, J. Paterson, R. Toulson & M. Marrington (Red.), *Innovation in Music*. Routledge. https://doi.org/10.4324/9781003118817-18

Wang, G. (2017). A History of Programming and Music. I N. Collins & J. D'Escrivan (Red.), *The Cambridge Companion to Electronic Music* (s. 58–74). Cambridge University Press.