KAPITTEL 4

Følger og differensligninger

Følger er et sentralt begrep i matematikk som dukker opp i mange ulike sammenhenger. Samtidig er det mange naturlige prosesser som kan beskrives ved hjelp av følger, og svært mange beregninger består i å generere følger av tall. Her vil vi fokusere spesielt på tre temaer i forbindelse med følger og differensligninger. I seksjon 4.2 ser vi litt på hvordan vi rent numerisk kan regne ut følger på datamaskin ved hjelp av differensligninger, og hvilke problemer avrundingsfeil kan gi i denne sammenhengen. I seksjon 4.3 viser vi hvordan vi kan generere tilfeldige tall ved hjelp av differensligninger, mens vi i seksjon 4.4 ser litt på digital lyd og hva det er. Digital lyd vil vi komme mer inn på i senere kapitler.

Et viktig problem i forbindelse med følger er hvordan vi kan avgjøre ut fra numeriske beregninger om en gitt følge konvergerer. Dette spørsmålet blir ikke diskutert i dette kapitlet, men vi vil komme inn på det i senere kapitler siden følger står sentralt i de fleste gjenværende kapitlene.

Mesteparten av dette kapitlet kan leses uavhengig av kapittel 4 i *Kalkulus*, men forklaringen på problemet med avrundingsfeil i forbindelse med numerisk simulering av differensligninger i seksjon 4.2 krever kjennskap til hvordan en løser andreordens differensligninger.

4.1 Noen forskjellige typer følger

I de fleste lærebøker i matematikk er en følge $\{a_n\}$ en *uendelig* sekvens av tall, men vi skal ikke være så strenge og vil også tillate *endelige* følger. Vi begynner med å se litt på hvordan følger ofte beskrives matematisk.

4.1.1 Matematiske følger

En uendelig følge er en samling av tall som er ordnet i rekkefølge, og det at følgen er uendelig betyr at til hvert naturlig tall n (som det er uendelig mange av) svarer det et ledd a_n i følgen. Det er mange måter å angi følger på i matematikk, men de to vanligste er ved en eksplisitt formel og implisitt, ved differensligninger.

Følger gitt ved eksplisitte formeler. Det aller enkleste er å gi en formel som lar oss regne ut leddene i følgen direkte. To enkle eksempler er følgene

$$\begin{aligned} \{n\} &= 1, 2, 3, 4, 5, 6, \ldots, n, \ldots, \\ \{1/n\} &= 1, 1/2, 1/3, 1/4, 1/5, 1/6, \ldots, 1/n, \ldots, \end{aligned} \tag{4.1}$$

der alle leddene er gitt med en eksplisitt formel som gjelder for alle naturlige tall. Slike følger opptrer ofte for eksempel i bevis der vi vet akkurat hvilken følge vi trenger. Når vi har en eksplisitt formel for følgen er den som regel forholdsvis enkel å analysere. Interessante spørsmål er: Konvergerer følgen, og hva er i tilfelle grensen? Går den mot uendelig, og i såfall, hvor fort vokser den? Disse spørsmålene kan som oftest besvares ved analyse uten datamaskin selv om plott og numeriske beregninger ofte kan være nyttig for å få litt 'følelse' for hvordan følgen oppfører seg.

Følger gitt ved differensligninger. En litt mer komplisert måte å angi en følge på er ved en differensligning, slik som Fibonaccifølgen

$$x_{n+2} = x_{n+1} + x_n, \qquad \text{med } x_1 = 1 \text{ og } x_2 = 1. \tag{4.2}$$

Her angir vi de to første leddene direkte, de kalles *startverdiene* eller *initialverdiene*, mens de resterende leddene er gitt ved en formel som sier hvordan et ledd kan regnes ut fra de to foregående leddene. Ved å begynne forfra kan vi da regne ut x_3, x_4 og så videre etter tur. En formel som (4.2) kalles en *differensligning* eller *rekursjonsformel* (navnet *rekurrensrelasjon* brukes også) og sies å være *rekursiv*.

Differensligninger går godt sammen med datamaskiner siden de gir oss en formel for å regne ut leddene som vi lett kan implementere i et program. Som vi skal se er dette allikevel ikke uproblematisk siden vi fort kan få problemer med avrundingsfeil når vi bruker flyttall.

Det fins mange typer differensligninger. De enkleste er de *lineære* ligningene, slik som ligningen for Fibonaccifølgen (4.2). Det at denne ligningen er lineær betyr at formelen på høyre side i (4.2) for å regne ut x_{n+2} bare inneholder summer av de tidligere leddene opphøyd i første potens og multiplisert med tall. Vi har altså ingen uttrykk som x_n^2, eller mer generelt, ledd av typen x_n^p med $p \neq 1$. Vi har heller ikke uttrykk som $f(x_n)$, for eksempel $\sin x_n$ på høyre side, kun enkle summer av tidligere ledd multiplisert med konstanter, såkalte *lineære kombinasjoner* av tidligere ledd.

Fibonacciligningen (4.2) er et eksempel på en *andreordens* differensligning. 'Andreordens' betyr her at høyresiden bare inneholder uttrykk som involverer de to foregående leddene x_{n+1} og x_n. Et eksempel på en første ordens ligning er $x_{n+1} = 3x_n$. Her involverer høyresiden kun det foregående elementet. Generelt kan vi ha en $m$te ordens ligning der høyresiden avhenger av de m foregående leddene i følgen. For eksempel er

$$x_{n+1} = x_n + x_{n-1} + x_{n-2} + x_{n-3} + x_{n-4}$$

en femteordens ligning siden høyresiden involverer 5 tidligere ledd. Den store fordelen med lineære differensligninger er at hvis vi har to løsninger vil også summen av løsningene være en løsning, og hvis vi multipliserer alle leddene i en løsning med det samme tallet får vi også en løsning. Dette kan vi utnytte i en matematisk analyse av lineære differensligninger, slik som i læreboka. Linearitet er forøvrig et svært viktig begrep i matematikk som dukker opp i mange ulike sammenhenger og som studeres systematisk i lineær algebra.

La oss også nevne at det fins *ikke-lineære* ligninger der formelen på høyre side er mer komplisert, slik som ligningen

$$x_{n+1} = \sin x_n + \sqrt{x_{n-1}}. \tag{4.3}$$

Selv om denne siste ligningen er noe mer komplisert enn Fibonacciligningen (4.2) har vi fremdeles en eksplisitt formel for det neste leddet i følgen. For beregninger på en datamaskin er derfor ikke (4.3) spesielt komplisert siden vi kjapt kan få beregnet mange ledd. Men den rent matematiske analysen av ikke-lineære ligninger er langt mer komplisert enn for de linære.

Den strenge matematiske definisjonen av en følge krever at den skal ha uendelig mange ledd, og for en matematiker er dette både naturlig og problemfritt siden det fins et godt apparat for å resonnere med uendelighetsbegrepet. Men i virkelighetens verden kan det kanskje virke litt overdrevet å operere med uendelige følger. Når vi leser i *Kalkulus* hvordan Fibonacci kom fram til Fibonacci-tallene ved å sette opp en modell for hvordan kaniner formerer seg, så er det utenkelig at vi i en slik sammenheng noen gang vil få bruk for de uendelig mange tallene som genereres av (4.2), siden vi aldri vil kunne få uendelig mange generasjoner av kaniner. På den annen side vil vi få problemer hvis vi stopper følgen etter et visst antall generasjoner, for vi vet jo ikke når kaninene vil slutte å formere seg. Det er derfor fruktbart å operere med en uendelig følge i denne sammenhengen. Da kan vi håndtere alle mulige antall generasjoner og slipper unna problemet med når kaninproduksjonen stopper. En helt annen sak er at denne modellen for hvordan kaniner formerer seg er svært forenklet og derfor ikke vil være riktig for særlig mange generasjoner.

4.1.2 Anvendelser av følger

Verden rundt oss er full av følger, og i praksis er de alle endelige (i den forstand at de inneholder et endelig antall ledd), selv om vi ikke kan sette en øvre grense på antall ledd vi noensinne vil ha i en følge. Hvis vi måler en størrelse med jevne mellomrom og fører en liste over resultatet får vi en følge, hvis vi kaster en terning mange ganger får vi en følge av tilfeldige heltall mellom 1 og 6, på ei CD-plate ligger det en følge av tall som representerer musikken på plata, og når vi prater på en ISDN-telefon eller GSM-mobiltelefon blir lyden vi hører overført som en følge av tall. Felles for alle disse eksemplene er at de genererte følgene er endelige, akkurat som i kanineksemplet.

Differensligningen (4.2) er en enkel *matematisk modell* for hvordan kaniner formerer seg, og ved hjelp av differensligninger kan vi modellere en vidt spekter av fenomener. Fibonaccis kaninmodell er et enkelt eksempel på en biologisk populasjonsmodell, og det viser seg generelt at differensligninger egner seg godt til å beskrive utviklingen av antallet medlemmer i en populasjon av dyr, selv om ligningene vanligvis må være betydelig mer kompliserte enn (4.2) for å kunne være realistiske. To andre eksempler på fenomener som kan beskrives ved differensligninger er hvordan kapital akkumulerer seg på en rentebærende bankkonto, og hvordan beregningstiden for et dataprogram utvikler seg når problemstørrelsen øker (tenk for eksempel på hvordan beregningstiden øker med n for programmet som finner løsningen på problemet med Hanois tårn med n ringer i kapittel 3).

4.1.3 Egenskaper ved følger

I forskjellige sammenhenger kan vi være interessert i ulike egenskaper ved følger. I matematikk er vi ofte interessert i om en følge konvergerer, og det er utviklet mye teori omkring dette. Også ved mange numeriske beregninger er dette et helt sentralt spørsmål, og det er derfor viktig med gode numeriske kriterier for når en følge konvergerer. Vi vil komme tilbake til dette i et senere kapittel.

Hvis vi har en følge som består av tilfeldige tall er vi interessert i andre egenskaper ved følgen. Vi kan for eksempel være interessert i om alle tallene i følgen forekommer like ofte eller om det er spesielle mønstre som gjentar seg. Hvis vi observerer fysiske størrelser som temperatur og nedbør er vi interessert i gjennomsnittet (eller 'normalen'), mens hvis følgen representerer musikk er vi kanskje interessert i hvor mye 'bass' og 'diskant' det er i følgen.

4.2 Simulering av differensligninger

Som regel er den første utfordringen som møter oss i forbindelse med en differensligning å finne ut hvordan følgen oppfører seg. I læreboka har vi sett hvordan vi ut fra differensligningen kan finne en formel for løsningen, iallfall for andreordens lineære ligninger. Men med en datamaskin tilgjengelig kan vi også veldig enkelt generere følgen numerisk direkte fra differensligningen, selv i de tilfellene der vi ikke kan finne løsningen rent matematisk (og det kan vi ikke for de aller fleste typer ligninger).

Som et eksempel ser vi på Fibonacciligningen (4.2) som vi nå skriver

$$x_n = x_{n-1} + x_{n-2}, \qquad \text{for } n \geq 3 \tag{4.4}$$

med startverdiene $x_1 = x_2 = 1$. Denne formen er bedre egnet til implementasjon i et program enn (4.2) siden vi har x_n på venstre side av ligningen.

Anta for eksempel at vi ønsker å generere følgen $\{x_n\}$ fram til og med ledd nummer 20. Det kan vi gjøre med kodebiten

```
x[1]=1;
x[2]=1;
for (n=3; n<=20; n++) {
    x[n] = x[n-1] + x[n-2];
    println("x[" + n + "]=" + x[n]);
  }
```

Hvis vi koder dette i Java med de rette nøkkelordene på de rette stedene, kompilerer og kjører programmet, vil vi få skrevet ut 18 linjer i terminalvinduet som ser ut som

```
x[3]=2
x[4]=3
x[5]=5
      .
      .
      .
x[18]=2584
x[19]=4181
x[20]=6764
```

Disse verdiene kan sammenlignes med den matematiske løsningen av differensligningen som vi fra *Kalkulus* vet er gitt ved

$$x_n = \frac{\sqrt{5}}{5}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right).$$

Numerisk generering av følgen bestemt av differensligningen på denne måten kalles ofte *simulering* av differensligningen.

Programkoden over er verdt noen kommentarer. Vi legger merke til at det ikke er sagt noe om hvilken type variable `x`'ene skal være, og i et språk som Java kommer vi ikke unna å spesifisere det. Vanligvis vil vi måtte bruke flyttall (`float` eller `double` i Java) i slike numeriske beregninger siden følgen ofte vil bestå av reelle tall, men i dette tilfellet vil vi faktisk kunne bruke heltallsvariable (`int` eller `long`) siden vi ser fra differensligningen (4.4) at alle leddene i følgen vil være heltall.

Vi legger også merke til at vi har brukt en tabell (`array` på engelsk) `x[n]` for å lagre leddene i følgen. Dette er hendig fra et programmeringssynspunkt og framhever sammenhengen med følgenotasjonen $\{x_n\}$. På den annen side er dette ikke nødvendig hvis vi bare ønsker at programmet vårt skal skrive ut leddene i følgen. Når vi for eksempel regner ut `x[18]` bruker vi bare de to verdiene `x[16]` og `x[17]`, de tidligere verdiene er uten interesse. Når vi så beveger oss videre til `x[19]` trenger vi ikke lenger `x[16]`. Dette kan vi utnytte til å skrive om koden til

```
x=1;
y=1;
for (n=3; n<=20; n++) {
    z = x + y;
    println("x[" + n + "]=" + z);
    x = y; y = z:
  }
```

Når vi kommer inn i løkka og skal beregne x_n inneholder `x` verdien til x_{n-2} mens `y` inneholder verdien til x_{n-1}. Vi regner så ut x_n, lagrer resultatet i z ved hjelp av tilordningen `z = x + y`, og skriver ut resultatet. Deretter må vi forberede oss på neste gjennomløp i løkka og komme i samme situasjon som da vi kom inn, men med `n` en større. Det gjør vi ved å la `x` få verdien x_{n-1} som ligger i `y` og la `y` få verdien x_n som ligger i `z` (legg merke til at det er viktig å gjøre disse tilordningene i denne rekkefølgen). Så går vi tilbake til toppen av løkka, øker `n` med 1 og gjentar det hele. I tillegg ser vi at koden er riktig ved første gjennomløp av løkka. Ved å utvide dette argumentet litt har vi faktisk et induksjonsbevis for at kodebiten er korrekt og gjør det den skal.

I vårt tilfelle kan vi på denne måten spare lagerplass for 17 variable (de tre variablene `x`, `y` og `z` kontra tabellen `x[n]` med `n` mellom 1 of 20). Dette er ubetydelig på dagens maskiner, men det er et godt prinsipp å ikke bruke mer plass enn nødvendig. Hvis vi skulle beregne $x_{1000000}$ hadde lagerplassen som den store tabellen ville ta opp vært betydelig. Forøvrig bør det nevnes at det er mulig å beregne $\{x_n\}$ med bare to variable x og y, se oppgave 1.

La oss forsøke å simulere en annen differensligning, for eksempel ligningen

$$x_n = \frac{1}{2}(x_{n-1} + x_{n-2}) \qquad \text{med } x_0 = 2 \text{ og } x_1 = 1/2. \tag{4.5}$$

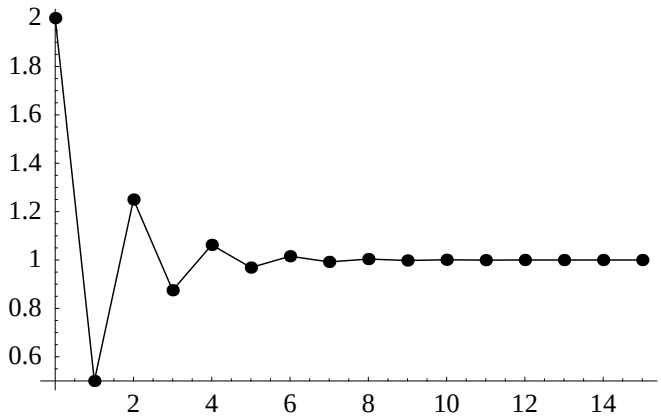
Denne ligningen er et spesialtilfelle av ligningen

$$x_n = \frac{1}{2}(x_{n-p} + x_{n-p-1})$$

som kan brukes til å generere gitarlignende lyder, se oppgave 8.

Siden ligning (4.5) har en koeffisient som er mindre enn 1 har vi ingen garanti for at leddene i følgen $\{x_n\}$ bare inneholder heltall. Hvis vi programmerer denne ligningen på samme måte som Fibonacciligningen bør vi derfor bruke flyttallsvariable. Gjør vi dette får vi generert tallene

```
x[0]=2
x[1]=0.5
x[2]=1.25
x[3]=0.875
     .
```



Figur 4.1. Plottet viser de 50 første leddene i følgen gitt ved differensligningen (4.5).

```
.
.
x[10]=1.0098
```

(Husk at tabeller i Java indekseres fra 0 så det å begynne med `x[0]` passer bra i Java.) Den eksakte løsningen kan vi finne ved å bruke framgangsmåten i *Kalkulus*. Resultatet blir

$$x_n = 1 + \left(-\frac{1}{2}\right)^n$$

og vi ser at de beregnede verdiene stemmer godt med disse eksakte verdiene. Løsningen skal altså konvergere mot 1 når n går mot uendelig. Beregner vi x_{30} og skriver ut verdien med 15 desimaler får vi

```
x[30]=1.000000000931323
```

som bekrefter konvergensen. Et plott av de første 50 leddene i følgen er vist i figur 4.1 og vi ser at verdiene konvergerer pent mot 1.

Vi fortsetter og ser på en tredje ligning, nemlig

$$x_n = \frac{10}{3}x_{n-1} - x_{n-2} \qquad \text{med } x_0 = 1 \text{ og } x_1 = 1/3. \tag{4.6}$$

Løser vi ligningen gitt ved (4.6) finner vi

$$x_n = \left(\frac{1}{3}\right)^n \tag{4.7}$$

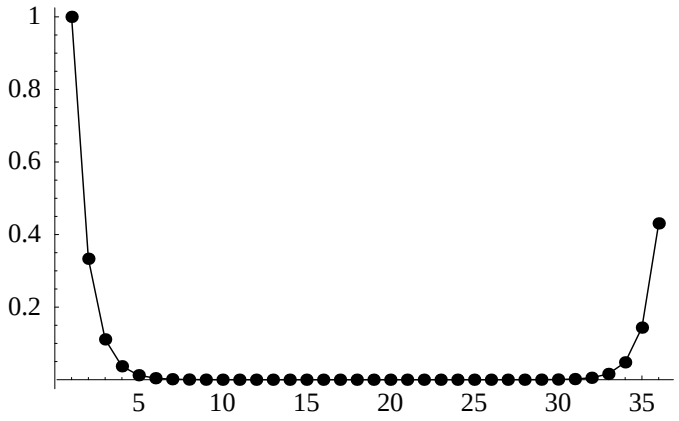
(eller vi kan vise ved induksjon at dette er løsningen). Denne ligningen kan programmeres på sammen måte som de andre to, og hvis vi kjører programmet og skriver ut resultatet med 10 desimaler får vi (vi bruker `double`-flyttall)

```
x[0]=1.0000000000
x[1]=0.3333333333
x[2]=0.1111111111
x[3]=0.0370370370
      .
      .
      .
x[20]=0.0000169351
```

Vi ser fra (4.7) at løsningen konvergerer mot 0 og de numeriske beregningene ser ut til å underbygge dette. La oss for ordens skyld sjekke et par verdier lenger ut i følgen. Hvis vi regner ut x_{100} og x_{500} numerisk får vi

```
x[100]=4.43755E30
x[500]=3.13073E221
```

(E-notasjonen er Javas måte å angi potenser av 10 på). Våre beregninger forteller oss altså at $x_{100} \approx 4 \cdot 10^{30}$ og at $x_{500} \approx 3 \cdot 10^{221}$ samtidig som vi vet at følgen er $x_n = (1/3)^n$ og konvergerer mot 0. Her er det åpenbart et eller annet som har gått svært galt!



Figur 4.2. Figuren viser de 35 første beregnede verdiene i følgen gitt ved (4.6). Punktene viser verdiene, som også har blitt forbundet med rette linjer.

Figur 4.2 viser et plott av verdiene slik de blir beregnet på datamaskin med verdiene av n langs den horisontale aksen. Vi ser at verdiene til å begynne med avtar pent mot null, slik vi også kan se fra de oppgitte verdiene over. Men så, når n passerer 30, begynner verdiene å vokse og verdiene vi regnet ut for x_{100} og x_{500} tyder på at de bare fortsetter å vokse når n øker. Årsaken til dette fenomenet er avrundingsfeil og forklaringen er forholdsvis enkel når vi vet hvordan vi kan finne en formel for løsningen til andreordens differensligninger.

Differensligningen gitt ved (4.6) har den karakteristiske ligningen

$$r^2 - \frac{10}{3}r + 1 = 0$$

som har røttene $r_1 = 1/3$ og $r_2 = 3$. Den generelle løsningen av (4.6) er derfor

$$x_n = C_1\Big(\frac{1}{3}\Big)^n + C_2 3^n \tag{4.8}$$

når vi ser bort fra de to startverdiene $x_0 = 1$ og $x_1 = 1/3$. For å få denne generelle løsningen til å stemme med disse startverdiene må vi velge $C_1 = 1$ og $C_2 = 0$ som gir løsningen (4.7). Hvis vi gjennomfører simuleringen av differensligningen med eksakte beregninger er det åpenbart denne løsningen som framkommer. Men som vi vet regner ikke datamaskiner helt nøyaktig med flyttall. Når vi regner ut $x_2 = 10x_1/3 - x_0$ får vi derfor ikke den eksakte verdien $1/9$, men noe som ligger nær dette tallet. Når beregningene fortsetter gjør maskinen stadig slike avrundingsfeil. Dette svarer til at vi har brukt en løsning av (4.6) på formen (4.8) der C_1 *ikke* er nøyaktig 1 og C_2 *ikke* er nøyaktig 0, altså en løsning

$$\hat{x}_n = (1 + \epsilon_1)\Big(\frac{1}{3}\Big)^n + \epsilon_2 3^n, \tag{4.9}$$

der både ϵ_1 og ϵ_2 er tall som er små i tallverdi. Problembarnet her er det siste leddet. Selv om ϵ_2 er liten vil den bli multiplisert med 3^n som kan bli vilkårlig stor for tilstrekkelig store verdier av n. Det siste leddet i (4.9) vil derfor etterhvert fullstendig dominere over det første leddet. Startverdiene vi bruker er $x_0 = 1$ og $x_1 = 1/3$, altså av størrelsesorden 1. Siden vi bruker 64 bits flyttall (`double` i Java) kan vi regne med omtrent 16 riktige desimale siffer i den første addisjonen, slik at både ϵ_1 og ϵ_2 vil være av størrelsesorden 10^{-17}. Det er omtrent for $n = 33$ at den beregnede følgen begynner å stige igjen så la oss forsøke å estimere $\hat{x}_{33}$ ut fra størrelsen på ϵ_1 og ϵ_2. Uttrykket for $\hat{x}_{33}$ er

$$\hat{x}_{33} = (1 + \epsilon_1)\Big(\frac{1}{3}\Big)^{33} + \epsilon_2 3^{33}. \tag{4.10}$$

Siden $3^{33} \approx 6 \cdot 10^{15}$ ser vi at det siste leddet er omtrent

$$\epsilon_2 3^{33} \approx 10^{-17} \cdot 6 \cdot 10^{15} = 0.06,$$

mens det første leddet i forhold til dette er så lite at vi kan overse det, $(1/3)^{33} \approx 10^{-16}$. Altså kan vi grovt regnet si at $\hat{x}_{33} \approx 0.06$. Fra figur 4.2 ser vi at den riktige verdien til $\hat{x}_{33}$ er omtrent 0.05 så estimatet vårt stemmer så nogenlunde.

I denne analysen har vi regnet som om det bare er beregningen av $\hat{x}_2$ som gir en avrundingsfeil, men i praksis vil vi få en liten avrundingsfeil for hvert ledd i følgen som vi beregner. Dette betyr at feilen vi får første gang nok er noe mindre

enn vi har regnet med her. Men for å få med oss feilene vi gjør i hvert steg på en enkel måte regner vi heller med litt større feil i det første steget.

I dette siste eksempelet ødelegger altså avrundingsfeil beregningene våre fullstendig etterhvert. Fra uttrykket for $\hat{x}_{33}$ i (4.10) ser vi at dette er uunngåelig hvis $\epsilon_2 \neq 0$. For uansett hvor liten ϵ_2 er, så vil før eller senere 3^n bli så stor at det andre leddet i (4.10) vil gi hovedbidraget til $\hat{x}_n$. Da vi studerte avrundingsfeil i kapittel 2 så vi at det store problemet er subtraksjon av to omtrent like store tall, men det er ikke det som er problemet her. Den generelle løsningen til ligningen (4.6) inneholder de to leddene $C_1(1/3)^n$ og $C_2 3^n$, og for store n vil alltid den andre løsningen dominere fullstendig over den første, bortsett fra i det ene tilfellet der C_2 er nøyaktig null. Men på grunn av avrundingsfeil vil vi aldri få C_2 eksakt lik 0 slik at det andre leddet alltid vil dominere for store verdier av n.

Legg merke til at dette fenomenet ikke er spesielt for denne ligningen. Hvis vi har en andreordens differensligning der den karakteristiske ligningen har to røtter r_1 og r_2 med $|r_1| < 1$ og $|r_2| > 1$, så vil alltid løsningen $C_2 r_2^n$ som stammer fra r_2 dominere for store verdier av n hvis $C_2 \neq 0$. På grunn av avrundingsfeil vil C_2 så og si aldri bli eksakt 0 hvilket betyr at vi så og si alltid vil få problemer med avrundingsfeil når vi simulerer slike ligninger numerisk.

Problemet er ikke forbeholdt lineære, andreordens ligninger. Hvis vi har en lineær, $m$te ordens ligning får vi tilsvarende problem hvis en av de m røttene er større enn 1 i tallverdi (selv om vi ikke kan finne eksakte formler for røttene kan vi alltid finne numeriske tilnærminger til røttene). Hvis ligningen ikke er lineær må vi også forvente tilsvarende problemer med avrundingsfeil, selv om vi da ikke har noen tilsvarende analysemetode, basert på røtter i en karakteristisk ligning.

Når vi ser problemene forbundet med numerisk simulering av (4.6), og vi samtidig har den eksplisitte formelen $x_n = (1/3)^n$ for løsningen, kan en lure på hva vitsen er med å gjøre den problematiske simuleringen? Poenget er at dersom vi modellerer et fenomen ved hjelp av differensligninger så vil som regel ligningene være så kompliserte at vi ikke har noen mulighet til å finne den eksplisitte løsningen. Den eneste muligheten for å se hvordan ligningene oppfører seg er derfor å gjøre numeriske simuleringer. Og siden vi kan få såpass dramatiske effekter av avrundingsfeil i en enkel ligning som (4.6) må vi være forberedt på lignende problemer i mer kompliserte ligninger også. Men uansett bør vi vite noe om størrelsen på feilen i simuleringene for å kunne stole på resultatene. Det er derfor utviklet analysemetoder som sier noe om feilen uten at vi vet hva den eksakte løsningen er.

4.3 Generering av pseudotilfeldige tall

I mange sammenhenger trenger vi å kunne trekke tilfeldige tall ved hjelp av en datamaskin. Hvis vi skal skrive et program som spiller Yatzy eller Ludo trenger vi åpenbart å kunne kaste terning, og mange mer avanserte spill har også behov for slik funksjonalitet. I mer seriøse sammenhenger er det også ofte bruk for tilfeldige

tall. Skal vi for eksempel skrive et program som skal simulere trafikken gjennom et lyskryss kan det kanskje være rimelig å anta at bilene ankommer krysset med tilfeldige mellomrom.

Støtten for å trekke tilfeldige tall varierer i ulike programmeringsspråk, men de fleste språk har i det minste en funksjon som gir tilfeldige flyttall mellom 0 og 1 (i Java heter denne funksjonen `random()` som ligger i klassen `java.lang.math`). I denne seksjonen skal vi se litt på hvordan det kan gjøres.

Vi merker oss først at hvis vi kan generere tilfeldige heltall mellom 0 og M, så kan disse regnes om til flyttall mellom 0 og 1 ved å dividere med M (så sant ikke M er for stor, da vil divisjonen bare gi 0). Problemet er dermed redusert til å generere tilfeldige heltall mellom 0 og M for et passende valgt heltall M, og for at vi skal kunne utnytte dette til å få mange flyttall mellom 0 og 1 bør M være et stort tall. Det fins flere metoder for å generere tilsynelatende tilfeldige heltall, men vi skal se litt på noen vanlige metoder som går under navnet *lineære kongruensgeneratorer*.

En lineær kongruensgenerator er bestemt av to heltall a og c, i tillegg til M. Det er ikke på noen måte likegyldig hvordan disse tallene velges, og det har blitt lagt ned mye arbeid i å finne gode verdier av a, c og M. Ett valg som har blitt foreslått er $a = 69069$, $c = 1$ og $M = 2^{32}$. Disse tre tallene bestemmer den enkle differensligningen (4.11) under , og genereringen av tilfeldige tall består i å simulere differensligningen numerisk: Vi velger først en heltallig startverdi x_0 mellom 0 og M som kalles frøet[1] ('seed' på engelsk). Vi lar så x_1 være resten vi får når $ax_0 + c$ divideres med M. Deretter dividerer vi $ax_1 + c$ med M, og lar x_2 være den resten vi nå får. Slik fortsetter vi inntil vi har fått generert alle de tilfeldige tallene vi trenger. Resten ved divisjon av to heltall m og n skrives ofte $m \bmod n$ i matematikk. Når startverdien x_0 er gitt kan derfor følgen av tilfeldige tall $\{x_n\}$ bestemmes ved rekursjonen

$$x_{n+1} = (ax_n + c) \bmod M \tag{4.11}$$

for $n = 0, 1, 2, \ldots$. (Husk at i Java er det operatoren `%` som angir resten ved heltallsdivisjon.) Tallene $x_1, x_2, x_3, \ldots$ er de tilfeldige heltallene mellom 0 og M som generatoren produserer, og etter divisjon med M får vi på denne måten fram tilfeldige flyttall mellom 0 og 1.

Vi ser av (4.11) at når a, c og M er gitt, så er følgen $\{x_n\}$ fullstendig bestemt ved x_0. De tilfeldige tallene som produseres av en lineær kongruens-generator er derfor ikke tilfeldige og kalles ofte *pseudotilfeldige* tall. Men pseudotilfeldige tall oppfører seg på en måte som gjør at vi i praksis ikke kan skille dem fra 'ordentlige' tilfeldige tall. Hva det egentlig betyr å trekke tilfeldige tall mellom 0 og 1, kommer

[1] Når vi kaller opp en funksjon som genererer tilfeldige tall kan vi som regel velge å oppgi frøet om vi ønsker det. Hvis vi ikke gjør det, vil generatoren selv velge frøet ved hjelp av klokka til datamaskinen.

vi nærmere inn på i et senere kapittel.[2].

Før eller siden vil differensligningen (4.11) gjenta seg selv. For hvis to tall er like, for eksempel x_m og x_{m+k}, så ser vi fra differensligningen at da vil også $x_{m+1} = x_{m+k+1}$, $x_{m+2} = x_{m+k+2}$, også videre. Hvis først x_m og x_{m+k} er like vil derfor den samme sekvensen gjenta seg med avstand k mellom like verdier; vi sier da at *perioden* er k. Det er dessuten klart at vi før eller siden må få tilbake et tall vi allerede har trukket. Tallene vi genererer er resten ved divisjon med M, og de eneste mulige verdiene for denne resten er heltallene fra 0 til $M-1$. Vi ser derfor at det er umulig å få en periode som er større enn M. Generelt vil perioden avhenge av a, c, M og frøet x_0, og det er klart ønskelig å velge a, c og M slik at generatoren får størst mulig periode uansett valg av frø. Til dette formålet fins det en velutviklet teori som gir kriterier som sikrer maksimal periode (uansett valg av frø).

4.4 Digital lyd

Svært mye av informasjonen som omgir oss i dag er lagret digitalt og blir overført digitalt. Vi har digital lyd på CD-plater, på internet og på kassettbånd, vi har digitale kameraer og digitale videokameraer, vi har digitale telefoner, vi har DVD-spillere der film er lagret digitalt, vi har bøker lagret digitalt på CD-rom plater, TV-signaler blir sendt digitalt via satelitt og så videre. I denne seksjonen skal vi se litt nærmere på hva ordet 'digitalt' betyr i sammenheng med lyd.

Fenomenet lyd er det vi oppfatter når lufttrykket ved trommehinnene i ørene varierer på bestemte måter. Nærmere bestemt må lufttrykket ossillere mellom 20 og 20 000 ganger i sekundet for at vi skal oppfatte ossillasjonene som lyd. Grensene på 20 og 20 000 er ytterpunktene — for de fleste ligger grensene litt over 20 og litt under 20 000. I forhold til det totale lufttrykket er svingningene vi oppfatter som lyd svært små, men øret er et følsomt organ som reagerer på mye mindre energimengder enn for eksempel øyet.

Mesteparten av lydene vi hører inneholder en ujevn blanding av mange ulike trykkvariasjoner slik at det er umulig å si at variasjonene ligger fast på for eksempel 1000 ossillasjoner i sekundet. Hvis ossillasjonene er jevne vil vi oppfatte lyden som om den ligger i en bestemt tonehøyde, slik som når et musikkinstrument spiller en enkelt tone. Det er antall ossillasjoner pr. sekund i slike 'rene' toner som må ligge innenfor 20 og 20 000, og vi refererer til antall ossillasjoner pr. sekund som *frekvensen* til tonen. Frekvens måles i Hz[3] slik at for eksempel en tonehøyde som svarer til 100 ossillasjoner pr. sekund angis som 100 Hz.

Det følger fra et grunnleggende resultat i en del av matematikken som kalles *Fourier-analyse* at enhver lyd kan skrives som en passende sum av 'rene' lyder

[2]De som er interessert kan lese mer om generering av tilfeldige tall og hvilke egenskaper vi ønsker at en generator skal ha, i kapittel 2 i boka *Stochastic simulation* av B. D. Ripley (Wiley, 1987).

[3]Uttales hertz etter den tyske fysikeren Heinrich Hertz (1857–1894).

med en veldefinert frekvens. Siden vårt øre bare kan oppfatte frekvenser mellom 20 Hz og 20 000 Hz er det nok å ta med rene lyder med en frekvens som ligger i dette hørbare området når vi spalter opp en lyd som en sum av rene lyder på denne måten.

For å lagre lyd må vi lagre størrelsen på trykkvariasjonene som den aktuelle lyden produserer. I utgangspunktet varierer lufttrykket kontinuerlig i tid slik at vi for hvert tidspunkt bør lagre hva lufttrykket skal være. Dette var tanken bak gamle vinylplater der trykkvariasjonene var kodet som små ujevnheter i vinylen som kunne plukkes opp av stiften på platespilleren.

På ei CD-plate lagres lyden *digitalt*. Dette betyr at størrelsen på ossillasjonene måles med jevne mellomrom og at hver av disse målingene lagres — vi sier at lyden *samples* og kaller målingene for *sampler*. Når lyden så skal spilles av igjen må avspillingsutstyret 'fylle inn' den informasjonen som skal ligge mellom samplene ved hjelp av en passende matematisk funksjon. For å få best mulig lydkvalitet er det viktig at tidsrommet mellom samplene er kort, og på ei CD-plate er det lagret 44 100 målinger pr. sekund, noe vi referer til ved å si at *samplingsraten* er 44100. Det viser seg at hvis vi bruker f sampler pr. sekund så kan vi ikke få med oss høyere lydfrekvenser enn $f/2$ Hz. For å være sikre på å få med oss frekvenser opp til 20 000 Hz må vi derfor ha minst 40 000 sampler pr. sekund, og med 44 100 sampler pr. sekund har vi da litt å gå på.

Det at lyd lagres digitalt innebærer også et annet aspekt, nemlig det at lydsamplene *kvantiseres*. I utgangspunktet burde vi egentlig lagre størrelsen på ossilasjonene med uendelige mange siffers nøyaktighet, men det lar seg selvsagt ikke gjøre. På CD-plater er hvert sample av lyden lagret med 16 bits (verdiene er lagret binært, som et tall i to-tallssystemet) hvilket betyr at de lagrede måleverdiene bare kan anta $2^{16} = 65536$ forskjellige verdier.[4] Typisk vil da verdiene kunne variere mellom -2^{15} og $2^{15} - 1$, og vi kan bruke datatypen `short` hvis vi vil arbeide med slike data i et Java-program. Vi må dessuten huske at lyden er lagret i stereo slik at hver gang musikken måles får vi 2 verdier som hver krever 16 bits. Dette betyr at det på en musikk-CD er lagret $2 \cdot 16 \cdot 44100 = 1411200$ bits pr. sekund som svarer til 176400 bytes pr. sekund (en byte er 8 bits). På en typisk CD med 1 times spilletid ligger det altså en informasjonsmengde på ca. 600 Mb (1 Mb er $1024 \cdot 1024$ bytes = 1048576 bytes). CD'er brukes også som lagringsmedium for andre typer informasjon enn musikk og da er kapasiteten 650 Mb.

Mer moderne lydformater utnytter forskjellige teknikker for å komprimere datamengden slik at musikk som opptar 600 Mb på en CD kan reduseres til så

[4]Det er 16 bits som er den effektive informasjonsmengden for hvert sample, men i tillegg er det for hver måleverdi lagret 33 bits med ekstra informasjon som brukes til *feilkorreksjon*. Ved hjelp av denne informasjonen kan CD-spilleren sjekke om den avleste måleverdien er riktig og eventuelt korrigere den. Dette gjør CD´er forholdsvis robuste overfor riper og fettmerker, men hvis mye av feilkorreksjonsinformasjonen også er ødelagt vil vi allikevel kunne få støy under avspilling.

lite som 50 Mb i MP3-format (vanlig format på Internett).

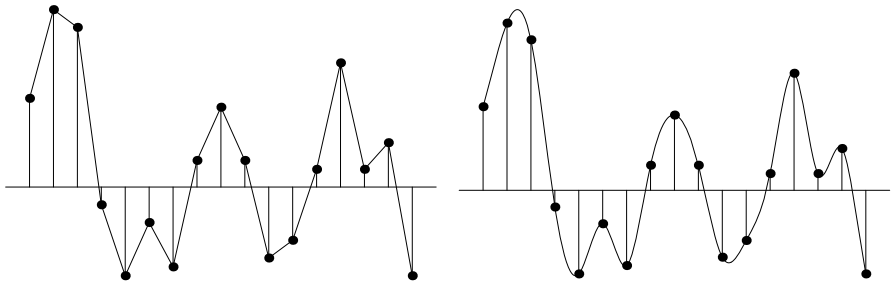
Til telefoni er det ikke bruk for den samme kvaliteten som til musikk, så både ISDN- og GSM-telefoner kommuniserer ved hjelp av digital lyd som er samplet 8000 ganger i sekundet. Dessuten brukes bare 8 bits til å representere hvert sample (egentlig 12 bit som komprimeres til 8). Til sammen gir dette en informasjonsstrøm på 64000 bits pr. sekund hver vei under en vanlig telefonsamtale. Med et vanlig ISDN-abonnement får vi tilgang på to slike linjer slik at vi kan overføre to samtaler samtidig eller 128000 bits pr. sekund. Har vi et ISDN-modem kan vi altså i beste fall overføre 128000 bits pr. sekund hvis vi bruker begge linjene samtidig.

4.4.1 Lyd på datamaskin

Ut fra vår diskusjon så langt ser vi at musikken på en CD kan betraktes som to endelige tallfølger $\{a_n\}$ og $\{b_n\}$, en følge for hver kanal (musikken er i stereo). For å illustrere de enkleste prinsippene bak digital lydbehandling er det tilstrekkelig å se på en kanal, så vi vil anta at et digitalt lydsignal er gitt ved en endelig følge av N tall $\{a_n\}_{n=1}^N$, og i denne sammenhengen refererer vi ofte til en følge som et *signal*. Hvis lyden er hentet fra en CD-plate kan det være naturlig å se på hver a_n som et 16 bits heltall i intervallet -2^{15} og $2^{15}-1$. Men ved å dividere alle tallene i følgen med 2^{15} vil vi få konvertert til flyttall som ligger i intervallet $[-1,1]$, noe som kan være mer hendig ved beregninger.

Veien fra en gitt tallfølge $\{a_n\}$ til lyd er kort. Så og si alle moderne datamaskiner er utstyrt med elektronikk som kan behandle lyd (et lydkort) og høytalere. Vi bør derfor velge et programmeringsspråk som gir adgang til lydkortet via en eller annen funksjon, la oss kalle den `Play`. Når vi kaller opp denne funksjonen med følgen $\{a_n\}$ som parameter, vil lydkortet omdanne følgen til et elektrisk signal som varierer på samme måte som følgen. Dette signalet sendes til maskinens høytalere der det setter høytalermembranene i svingninger svarende til signalet. Disse svingningene setter lufta i bevegelse med det resultat at vi hører den aktuelle lyden.

Denne beskrivelsen av veien fra følge til lyd får det hele til å høres ganske greit ut, men fra et matematisk synspunkt er det i allefall en stor utfordring. Følgen vår $\{a_n\}$ består av tall som angir styrken på lydsignalet ved en del tidspunkter som er adskilt med et fast tidsintervall. Har vi for eksempel en samplingsrate på 8000 er avstanden mellom hver verdi $1/8000s = 0.000125s$. På den annen side er lyd et fenomen som er kontinuerlig i tid slik at lufttrykket varierer hele tiden, og ikke bare ved isolerte tidspunkter. På en eller annen måte må vi derfor 'fylle inn' verdier for lydsignalet mellom de verdiene som følgen gir. Dette kan gjøres på mange måter, og kvaliteten på lyden vi hører i høytalerene er svært avhengig av hvor godt vi gjetter på hvordan lyden er mellom verdiene vi har fra $\{a_n\}$. På den annen side er det ikke så kritisk hvordan dette gjøres hvis vi har høy samplingsrate siden to naboverdier da som regel er ganske lite. I en slik situasjon



Figur 4.3. To forskjellige måter å konvertere et digitalt signal til et analogt. I (a) har vi trukket rette linjer mellom hver måleverdi, mens vi i (b) brukt tredjegrads polynomer mellom hver måleverdi.

vil de ulike metodene som regel gi omtrent samme svar. I figur 4.3 har vi vist to måter å fylle inn mellomliggende verdier på.

Kontinuerlige lydsignaler kalles ofte *analoge* signaler i motsetning til digitale signaler, og det å fylle inn mellomliggende verdier kalles ofte digital-til-analog konvertering eller *DA-konvertering*. Den motsatte prosessen, det å danne et digitalt signal fra et analogt, er noe enklere, og ikke overaskende kalles dette ofte *AD-konvertering* (analog-til-digital konvertering).

Som regel har en liten kontroll over DA-konverteringen, den er implementert i elektronikken og vi må ta det vi får. Det vi har til rådighet er en funksjon som `Play`, og ikke minst den generelle regnekraften i datamaskinen.

Før vi kan avspille lyden gitt ved følgen $\{a_n\}$ må vi bestemme oss for en samplingsrate. Hvis vi for eksempel har 16000 verdier kan dette svare til en lyd som varer i 2 sekunder hvis samplingsraten er 8000, mens lyden bare vil vare i 1 sekund hvis samplingsraten er 16000. Samplingsraten er en parameter som vi må gi til funksjonen `Play` og som vi har full kontroll over. Vi kan derfor spille av våre 16000 verdier med ulike samplingsrater. Hvis lyden er generert med en samplingsrate på 8000 vil det høres riktig ut om vi avspiller med den samme samplingsraten. Hvis vi derimot spiller av med en samplingsrate på 16000 når lyden ble generert med en samplingsrate på 8000 vil alle ossillasjoner bli dobbelt så raske som opprinnelig, slik at tonehøyden vil fordobles ved avspilling. En lyd med en frekvens på 1000 Hz vil derfor bli til en lyd på 2000 Hz. På samme måte vil frekvensen bli redusert når samplingsraten reduseres.

Som vi så tidligere er samplingsraten på CD´er 44100, mens den for digital telefoni er 8000. For eksperimenter er det som regel lurt å bruke en samplingsrate på 8000 så slipper vi å arbeide med så store datamengder.

4.4.2 Filtrering av lyd

La oss nå anta at vi har et lydsignal $\{a_n\}_{n=0}^{N-1}$ som vi har lagret i en tabell `a` av lengde N. Det som gjør kombinasjonen digital lyd og datamaskiner så spennende er at vi nå kan kombinere matematikk med datamaskinens regnekraft til å endre eller *filtrere* lyden. La oss for eksempel tenke oss at vi har fått tilgang på en digital utgave av et gammelt opptak med en artist som levde på første halvdel av 1900-tallet. Vi kan da forsøke å forbedre lydkvaliteten ved å fjerne noe av støyen som et slikt gammelt opptak uvegerlig vil være befengt med.

En annen utfordrende oppgave er å komprimere følgen slik at informasjonsmengden blir redusert til 1/12, slik som det gjøres i MP3-formatet, men uten at lydkvaliteten blir hørbart dårligere.

Her skal vi ikke forsøke oss på slike krevende oppgaver. I stedet skal vi se på noen enkle operasjoner vi kan gjøre med lyden.

Avspille lyden med forskjellige samplingsrater. Kanskje den enkleste måten å endre lyden på er å endre samplingsraten. Hvis vi har et lydsignal der samplingsraten var s ved opptak kan vi ved avspilling forsøke samplingsratene $s/2$, s og $2s$. Det vil endre en tone med frekvens f til en tone med frekvens lik henholdsvis $f/2$, f og $2f$.

Spille av lyden baklengs. Opp gjennom årene har det gått rykter om skjulte budskap på en del rockeplater. Ved å spille platen baklengs skal budskapene komme fram. Slikt er lett å sjekke hvis vi har en digital versjon av plata.

For å spille av lyden $\{a_n\}_{n=0}^{N-1}$ baklengs danner vi lyden $\{b_n\}_{n=0}^{N-1}$ der b_n er gitt ved

$$b_n = a_{N-1-n}, \qquad \text{for } n = 0,\, 1,\, \ldots,\, N-1.$$

Vi ser da at $b_0 = a_{N-1}$, $b_1 = a_{N-2}$ og så videre fram til $b_{N-2} = a_1$ og $b_{N-1} = a_0$. Lyden gitt ved følgen $\{b_n\}$ vil derfor være lyden gitt ved $\{a_n\}$ spilt baklengs.

Legge til støy. Vi skal ikke her gå inn på hvordan en kan fjerne støy fra en følge, men det å legge til støy er ikke så vanskelig. Nå er ikke støy noe entydig begrep, vi bruker det om all mulig uønsket lyd. Her mener vi med støy tilfeldig sus uten noen spesiell struktur. Det viser seg at denne typen støy kan vi få fram ved hjelp av tilfeldige tall.

La oss anta at vi har lydsignalet $\{a_n\}$, og at verdiene er flyttall i intervallet $[-1, 1]$. For å legge til støy kan vi legge et tilfeldig tall i intervallet $[-0.1, 0.1]$ til hver verdi a_n. Dette oppnår vi ved å danne en ny følge $\{b_n\}$ der b_n er gitt ved (i et Javalignende språk)

```
b[n]=a[n] + 0.2*(random()-0.5)
```

Siden `random()` gir et tilfeldig tall mellom 0 og 1 får vi et tilfeldig tall mellom -0.5 og 0.5 ved å trekke fra 0.5, og ved å multiplisere resultatet med 0.2 får vi et tilfeldig tall mellom -0.1 og 0.1. Vi kan selvsagt gjøre støyen sterkere eller svakere ved å endre faktoren 0.2 over.

Legge til ekko. Et ekko er bare en svakere kopi av den opprinnelige lyden med litt forsinkelse. Ved å variere tidsforsinkelsen kan vi få fram forskjellige effekter. Svært kort forsinkelse vil ikke oppfattes som ekko, men som en litt 'mykere' variant av den opprinnelige lyden, mens litt større forsinkelse (ca. 20 ms[5]) gir et naturlig ekko. Hvis vi skal måle forsinkelsen i millisekunder må vi kjenne samplingsraten. Som vi så over er tidsintervallet mellom to sampler 1.25 ms ved en samplingsrate på 8000. En forsinkelse på 25 ms svarer derfor til en forsinkelse på 20 tidsintervaller. Dette oppnår vi ved å danne et nytt lydsignal $\{b_n\}$ ved

$$b_n = a_n + da_{n-20}, \tag{4.12}$$

der d er en dempningsfaktor, for eksempel $d = 0.5$. Legg merke til at denne definisjonen av $\{b_n\}$ skaper problemer ved begynnelsen av følgen siden $n - 20$ er negativ når $n < 20$, og vi har antatt at signalet $\{a_n\}$ starter med a_0. Mer presist bør vi derfor definere $\{b_n\}$ ved

$$b_n = \begin{cases} a_n, & \text{for } 0 \le n \le 19, \\ a_n + da_{n-20}, & \text{for } 20 \le n \le N-1. \end{cases}$$

Hvis vi vil ha mer eller mindre forsinkelse kan vi erstatte 20 med et annet passende tall. Vi må også huske på at samplingsraten påvirker forsinkelsen. Med en samplingsrate på 44100 må forsinkelsen være omtrent 110 sampler for å få en tidsforsinkelse på omtrent 25 ms.

Ved å variere tidsforsinkelsen kan vi få fram ulike effekter. Vi kan for eksempel la forsinkelsen variere periodisk mellom 10 og 30 ved å beregne $\{b_n\}$ ved

$$b_n = a_n + da_{n-g_n}$$

der g_n er gitt ved

$$g_n = 10\big(2 + \sin(\mu n)\big). \tag{4.13}$$

Vær oppmerksom på at g_n vanligvis ikke blir noe heltall, så vi bør runde av høyresiden til det nærmeste heltallet. Faktoren 10 kan selvsagt endres etter behov, og som nevnt over bør den økes hvis samplingsraten økes.

Parameteren μ i (4.13) er en konstant som bør velges nokså liten, ellers blir variasjonene for voldsomme. Hvis vi ønsker å få inn 5 svingninger pr. sekund og

[5] 1 ms står for 1 millisekund som er det samme som 0.001 s.

samplingsraten er 8000 så må μn variere over et intervall på 10π når n varierer over et intervall på 8000. Altså må vi ha $8000\mu = 10\pi$ eller

$$\mu = \frac{10\pi}{8000}.$$

Mer generelt ser vi at hvis vi vil ha ν svingninger pr. sekund når samplingsraten er s må vi velge

$$\mu = \frac{2\pi\nu}{s}.$$

Dempe diskanten. Fra stereoanlegg er vi vant til å ha muligheten til å kunne justere bass og diskant i musikken. For å gjøre dette ordentlig trenger vi en presis definisjon av frekvensbegrepet og en måte å måle frekvensinnholdet i et signal. Det skal vi ikke forsøke oss på her, men vi kan få til slike effekter med en intuitiv tilnærming til problemet. Vi begynner med å se hvordan vi kan dempe de høye frekvensene og dermed markere bassen bedre.

De høye frekvensene kommer fra de raskeste svingningene i lydsignalet. Hvis vi derfor gjør en operasjon på signalet som gjør svingningene mindre så får vi redusert innholdet av høye frekvenser og dermed diskanten i lyden. Et slikt filter kalles et *lavpassfilter*. En enkel måte å redusere svingningene på er danne et nytt signal $\{b_n\}$ ved å ta *gjennomsnitt* av noen nabosampler. For eksempel kan vi definere $\{b_n\}$ ved

$$b_n = \frac{a_{n-1} + a_n + a_{n+1}}{3}.$$

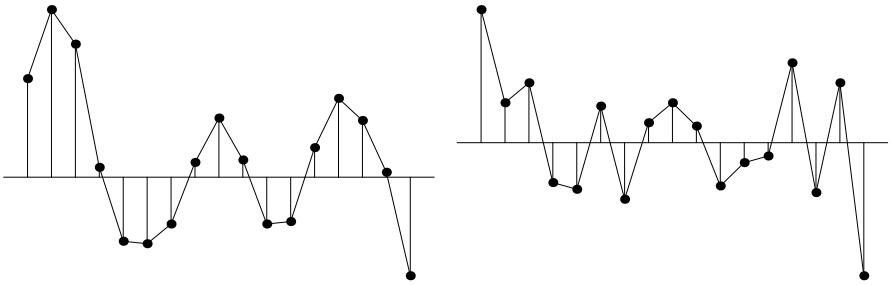
Som for ekko får vi problemer i endene av signalet, men det kan vi ordne med å bruke det opprinnelige signalet i endene,

$$b_n = \begin{cases} a_n, & \text{for } n = 0, \\ (a_{n-1} + a_n + a_{n+1})/3, & \text{for } 1 \le n \le N-2, \\ a_n, & \text{for } n = N-1. \end{cases}$$

Vi erstatter altså a_n med gjennomsnittet av a_n og de to naboverdiene og gir de tre verdiene like stor vekt i gjennomsnittet. Siden gjennomsnittet skal settes inn på plassen til a_n er det ikke urimelig å la a_n telle mer enn de to naboene. Det kan vi oppnå ved å bruke filteret

$$b_n = \begin{cases} a_n, & \text{for } n = 0, \\ (a_{n-1} + 2a_n + a_{n+1})/4, & \text{for } 1 \le n \le N-2, \\ a_n, & \text{for } n = N-1. \end{cases} \tag{4.14}$$

Effekten av dette på signalet i figur 4.3 er vist grafisk i figur 4.4, og vi ser ganske tydelig at kantene har blitt glattet ut litt.



(a) (b)

Figur 4.4. Glatting og framheving av kanter i et signal. I (a) har vi erstattet signalet i figur 4.3 med gjennomsnittssignalet gitt ved (4.14), mens vi i (b) har erstattet signalet i figur 4.3 med signalet gitt ved (4.15) som framhever kantene i signalet.

For å få en penere dempning av diskanten kan vi bruke lengre gjennomsnittsfiltre, for eksempel

$$b_n = \begin{cases} a_n, & \text{for } 0 \le n \le 1, \\ (a_{n-2} + 4a_{n-1} + 6a_n + 4a_{n+1} + a_{n+2})/16, & \text{for } 2 \le n \le N-3, \\ a_n, & \text{for } N-2 \le n \le N-1. \end{cases}$$

Legg merke til at koeffisientene foran a'ene er plukket ut fra rad 4 i Pascals trekant, mens tallet vi dividerer med er summen av disse koeffisientene. Det viser seg at det å plukke koeffisienter fra en rad i Pascals trekant med like nummer generelt er en god måte å dempe diskanten på.

Dempe bassen. På samme måte som vi kan dempe diskanten kan vi dempe bassen og dermed framheve diskanten. En enkel måte å gjøre dette på er å snu annenhvert fortegn i koeffisientene vi brukte ved glatting. Hvis vi endrer filteret i (4.14) på denne måten og anvender det på $\{a_n\}$ får vi det nye signalet $\{b_n\}$ gitt ved

$$b_n = \begin{cases} a_n, & \text{for } n = 0, \\ (-a_{n-1} + 2a_n - a_{n+1})/4, & \text{for } 1 \le n \le N-2, \\ a_{n-1}, & \text{for } n = N-1. \end{cases} \tag{4.15}$$

Figur 4.4 (b) viser resultatet av å gjøre denne operasjonen på signalet i figur 4.3. Vi ser at dette signalet er mer kantete enn både det opprinnelige signalet og det glattede signalet i figur 4.4 (a). Denne 'kantetheten' er det som representerer de høye frekvensene, eller diskanten, i det opprinnelige signalet. Et filter av denne typen, som bevarer de høye frekvensene, kalles et *høypassfilter*. Vi kan danne andre høypassfiltere ved å endre passende glattingsfiltere på samme måten.

4.4.3 Signalbehandling.

Her har vi skissert hvordan vi ved hjelp av litt matematikk kan gjøre operasjoner på lydsignaler. Slik filtrering er en del av feltet *signalbehandling*, og er en viktig del av grunnlaget for moderne teknologi. Selv om vi enkelt kan se prinsippene bak signalbehandling, er det å lage gode filtere som gjør akkurat det de skal en omfattende prosess som ofte involverer både matematikk og statistikk. I tillegg er det viktig med en god forståelse for hvordan vi oppfatter lyd. Ved for eksempel kompresjon av lyd utnyttes det faktum at dersom vi hører en lyd med en dominerende frekvens som samtidig inneholder en nærliggende frekvens som ikke er så kraftig, så registrerer ikke øret denne nabofrekvensen. Denne kan derfor fjernes fra signalet uten at vi hører nevneverdig forskjell, og det viser seg at det nye signalet kan lagres på en mer kompakt form en det opprinnelige.

Her har vi fokusert på lydsignaler, men det er svært mange typer signaler (følger) som kan behandles på tilsvarende måte. Noen eksempler er radiosignaler, radarsignaler, ultralyd og digitale bilder.

Oppgaver

4.1 a) Skriv et program som beregner de 50 første Fibonaccitallene $\{x_n\}_{n=1}^{50}$ definert ved (4.2). Bruk den siste av de to algoritmene som er skissert i teksten.

b) En liten utfordring: Klarer du å programmere Fibonaccitallene ved hjelp av algoritmen i (a), men med bare to variable `x` og `y` i tillegg til `n`.

4.2 I denne oppgaven skal vi se på Fibonacciligningen, men med litt andre startverdier enn de vanlige.

a) Finn den eksakte løsningen til Fibonacciligningen $x_{n+1} = x_n + x_{n-1}$ med startverdiene

$$x_0 = 1, \quad x_1 = \frac{1}{2}\big(1 - \sqrt{5}\big).$$

b) Beregn x_{100} numerisk ved å simulere ligningen på datamaskin, og sammenlign med den eksakte løsningen du fant i (a). Forklar resultatet.

4.3 a) Finn den andreordens lineære differensligningen som har karakteristisk ligning med røtter $r_1 = 1/2$ og $r_2 = 2$ og startverdier slik at løsningen blir $x_n = (1/2)^n$.

b) Simuler ligningen du fant i (a) numerisk og forklar resultatet.

4.4 Lag og test noen kongruensgeneratorer med $a = c = 1$ og forklar hvorfor dette valget ikke gir særlig 'tilfeldige' tall.

4.5 Den lineære kongruensgeneratoren gitt ved $a = 2^{16} + 3 = 65539$, $c = 0$ og $M = 2^{31}$ ble i sin tid brukt på noen kjente kommersielle datamaskiner. Det ble snart kjent at denne generatoren fungerte dårlig. Programmer generatoren og generer den avledede følgen

$$y_n = x_n - 6x_{n-1} + 9x_{n-2}.$$

Plott så y_n og tenk over hvorfor plottet sier noe om at følgen $\{x_n\}$ ikke er så tilfeldig.

4.6 Programmer generatoren av tilfeldige tall gitt ved (4.11) med $a = 69069$, $c = 1$ og $M = 2^{32}$. Velg selv x_0 og generer 100 tilfeldige tall.

4.7 Gjør eksperimenter med de ulike typene filtrering som er beskrevet i seksjon 4.4.2. Bruk både musikk, tale og evetuelt kunstige lyder i eksperimentene.

4.8 I denne oppgaven skal vi se på en enkel differensligning for å generere lyd som minner om et strengeinstrument. Metoden kalles Karplus-Strong algoritmen og den ble oppdaget i 1979 av en forsker, Kevin Karplus, og en student, Alexander Strong, på Stanford University i USA.

Metoden produserer særlig gode gitar-liknende lyder. Til å være så enkel og beregningseffektiv gir den utrolig god lyd. Differenseligningen er gitt ved

$$x_n - \frac{1}{2}(x_{n-p} + x_{n-p-1}) = 0. \tag{4.16}$$

Her er p et positivt heltall som bestemmer frekvensen til lyden. Hvis vi for eksempel bruker en samplingsrate på 44100 og en ønsker en tone med frekvens 440 Hz, må P være $P = 44100/440 = 100,22 \approx 100$. (På grunn av avrunding til heltall får vi altså en frekvens på 441 Hz steden for 440 Hz, men det fins en variant av algoritmen som forbedrer dette.) Vi legger ellers merke til at for $p = 1$ så er ligning (4.16) identisk med ligningen (4.5) som vi simulerte i seksjon 4.2.

Vi ser at differensligningen (4.16) er av $p + 1$te orden så vi trenger $p + 1$ startverdier x_0, x_1, x_2, ..., x_p. Når disse er gitt kan vi generere nye verdier fra formelen

$$x_n = \frac{1}{2}(x_{n-p} + x_{n-p-1}).$$

For å få fram ulike toner bruker vi forskjellige startverdier.

a) Velg en passende samplingsrate og bruk verdiene $x_0 = 1$ og $x_1 = x_2 = \cdots = x_p = 0$ som startverdier og generer lyden med forskjellig frekvens. Husk at frekvensen er gitt ved s/p der s er samplingsraten.

b) Bruk tilfeldige tall mellom $[-1, 1]$ som startverdier i steden og gjenta forsøket. Bruken av tilfeldige tall som startverdi har den fordelen at lyden blir mer realistisk siden den ikke er akkurat den samme hver gang.

c) Prøv andre startverdier som kan gi spennende lyder. Husk at startverdiene må ossilere for at det skal bli noe lyd.

d) Forsøk å endre differensligningen litt og se hvordan det påvirker lyden.

4.9 I denne oppgaven skal vi se litt på differensligninger der den karakteristiske ligningen har to komplekse røtter.

a) Gjør en numerisk simulering av ligningen

$$x_n = \frac{x_{n-1}}{2} - x_{n-2}, \qquad x_0 = 0,\, x_1 = 1.$$

Ser løsningen ut til å gå mot 0 eller ∞ når n blir stor eller forblir løsningen begrenset, men ulik 0?

Spill av lyden som løsningen representerer (generer nok verdier til ett sekund med lyd).

b) Verifiser at observasjonen du gjorde i (a) om hva som skjer med x_n når n blir stor faktisk er riktig.

c) La oss nå se på den generelle ligningen

$$x_n + bx_{n-1} + cx_{n-2} = 0 \tag{4.17}$$

der b og c er reelle tall, men vi antar at de er valgt slik at begge røttene i den karakteristiske ligningen er komplekse.

Finn et uttrykk for tallverdien og argumentet til røttene i den karakteristiske ligningen.

Som startverdier bruker vi i resten av oppgaven $x_0 = 0$ og $x_1 = 1$, slik som i (a).

d) Velg verdier av b og c slik at løsningen av (4.17) gir en lyd med fast volum og høy frekvens.

e) Velg verdier av b og c slik at løsningen av (4.17) gir en lyd med fast volum og lav frekvens.

f) Velg verdier av b og c slik at løsningen representerer en lyd med avtagende volum. Forsøk å få til både lav og høy frekvens.