KAPITTEL 5

Funksjoner og kontinuitet

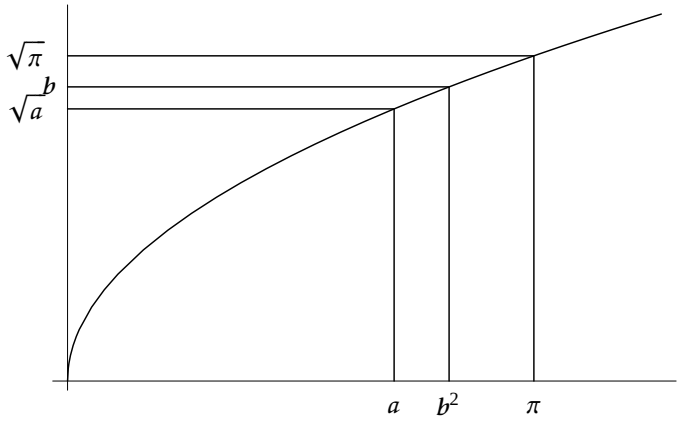
I dette kapitlet skal vi se litt på hva kontinuitet betyr for numeriske beregninger og plotting av funksjoner, og vi skal se at skjæringssetningen gir opphav til en enkel og robust metode for å finne numeriske tilnærminger til nullpunkter i funksjoner, nemlig halveringsmetoden. Vi skal dessuten se at de vanlige funksjonene fra skolen (særlig de trigonometriske) kan være nyttige for å generere ulike typer lyd, og at skalaene som vi kjenner fra musikk kan beskrives ved hjelp av litt enkel matematikk.

5.1 Kontinuitet og beregninger med flyttall

I kapittel 2 så vi at flyttallsberegninger som regel vil være befengt med avrundingsfeil, og vi så i kapittel 4 at dette kan få katastrofale følger for numerisk simulering av differensligninger. Store deler av matematikken gjør utstrakt brukt av funksjonsbegrepet, og programmeringsspråk har konstruksjoner som ofte kalles funksjoner og som har mye til felles med matematiske funksjoner. I denne seksjonen skal vi se hvilke konsekvenser avrundingsfeil kan få når vi bruker flyttall til å beregne verdien av funksjoner.

La oss anta at vi skal beregne tallet $\sqrt{\pi}$. En typisk lommeregner gir svaret 1.772453851. Nå vet vi at π er et irrasjonalt tall så hverken lommeregnere eller datamaskiner kan representere dette tallet eksakt. Det lommeregneren forsøker å regne ut er derfor ikke $\sqrt{\pi}$, men $\sqrt{a}$ der a er tallet som lommeregneren bruker som tilnærming til π. Dette er hva lommeregneren forsøker å regne ut, men siden $\sqrt{a}$ neppe vil være et flyttal vil det endelige svaret b være lommeregnerens tilnærming til $\sqrt{a}$. Dette tallet b ser vi er resultatet om vi tar kvadratroten av b^2. Vi kan derfor oppsummere det hele med å si at når vi forsøker å beregne $\sqrt{\pi}$ får vi et resultat b som svarer til at vi tar den eksakte kvadratroten til b^2. Med så mange tilnærminger er spørsmålet om vi kan stole på at b er en god tilnærming til $\sqrt{\pi}$.

For å presisere det vi har sagt, la oss nå anta at vi gjør beregningene på en datamaskin med 64 bits flyttall. Det å erstatte π med a er i utgangspunktet ikke



Figur 5.1. De forskjellige størrelsene som er involvert i å beregne $\sqrt{\pi}$.

dramatisk siden vi kan regne med at a er den beste tilnærmingen til π med 64 bits flyttall. Rundt regnet vil derfor de 16 første desimale sifrene i a og π være like. På samme måte vil det endelige svaret b være den beste tilnærmingen til $\sqrt{a}$ med 64 bits flyttall slik at vi kan regne med at de 16 første sifrene i disse to tallene også stemmer overens. Det kritiske punktet er derfor om $\sqrt{a}$ er en god tilnærming til $\sqrt{\pi}$. Det er her kontinuiteten kommer oss til unnsetning. Siden a ligger nær π og kvadratrotfunksjonen er kontinuerlig må $\sqrt{a}$ ligge nær $\sqrt{\pi}$.

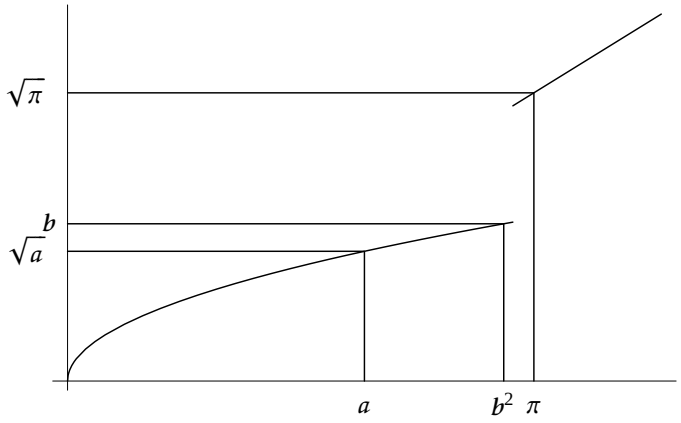
Situasjonen er illustrert i figur 5.1. På figuren er størrelsen på avrundingsfeilen betydelig overdrevet for å gjøre effekten tydelig. I følge figuren er tilnærmingen a til π mindre enn π, mens tilnærmingen b til $\sqrt{a}$ igjen er mindre enn $\sqrt{a}$ slik at b^2 er mindre enn a. De to feilkildene trekker derfor i samme retning i vårt eksempel, men i andre situasjoner kan de trekke hver sin vei.

Det kritiske punktet i dette eksempelet er altså om $\sqrt{a}$ er nær $\sqrt{\pi}$ når a er nær π, og siden kvadratrotfunksjonen er kontinuerlig ser vi at så er tilfelle. Dette blir enda tydeligere hvis vi gjør det samme forsøket med en diskontinuerlig funksjon.

I figur 5.2 illustrerer vi beregning av $f(\pi)$ for funksjonen f definert ved

$$f(x) = \begin{cases} \sqrt{x}, & \text{for } x < \pi, \\ 6 - x, & \text{for } x \geq \pi. \end{cases}$$

La oss igjen anta at vi skal beregne $f(\pi)$. Som i det foregående eksempelet antar vi at det flyttallet som ligger nærmest π er a, og vi antar som før at a er mindre enn π. På grunn av avrundingsfeil regner vi derfor ut $f(a)$ i steden for $f(\pi)$, og på grunn av nok en avrundingsfeil vil $f(a)$ bli rundet av til b. Som før består de to avrundingene bare i å erstatte et irrasjonalt tall med det nærmeste flyttallet, så feilen i disse prosessene er små. Men vi ser at selv om a og π ligger nær hverandre



Figur 5.2. Beregning av et punkt på en diskontinuerlig funksjon.

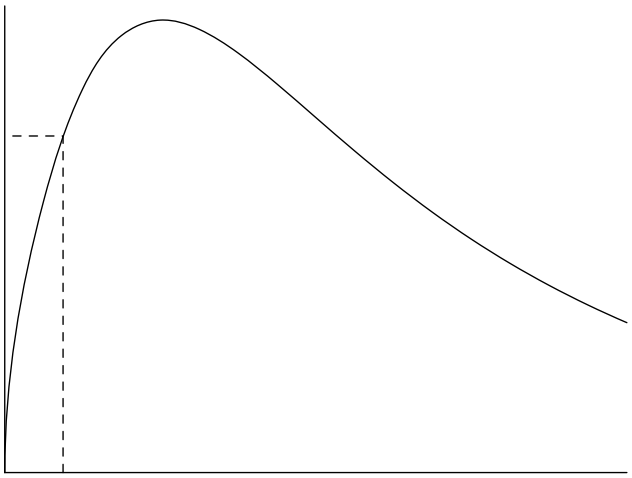
så er avstanden mellom $f(\pi)$ og $f(a)$ stor på grunn av diskontinuiteten som f har i punktet $x = \pi$.

Dette er et generelt problem med diskontinuerlige funksjoner. Når vi skal beregne en funksjonsverdi $f(x_0)$ nær diskontinuiteten risikerer vi at avrundingsfeil flytter oss fra den ene til andre siden av diskontinuiteten slik at vi i steden regner ut $f(\hat{x}_0)$. Differansen $f(x_0) - f(\hat{x}_0)$ vil da bli omtrent like stor som spranget f gjør i diskontinuiteten, uansett hvor liten avstanden er mellom x_0 og $\hat{x}_0$.

Hvis vi går tilbake til figur 5.1 så legger vi merke til at forskjellen mellom $\sqrt{\pi}$ og $\sqrt{a}$ faktisk er mindre enn forskjellen mellom π og a. Kvadratrotfunksjonen demper altså effekten av avrundingsfeilen som ble gjort da π ble erstattet med a. Dette er ikke noe uvanlig fenomen, men det motsatte kan like gjerne inntreffe, slik som i figur 5.3. I dette tilfellet ser vi at forskjellen mellom $f(p)$ og $f(a)$ er større enn forskjellen mellom p og a fordi f er ganske bratt i det aktuelle området. Hvis a er det flyttallet som ligger nærmest p ser vi derfor at feilen som gjøres ved å erstatte p med a blir forstørret opp når vi anvender f med det som konsekvens at forskjellen mellom $f(p)$ og $f(a)$ blir større enn forskjellen mellom p og a (i tillegg kommer effekten av å erstatte $f(a)$ med det nærmeste flyttallet).

Vi har altså sett at når vi skal beregne en funksjonsverdi $f(x)$ så er det hvor bratt funksjonen er i området rundt x som avgjør hvor stor avrundingsfeilen i $f(x)$ vil være. Hvis f er flat i nærheten av x vil avrundingsfeilen bli liten, mens hvis f er bratt blir avrundingsfeilen større. Det verste er hvis f er diskontinuerlig i x, da risikerer vi at feilen blir like stor som spranget i funksjonsverdi. Siden størrelsen på avrundingsfeilen er avhengig av hvor bratt funksjonen er kommer det kanskje ikke som noen overaskelse at den kan uttrykkes ved hjelp av den deriverte av f i nærheten av x. Dette skal vi se nærmere på i neste kapittel.

De funksjonene vi har sett på her har vært 'snille'. Selv den diskontinuerlige



Figur 5.3. Beregning av et punkt på en bratt funksjon.

funksjonen i figur 5.2 oppfører seg pent overalt bortsett fra i diskontinuitetspunktet. Heldigvis er de fleste funksjonene vi møter i praksis såpass pene. Men hvis alt vi krever av funksjonene våre er at de skal være kontinuerlige så kan de være betraktelig mye styggere enn det vi har sett her. Et eksempel er vist i begynnelsen av kapittel 5 i *Kalkulus*. Men selv for slike ville funksjoner er det altså sånn at hvis vi bare klarer å få x tilstrekkelig nær a så kan vi få $f(x)$ så nær $f(a)$ som vi måtte ønske. Problemet med flyttallsberegninger er at det er en nedre grense for hvor liten vi kan få avstanden $|x - a|$, og for slike stygge funksjoner som den i *Kalkulus* kan det godt tenkes at denne avstanden er for stor til at avstanden $\big|f(x) - f(a)\big|$ er en akseptabel avrundingsfeil. Konklusjonen er derfor at vi trenger kontinuitet for å kunne stole på våre flyttallsberegninger, men vi trenger mer enn det, nemlig 'pene' funksjoner. I praksis betyr ofte det at de laveste deriverte av funksjonen også er kontinuerlige.

Når dette er sagt bør det understrekes at diskontinuiteter som den i funksjonen i 5.2 i mange sammenhenger ikke skaper problemer. Senere i dette kapitlet skal vi for eksempel generere lyd fra en diskontinuerlig funksjon. Så lenge det er enkle diskontinuiteter i noen få punkter som vi kjenner går det meste bra, men kompliserte diskontinuiteter i punkter som vi ikke kjenner kan skape store problemer. Dette kan for eksempel være tilfelle hvis funksjonsverdiene ikke er gitt eksplisitt, men bare som løsning av en ligning.

5.2 Kontinuitet og plotting av funksjoner

Det visuelle bildet av en funksjon $f : A \mapsto \mathbb{R}$ som i matematikk kalles *grafen* til funksjonen, er definert som mengden av par i planet gitt ved

$$\big\{\big(x, f(x)\big) \mid x \in A\big\}.$$

Det er selvsagt umulig å eksplisitt angi alle disse punktene siden det vanligivis er uendelig mange av dem. Heldigvis er dette heller ikke nødvendig for å få et godt bilde av de funksjonene vi vanligvis møter.

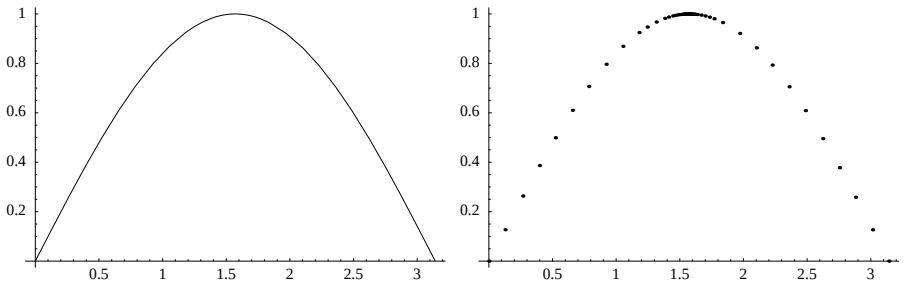
Det å skissere grafen til en funksjon kalles ofte *å plotte* funksjonen, og for å plotte en funksjon ved hjelp av papir og blyant er det vanlig å finne fram til en del viktige punkter så som funksjonens nullpunkter, dens ekstremalpunkter (punkter der den deriverte er 0) og vendepunkter (punkter der den andrederiverte er 0). Ut fra dette kan vi så lage en grov skisse av grafen. På en datamaskin fungerer ikke en slik metode så godt. For det første er det ikke alltid så lett å finne alle disse 'viktige' punktene, og for det andre er det ikke så lett å gi en presis oppskrift på hvordan punktene i mellom skal fylles inn.

5.2.1 Litt om grafikk

Før vi går videre må vi minne om hvordan bilder framstilles på en dataskjerm. En dataskjerm består av mange små punkter eller *piksler*[1], som er ordnet i et regulært, rektangulært mønster (kalles ofte *raster*). En vanlig konstellasjon er 1280 punkter i horisontal retning og 1024 punkter i vertikal retning, noe som gir et rektangulært mønster med tilsammen 1310720 piksler. Hvert av disse punktene kan så farvelegges slik vi måtte ønske. Det å lage en tegning på skjermen består derfor i å gi hvert slikt punkt riktig farve. Skrivere fungerer på samme måten, forskjellen er bare at punktene sitter mye tettere sammen enn på en skjerm (det vanlige er 5000–10000 punkter horisontalt og 7000–14000 vertikalt på et A4-ark). Når informasjon skal presenteres visuelt på en datamaskin skjer det altså punktvis eller *digitalt*, akkurat som med lyd.

På laveste nivå opererer datamaskinen med et heltallig koordinatsystem som reflekterer den rektangulære punktstrukturen. Origo er i øverste venstre hjørne, og x-koordinatene øker med 1 når vi beveger oss et piksel mot høyre, mens y-koordinatene øker med 1 når vi beveger oss et piksel nedover på skjermen. Heldigvis slipper vi vanligvis å bruke dette koordinatsystemet. I steden gir de fleste grafikkomgivelser programmereren muligheten til å definere sitt eget koordinatsystem som er tilpasset den spesifikke anvendelsen. Skal vi plotte funksjonen $\sin x$ når x varierer mellom 0 og π er det naturlig å ha et koordinatsystem der x varierer mellom 0 og π og y varierer mellom 0 og 1 (dersom x varierer over et større område bør vi ha et koordinatsystem der y varierer mellom -1 og 1). Bak

[1] Piksel kommer av det engelske *pixel* som igjen kommer av *picture element*.



(a) (b)

Figur 5.4. Plott av $\sin x$. Plottet i (a) er produsert ved å trekke rette linjer mellom punktene som er vist i (b).

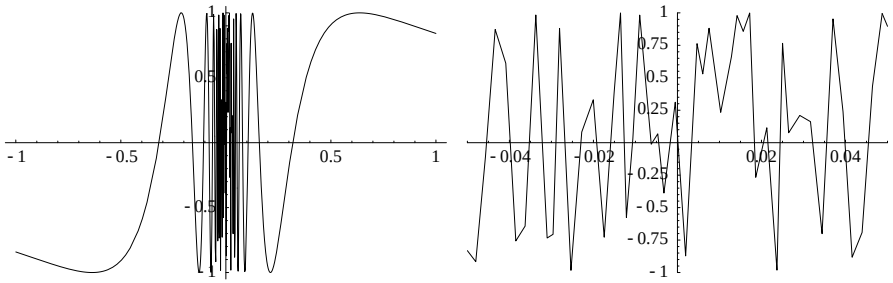
kulissene vil så passende programvare gjøre de konverteringene som er nødvendige for å oversette slike brukerkoordinater til heltallige skjermkoordinater.

Dersom vi skal trekke en rett linje mellom to punkter X og Y trenger vi ikke selv å avgjøre hvilke piksler som skal ha hvilken farve, dette blir vanligvis gjort av grafikksystemet på maskinen. Dette har den store fordelen at vi bare kan gi en kommando som `Line(X,Y)` og så vil linja bli tegnet på best mulig måte på det mediet vi arbeider med i øyeblikket. Det å angi bilder ved slike matematiske operasjoner (linjestykker angitt ved endepunkter, sirkler angitt ved sentrum og radius, også videre) kalles ofte vektorgrafikk, mens det å angi pikselverdier direkte kalles rastergrafikk. Fordelen med vektorgrafikk er at et slikt bilde lett kan forstørres eller roteres ved hjelp av enkel matematikk før det oversettes til et pikselmønster, mens når pikselmønsteret først er dannet blir slike operasjoner mer tidkrevende samtidig som resultatet ikke blir så bra. Dessuten opptar et bilde lagret som rastergrafikk stor lagerplass, mens et enkelt bilde i vektorgrafikk kan lagres svært kompakt (hvis bildet bare består av en rett linje er det nok å kjenne linjas endepunkter og koordinatsystemet som brukes).

5.2.2 Plotting av funksjoner

Ut fra hva vi nå vet om hvordan datamaskiner håndterer grafikk så ser vi at en mulig måte å plotte en funksjon f på er å la x gjennomløpe alle punktene i definisjonsområdet A og så for hver x markere det pikselet som ligger nærmest punktet $\big(x, f(x)\big)$. Problemet er at det vanligvis er uendelig mange tall i A, så dette er ikke direkte gjennomførbart i praksis.

Den vanlige plottemetoden består i å beregne et endelig antall punkter på grafen til f og så trekke en rett linje mellom hvert par av nabopunkter. For at dette skal fungere bra må funksjonen oppføre seg slik at det er rimelig å tilnærme den med en rett linje på små intervaller. Et eksempel er vist i figur 5.4. Til venstre



(a) (b)

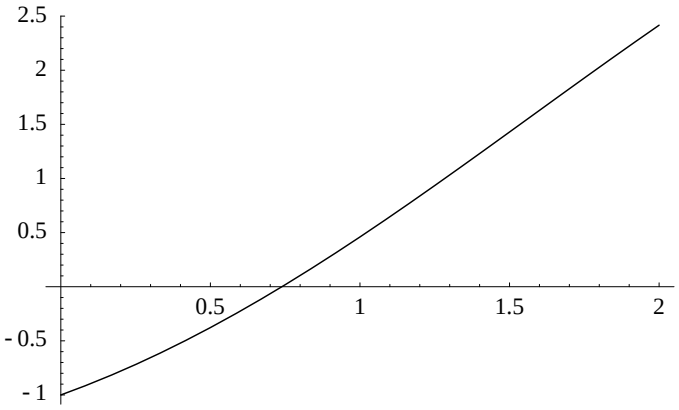
Figur 5.5. I (a) vises et plott av funksjonen $\sin(1/x)$ på intervallet $[-1, 1]$. I (b) vises et utsnitt av plottet i (a) tatt fra intervallet $[-0.05, 0.05]$.

ser vi et plott av $\sin x$ på intervallet $[0, \pi]$ (produsert av Mathematica), og til høyre er punktene som er brukt i plottet vist. Nær toppen, der sinusfunksjonen krummer mest, må vi ha mange punkter, mens vi ikke trenger så mange punkter på sidene der grafen er nesten rett.[2]

Siden denne plotteteknikken alltid vil gi en sammenhengende kurve er det en underliggende forutsetning at funksjonen er kontinuerlig, og helst bør grafen ligne en rett linje når vi 'zoomer' inn og ser på grafen i detalj. Hvis grafen ikke har denne oppførselen vil resultatet ved første øyekast kunne se bra ut, men dersom vi forstørrer opp bildet vil vi kunne se feilene. Et eksempel er vist i figur 5.5 der vi vedhjelp av Mathematica har plottet funksjonen $\sin(1/x)$ som har en problematisk diskontinuitet for $x = 0$. Plottet til venstre viser ikke særlig annet enn at det skjer noe dramatisk nær $x = 0$. Ved å se på et lite delområde av x-aksen rundt $x = 0$ kan vi se tydeligere hvordan Mathematica håndterer singulariteten. Vi ser der at punktene som Mathematica har valgt å bruke er nokså tilfeldige, men når vi betrakter funksjonen på hele intervallet $[-1, 1]$ gir altså de valgte punktene informasjon nok.

Når funksjoner studeres ved plotting må en altså alltid være klar over at det som vises på skjermen bare er en tilnærming til den underliggende matematiske funksjonen. For å produsere plottet i figur 5.4 (a) brukte Mathematica punktene i figur 5.4 (b). Hvis vi hadde en annen funksjon som var lik $\sin x$ i disse punktene, men beveget seg mye mellom punktene ville altså plottet kunne bli det samme

[2]Noen plotteprogrammer vil beregne punkter med en gitt, fast avstand, uansett hvor mye funksjonen som skal plottes krummer seg i ulike områder. Fordelen med dette er at vi ikke trenger å bekymre oss om hvilke punkter vi skal beregne, vi må bare velge avstanden mellom punktene tilstrekkelig liten til at resultatet blir bra. Ulempen er at det blir beregnet mange flere punkter enn det som faktisk er nødvendig. I en del sammenhenger er dette akseptabelt, mens hvis funksjonen er svært komplisert og tung å beregne vil en slik metode kunne bli for tidkrevende.



Figur 5.6. Funksjonen $f(x) = x - \cos x$ på intervallet $[0, 2]$.

selv om funksjonene var helt forskjellige.

5.3 Numerisk løsning av ligninger med halveringsmetoden

Det er bare de færreste ligninger som kan løses eksakt i den forstand at vi kan finne en eksplisitt formel for løsningen. Vi har en formel for løsningen til lineære ligninger og for annegradsligninger, og det fins formler for løsningen til tredje- og fjerdegradsligninger. I tillegg er det enkelte andre ligninger der vi kan finne eksplisitte formler for løsningen, men i de aller fleste tilfeller kan vi ikke finne noen slik formel.

Skjæringssetningen forteller oss at en kontinuerlig funksjon som har motsatt fortegn i de to endene av et intervall må være 0 i minst et punkt i det indre av intervallet. Den forteller oss altså ikke hva nullpunktet er, men garanterer at det *eksisterer* et nullpunkt. Et eksempel er vist i figur 5.6. Funksjonen er $f(x) = x - \cos x$ på intervallet $[0, 2]$, og vi ser at $f(0) < 0$ mens $f(2) > 0$. Siden f er kontinuerlig, sier skjæringssetningen at det fins et tall c som ligger mellom 0 og 2 med den egenskapen at $f(c) = 0$, og vi ser fra figuren at en slik c fins med $c \approx 0.75$.

Selv om vi ikke kan finne en formel for nullpunktet c kan vi finne en numerisk tilnærming til c. I de fleste anvendelser er dette faktisk å foretrekke framfor det å ha en eksplisitt, men komplisert formel for et nullpunkt. Halveringsmetoden er en metode som utnytter skjæringssetningen til å finne en slik numerisk tilnærming. Vi skal senere se på Newtons metode som også er en metode for å finne numersike tilnærminger til nullpunkter.

For å beskrive halveringsmetoden tenker vi oss at vi har en kontinuerlig funksjon f definert på et lukket intervall $[a, b]$ og at $f(a)$ og $f(b)$ har motsatt fortegn, slik som i figur 5.6. Skjæringssetningen forteller oss altså at da fins det et tall c mellom a og b slik at $f(c) = 0$. Utfordringen vår er å finne en god tilnærming

til c. Dette gjør vi på en indirekte måte. Vi finner midtpunktet $m_1 = (a+b)/2$ og regner ut funksjonsverdien $f(m_1)$. Hvis nå $f(m_1) = 0$ har vi funnet et nullpunkt så vi kan stoppe prosessen. Hvis ikke ser vi på fortegnet til $f(m_1)$. Hvis $f(m_1)$ har motsatt fortegn av $f(a)$ så vet vi fra skjæringssetningen at f må ha et nullpunkt mellom a og m_1. Vi setter derfor $a_2 = a$ og $b_2 = m_1$ og fortsetter prosessen med dette nye intervallet. Hvis derimot $f(m_1)$ har motsatt fortegn av $f(b)$ vet vi at f må ha et nullpunkt mellom m_1 og b så vi setter $a_2 = m_1$ og $b_2 = b$ og fortsetter med dette intervallet.

Når intervallet $[a_2, b_2]$ er bestemt beregner vi $m_2 = (a_2 + b_2)/2$ og regner ut $f(m_2)$. Hvis $f(m_2) = 0$ har vi funnet et nullpunkt og kan stoppe. Hvis $f(m_2) \neq 0$ så har enten $f(a_2)$ og $f(m_2)$ eller $f(m_2)$ og $f(b_2)$ motsatt fortegn slik at vi kan fortsette prosessen med et av intervallene $[a_2, m_2]$ og $[m_2, b_2]$. I praksis må vi passe oss så ikke vi blir stående å halvere intervaller i all evighet. Dette unngår vi ved å telle opp antall halveringer og stoppe når vi når et avtalt antall som vi kan kalle n. Alt dette kan vi lett formulere som en algoritme i et programmeringsspråk. I et Java-lignende språk vil det se ut som følger.

Algoritme 5.1 (Halveringsmetoden). *La f være en kontinuerlig funksjon definert på intervallet $[a, b]$ og la n være et naturlig tall som angir det maksimale antall halveringer av intervallet. Følgende kodebit vil produsere et tall m som er en tilnærming til et nullpunkt c for f:*

```
double a1, b1, m, fm;
int i;
a1 = a; b1 = b;
i = 0;
m = (a1+b1)/2;
fm = f(m);
while (i<n && fm != 0) {
  if (sign(fa*fm) < 0) b1 = m
  else
    a1 = m;
  m = (a1+b1)/2;
  fm = f(m);
  i++;
}
```

(Funksjonen `sign` *gir fortegnet til argumentet.) Etter at denne koden er utført vil m enten være et nullpunkt for f eller så vil avstanden mellom m og et nullpunkt være liten i den forstand at*

$$|c - m| \leq \frac{b-a}{2^n}. \tag{5.1}$$

I beskrivelsen av metoden over gjorde vi bruk av variablene a_1, a_2, ..., og b_1, b_2, ... I praksis trenger vi ikke ta vare på alle disse tallene, så når midtpunktet er

beregnet og vi vet hva det nye intervallet skal være kan vi trygt legge midtpunktet i `a1` eller `b1`, avhengig av fortegnet på `fm`. På denne måten vil alltid intervallet `[a1,b1]` være det minste intervallet vi har funnet som er slik at f har motsatt fortegn i endepunktene og dermed et nullpunkt i det indre av intervallet.

Strengt tatt trenger vi ikke en egen variabel til `fm` for å ta vare på funksjonsverdien $f(m)$, men det å beregne en funksjonsverdi tar tid så det er god programmeringspraksis å bare gjøre det en gang og lagre verdien i en variabel.

Feilestimatet i (5.1) kan se mystisk ut, men er ikke så overraskende. Hvis $n = 1$ så halveres intervallet bare en gang slik at når algoritmen er gjennomløpt vil m være midtpunktet i det opprinnelige intervallet $[a, b]$. Det tilfellet som da gir størst feil er om nullpunktet c ligger langt fra m, altså svært nær a eller b. Avstanden fra a eller b til m gir derfor en øvre grense for feilen. Altså har vi

$$|c - m| \leq b - m = m - a = (b - a)/2$$

når $n = 1$. Hver gang vi halverer intervallet vil også den maksimale feilen bli halvert slik at vi med totalt n halveringer ender opp med feilestimatet (5.1).

Hvis vi kaller det siste midtpunktet som blir generert av halveringsmetoden for m_n så kan vi skrive (5.1) som

$$|c - m_n| \leq \frac{b - a}{2^n}. \tag{5.2}$$

Ved å la n øke kan vi generere en følge av midtpunkter $\{m_n\}$, og vi ser fra denne ulikheten at en slik følge må konvergere mot c siden høyresiden går mot 0 når n går mot uendelig.

Vi kan utnytte estimatet (5.2) til å finne en tilnærming m_n til c slik at feilen garantert er mindre enn en toleranse som vi velger. Hvis toleransen er ϵ så ser vi at hvis vi velger n så stor at

$$\frac{b - a}{2^n} \leq \epsilon \tag{5.3}$$

så får vi automatisk fra (5.2) at $|c - m_n| \leq \epsilon$. Ulikheten (5.3) er ekvivalent med at

$$2^n \geq \frac{b - a}{\epsilon}.$$

Tar vi logaritmer på begge sider av denne ulikheten får vi

$$n \ln 2 \geq \ln(b - a) - \ln \epsilon$$

som gir

$$n \geq \frac{\ln(b - a) - \ln \epsilon}{\ln 2}. \tag{5.4}$$

Når intervallet $[a, b]$ og toleransen ϵ er gitt forteller denne ulikheten oss hvor stor vi må velge n for at feilen $|c - m_n|$ i halveringsmetoden skal bli mindre enn ϵ.

Som et eksempel kan vi tenke oss at vi skal finne nullpunktet i funksjonen $f(x) = x - \cos x$ i figur 5.6 med feil mindre enn 10^{-6} når vi starter med intervallet $[a, b] = [0, 2]$. Regner vi ut høyresiden i (5.4) i dette tilfellet så ser vi at den er omtrent 20.93. Vi velger derfor n som det minste heltallet som er større enn dette, altså $n = 21$. Dersom vi ønsker større nøyaktighet og velger $\epsilon = 10^{-15}$ blir høyresiden omtrent 50.82 slik at vi må velge $n = 51$ for å være sikre på at feilen ligger innenfor denne grensen.

Dette er en svært vanlig måte å kontrollere nøyaktigheten i numeriske beregninger på: Vi vet ikke eksakt hva feilen er, men vi har en øvre grense (som her er gitt ved (5.2)). For å få feilen mindre enn den gitte toleransen krever vi at den øvre grensen for feilen skal være mindre enn toleransen, og dette gjør oss i stand til å finne ut hvor stor n må være for at vi skal kunne garantere at feilen (egentlig den øvre grensen for feilen) skal være mindre enn toleransen. Ofte vil denne verdien på n være litt større enn det som faktisk er nødvendig, men det å finne den minst mulige n er som regel så krevende at det er bedre å bruke et enkelt estimat som gir en verdi som er litt i største laget.

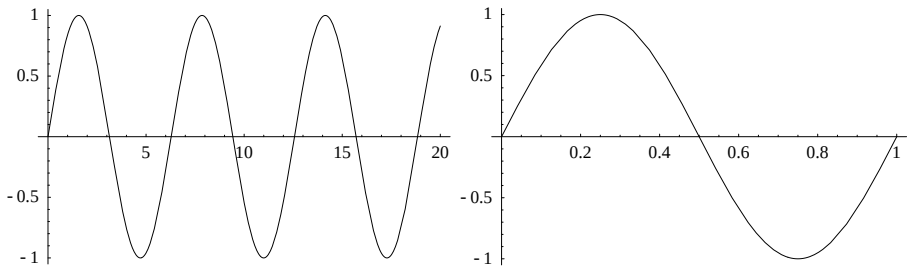
Halveringsmetoden er en svært robust metode for å finnne numeriske tilnærminger til nullpunkter siden alt vi trenger av antagelser er at f er kontinuerlig og har motsatt fortegn i de to endene av intervallet vi starter med. Under disse forutsetningene vil midtpunktene som metoden genererer alltid konvergere mot et nullpunkt for f. Det som ikke er så positivt med metoden er at den konvergerer forholdsvis sakte. Fra feilestimatet (5.1) ser vi at om vi øker n med 1 så vil feilen bli halvert. Dette svarer til at antall riktige binære siffer i m øker med 1 hver gang n økes med 1. Newtons metode, som vi skal se på i neste kapittel, konvergerer mye raskere enn dette. Når denne metoden konvergerer dobles antall riktige sifre hver gang n økes med 1. På den annen side er det andre problemer forbundet med Newtons metode.

Selv om halveringsmetoden er robust og alltid konvergerer mot et nullpunkt når forutsetningene er oppfylt vet vi ikke noe om *hvilket* nullpunkt metoden vil konvergere mot. Hvis vi på forhånd vet at intervallet $[a, b]$ bare inneholder ett nullpunkt vil metoden finne dette, men hvis intervallet inneholder flere nullpunkter kan vi ikke på noen enkel måte si hvilket av disse som metoden finner.

5.4 Lyd fra funksjoner

I kapittel 4 så vi på de grunnleggende prinsippene for digital lydbehandling, og i denne seksjonen skal vi se hvordan vi kan utnytte vanlige matematiske funksjoner til å generere lyd. Kontinuerlige lydsignaler (i motsetning til digitale signaler) kalles ofte *analoge signaler*, og det å behandle slike signaler kalles ofte *analog signalbehandling*.

Vi husker at en følge som ossilerer vil generere lyd når den avspilles. En måte å generere slike ossilerende følger er å plukke verdier fra en ossilerende funksjon, og prototypen på ossilerende funksjoner er $\sin x$ (eller $\cos x$). I figur 5.7 har vi vist



(a) (b)

Figur 5.7. I (a) vises funksjonen $\sin t$ på intervallet $[0, 20]$ mens (b) viser funksjonen $\sin 2\pi t$ på intervallet $[0, 1]$.

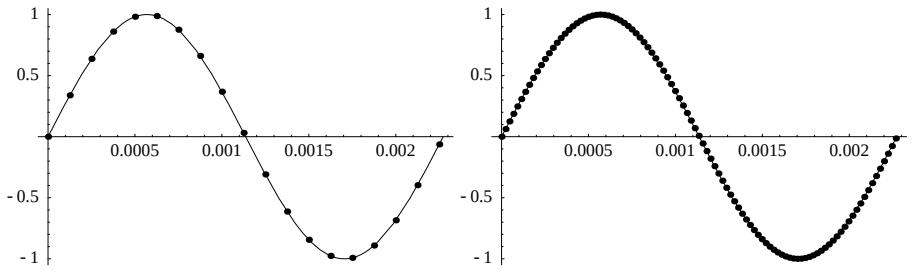
et utsnitt av $\sin x$ hentet fra intervallet $[0, 20]$ (husk at x er målt i radianer). Som vi vet ossilerer $\sin x$ jevnt mellom -1 og 1 og vi vet også at når vi har beveget oss over en intervallbredde på 2π vil $\sin x$ gjenta seg — den er en *periodisk* funksjon med *periode* 2π. I figur 5.7 ser vi at vi har fått med litt over 3 perioder av $\sin x$ siden $3 \cdot 2\pi \approx 18.9$ og vi har en intervallbredde på 20.

For å få lyd ut av en sinusfunksjon må vi ha kontroll på hvor mange ganger den ossilerer pr. sekund. Siden tiden nå skal løpe langs x-aksen erstatter vi x med t. Vi legger merke til at funksjonen $\sin 2\pi t$ inneholder en full periode av sinusfunksjonen når t varierer over intervallet $[0, 1]$, se figur 5.7 (b). Hvis vi betrakter denne funksjonen over intervallet $[0, 440]$ vil vi derfor få med oss 440 perioder av sinusfunksjonen. Disse ossilasjonene kan vi få inn på intervallet $[0, 1]$ hvis vi bruker funksjonen $\sin 2\pi 440t$. For når t nå varierer over intervallet $[0, 1]$ vil $440t$ variere over intervallet $[0, 440]$ slik at vi får med oss 440 perioder av sinusfunksjonen. Siden vi tenker oss at t måles i sekunder har vi nå en funksjon som ossilerer 440 ganger pr. sekund, så hvis vi kan omdanne dette til lyd burde vi kunne høre denne funksjonen siden vi oppfatter som lyd alt mellom 20 og 20000 ossilasjoner pr. sekund.

En datamaskin kan bare håndtere digital lyd så vi må lage en følge fra funksjonen vår. Men det er enkelt. La oss anta at vi bruker en samplingsrate på 8000. Da plukker vi bare 8000 jevnt fordelte verdier fra funksjonen $\sin 2\pi 440t$ og gir disse til `Play`-funksjonen i vår programmeringsomgivelse. Hvis vi kaller disse verdiene $\{a_i\}$ kan de beregnes ved

$$a_i = \sin(2\pi 440i/8000)$$

for $i = 0, 1, \ldots, 7999$. Når denne følgen avspilles med samplingsrate 8000 vil vi høre en lyd med frekvens 440 Hz. Denne tonen kalles *kammertonen* i musikk og har samme tonehøyde som summetonen i telefonen.



(a) (b)

Figur 5.8. I (a) vises en periode av funksjonen $\sin 2\pi 440t$ samplet med 8000 verdier pr. sekund, og i (b) samples det med 44100 verdier pr. sekund.

Hvis vi ønsker en høyere samplingsrate må vi plukke flere verdier fra hver periode. Med for eksempel en samplingsrate på 44100 må vi plukke 44100 verdier pr. sekund. I vårt tilfelle gir det en følge $\{\hat{a}_i\}$ der a_i er definert ved

$$\hat{a}_i = \sin(2\pi 440i/44100)$$

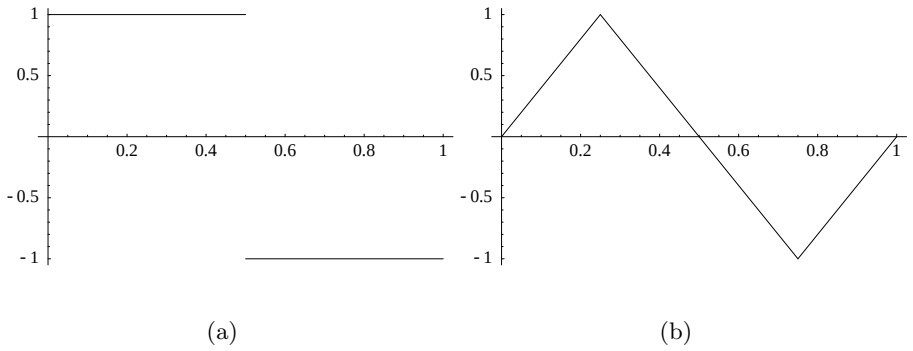
for $i = 0, 1, \ldots, 44099$. Effekten av å øke samplingsraten er altså at vi tar med flere verdier og på den måten får en bedre representasjon av funksjonen. I figur 5.8 har vi vist hvor tett samplene ligger på en periode av funksjonen når samplingsraten er 8000 (i (a)) og 44100 (i (b)).

Hvis vi mer generelt ønsker en lyd med frekvens f oppnår vi det ved å plukke verdier fra funksjonen $\sin 2\pi ft$. Som nevnt i kapittel 4 så er det slik at hvis vi bruker samplingsrate s får vi ikke med oss høyere frekvenser enn $s/2$. Frekvensen f bør derfor ikke overstige $s/2$, men det kan være morsomt å eksperimentere med å overstige denne grensen.

De trigonometriske funksjonene er prototypene på ossilerende funksjoner, og definisjonen på en 'ren' tone med frekvens f er den som genereres av funksjonen $\sin 2\pi ft$ (eller $\cos 2\pi ft$; begge genererer den samme lyden). Men det er andre funksjoner som også kan generere lyder med gitt frekvens, selv om vi da ikke får 'rene' toner. For eksempel kan vi bruke en 'firkantpuls' der en periode ser ut som i figur 5.9 (a). Denne funksjonen er definert ved

$$P_0(t) = \begin{cases} 1, & \text{når } 0 \le t < 1/2, \\ -1, & \text{når } 1/2 \le t < 1. \end{cases}$$

Funksjonen er altså diskontinuerlig, men det er ikke tvil om at den ossilerer en gang. For å lage en lyd med frekvens 440 Hz fra denne funksjonen bruker vi samme oppskrift som over og presser 440 perioder inn på et sekund. Det oppnår vi ved å



Figur 5.9. Plott av firkantpuls (a) og trekantpuls (b).

bruke funksjonen $P_0(440t)$. Spiller vi av denne er det ikke tvil om at den har en grunnfrekvens på 440 Hz, men i tillegg hører vi noe de fleste vil synes er ubehagelig lyd, på grensen til støy. Matematisk kommer dette av at diskontinuitetene gir funksjonen noen kraftige, ekstra frekvenser, i tillegg til grunntonen på 440 Hz.

I figur 5.9 (b) har vi en annen periodisk funksjon som kan brukes som byggekloss for å lage lyd. Denne kalles en trekantpuls og er gitt ved

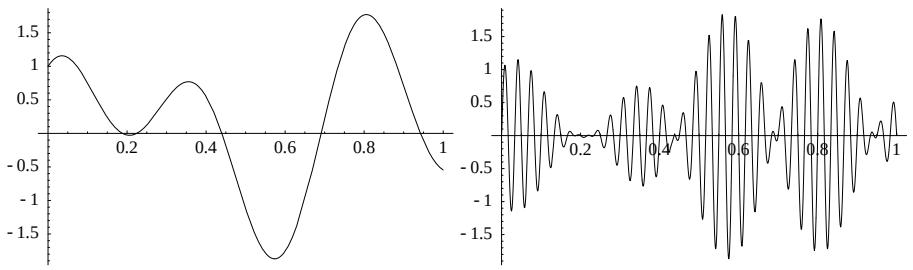
$$P_1(t) = \begin{cases} 4t, & \text{når } 0 \leq t < 1/4, \\ 2 - 4t, & \text{når } 1/4 \leq t < 3/4, \\ 4t - 4, & \text{når } 3/4 \leq t < 1. \end{cases}$$

Igjen kan vi danne en lyd med frekvens f ved å sample fra funksjonen $P_1(ft)$. Lyder generert fra P_1 er mer behagelige enn de som produseres fra P_0 siden vi ikke har diskontinuiteter. Men P_1 er ikke så glatt og pen som sinusfunksjonen (den deriverte av P_1 er diskontinuerlig) så den oppfattes av de fleste som noe skarpere.

Ved å gjenta andre funksjoner f ganger pr. sekund kan vi få fram andre lydkvaliteter med frekvens f og i prinsippet lyden fra alle mulige musikkinstrumenter. Men merk at dette er bare prinisipielt, i praksis vil det være svært vanskelig å modellere musikkinstrumenter godt på denne måten.

5.4.1 Modulasjon

De funksjonene vi har sett på så langt genererer en fast tone med en gitt frekvens, men med forskjellig kvalitet, omtrent slik forskjellige musikkinstrumenter kan produsere den samme tonen. For å generere lyd som inneholder informasjon må en bruke mer sofistikerte teknikker. De fleste bruker fremdeles analoge radioer der lydsignalene blir overført som kontinuerlige funksjoner. Utfordringen består i å få kodet et gitt lydsignal ved hjelp av kontinuerlige funksjoner slik at det kan



Figur 5.10. Funksjonen i (a) er i (b) multiplisert med $\sin 2\pi 30t$ for å illustrere amplitudemodulasjon.

overføres ved hjelp av elektromagnetiske bølger, siden slike bølger kan spres over store avstander før de plukkes opp av en antenne. (Det fungerer jo ikke så bra å sette en høytaler på en fjelltopp for å spre lyd!) FM-radio er basert på såkalt *frekvensmodulasjon* mens AM-radio er basert på *amplitudemodulasjon*.

Ved amplitudemodulasjon gis hver radiostasjon en fast sinusfunksjon med frekvens i området 150 kHz til 1500 kHz.[3] Dette svarer til senderfrekvensen, og det elektromagnetiske signalet som sendes ut har denne grunnfrekvensen. Et lydsignal som skal overføres legges inn ved å multiplisere denne grunnleggende sinusfunksjonen med en enkel variant av signalet som skal overføres. Hvis for eksempel lyden er gitt ved en funksjon $g(t)$ som sier hvordan lufttrykket skal variere og senderfrekvensen er 600 kHz vil signalet som sendes ut være

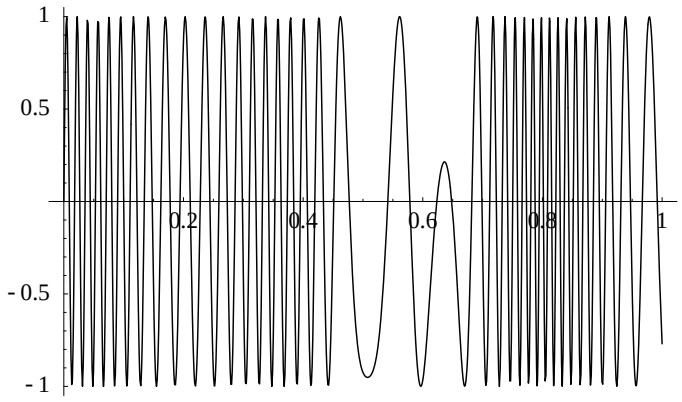
$$G(t) = C_1\big(1 + C_2 g(t)\big)\sin(2\pi 600000t), \tag{5.5}$$

der C_1 og C_2 er konstanter som tilpasses signalet og sendertypen. I utgangspunktet svinger sinusfunksjonen mellom -1 og 1 — vi sier at utslaget eller *amplituden* er 1. Vi ser at i forhold til dette er amplituden i funksjonen $G(t)$ i (5.5) justert med funksjonen $C_1\big(1 + C_2 g(t)\big)$ som involverer lydsignalet som skal overføres. Med andre ord vil amplituden variere med tiden og med $g(t)$ — det er dette som har gitt opphav til begrepet amplitudemodulasjon. For å plukke opp lydsignalet må mottageren inneholde elektronikk for å skille $g(t)$ fra $G(t)$.

Figur 5.10 (a) viser et plott av et lite utsnitt av et amplitudemodulert signal. For lettere å vise hva som skjer når en sinusfunksjon multipliseres med en annen funksjon er funksjonen som vises i (b) bare produktet av funksjonen i (a) og $\sin 2\pi 30t$.

FM-radio bruker et litt annet prinsipp for å overføre lyd. Igjen gis hver radiostasjon en fast frekvens ω, men nå i området 87.5 MHz til 108 MHz.[4] Lydsig-

[3] 1 kHz er det samme som 1000 Hz.
[4] 1 MHz er det samme som 10^6 Hz.



Figur 5.11. Funksjonen $g(t)$ fra figur 5.10 (a) kodet ved hjelp av frekvensmodulasjon.

nalet $g(t)$ overføres nå ved å overføre funksjonen

$$G(t) = A\sin\Big(2\pi\omega t + 2\pi k\int_0^t g(s)\,ds\Big), \tag{5.6}$$

der A og k er passende konstanter. Signalet $g(t)$ brukes altså til å justere frekvensen i steden for amplituden, derav navnet frekvensmodulasjon.

AM- og FM-modulasjon kan også brukes til å lage spennende lyder, og mange av lydeffektene i synthesizere er basert på modulasjon. Mye av dette kan vi også ganske enkelt få til på en datamaskin. For eksempel kan vi mulitplisere $\sin 2\pi 440t$ med funksjonen e^{-t} eller mer generelt e^{-at} der a er en positiv konstant for å dempe lyden. Lydsignalet

$$e^{-t}\sin 2\pi 440t$$

vil derfor være en ren tone med frekvens 440 Hz som dør ut med tiden.

Vi kan også danne lyd ved hjelp av frekvensmodulasjon. Funksjonen

$$\sin(2\pi 440 + 20\sin 5t) \tag{5.7}$$

gir en lyd med en frekvens som varierer rundt 440 Hz $\pm$20 Hz. Funksjonen

$$b(t) = e^{-\alpha t}\sin\big(2\pi f_c t + be^{-\beta t}\sin(2\pi f_m t)\big) \tag{5.8}$$

er basert på en kombinasjon av amplitude og frekvensmodulasjon. Ved å velge konstantene som $f_c = 80$ Hz, $f_m = 112$ Hz, $\alpha = 0.06$, $\beta = 0.09$ og $b = 4$ så framkommer en klokkelignende lyd. Når denne lyden avspilles bør tida løpe en stund, i allefall 10 sekunder og gjerne mer. Ved å eksperimentere med parametrene i (5.8) er det mulig å få fram et vidt spekter av lyder.

5.4.2 Musikkskalaer og matematikk

Vi har nå tilgjengelig funksjoner som gir oss toner med forskjellig frekvens, både rene toner og toner med litt 'skarpere kanter'. Hvordan kan disse kombineres for å produsere musikk?

I musikk opererer en ikke med toner med vilkårlig frekvens. For at musikken skal høres 'pen' ut må de ulike tonene ha frekvenser som står i et visst forhold til hverandre. Disse forholdene varierer innen ulike kulturer, men i vår vestlige verden er tolvtoneskalaen dominerende. Utgangspunktet for en slik skala er at når en tone svinger dobbelt så fort som en annen så høres de to tonene like ut selv om de er forskjellige — vi sier at de ligger en *oktav* fra hverandre. På et standard piano har den laveste tonen (A'en lengst til venstre) en frekvens på 27.5 Hz. Når vi beveger oss mot høyre ligger hver A en oktav over den foregående, og totalt er det på et standard piano åtte A´er med frekvenser (målt i Hz),

$$27.5,\ 55,\ 110,\ 220,\ 440,\ 880,\ 1760,\ 3520.$$

Den siste A'en er ikke den høyeste tonen på pianoet, det ligger 3 tangenter til høyre for denne slik at den høyeste tonen er en C. Tilsammen gir dette 88 tangenter som også er det vanlige på andre tangentinstrumenter av full størrelse. Mellom to A'er som er naboer ligger det 11 tangenter (både hvite og svarte). Hver av disse har igjen slektninger som ligger en oktav over eller under i frekvens. Innenfor en oktav er det derfor 12 forskjellige toner, og frekvensen til disse er jevnt fordelt i den forstand at forholdet i frekvens mellom to nabotangenter er konstant.

Hvis vi oversetter dette til matematikk så ser vi at de to funksjonene $\sin 2\pi ft$ og $\sin 2\pi 2ft$ skiller seg med en oktav. Vi har totalt 12 toner i en oktav, og den trettende tonen ligger altså en oktav over den første og har dobbel frekvens av denne. La oss betegne de 13 frekvensene med f_0, f_1, f_2, ..., f_{12}, der $f_{12} = 2f_0$. De mellomliggende tonene skal ha frekvenser som er jevnt fordelt mellom f_0 og f_{12} slik at forholdet mellom nabotoner skal være konstant. Hvis vi kaller dette forholdet c så skal vi altså ha

$$\begin{aligned}
f_1 &= cf_0, \\
f_2 &= cf_1 = c^2 f_0, \\
f_3 &= cf_2 = c^3 f_0, \\
&\vdots \\
f_i &= cf_{i-1} = c^i f_0, \\
&\vdots \\
f_{12} &= cf_{11} = c^{12} f_0.
\end{aligned}$$

Siden vi i tillegg har $f_{12} = 2f_0$ så ser vi fra den siste av disse ligningene at $c^{12} = 2$ eller $c = 2^{1/12} \approx 1.0594$. Tar vi utgangspunkt i A´en med frekvens 440 Hz kan vi

derfor representere alle tonene på pianoet ved funksjonene

$$\sin 2\pi 440 c^i t, \qquad i = -48, -47, \ldots, 38, 39.$$

Spesielt ser vi at tonene innenfor oktaven som går fra 220 Hz til 440 Hz har frekvensene

$$\begin{gathered} 220,\ 233.08,\ 246.94,\ 261.63,\ 277.18,\ 293.67, \\ 311.13,\ 329.63,\ 349.23,\ 369.99,\ 391.00,\ 415.31,\ 440. \end{gathered} \tag{5.9}$$

For å spille av en av disse tonene trenger vi bare sample funksjonen $\sin 2\pi f_i t$ der f_i er en av frekvensene over, og sende resultatet gjennom `Play`. Hvis vi ønsker en annen type lyd kan vi bruke funksjonene P_0 eller P_1 over, eller eventuelt en annen funksjon.

Frekvensene listet opp i (5.9) er alle tonene mellom de to A'ene, og ikke alle disse hører hjemme i tonearten A-dur. A-dur inneholder frekvensene (navnet på tonene over)

A	H	C$^\sharp$	D	E	F$^\sharp$	G$^\sharp$	A
f_0	f_2	f_4	f_5	f_7	f_9	f_{11}	f_{12}

Hvis disse frekvensene avspilles i rekkefølge er resultatet en pen A-dur skala. Vi har også A-moll skalaen som er gitt ved

A	H	C	D	E	F	G	A
f_0	f_2	f_3	f_5	f_7	f_8	f_{10}	f_{12}

(det fins også andre typer moll-skalaer).

Ved å begynne på andre frekvenser enn f_0, men bruke samme sprangfaktor mellom frekvensene, får vi fram de andre dur- og moll-skalaene. Begynner vi for eksempel på f_3 får vi henholdsvis C-dur og C-moll.

A-dur skalaen over er et eksempel på en *temperert* skala og er et kompromiss for å få det hele til å gå opp slik at vi for eksempel på et piano kan spille ikke bare i A-dur, men også i C-dur og F-dur. Frekvensene til den midterste oktaven i den tempererte A-dur skalaen er altså

A	H	C$^\sharp$	D	E	F$^\sharp$	G$^\sharp$	A
220	246.94	277.18	293.67	329.63	369.99	415.31	440

Nå fins det også andre skalaer der forholdet mellom frekvensene er basert på at det innen en oktav skal være tolv toner med et fast forhold mellom frekvensene. En *renstemt* dur-skala er, som navnet tilsier ingen kompromisskala, og er basert på at frekvensene til de 8 tonene i skalaen har følgende forhold til den første tonen,

$$1,\ \frac{9}{8},\ \frac{5}{4},\ \frac{4}{3},\ \frac{3}{2},\ \frac{5}{3},\ \frac{15}{8},\ 2.$$

Dette vil gi en A-dur skala med frekvenser

A	H	C$^\sharp$	D	E	F$^\sharp$	G$^\sharp$	A
220	247.5	275	293.33	330	366.67	412.5	440

Som vi ser er ikke forskjellen så stor mellom den renstemte og den tempererte skalaen, men forskjellen er mer enn stor nok til at to instrumenter som er stemt etter hver sin skala ikke vil lyde bra sammen.

Forholdene som danner grunnlaget for den renstemte skalaen har sitt utspring i den *pytagoreiske* skalaen som ble utviklet av pytagorerne. Den pytagoreiske skalaen ble senere justert litt av Ptolemeus og ble dermed til den renstemte skalaen over. Forholdene 1, 9/8, 4/3, 3/2 og 2 er felles for begge de to skalaene og er 'naturlige' i den forstand at disse tonene enkelt kan produseres på et enstrenget instrument.

Problemet med den renstemte skalaen er at den er 'helt riktig' i A-dur (eller den duren som renstemmes), men ikke så god i andre tonearter. Siden det tar såpass lang tid å stemme et piano er det vanligvis stemt temperert siden det gir et godt kompromiss for alle tonearter. Strykeinstrumenter kan derimot godt bruke renstemte skalaer siden de er raske å stemme og musikeren selv har kontroll på nøyaktig hvilken tonehøyde som skal brukes.

I musikk er det sjelden at enkelttoner lyder alene. Som regel settes flere toner sammen slik at vi får *harmonier* eller *akkorder*. Teorien for dette kalles *harmonilære* og denne læren kan også studeres ved hjelp av matematikk. Dette skal vi ikke gå inn på her, men vi skal vise hvordan vi kan generere en A-dur akkord. En slik akkord får vi ved å spille de tre tonene A, C$^\sharp$ og E samtidig. Hvis vi tar utgangspunkt i A´en med frekvens $f_0 = 220$ Hz skal vi altså spille de tre tonene med frekvens f_0, $c^4 f_0$ og $c^7 f_0$. Dette oppnår vi ved å avspille funksjonen

$$\sin 2\pi f_0 t + \sin 2\pi c^4 f_0 t + \sin 2\pi c^7 f_0 t.$$

Ved å variere f_0 får vi fram andre dur akkorder. Moll-akkorder får vi når den midterste frekvensen endres fra $c^4 f_0$ til $c^3 f_0$.

Oppgaver

5.1 a) Programmer halveringsmetoden slik den står beskrevet i algoritme 5.1. Test metoden på funksjonen $f(x) = x - \cos x$ på intervallet $[0, 2]$ og beregn nullpunktet med 10 riktige siffer. Finn en passende verdi for n ut fra ulikheten (5.4).

b) Finn en ligning som har $c = \sqrt{3}$ som nullpunkt og finn en tilnærming til c med 15 riktige siffer ved å anvende halveringsmetoden på denne ligningen.

c) Som (b), men med $c = 2^{1/12}$.

d) Som (b), men med $c = \pi$.

e) Som (b), men med $c = e$ (grunntallet for naturlige logaritmer).

5.2 a) Skriv et program som genererer en lyd på 440 Hz med utgangspunkt i de tre funksjonene P_0, P_1 og sin, slik som beskrevet i teksten over. (Funksjonene P_0 og P_1 kan programmeres ved hjelp av `if`-tester.)

b) Bruk samplingsrate 8000 og eksperimenter med å spille av lyder med frekvens i nærheten av 4000, altså halvparten av samplingsraten. Forsøk å danne deg et bilde av hva som skjer i det du passerer grensen på 4000.

c) Finn andre funksjoner enn P_0, P_1 og sin som kan brukes til å generere spennende lyder.

5.3 Ta utgangspunkt i funksjonen gitt ved (5.8) og varier parametrene slik at du får fram ulike 'pene' lyder.

5.4 a) Skriv et program som spiller av de 8 tonene i A-dur skalaen. Bruk en samplingsrate på 8000 og la hver tone vare i 0.4 s med en pause på 0.1 s mellom hver tone.

b) Skriv et program som lar f_0 gjennomløpe de 13 frekvensene svarende til tonene i oktaven fra fra 220 Hz opp til 440 Hz, og så for hver verdi av f_0 spiller de 8 tonene i den tilsvarende dur-skalaen.

c) Skriv et program som først spiller en A-dur akkord i den tempererte skalaen og deretter en A-dur akkord i den renstemte skalaen.

d) Skriv et program som først spiller en A-moll akkord i den tempererte skalaen og deretter en A-moll akkord i den renstemte skalaen.