Kompendium med oppgaver for MAT-INF 1100

Høsten 2004

Knut Mørken

6. desember 2004

Innhold

Forord

Sammen med *Kalkulus* av Tom Lindstrøm utgjør dette kompendiet læremidlene i kurset MAT-INF 1100, *Modellering og beregninger*, ved Universitetet i Oslo. Målet med kompendiet er å gi et perspektiv på kalkulus som er fokusert mot beregninger ved hjelp av datamaskin, og dessuten vise litt av hvordan matematikken som gjennomgås i MAT 1100 og MAT-INF 1100 har anvendelser i ulike fagfelt. Både dette kompendiet og undervisningen i MAT 1100 er basert på at studentene kan programmere fra før eller samtidig følger undervisningen i INF 1000 (grunnkurset i programmering ved Institutt for informatikk).

Deler av dette kompendiet har tidligere vært brukt i kurset MAT 100B, og i planleggingen av den første versjonen både av MAT 100B og dette kompendiet høsten 2000 hadde jeg nyttig hjelp av en referansegruppe bestående av Ørnulf Borgan, Sverre Holm, Hans-Petter Langtangen, Ole Christian Lingjærde og Harald Osnes. I tillegg til å være nyttige diskusjonspartnere bidro flere av disse også direkte til kompendiet. Sverre Holm bidro med ideer til deler av stoffet om lyd, og til dette fikk jeg også tips fra Hermann Lia. Seksjonen om stokastisk modellering og sammenhengen mellom integrasjon og sannsynlighetsregning i kapittel 7 ble skrevet av Ørnulf Borgan, og han er også opphavsmann til seksjon 4.3 om generering av tilfeldige tall. Hans Petter Langtangen leverte kjernematerialet til delen om fallskjermhopping i kapittel 8. I tillegg skrev Erik Bølviken hoveddelen av seksjonen om kapitalvekst i kapittel 6.

I MAT-INF 1100 er det plass til mer stoff om beregninger enn i det tidligere MAT 100B, og i høstens versjon av kompendiet kommer det et nytt kapittel om Taylorpolynomer og approksimasjoner med anvendelser innen kompresjon av lyd og representasjon av geometri.

Pål Hermunn Johansen har bidratt med utstrakt og meget kompetent programmeringshjelp, finansiert av Matematisk institutt. Mine nærmeste kolleger Geir Dahl, Tom Lyche og Ragnar Winther har lest de fleste kapitlene og gitt nyttige kommentarer og tips. Geir Ellingsrud, Arne B. Sletsjøe og særlig Tom Lindstrøm, som har undervist de andre MAT 100 variantene, har vært gode støttespillere sammen med administrasjonen ved Matematisk institutt. I administrasjonen vil jeg særlig berømme Helge Galdal som med imponerende iver dro i gang de nye kursvariantene høsten 2000 og Heidi Raude som tålmodig og velvillig har vært støtfanger mot studentene og organisator for våre stadig nye påfunn omkring evalueringsformer. Jeg vil berømme instituttet for satsingen på begynnerundervisningen, og takke for at jeg som utenforstående har fått være med på dette. Takk også til mitt eget institutt, Institutt for informatikk, som alltid har vært et godt hjem, og som meget velvillig har sluppet meg utenfor husets vegger.

Studentene som har fulgt undervisningen i MAT 100B har vært tolerante og overbærende med våre eksperimenter med undervisningsform og innhold. Selv om det sikkert er mange trykkfeil igjen ville det vært enda flere om ikke mange årvåkne studentøyne hadde saumfart teksten og rapportert feil til meg. Framfor alt har mange av dem vært entusiastiske og gitt positive tilbakemeldinger og derved gjort en stor utfordring til en morsom og trivelig oppgave.

Blindern, August 2003

KAPITTEL 1

Innledning

For mange er matematikk det faget som står lengst fra virkeligheten—en utbredt holdning er at matematikk ikke kan brukes til noe. Faktum er at dette er helt feil! Matematikk generelt, og spesielt hovedtemaet i MAT 1100 og MAT-INF 1100, kalkulus, har en mengde anvendelser. Faktisk er det slik at store deler av vårt moderne samfunn rett og slett er utenkelig uten matematikk. Mobiltelefoner og annet kommunikasjonsutstyr er basert på grunnleggende matematisk forståelse, og utvikling av datamaskiner er umulig uten utstrakt bruk av matematikk. På samme måte er store deler av moderne industri helt avhengig av matematikk.

Slike anvendelser av matematikk er videreføringen av tekststykkene som vi kjenner fra skolen. Disse uoppstilte oppgavene der en blir stilt overfor et mystisk problem som så skal løses ved hjelp av matematikk, har vært til glede og inspirasjon for noen, og til fortvilelse for andre. Vi blir stilt overfor et problem fra den virkelige verden, og utfordringen er å få oversatt problemet til matematikk i form av ligninger eller en annen passende *matematisk modell*. Når vi har fått fram den matematiske modellen er neste steg å løse det matematiske problemet. Til slutt må vi tolke løsningen i forhold til det opprinnelige problemet. Denne prosessen kalles *matematisk modellering*. Kort oppsummert består matematisk modellering av:

1. Oversett det gitte problemet til matematikk i form av en matematisk modell (ofte ligninger av et eller annet slag).

2. Løs det matematiske problemet.

3. Tolk løsningen av det matematiske problemet tilbake i rammene til det opprinnelige problemet.

Det er verdt å merke seg at programmering har mange likhetstrekk med matematisk modellering. Når vi skriver et dataprogram er utgangspunktet et 'virkelig' problem som skal løses med programmet vi skriver. For å komme fra problem til program må vi innføre datastrukturer og finne en strategi for å løse problemet. Vi må altså oversette fra det virkelige problemet til en idealisert verden som bare kan eksistere i datamaskinen. Ofte

vil vi måtte ty til en eller annen form for matematikk som hjelpemiddel i programmeringsprosessen. Når programmet er ferdig kan det utføres på en maskin og vi kan så tolke resultatene i forhold til det opprinnelige problemet.

Matematikkunnskaper er ofte nyttige når en skal programmere. På den annen side er programmering og datamaskiner ofte nyttige, for ikke å si essensielle, i det meste av matematisk modellering. Dette gjelder særlig i del 2 av modelleringsprosessen som er skissert over. Idag er det bare de aller enkleste matematiske modeller vi kan ha forhåpninger om å analysere og løse med 'papir og blyant'. Ettersom datamaskinene blir kraftigere blir også de matematiske modellene mer kompliserte og gir dermed arbeid til de stadig raskere datamaskinene. Men uansett hvor kraftig datamaskinen er må den programmeres for å kunne løse problemene. I mange tilfeller kan vi benytte standard programmer som andre har skrevet, men ofte vil ikke det bli raskt eller nøyaktig nok, eller det fins ingen programmer eller metoder som løser vårt problem. Utvikling av nye og bedre metoder for å løse ulike matematiske problemer, med tilhørende programvare, er derfor et fagfelt som beskjeftiger mange forskere.

Datamaskinen er også et viktig verktøy i del 3 av modelleringsprosessen. Metoden som løser det matematiske problemet gir ofte en løsning i form av en enorm samling tall. Store tallmengder er vanskelige for oss mennesker å tolke direkte, så i steden kan det være bedre å vise løsningen i form av bilder, lyd, video eller et annet format som passer for det aktuelle problemet som vi modellerer. Hvis vi for eksempel modellerer havbølger er det som regel bedre å få presentert løsningen i form av en video av de kunstige bølgene, enn alle tallene som representerer bølgene. Prosessen å oversette fra tall til bilder, lyd eller video krever igjen matematiske teknikker når det skal gjøres på en datamaskin.

Utviklingen av kraftigere datamaskiner betyr derfor ikke at matematikkstudier og matematikkunnskaper blir overflødige. Tvert i mot, bruken av datamaskiner gir nye anvendelser for matematikk, og det dukker stadig opp nye matematiske utfordringer både i modelleringsfasen, i behovet for nye løsningsmetoder og i tolkningsfasen. En godt kvalifisert fagperson er derfor fortrolig med matematikk og behersker datamaskinen som et naturlig verktøy, samtidig som hun har noe kunnskap om fagområdet som ga opphav til den matematiske modellen.

Det er nesten bare fantasien som setter grenser for hva som kan modelleres ved hjelp av matematikk. Matematiske modeller står sentralt i så ulike fag som biologi, fysikk, meteorologi, informatikk, medisin, økonomi, arkitektur, filmproduksjon, telefoni, musikk og landbruk, bare for å nevne noen. Noen vanlige grunner for å implementere matematiske modeller av fenomener på en datamaskin er:

- Ved å simulere den matematiske modellen på en datamaskin kan en sammenligne simuleringene med virkelige observasjoner for å undersøke om den underliggende teorien er riktig.

- I stedet for å gjøre virkelige eksperimenter kan eksperimentene utføres ved hjelp av en matematisk modell i en datamaskin. Hvis vi for eksempel har en matematisk modell av en oljeplattform og matematiske modeller for oppførselen til havet der plattformen skal plasseres, kan vi se om plattformen tåler de påkjenningene den vil

kunne bli utsatt for, allerede før den er bygget. På denne måten kan vi få hjelp til å dimensjonere plattformen riktig.

- Ved hjelp av matematikk kan vi representere informasjon i en datamaskin. I databasert animasjon og filmproduksjon brukes matematiske funksjoner til å representere både geometrien til figurene og banene til kameraene, og for å kunne lagre musikk, bilder og video mest mulig kompakt i en datamaskin lagres ikke 'rådata', men en passende matematisk tilnærming til originalen som kan beskrives med færre parametre (tall) enn den opprinnelige datamengden.

Når alt dette er sagt om anvendelser av matematikk bør det også sies at matematikk er et spennende og levende fag som står godt på egne ben og er en viktig del av vår kultur. Matematikken selv genererer nye matematiske modeller og gir opphav til utfordrende problemer, uavhengig av ulike anvendelser.

Resten av dette kompendiet består av syv kapitler. Kapittel 2 tar for seg hvordan tall representeres i datamaskiner. De fleste reelle tall kan bare representeres med tilnærminger. Dette gir opphav til avrundingsfeil, og bevissthet om hvordan numeriske beregninger innflueres av avrundingsfeil er et tema som går igjen i senere kapitler.

I kapittel 4 er hovedtema anvendelser av matematiske følger. Vi ser litt på numerisk simulering av diffensligninger, generering av tilfeldige tall og behandling av lyd på datamaskiner.

Hovedtema i kapittel 5 er funksjoner. Vi tar for oss definisjonen av kontinuitet og hva den sier om mulighetene for nøyaktig beregning og plotting av funksjoner. Dernest ser vi hvordan vi fra skjæringssetningen i matematikk får en naturlig måte for numerisk bestemmelse av nullpunkter for funksjoner, nemlig halveringsmetoden. Siste tema i kapittel 5 er hvordan vi kan lage lyd fra funksjoner og en liten smakebit på sammenhengen mellom musikk og matematikk.

I kapittel 6 kommer vi til derivasjon. Vi viser hvordan den deriverte av en funksjon gir et mål på funksjonens følsomhet for avrundingsfeil og viser hvordan Newtons metode kan implementeres. Kapitlet avsluttes med en seksjon om kapitalvekst. Vi starter med en beskrivelse basert på følger og en diskret måling av tid. Ved å la tidsstegene gå mot null får vi inn kontinuerlig tid og differensligningene blir til differensialligninger.

Neste tema er integrasjon i kapittel 7. Vi viser først hvordan et par metoder for numerisk integrasjon kan kodes effektivt og ser så litt på hvordan integrasjon er et naturlig verktøy innen sannsynlighetsregning.

Det siste kapitlet tar for seg differensialligninger. Første tema er implementasjon av Eulers metode for numerisk løsning av differensialligninger. Deretter ser vi på et konkret, fysisk problem, nemlig simulering av fallskjermhopping.

En sentral del av kompendiet er modelleringen, som delvis foregår i oppgavene. Modelleringen er konsentrert om fire hovedemner. I kapittel 4 og 5 er det lyd som modelleres ved hjelp av følger, differensligninger og funksjoner, og i kapittel 6 modelleres kapitalvekst ved hjelp av differens- og differensialligninger. I kapittel 8 er det modellering av naturlover som er tema, i dette tilfellet simulering av fallskjermhopping. Modelleringen i kapittel 7 er grunnleggende annerledes i og med at sannsynlighetsbegrepet er sentralt. Det konkrete eksempelet vi ser på er fordelingen av vekt for nyfødte.

KAPITTEL 2

Tall og datamaskiner

I dette kapitlet skal vi se på heltall og reelle tall og hvordan disse representeres på datamaskin. Heltall skaper ingen store problemer for datamaskiner annet enn at vi må begrense størrelsen på dem. Reelle tall skaper derimot er mer problematiske fordi vi ikke bare må begrense størrelsen, men også antall siffer. Dette kan gi opphav til avrundingsfeil som i ekstreme tilfeller kan gjøre at resultatet av numeriske beregninger blir fullstendig feil selv om beregningene rent matematisk er riktige. Vi diskuterer også to ulike måter å måle feil på, og avslutter med noen generelle kommentarer om objektorientert programmering og matematikk.

2.1 Naturlige, hele, rasjonale, reelle og komplekse tall

Fra *Kalkulus* vet vi at tallene kan deles inn i forskjellige klasser. Den minste klassen er de *naturlige tallene* som vi bruker når vi teller,

$$\mathbb{N} = \{1, 2, 3, 4, 5, 6, \ldots\}.$$

Tar vi med 0 og de negative tallene får vi mengden av *hele tall*,

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}.$$

Det neste steget kommer når vi begynner å dele opp ting i mindre deler, da trenger vi brøkene eller de *rasjonale tallene* som vi betegner med $\mathbb{Q}$. Dette er tall på formen

$$\frac{p}{q}$$

der p og q er hele tall med den begrensningen at q må være ulik 0. Setter vi $q = 1$ ser vi at alle de hele tallene er en delmengde av brøkene.

Som vi vet fra skolen, kan de rasjonale tallene skrives som desimaltall. For eksempel

er[1]

$$\begin{aligned} \frac{1}{2} &= 0.5, \\ \frac{3}{16} &= 0.1875, \\ \frac{97}{25} &= 3.88. \end{aligned} \tag{2.1}$$

For de fleste rasjonale tall får vi uendelig mange desimaler når vi skriver dem som desimaltall,

$$\begin{aligned} \frac{1}{3} &= 0.333333333\cdots \\ \frac{3}{7} &= 0.42857142857142\cdots \\ -\frac{10}{13} &= -0.7692307692307692307\cdots \\ \frac{11}{17} &= 0.64705882352941176470588235294117\cdots \end{aligned} \tag{2.2}$$

I eksemplene over må vi altså ha uendelig mange desimaler, men heldigvis er det et system, sifrene gjentar seg etter en stund. Dette gjelder generelt: Når rasjonale tall skrives som desimaltall får vi enten et endelig antall desimaler slik som i (2.1) eller en endelig sekvens av desimaler som gjentar seg uendelig mange ganger.

I dagliglivet kommer vi langt med de rasjonale tallene, men i matematikken trenger vi flere tall. For eksempel trenger vi tallet a som er slik at $a^2 = 2$, altså kvadratroten $a = \sqrt{2}$. Som vi skal se senere er dette tallet ikke rasjonalt, det sies derfor å være irrasjonalt. Det finnes uendelig mange irrasjonale tall, for eksempel er alle kvadratrøtter av naturlige tall irrasjonale dersom de ikke er heltallige, og e og π er også irrasjonale. Faktisk er det slik at det i en presis forstand fins flere irrasjonale tall enn rasjonale tall.

Dersom vi legger de rasjonale tallene til de irrasjonale tallene får vi de reelle tallene $\mathbb{R}$. Denne mengden inneholder altså både de naturlige tallene, de hele tallene, de rasjonale tallene og de irrasjonale tallene og fyller tallinja fullstendig slik at det ikke blir hull. *Kalkulus* inneholder mye stoff om de reelle tallene, og vi skal komme tilbake til noe av dette senere.

Den siste utvidelsen av tallene som vi skal studere er de *komplekse tallene* som vi vanligvis betegner med $\mathbb{C}$. Fordelen med å innføre disse tallene er at da kan vi løse ligningen $x^2 = -1$ og andre ligninger som krever at vi tar kvadratroten og mer generelle røtter av negative tall. Et helt kapittel i læreboka er tilegnet de komplekse tallene, men vi skal ikke si så mye om disse i dette kompendiet.

[1] Legg merke til at vi ikke bruker desimalkomma, men desimalpunktum. Desimalpunktum er vanlig i de fleste andre land og skaper ikke så lett forvirring når vi skriver matematikk. Skal vi liste opp de tre tallene 3,2, 2,8, 4,1 blir det lett uoversiktlig med desimalkomma. Med desimalpunktum blir det mye klarere, 3.2, 2.8, 4.1.

2.2 Datamaskiner og heltall

Datamaskiner ble før ofte kalt for regnemaskiner fordi de var bygd spesielt for å automatisere regning med tall. I dag brukes datamaskiner til å manipulere både tekst, lyd og bilder (og mye annet) i tillegg til tall, men på laveste nivå representeres allikevel alt som tall. La oss se litt overfladisk på hvordan dette gjøres.

2.2.1 Representasjon av heltall

På en datamaskin skiller vi mellom *heltall* (*integers* på engelsk) og *flyttall* (*floating point numbers* eller *floats* på engelsk). Heltallene på en datamaskin er de samme som de matematiske heltallene $\mathbb{Z}$ som vi nevnte over. Forskjellen er bare at det er umulig å få en datamaskin til å håndtere alle mulige heltall. Siden det er uendelig mange av dem ville det kreve uendelig med ressurser (lagerplass og regnekapasitet) å regne med alle mulige heltall. Vi må derfor sette en grense for hvor store heltall maskinen skal arbeide med.

For å forstå begrensningene som maskinen setter er det nyttig å vite hvordan heltallene representeres. Tall kan representeres i mange tallsystemer. Vi er vant til å bruke titallsystemet og sifrene 0—9 slik at når vi skriver 3489 så angir det tallet

$$3489 = 3 \cdot 10^3 + 4 \cdot 10^2 + 8 \cdot 10^1 + 9 \cdot 10^0,$$

men det er ingenting i veien for å bruke andre tallsystemer med andre grunntall enn 10. Bruker vi for eksempel grunntall 7, så har vi sifrene 0, 1, 2, 3, 4, 5 og 6 til rådighet og tallet 235_7 (grunntallet angis med indeks hvis det ikke er 10) er det samme som

$$235_7 = 2 \cdot 7^2 + 3 \cdot 7^1 + 5 \cdot 7^0 = 2 \cdot 49 + 3 \cdot 7 + 5 = 124.$$

Datamaskiner bruker som regel to-tallsystemet. Dette har den fordelen at vi bare trenger de to sifrene 0 og 1, de to *binære* sifrene. Dette er en fordel fordi det gir slingringsmonn når maskinen skal tolke hva den har lagret. For å lese et lagret tall må maskinen lese hvert av sifrene og da er det bare to muligheter: enten er det 0 eller så er det 1. Hvis sifferet i maskinen for eksempel er representert ved en spenning så kan vi tillate at alt mellom 0 volt og 2 volt skal tolkes som 0 mens alt mellom 4 volt og 6 volt skal tolkes som 1 (andre spenninger vil rapporteres som feil). Dersom vi opererte i et tallsystem med flere siffer ville vi ikke kunne bruke et så stort spenningsområde for hvert siffer og dermed ville vi bli mer utsatt for feil.[2]

[2]Dette er forøvrig en av de store fordelene med dagens *digitale* teknologi som omgir oss på alle kanter. Når musikk lagres digitalt betyr det at lydsignalet leses av med jevne mellomrom (44 100 ganger pr. sekund på en CD-plate) og lagres som et tall på binær form. Når dette skal leses ved avspilling har vi samme fordel som over, det skal bare avgjøres om det er 0 eller 1. En del fett og støv kan derfor tolereres på en CD-plate siden 'vi har en del å gå på´. På en gammeldags kassett derimot lagres lydsignalet ved å magnetisere kassettbåndet og det er intensiteten i magnetiseringen som sier hvor sterkt lydsignalet er, på en *analog* (kontinuerlig) skala. Litt fett eller andre forstyrrelser på båndet vil svært lett endre magnetiseringen og dermed det avleste lydsignalet, og dette vil høres som støy eller sus. Det samme forholdet gjør seg gjeldende når vi sammenligner digital (GSM og ISDN) og analog telefoni eller analog eller digital overføring av TV-signaler.

Datamaskiner representerer altså tall i to-tallsystemet og bruker kun sifrene 0 og 1. I dette tallsystemet skrives for eksempel det desimale tallet 13 som

$$1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13,$$

mens 121 binært blir 1111001_2. Dersom vi bruker 2 desimale siffer kan vi på en naturlig måte representere 100 heltall, nemlig tallene 0–99, og hvis vi bruker n desimale siffer kan vi representere 10^n tall (fra 0 opp til $10^n - 1$). På samme måte kan vi med n binære siffer representere 2^n heltall, nemlig tallene fra og med 0 opp til og med $2^n - 1$. Dagens datamaskiner har *maskinvare* (elektronikk) som håndterer 32 eller 64 binære siffer (ofte kalt *bits*) svært effektivt slik at programmer og programmeringsspråk som regel holder seg til dette. Dersom vi bruker 32 bits kan vi representere

$$2^{32} = 4294967296$$

forskjellige heltall. Siden vi skal ha både negative og positive tall fordeler vi disse tallene så jevnt vi kan på hver side av null og tillater heltallene mellom -2^{31} og $2^{31} - 1$, hvilket vil si tallene

$$-2147483648, \ldots, -3, -2, -1, 0, 1, 2, 3 \ldots, 2147483647.$$

Dette svarer til at vi bruker ett bit til å angi fortegnet til tallet og de resterende 31 bits til å angi tallverdien. Dessuten regner vi 0 til de positive tallene.

Hvis vi går opp til 64 binære siffer kan vi håndtere betydelig større heltall, nemlig tall fra -2^{63} opp til $2^{63} - 1$,

$$-9223372036854775808, \ldots, -2, -1, 0, 1, 2, \ldots, 9223372036854775807.$$

Det er støtte for disse to typene heltall i de fleste programmeringsspråk. I Java får vi tilgang på 32 bits heltall ved å benytte variabeltypen `int` mens 64 bits heltall er gitt ved variabeltypen `long`. I Java kan vi også benytte 8 bits heltall ved typen `char` og 16 bits heltall ved typen `short`. Beregninger med `char` og `short` tall foregår ved at de internt i maskinen blir betraktet som 32 bits heltall av typen `int`. Det er derfor ikke noe raskere å regne med disse mindre heltallstypene.

Det er verdt å merke seg at standardtypen for heltall i Java er `long`, altså 64 bits heltall. Når Java finner et heltallsuttrykk vil det derfor regnes ut som et 64 bits uttrykk.

I de fleste språk fins det predefinerte konstanter som inneholder største og minste heltall for en gitt heltallstype. I Java ligger disse konstantene i klassene `Long` (64 bits heltall) og `Integer` (32 bits heltall). Disse klassene inneholder også en del andre funksjoner som kan være nyttige ved håndtering av heltall.

2.2.2 Heltallsaritmetikk

I tillegg til å kunne lagre heltall i datamaskinen må vi også kunne regne med dem. Maskinen har egne elektroniske kretser som utfører aritmetikk med tall, og vi får tilgang til dette via et passende programmeringsspråk. I de fleste språk ligger *syntaksen* (notasjonen) for de vanlige aritmetiske operasjonene nær opptil det vi er vant til fra matematikkbøker, men i tillegg er det ofte støtte for andre operasjoner som det ikke fins

noen standard notasjon for i matematikk. I Java som i de fleste andre språk, betegner `+`, `-`, `*` og `/` henholdsvis addisjon, subtraksjon, multiplikasjon og divisjon, mens `a % b` gir resten når a divideres med b. En fullstendig liste over tillatte operasjoner for variable av typen `int` og `long` kan du finne i en lærebok for Java.

Det er essensielt bare en ting som kan gå galt ved behandling av heltall på en datamaskin og det er at vi støter på et tall som er for stort (i tallverdi) til å representeres i det formatet vi opererer i. Som vi har sett går grensen for 32-bits heltall ved tall som er mindre enn -2^{31} eller større enn $2^{31}-1$ mens den for 64-bits heltall går ved tall som er mindre enn -2^{63} eller større enn $2^{63}-1$. En typisk situasjon der vi vil kunne overskride formatet er ved addisjon av to store tall. Dersom vi bruker 64-bits heltall og forsøker å addere de to tallene $a = 2^{63}-2$ og $b = 2^{63}-7$ blir resultatet

$$a + b = 2^{63} - 2 + 2^{63} - 7 = 2 \cdot 2^{63} - 9 = 2^{64} - 9.$$

Tallet $2^{64}-9$ er større enn $2^{63}-1$ og kan derfor ikke representeres som et 64-bits heltall. I en slik situasjon vil maskinen bare overse noen siffer slik at resultatet kan virke nokså vilkårlig (denne oppførselen er definert av elektronikken i maskinen og ikke det aktuelle programmeringsspråket). En slik feilsituasjon kalles *overflow*, og vil alltid oppstå når resultatet av en operasjon på heltall resulterer i et tall med for stor tallverdi. I tillegg til ved addisjon kan dette også skje ved subtraksjon og multiplikasjon og ved tilordning av en for stor konstant til en variabel.

En annen feilsituasjon oppstår når vi forsøker å dividere noe med 0. Når dette skjer i beregninger som involverer heltallsvariable vil Java protestere og gi en såkalt `exception`.

2.2.3 Vilkårlig store heltall

Selv om vi med 64 bits heltallsvariable kan håndtere ganske store tall er det altså en øvre grense for størrelsen på tallene. Dersom maskinen ved en utregning ender opp med et tall som går utover de tillatte grensene får vi ingen feilmelding, men tallene som er for store blir gjort om til tall som ligger innenfor det lovlige området. Dette kan ofte være vanskelig å oppdage hvis vi ikke har klare tanker om hvordan resultatet av beregningene skal være. Det er derfor viktig å vite på forhånd at datatypen vi bruker (`int` eller `long`) er stor nok til å håndtere beregningene våre. Siden det i matematikkens verden finnes vilkårlig store heltall er det altså uendelig mange heltall vi ikke kan bruke på en fornuftig måte, selv med variable av typen `long` (64 bits). I de fleste sammenhenger er dette uten betydning siden det er svært sjelden vi har behov for å regne med heltall som i tallverdi er større enn 2^{63}. Men dersom behovet skulle melde seg så fins det en løsning.

Løsningen består i å skrive et program som håndterer store heltall. Det vanlige er å la størrelsen på tallene være helt fri, uten noen begrensninger. Når programmet møter et heltall, enten som inngangsverdi fra brukeren eller som resultatet av en beregning, vil det avsette plass nok til å representere hele tallet, uansett størrelse. For å utføre operasjoner på slike store tall må tilhørende aritmetiske algoritmer være implementert. På denne måten kan vi komme unna begrensningene som maskinvaren setter.

Ut fra dette kan man forledes til å tro at datatypene `long` og `int` som er definert ut fra maskinvaren er overflødige. Det er ikke tilfelle. Beregninger med store heltall som

krever egne programmer for aritmetiske operasjoner går svært, svært sakte i forhold til operasjoner på 'normale´ heltall som har aritmetiske algoritmer implementert i maskinvare. Dessuten skal man også være klar over at det ikke på noen datamaskin er mulig å representere alle mulige heltall. Selv om programmet som håndterer tallene ikke setter noen eksplisitt grense vil størrelsen på maskinens hukommelse og disk alltid sette begrensninger siden tallene må lagres internt i maskinen. Ved å øke disse ressursene kan vi arbeide med større heltall, men det vil aldri bli mulig å regne med uendelig mange siffer når vi har endelige ressurser. I tillegg vil det altså kunne ta svært lang tid å utføre beregninger med mange siffer.

Heldigvis har andre allerede gjort jobben med å implementere rutiner for store heltall. Programmer som Maple og Mathematica har innebygd støtte for slike tall og vil selv konvertere fra 64 bits til store heltall etter behov. I tradisjonelle programmeringsspråk er det vanligvis tilgang på biblioteker for å regne med store heltall. Java har klassen `java.math.BigInteger` som implementerer heltallsaritmetikk med vilkårlig mange siffer.

Når vi bruker en datatype som kan håndtere vilkårlig store heltall kan vi ikke få overflow siden det i prinsippet ikke er noen øvre grense for størrelsen på tallene vi kan håndtere. Men som nevnt over, setter de tilgjengelige maskinressursene (hukommelse og hastighet) begrensninger i praksis. Hastighetsgrensen skaper ikke større problemer enn at vi risikerer å måtte vente svært lenge på et resultat, men hukommelsegrensen er mer alvorlig. Dersom vi støter på et heltall som er så stort at programmet ikke har tilgang på nok hukommelse til å lagre tallet vil det måtte gi en feilmelding og beregningene må avbrytes.

2.3 Datamaskiner og reelle tall

For å kunne utnytte datamaskiner innen matematikk må maskinene kunne arbeide med reelle tall på en tilfredstillende måte, og dette gjøres vanligvis ved hjelp av *flyttall*. Flyttall og deres særegenheter er et omfattende tema som vi ikke skal gå inn på i detalj her, men det er nødvendig å kjenne de grunnleggende prinsippene for hvordan flyttall håndteres for å kunne forstå noen av problemene som kan oppstå i forbindelse med numeriske beregninger.

Mange har opplevd at lommeregneren ikke alltid regner helt riktig. For eksempel fins det en lommeregner (HP 48G) som regner ut $\sqrt{2}$ som

$$1.41421356237.$$

Kvadrerer vi dette tallet på den samme lommeregneren får vi ikke 2, men 1.99999999999 som svar. Feilen som lommeregneren gjør er svært liten, og hvis vi runder av til 11 siffer blir kvadratet 2, slik vi forventer. Problemet er at den aktuelle kalkulatoren bare regner med 12 siffer mens $\sqrt{2}$ er et irrasjonalt tall med uendelig mange desimaler. Når vi forsøker å representere $\sqrt{2}$ som et tall med 12 siffer (11 desimaler) gjør kalkulatoren derfor en liten feil, og når den tilnærmede kvadratroten multipliseres med seg selv trekker vi denne feilen med oss. I tillegg gjør også lommeregneren en liten feil i selve multiplikasjonen med den konsekvens at den forteller oss at kvadratroten av 2 opphøyd i annen ikke er nøyaktig 2. Dette fenomenet kalles *avrundingsfeil*.

Prosessen vi har skissert over gjelder også for moderne kalkulatorer og datamaskiner, selv om de fleste vil vise det riktige svaret 2 på regnestykket over. Dette kan skje fordi de regner med flere siffer enn de viser. Maskinen kan for eksempel regne med 16 siffer. Selv om det sekstende sifferet er feil vil svaret bli det riktige tallet 2 når det avrundes til 12 siffer. Men selv om svaret som maskinen viser er riktig i dette tilfellet er problemet med avrundingsfeil altså fremdeles til stede.

De fleste beregninger med desimaltall på kalkulatorer og datamaskiner er ikke helt nøyaktige og gir avrundingsfeil, og jo flere beregninger som er nødvendige før vi er framme ved det endelige resultatet dess større vil feilen i det endelige resultatet vanligvis være. Stort sett er feilene så små at det bare er de aller siste sifrene som er feil, men enkelte beregninger er så problematiske at en betydelig andel av de utregnede sifrene blir feil, ja vi skal senere se eksempler der svaret som datamaskinen produserer ikke inneholder noen riktige siffer i det hele tatt. Slike problematiske beregninger sies å være *dårlig kondisjonerte*, mens de som det kan påvises ikke gir særlig avrundingsfeil sies å være *godt kondisjonerte*.

Hvis det er mulig at alle sifrene i et beregnet tall kan bli feil er det åpenbart av stor betydning å vite når dette kan skje. Hvis en flyfabrikant for eksempel forsøker å estimere belastningene som vingene på et fly som er under planlegging kan bli utsatt for bør vi være sikre på at tallene som regnes ut er til å stole på, ellers kan følgene bli katastrofale. Slik analyse av beregninger med tanke på å påvise nøyaktigheten er en del av fagfeltet *numerisk analyse*.

Som nevnt over er studiet av avrundingsfeil for omfattende til at vi kan gå inn på alle detaljer her, men i resten av denne seksjonen skal vi se litt på hvordan reelle tall vanligvis representeres i en datamaskin og hvordan dette kan gi opphav til avrundingsfeil.

2.3.1 Reelle tall på normalisert form, flyttall

Siden en datamaskin har endelige ressurser kan vi ikke ha noen forhåpning om å kunne representere alle mulige reelle tall, spørsmålet er bare hvilke som skal tas med og hvilke som skal utelates. Utgangspunktet er at vi betrakter et reelt tall som et desimaltall som kan ha vilkårlig mange siffer både til venstre og høyre for desimalpunktum (merk at vi bruker punktum for desimalkomma i dette kompendiet). For å kunne representere slike tall i datamaskinen må vi gjøre to begrensninger. Som for heltall må vi begrense det totale antall siffer vi tillater i et tall. Dette begrenser nøyaktigheten vi kan operere med. Hvis vi for eksempel begrenser antall siffer til fire vil den beste representasjonen av π, $1/7$ og $100003/17$ være

$$\begin{aligned} \pi &\approx 3.142, \\ \frac{1}{7} &\approx 0.1429, \\ \frac{100003}{17} &\approx 5883, \end{aligned}$$

i følge de vanlige avrundingsreglene. Disse tallene kan vi greit runde av til fire siffer, men hva med større tall, for eksempel 58830? Vi ser at når tallene blir større enn eller lik 10000 så trenger vi fem siffer for å skrive dem, selv om vi egentlig bare er interessert i de

fire første sifrene. Poenget er at vi må vite størrelsen på tallet, altså hvor desimalpunktet skal være i forhold til de fire sifrene. Og her kommer den andre begrensningen inn; vi må sette en grense for hvor langt desimalpunktet kan være fra de fire sifrene. En grei måte å ordne dette på er å skrive tallene på en standardisert form, og det viser seg at formatet

$$\begin{aligned} 3.142 &= 0.3142 \cdot 10^1, \\ 0.1429 &= 0.1429 \cdot 10^0, \\ 5883 &= 0.5883 \cdot 10^4, \\ 0.003581 &= 0.3581 \cdot 10^{-3} \end{aligned}$$

er hendig. Hvis det gitte tallet er a og er ulik 0, omskriver vi det altså som

$$a = b \cdot 10^n, \tag{2.3}$$

der $|b|$ ligger i intervallet

$$0.1 \leq |b| < 1, \tag{2.4}$$

og n er heltallet som gjør at denne likheten holder (legg merke til at det bare er en verdi av n som gjør likheten (2.3) med begrensningen (2.4) gyldig—vi sier at n er *entydig* bestemt). Med andre ord er 10^{-n} den potensen av ti som vi må multiplisere a med for at resultatet b skal bli et tall der første siffer står rett til høyre for desimalpunktum. Tallet b kalles ofte for *mantissen* mens n kalles *eksponenten* til a, og høyresiden av (2.3) kalles den *normaliserte formen* av a.

Når a er omskrevet som i (2.3) er det lett å gjøre de to begrensningene som må til for at a kan representeres med endelige ressurser: i tillegg til å begrense antall siffer i b må vi også begrense antall siffer i n, og dermed begrense hvor langt desimalpunktet er fra de sifrene vi bruker for å skrive a.

For å illustrere representasjon og aritmetikk med flyttall, skal vi i resten av denne seksjonen, som over, bruke en tallmodell der vi begrenser antall siffer i b til fire samt et fortegn, mens n begrenses til å ha ett siffer samt et fortegn. Da vil det største og det minste positive tallet vi kan representere være de to tallene

$$\begin{aligned} 0.9999 \cdot 10^9 &= 999900000, \\ 0.1000 \cdot 10^{-9} &= 0.0000000001. \end{aligned} \tag{2.5}$$

Det minste og det største negative tallet som kan representeres med firesifret mantisse og ensifret eksponent er tallene i (2.5) med motsatt fortegn,

$$\begin{aligned} -0.9999 \cdot 10^9 &= -999900000, \\ -0.1000 \cdot 10^{-9} &= -0.0000000001. \end{aligned}$$

Tilfellet $a = 0$ må behandles spesielt. I dette tilfellet må vi ha $b = 0$, mens n kan være hva som helst. Det er da vanlig å bruke $n = 0$, slik at vi omskriver 0 som

$$0 = 0.0000 \cdot 10^0.$$

Valget vi gjorde med å bruke fire siffer til mantissen og ett siffer til eksponten var tilfeldig; et mer realistisk valg ville for eksempel være å avsette ti siffer til mantissen og to siffer til eksponenten. I tillegg må vi også ta vare på fortegnene til mantissen og eksponenten.

Eksponenten forteller oss størrelsen på tallet (i tallverdi). Hvis eksponenten er stor er tallet enten et stort positivt tall eller et negativt tall med stor tallverdi. Hvis eksponenten er liten, nærmere bestemt et negativt tall med stor tallverdi (for eksempel -9 når ett siffer er avsatt til eksponenten) er tallet lite i den forstand at det ligger nær null (tallverdien er liten).

Mantissen inneholder sifrene lengst til venstre i tallet, uavhengig av tallets størrelse. For eksempel kan en mantisse på 0.14 angi tallet 14000, tallet 14, tallet 0.0014 og mange andre, alt avhengig av størrelsen på eksponenten.

2.3.2 Addisjon og subtraksjon av flyttall

Når vi har funnet fram til en standardisert måte for å representere reelle tall som flyttall er det ikke så vanskelig å tenke seg hvordan aritmetikk med flyttall foregår. La oss se på noen eksempler der vi, som over, bruker flyttall med fire desimale siffer (pluss fortegn) til mantissen og ett desimalt siffer (pluss fortegn) til eksponenten. For å addere de to tallene 2.35 og 4.64 skrives de først på normalisert form,

$$2.35 = 0.2350 \cdot 10^1,$$
$$4.64 = 0.4640 \cdot 10^1.$$

Deretter adderes de to mantissene, noe som gir svaret

$$0.699 \cdot 10^1,$$

og dette gjenkjenner vi som tallet 6.99.

Et litt mer komplisert eksempel er $4.78 + 6.01$. På normalisert form er disse tallene

$$4.78 = 0.4780 \cdot 10^1,$$
$$6.01 = 0.6010 \cdot 10^1.$$

Adderer vi nå de to mantissene får vi svaret

$$1.079 \cdot 10^1.$$

Selv om de to leddene i summen var på normalisert form er svaret altså ikke på normalisert form siden mantissen er større enn en. Dette kommer av at vi fikk 'en i mente´ da vi adderte de to sifrene 4 og 6 lengst til venstre i mantissen. Addisjonsalgoritmen vil kompensere for dette ved å flytte desimalpunktet en plass til venstre og dermed få svaret over på normalisert form,

$$1.079 \cdot 10^1 = 0.1079 \cdot 10^2.$$

I vanlig notasjon gjenkjenner vi tallet som 10.79

Legg merke til at vi i det siste eksemplet begynte med to tresifrede tall og endte opp med et firesifret tall. I det neste eksempelet øker antall siffer fra fire til fem. Vi adderer de to tallene 0.5645 og 0.7821 som på normalisert form er

$$0.5645 = 0.5645 \cdot 10^0,$$
$$0.7821 = 0.7821 \cdot 10^0.$$

Adderer vi mantissene og konverterer til normalisert form blir svaret

$$0.13466 \cdot 10^1.$$

Denne mantissen består av mer enn fire siffer og kan derfor ikke representeres med de flyttallene vi opererer med. I en slik situasjon vil maskinen runde av[3] svaret til

$$0.1347 \cdot 10^1.$$

Så langt har addisjonene vi har sett på enten bevart eller økt antall siffer i mantissen. Ved subtraksjon kan antall siffer reduseres, og dette kan gjøre det problematisk å vurdere nøyaktigheten i en sekvens av beregninger. La oss først se på et eksempel som ikke skaper alvorlige problemer. For å utføre subtraksjonen $1.438 - 1.113$ gjør vi som ved addisjon og omskriver først tallene til normalisert form

$$1.438 = 0.1438 \cdot 10^1,$$
$$1.113 = 0.1113 \cdot 10^1.$$

Subtraksjon av de to mantissene gir tallet

$$0.0325 \cdot 10^1.$$

Som vi ser er dette tallet ikke på normalisert form siden mantissen er mindre enn 0.1. Vi kan konvertere til normalisert form ved å flytte desimalpunktet en plass mot høyre,

$$0.0325 = 0.3250 \cdot 10^0.$$

Vi ser at dersom vi skal skrive svaret med fire siffer (som maskinen vil gjøre i sin interne representasjon) må vi ta med en null lengst til høyre. I dette tilfellet er det greit å legge til en null, men i andre sammenhenger kan dette skape problemer. Et eksempel vil illustrere hva som kan skje.

La oss regne ut uttrykket $x - \sin x$ når $x = 1/12$, målt i radianer. På normalisert form har vi

$$x = 1/12 \approx 0.8333 \cdot 10^{-1},$$
$$\sin x \approx 0.8324 \cdot 10^{-1}.$$

[3]Enkelte 'sære´ maskiner vil kunne trunkere svaret til $0.1346 \cdot 10^1$, altså bare stryke det femte sifferet. Dette er ikke å anbefale siden det gir en større feil enn avrunding.

Subtraherer vi på samme måte som over får vi derfor det normaliserte tallet

$$x - \sin x \approx 0.9000 \cdot 10^{-4}.$$

Men det riktige svaret, skrevet på normalisert form med fire siffer, er $0.9642 \cdot 10^{-4}$. Vi ser altså at bare det første av de fire sifrene er riktige. Problemet er at maskinen har lagt til ekstra nuller slik som over, men i motsetning til det tidligere eksempelet er det de tre sifrene 6, 4 og 2 vi skal legge til nå, og ikke tre nuller.

Så langt har tallene vi har regnet med vært omtrent like store. Dersom dette ikke er tilfelle må addisjonsalgoritmen justeres noe. Ved addisjon av tallene $10 = 0.1000 \cdot 10^2$ og $0.1 = 0.1000 \cdot 10^0$ kan vi ikke bare addere mantissene til de to tallene siden de har forskjellige eksponenter. I slike tilfeller omskrives det minste tallet slik at det får samme eksponent som det største før mantissene adderes,

$$10 = 0.1000 \cdot 10^2,$$
$$0.1 = 0.0010 \cdot 10^2.$$

Svaret blir $0.1010 \cdot 10^2$ som vi gjenkjenner som 10.1.

Når forskjellen mellom de to tallene ikke er så stor går dette bra, men det er ikke vanskelig å finne problematiske situasjoner. La oss se på addisjonen $1000 + 0.001$. På normalisert form er de to tallene $1000 = 0.1000 \cdot 10^4$ og $0.001 = 0.1000 \cdot 10^{-2}$. For å utføre addisjonen må det minste tallet skrives som et tall med 4 som eksponent,

$$0.001 = 0.1000 \cdot 10^{-2} = 0.0000001 \cdot 10^4.$$

Men husk at vi bare har fire siffer tilgjengelig for mantissen, så før addisjonen kan utføres må dette tallet avrundes til et tall med fire siffers mantisse og eksponent 4. Denne avrundingen gir $0.001 \approx 0.0000 \cdot 10^4$. Addisjonen vi faktisk ender opp med blir derfor

$$0.1000 \cdot 10^4 + 0.0000 \cdot 10^4 = 0.1000 \cdot 10^4 = 1000.$$

På en maskin med fire siffers mantisse og ett siffers eksponent vil altså addisjonen $1000 + 0.001$ gi svaret 1000 i steden for det riktige 1000.001. Dette er forsåvidt ikke dramatisk siden svaret er riktig avrundet til fire siffer, men som vi skal se under er dette fenomenet noe vi må ta hensyn til i visse situasjoner når vi programmerer.

Et problem som kan oppstå ved addisjon eller subtraksjon er at svaret blir for stort (i tallverdi) til å kunne representeres som et flyttall. Et eksempel innenfor vår flyttallsmodell er addisjonen $0.8 \cdot 10^9 + 0.4 \cdot 10^9$. På normalisert form blir svaret $0.12 \cdot 10^{10}$—et for stort tall siden vi trenger 2 siffer til eksponenten. Dersom det i en sekvens av beregninger på en datamaskin forekommer et tall med for stor tallverdi vil dette tallet på de fleste moderne datamaskiner få verdien `Positive_Infinity` eller `Negative_Infinity`.

2.3.3 Multiplikasjon og divisjon med flyttall

Multiplikasjon og divisjon av flyttall på normalisert form utføres som forventet. Ved multiplikasjon multipliseres mantissene sammen mens eksponentene adderes, og hvis mantissen

ble mindre enn 0.1 (i tallverdi) vil svaret justeres til normalisert form. Divisjon foregår på lignende måte. Mantissene divideres og eksponentene subtraheres, og om nødvendig justeres resultatet til normalisert form. Som vi så over kan addisjon av flyttall skape store problemer fordi vi kan miste siffer. Slikt kan ikke skje ved multiplikasjon og divisjon så vi går ikke inn på detaljer i algoritmene her.

Ved multiplikasjon og divisjon kan det lett skje at et tall blir for stort eller for lite, og som nevnt over fører dette til at variabelen som skal ha det endelige resultatet får verdien `Positive_Infinity` eller `Negative_Infinity`. Men ved multiplikasjon og divisjon kan det også skje at et tall blir mindre (i tallverdi) enn det minste tallet (i tallverdi) som kan representeres på maskinen. I modellen som vi har brukt over skjer for eksempel dette dersom vi multipliserer $0.1 \cdot 10^{-6}$ med seg selv. Det riktige svaret er da $0.1 \cdot 10^{-13}$, og for å representere dette tallet trenger vi to siffer i eksponenten, samt fortegnet. En slik situasjon vil ikke gi noe feilmelding, men svaret vil bli avrundet til 0.0.

Ved divisjon kan en annen feilsituasjon oppstå, nemlig divisjon med 0. Dette er en udefinert situasjon rent matematisk og håndteres ved at resultatet blir `NaN` som er en forkortelse for Not a Number. Verdien `NaN` gir alltid `NaN` uansett hva den kombineres med og vil derfor lett infisere en beregningskjede fullstendig hvis den først dukker opp. På denne måten er det lett å oppdage at noe har gått galt.

2.3.4 Binær representasjon av flyttall

Da vi diskuterte representasjon av heltall argumenterte vi for å bruke to-tallsystemet siden dette var mer robust i forhold til støy. Det samme gjelder selvsagt ved representasjon av reelle tall. Representasjonen som brukes i de fleste datamaskiner er derfor en to-tallsversjon av representasjonen over. Hvis a er tallet vi skal representere omskriver vi det som

$$a = c \cdot 2^m, \tag{2.6}$$

der c ligger i området $0.5 \leq c < 1$ hvis a er positiv og $-1 < c \leq -0.5$ hvis c er negativ, mens m er den heltallige eksponenten som gjør at de to sidene av likhetstegnet blir like. Hvis $a = 0$ er både c og m null. Vi setter så av det ønskede antallet binære siffer til representasjon av c og m. De aritmetiske operasjonene utføres så i to-tallssystemet, men de sammme problemene som vi så over (i ti-tallssystemet) vil kunne forekomme. Spesielt må det understrekes at vi ikke kommer unna problemet med avrundingsfeil.

Dagens datamaskiner følger stort sett en internasjonal standard for flyttallsaritmetikk, ofte omtalt som IEEE-standarden.[4] Maskinene har mulighet for å representere flyttall med enten 32 eller 64 binære siffer (bits) (noen maskiner kan også arbeide med 80 eller 128 bits flyttall), og i Java kalles disse flyttallstypene `float` og `double`. For flyttall av typen `float` er det avsatt 24 binære siffer til mantissen og 8 binære siffer til eksponenten, begge inklusive fortegn. Dette betyr at eksponenten må være et heltall i området fra -127 til 128. Det største positive tallet som kan representeres med typen `float` er derfor omtrent

[4]IEEE er forkortelse for *Institute of Electrical and Electronic Engineers* som er en stor forening for ingeniører i USA. Flyttallsstandarden er beskrevet i det som kalles *IEEE standard reference 754*.

$2^{129} \approx 6.8 \cdot 10^{38}$.[5] Det minste positive tallet av typen `float` er omtrent $2^{-128} \approx 2.9 \cdot 10^{-39}$.

Variabeltypen `double` har 64 binære siffer, og av disse er 53 avsatt til mantissen, mens de resterende 11 er avsatt til eksponenten. Dette betyr at det største postive tallet som kan representeres med denne datatypen er omtrent $1.7 \cdot 10^{308}$ mens det minste positive tallet er omtrent $4.9 \cdot 10^{-324}$. En mantisse på 53 binære siffer svarer til omtrent 15 desimale siffer. Med andre ord kan vi altså si at vi arbeider med 15 desimale siffer når vi bruker 64 bits flyttall.

Java har to klasser `Float` og `Double` som inneholder en del nyttige konstanter og funksjoner for de to flyttallstypene `float` og `double`. Blant annet inneholder disse klassene konstanter som gir største og minste positive tall for de respektive variabeltypene og konstantene `NaN`, `Negative_Infinity` og `Positive_Infinity`.

Siden moderne datamaskiner regner omtrent like raskt med 64 bits flyttall som med 32 bits, er det nå vanlig å bruke `double` variable i de fleste sammenhenger. Dette gir større nøyaktighet uten ekstra kostnad. Det største problemet med 64 bits flyttall er at de tar opp dobbelt så stor plass når de skal lagres.

2.3.5 Programmering med flyttall

For mange har den grunnleggende kunnskapen om flyttall kommet som resultat av dyrekjøpt erfaring etter flere dagers feilsøking i et dataprogram som oppfører seg helt uforståelig. Når vi programmerer gjør vi ofte, eksplisitt eller implisitt, bruk av tallenes matematiske egenskaper, men det er altså ikke alltid disse gjelder for flyttall. Informasjonen vi nå har bør gjøre det lettere å gjennomskue feil relatert til flyttall, og følgende enkle tips bør kunne redusere antall feil betraktelig.

Bruk av heltall. I forhold til flyttall har heltall den store fordelen at de er helt nøyaktige. Dersom det i en beregning er tall som bare antar heltallige verdier bør slike tall derfor ikke legges i flyttallsvariable. På samme måte kan beregninger ofte involvere enkle reelle tall som raskt kan regnes ut fra heltall. For eksempel er det ofte nødvendig å la en variabel x anta reelle verdier med fast avstand, la oss si

$$x = 0.0, \quad x = 0.1, \quad x = 0.2, \quad \ldots, \quad x = 0.9, \quad x = 1.0.$$

En måte å implementere dette på er å la x starte med verdien 0.0 og så øke x med 0.1 etterhvert som beregningene utvikler seg. Problemet med dette er at vi da gjør en liten avrundingsfeil hver gang x økes, og en test på om x er lik 1.0 ved slutten av beregningene vil kunne gi negativt svar. En bedre måte å gjøre disse beregningene på er å la en heltallsvariabel i anta verdiene 0, 1, 2, ..., 9, 10 og så regne ut x som $x = 0.1 * i$. Da får vi fremdeles en liten avrundingsfeil i multiplikasjonen, men ingen akkumulert (oppsamlet) avrundingsfeil. Dessuten kan vi gjøre den problematiske sammenligningen av x med 0.1 ved i steden å sjekke om i har blitt lik 10.

[5] Dette tallet framkommer ikke direkte fra en binær versjon av den desimale modellen vi så på over, en del tekniske detaljer kommer i tillegg.

Antall siffer i konstanter. Selv om vi regner aldri så nøyaktig, vil vanligvis ikke resultatene vi kommer fram til være mer nøyaktig enn de mest unøyaktige tallene vi baserer beregningene på (som vi har sett vil sluttresultatet ofte være enda mindre nøyaktig). Det er derfor viktig å passe på at våre inngangsdata har tilstrekkelig nøyaktighet. Mange tall som for eksempel måledata har vi ingen kontroll over, men mange beregninger er også basert på matematiske konstanter og disse må vi passe på at blir angitt med tilstrekkelig nøyaktighet. Regner vi med 32 bits flyttall bør derfor konstanter angis med 8 siffer, mens de bør angis med 17-18 siffer hvis vi regner med 64 bits flyttall. Mange språk har de vanligste tallene som π og e tilgjengelige som predefinerte konstanter. Ellers er det en god regel å regne ut aktuelle konstanter i programmet i steden for å taste dem inn når det er mulig, siden de innebygde matematiske funksjonene (trigonometriske funksjoner, logaritmer og eksponensialfunksjoner, rotutdragning) vanligvis gir maksimal nøyaktighet for den aktuelle datatypen.

Håndtering av `NaN`. Som nevnt over gir maskinen beskjed om at noe tvilsomt har skjedd under flyttallsberegninger ved å gi svaret `NaN`. Den vanligste grunnen til dette er at det et eller annet sted har blitt en divisjon med 0. Noen ganger vil bare en eller noen få variable være infisert av `NaN`, og de andre verdiene som skrives ut være riktige, men uansett er dette et tegn på at noe er grunnleggende galt i programmet og det er på tide med feilsøking.

Sammenligning av flyttall. En viktig del av numeriske beregninger er tester på likhet mellom tall. Som et eksempel kan vi tenke oss at vi skal skrive et program for å finne nullpunkter for en funksjon f. Hvis vi har et tall x som vi tror er nullpunktet, må vi teste om $f(x)$ er null. Men ut fra hva vi har sett over vil avrundingsfeil gjøre at vi neppe kan håpe på å finne et nullpunkt helt nøyaktig og da vil heller ikke $f(x)$ bli null. I steden bør vi teste på om tallverdien til $f(x)$ er 'liten'. På grunn av problemene med avrundingsfeil gjelder dette generelt: test aldri om to flyttall er like, test i steden om differansen i tallverdi er nær null. En annen variant av samme fenomen er at når a og b er flyttall som begge er forskjellige fra null kan det allikevel godt hende at $a + b = a$, noe som er umulig for reelle tall, se eksempelet over der vi adderte 1000 og 0.001.

Sammenligning av flyttall kan bli svært mystisk dersom et av tallene som sammenlignes ved en feil har blitt til `NaN`. Det er nemlig slik at et logisk uttrykk som involverer `NaN` alltid gir verdien `false`, selv uttrykket `NaN == NaN` er `false`! Måten å teste om en variabel `a` av typen `double` i Java inneholder `NaN` er

```
if (Double.isNaN(a)) . . . // Hvis dette slår til har vi NaN
```

Avrundingsfeil. Som vi har sett kan antall riktige siffer i flyttall variere. Selv om utgangspunktet for beregningene var tall der alle sifrene var riktige kan vi underveis få kansellering og avrundingsfeil som gjør at langt fra alle siffer i resultatet trenger være riktige. Det er ofte vanskelig å avgjøre om en metode kan gi store avrundingsfeil, men et godt tips er å teste koden med enkle data der resultatet er kjent. På den måten kan man få et godt inntrykk av følsomheten for avrundingsfeil. Det aller beste er selvsagt

å gjennomføre en matematisk analyse av beregningene og gi nøyaktige estimater for avrundingsfeilen, men dette er ofte svært krevende eller umulig i praksis.

2.3.6 Flyttall med vilkårlig presisjon

Problemene med flyttall kommer av de to grunnleggende begrensningene at antall siffer i både mantisse og eksponent er begrenset. Ved å øke antall siffer blir problemene umiddelbart mindre framtredende. For eksempel er det mindre sannsynlig å få flyttallsrelaterte problemer med 64 bits flyttal en med 32 bits flyttall. På den annen side er det klart at de grunnleggende problemene ikke forsvinner selv om vi øker antall siffer; selv om antall siffer har økt er antallet fremdeles begrenset. På de aller fleste datamaskiner fins det bare støtte i elektronikken for å regne med 32 eller 64 bits flyttall, men man kan selvsagt selv skrive programmer som håndterer flyttall med flere siffer. Dette er gjort i de store matematikkprogrammene Maple og Mathematica, og i Java håndterer klassen `java.math.BigDecimal` flyttall med vilkårlig mange siffer. Selv om det kan være imponerende å kunne regne ut π med en million siffer er det viktig å være klar over at begrensningene med flyttall stadig er til stede. Dessuten setter selvsagt de tilgjengelige maskinressursene (hastighet og hukommelse) en øvre grense for det antall siffer som kan håndteres i praksis.

2.3.7 Regnehastighet

Det er vanskelig å si noe presist om regnehastigheten på en datamaskin siden den endrer seg raskt, i takt med teknologiutviklingen, og varierer mellom maskintypene. Tommelfingerregelen er at regnehastigheten dobles hver 18. måned.[6] Dette har grovt sett holdt stikk i mange år nå, men kan selvsagt ikke vare evig.

Den viktigste faktoren som påvirker regnehastigheten er *klokkefrekvensen* i maskinen. Litt forenklet angir denne hvor fort klokka i maskinen 'tikker', og poenget er at de enkleste regneenhetene i maskinen vanligvis klarer å levere et resultat hver gang klokka 'tikker'. Dette betyr at maskinen kan levere resultatet av en addisjon eller multiplikasjon ved hvert tikk, mens divisjon lett tar 3–4 tikk. Nå er dette bare riktig hvis maskinen får lov til å utføre en lang rekke med like operasjoner, for eksempel addisjoner eller multiplikasjoner. I praksis må den ofte veksle mye på hva som skal gjøres slik at den optimale hastigheten ikke oppnås.

Klokkefrekvensen (antall tikk i sekundet) på dagens (år 2000) PC'er ligger fra ca. 500 MHz til ca. 1 GHz, altså mellom 500 000 000 og 1 000 000 000 tikk i sekundet. Dette betyr at maskinene under optimale forhold kan levere et sted mellom en halv milliard og en milliard resultater pr. sekund (en milliard flyttallsoperasjoner pr. sekund refereres ofte til som en gigaflop). Det må understrekes at her er det variasjoner, og det fins 'småfinesser' i mange maskiner som kan gjøre at de under optimale forhold kan levere noe mer enn et resultat pr. tikk.

En måte å øke regnehastigheten er å øke klokkefrekvensen i maskinen, men dette er krevende rent elektronisk. Den andre muligheten er å hekte mange maskiner sammen

[6] Dette kalles Moores lov etter Gordon Moore. Han var en av grunnleggerne av selskapet Intel som produserer PC-brikker, og han kom med denne spådommen i 1965. Nå er det stadig flere forskere som uttaler at det kan bli vanskelig å leve opp til Moores lov selv om den har holdt ganske godt fram til nå.

og la dem arbeide sammen på samme problem, vi sier at de arbeider i *parallel*. Dersom vi setter 10 maskiner til å arbeide på et problem skulle en tro at det ville kunne løses på en tiendedel av tiden, men dette er bare unntaksvis tilfelle, mer realistisk kan vi kanskje få redusert tiden ned til 20 eller 30 % av den tiden en enslig slik maskin bruker. Dette kommer av at et problem sjelden består av 10 uavhengige småproblemer som kan løses hver for seg. Som regel er det derfor slett ikke opplagt hvordan problemet kan deles opp i mindre problemer som kan fordeles til de forskjellige maskinene, og vanligvis kommer vi ikke unna at noen maskiner må bruke en del av tiden til å vente på at andre maskiner har regnet ferdig. Dette gjør parallelisering utfordrende og til et svært aktivt forskningsområde i miljøer som har stort behov for regnekraft.

2.3.8 Valg av variabelnavn

Ved valg av variabelnavn fins det noen uskrevne regler i matematikk som delvis har blitt arvet i programmering, særlig i eldre språk. I matematikk brukes ofte bokstavene i, j, k, l, m og n om heltall, for eksempel som indekser i summasjoner som i $\sum_{i=1}^{n} i^2$. Bokstavene x, y og z brukes ofte om reelle variable som i $\sin x$, mens f, g og h ofte brukes som funksjonsnavn. I tillegg fins det mindre utbredte konvensjoner for bruk av store og små bokstaver og greske bokstaver.

I matematikk er det uvanlig å bruke variable som består av mer enn en bokstav. I programmering bruker vi som regel lengre variabelnavn, men særlig i matematisk programmering har det vært vanlig å bruke konvensjonene fra matematikk og la heltallsvariable begynne med en av bokstavene i, j, k, l, m eller n. Dette er grunnen til at det ikke er uvanlig å finne et variabelnavn som `iant` i et program skrevet av en nordmann. Dette vil typisk betegne en variabel som inneholder et antall, og *i´*en er der fordi dette er et heltall. I dag er det vanlig (og god) praksis i programmering å la variabelnavn reflektere hva variabelen betegner, og dette gjelder selvsagt også i matematisk programmering. På grunn av konvensjonene i matematikk betyr dette at `i` og `k1` ofte betegner heltall, mens `x` og `z3` ofte betegner reelle tall i et matematisk inspirert program.

2.4 Absolutt og relativ feil

Som vi har sett er det ikke til å unngå at vi får avrundingsfeil når vi arbeider med flyttall, og det er derfor viktig å ha en viss kontroll på denne feilen. Men for å kunne ha kontroll på feilen må vi aller først vite hvordan vi skal måle feilen, og det er tema for denne seksjonen.

For å måle feil trenger vi ikke skille mellom flyttall og reelle tall, så vi tenker oss at vi har et reelt tall a og en tilnærming $\tilde{a}$ til a. Den opplagte måten å definere feilen i denne tilnærmingen er ved den *absolutte feilen*, som er gitt ved

$$|a - \tilde{a}|. \tag{2.7}$$

Her har vi satt på tallverditegn siden vi ikke bryr oss med om feilen er positiv eller negativ. Hvis for eksempel $a = 1$ og $\tilde{a} = 1.001$ ser vi at den absolutte feilen er $|-0.001| = 0.001$.

I mange tilfeller er den absolutte feilen en god måte å måle feil på, men enkelte ganger gir den oss ikke den informasjonen vi er ute etter. Hvis vi har tallet $b = 1000$ og

en tilnærming $\tilde{b} = 1000.001$ så er igjen den absolutte feilen 0.001. Med denne tolkningen av feil er altså $\tilde{b}$ en like god tilnærming til b som det $\tilde{a}$ er til a. Men hvis vi ser på antall siffer så inneholder $\tilde{a}$ bare tre riktige siffer mens $\tilde{b}$ inneholder 6 riktige siffer. Utfra en slik betraktning er det derfor ikke urimelig å si at $\tilde{b}$ er en mye bedre tilnærming til b enn det $\tilde{a}$ er til a. For å få fram denne forskjellen definerer vi den *relative feilen* ved

$$\frac{|a - \tilde{a}|}{|a|}. \tag{2.8}$$

når a er forskjellig fra 0 (når $a = 0$ er ikke den relative feilen definert). Den relative feilen framkommer altså ved å skalere den absolutte feilen med det tallet vi betrakter som det 'riktige'. Med andre ord angir den relative feilen hvor stor andel den absolutte feilen utgjør av det 'riktige' tallet.

Vi kan nå regne ut den relative feilen i $\tilde{a}$ og $\tilde{b}$ og finner

$$\frac{|a - \tilde{a}|}{|a|} = 1.0 \cdot 10^{-3}, \qquad \frac{|b - \tilde{b}|}{|b|} = 1.0 \cdot 10^{-6}.$$

I vårt tilfelle er derfor den relative feilen i $\tilde{b}$ mye mindre enn den relative feilen i $\tilde{a}$ og vi skal se senere at størrelsen på den relative feilen er nært knyttet til antall riktige siffer i tilnærmingen vår.

En god analogi til absolutt og relativ feil er hvordan vi måler avkastning av penger. Anta at vi har kr. 2430 i banken ved inngangen til ett år. Vi rører ikke pengene, og når året er omme har pengene vokst til kr. 2551.50. Vi ser at vi har hatt en avkastning på kr. 121.50 siden $2551.50 - 2430 = 121.50$. Hvis vi tenker på $c = 2430$ som det opprinnelige beløpet og $\tilde{c} = 2551.50$ som en tilnærming til dette så svarer den absolutte feilen $|c - \tilde{c}| = 121.50$ nettopp til avkastningen i kroner. Hvis vi regner ut den relative feilen i dette tilfellet så får vi

$$\frac{|c - \tilde{c}|}{|c|} = \frac{121.50}{2430} = 0.05.$$

Vi tar altså avkastningen i kroner og dividerer med det beløpet vi begynte med. Vi finner dermed hvor stor andel avkastningen er av beløpet vi startet med og svaret er 0.05 eller 5%. Dette svarer selvsagt til at vi har hatt en rente på 5% det året vi har hatt pengene i banken. Dette enkle eksemplet illustrerer at absolutt feil kan sammenlignes med avkastning i kroner mens relativ feil svarer til avkastning i %, altså rentefoten.

2.4.1 Relativ feil angir antall riktige siffer

Hvis vi går tilbake til de to eksemplene over med a og b og deres tilnærminger, så ser vi at $\tilde{a}$, som inneholder 3 riktige siffer, har en relativ feil på 10^{-3}, mens $\tilde{b}$, som har 6 riktige siffer, har en relativ feil på 10^{-6}. Generelt er det slik at hvis den relative feilen i $\tilde{c}$ er

$$r = \frac{|c - \tilde{c}|}{|c|} = x \cdot 10^{-m} \qquad \text{med } 0.5 \leq |x| < 5, \tag{2.9}$$

så vil de m første sifrene i c og $\tilde{c}$ stemme overens. Dette er en tommelfingerregel som må tolkes litt romslig, men la oss forsøke oss på et litt upresist argument for at dette er omtrent riktig.

Hvis c er 1 er ikke dette så overraskende. I dette tilfellet er den relative og den absolutte feilen like, slik at hvis de m første sifrene i c og $\tilde{c}$ er like (ett siffer til venstre for desimalkomma og $m-1$ siffer til høyre), så er m'te desimal det første sifferet der de to tallene skiller seg. Men dette er åpenbart det samme som at $r = |c - \tilde{c}| = x \cdot 10^{-m}$ for en passende x som er omtrent 1, slik som i (2.9). Et eksempel er gitt ved a og $\tilde{a}$ over der $|a - \tilde{a}| = 0.001 = 1.0 \cdot 10^{-3}$.

Hvis c er nær 1 i den forstand at $0.5 \leq |c| < 5$, gjør vi ikke så stor feil om vi tilnærmer den relative feilen med den absolutte feilen og bruker argumentet over. Et slikt eksempel er gitt ved $c = 2.135$ og $\tilde{c} = 2.133$. Den absolutte feilen er $2 \cdot 10^{-3}$ og vi ser at de 3 første sifrene i c og $\tilde{c}$ stemmer overens.

I det generelle tilfellet der $c \neq 1$ skriver vi c på formen

$$c = d \cdot 10^{-n}, \qquad \text{med } 0.5 \leq |d| < 5.$$

Tallet d inneholder nøyaktig de samme sifrene som c, men skalert slik at $|d|$ er så nær 1 som mulig. Vi skalerer tilnærmingen $\tilde{c}$ på tilsvarende vis,

$$\tilde{c} = \tilde{d} \cdot 10^{-n},$$

med samme eksponent som c. Vi ser da at $|\tilde{c} - c| = |\tilde{d} - d| \cdot 10^{-n}$, slik at vi i uttrykket for den relative feilen r kan forkorte bort 10^{-n}. Dette gir

$$r = \frac{|c - \tilde{c}|}{|c|} = \frac{|d - \tilde{d}|}{|d|}.$$

Den relative feilen i $\tilde{c}$ er altså lik den relative feilen i $\tilde{d}$. Men siden $|d|$ er ganske nær 1, er vi tilbake i en situasjon der den absolutte feilen $|d - \tilde{d}|$ gir en god tilnærming til den relative feilen. Altså har vi

$$r \approx |d - \tilde{d}| = x \cdot 10^{-m}, \qquad \text{der } 0.5 \leq |x| < 5,$$

hvilket betyr at d og $\tilde{d}$ har omtrent m siffer felles. Siden c og $\tilde{c}$ er tall med nøyaktig samme siffer som d og $\tilde{d}$ må disse også ha omtrent m siffer felles.

Siden den relative feilen måler hvor mange siffer som er riktige egner den seg svært godt til å måle feilen i flyttallsberegninger som jo foregår med et fast antall siffer.

2.5 Noen ord om objektorientert programmering og matematikk

På laveste nivå opererer en datamaskin på binære siffer ved hjelp av et knippe tillatte operasjoner. For noen svært spesielle anvendelser gir dette tilstrekkelig funksjonalitet, men som regel er det nødvendig med mer komplekse datatyper. I vår sammenheng trenger vi særlig å kunne arbeide med tall, og elektronikken i maskinen gir god tilgang til å kunne arbeide både med heltall og reelle tall (i form av flyttall). I Java (som i de fleste programmerimngsspråk) fins det datatyper for tall som er direkte tilpasset elektronikken i maskinen, for eksempel `int`, `long`, `float` og `double`, med sine tilhørende aritmetiske operasjoner og andre metoder knyttet til datatypen. Men matematikk handler ikke bare

om heltall og reelle tall, det fins mange andre matematiske objekter som kan dra nytte av en datamaskins regneferdigheter.

Et opplagt eksempel er komplekse tall. I Java fins det ingen datatype for komplekse tall så en slik datatype må vi eventuelt implementere selv (eller finne fram til biblioteker som andre har skrevet). Java er et språk som er godt tilpasset *objektorientering*, og denne programmeringsmetodologien er godt egnet til å implementere matematiske objekter med sine metoder. For å kunne regne med komplekse tall på en ryddig måte er det derfor rimelig å definere en klasse `complex(x,y)` med passende metoder. Et objekt i denne klassen har da to parametre x og y som angir henholdsvis realdelen og imaginærdelen til tallet. For å kunne gjøre beregninger trenger vi metoder som implementerer aritmetiske operasjoner med komplekse tall, representasjon med polarkoordinater, evaluering av elementære funksjoner med komplekse argumenter og lignende. I tillegg trenger vi metoder for å kunne skrive ut og lese inn komplekse tall.

Matematikken er full av andre objekter med tilhørende metoder som kan implementeres naturlig i en objektorientert omgivelse. Nært beslektet med tallene har vi vektorer både i planet, rommet og flere dimensjoner. De enkleste metodene er da addisjon av vektorer og multiplikasjon av en vektor med en skalar (et tall). Vektorregning er en del av lineær algebra[7] der vektorer er et spesielt eksempel på et mer generelt objekt, nemlig matriser. Disse har sine tilhørende metoder som addisjon, multiplikasjon med skalar, multiplikasjon av matriser, invertering, og dermed divisjon av matriser også videre.

Et eksempel som ikke er en direkte generalisering av tallene er funksjoner. Som objekt er en funksjon f bestemt ved sin definisjonsmengde og en presis beskrivelse av hva den gjør med sitt argument x, altså hvordan den beregner resultatet $f(x)$. Metodene i en funksjonsklasse kan være så mangt siden det er svært mye vi kan gjøre med funksjoner. For eksempel kan vi addere, multiplisere og dividere funksjoner, og hvis vi er interessert i derivasjon er det naturlig å implementere derivasjonsreglene som metoder.

Oppgaver

2.1 Programmer summene under og skriv ut resultatet. Husk at Java har en operator `+=` som kan være nyttig her.

a) $\sum_{i=1}^{5} i^2$.
b) $\sum_{i=1}^{500} 1/i$.
c) $\sum_{i=-15}^{15} i^4$.
d) $\sum_{i=6}^{20} 5$.
e) $\sum_{i=1}^{1000} \sin(i/50)$.

2.2 I matematikk brukes ofte en notasjon for produkter tilsvarende den for summer.

[7] Lineær algebra er en viktig del av grunnutdanningen i matematikk og står sentralt i de to neste grunnkursene (MAT 1110 og MAT 1120).

Uttrykket $\prod_{i=1}^{6} i$ er definert ved

$$\prod_{i=2}^{6} i = 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6.$$

Symbolet $\prod$ er altså helt analogt med $\sum$ bortsett fra at plusstegnene blir endret til multiplikasjonstegn, slik at de aktuelle tallene multipliseres i steden for å adderes. Programmer og skriv ut resultatet av følgende produkter.

a) $\prod_{i=1}^{30} i$.
b) $\prod_{i=3}^{20} 2$.
c) $\prod_{i=1}^{10} i^3$.
d) $\prod_{i=5}^{100} 1/i$.
e) $\prod_{i=-4}^{4} i/(i+1)$.

2.3 Programmer for-løkka

```
double x;
for (x=0.0; x<= 2.0; x+=0.1)
  println("x er " + x);
```

Hvorfor blir aldri `x` lik 2.0?

2.4 Ut fra teksten er det klart at på en datamaskin vil addisjonen $1 + \epsilon$ bli regnet ut til 1 hvis bare ϵ er liten nok. Skriv et program som kan hjelpe deg til å finne det minste heltallet n slik at $1 + 2^{-n}$ blir 1. Gjør dette både for 32- og 64-bits flyttall (`float`- og `double`-variable i Java).

2.5 Binomialkoeffisientene $\binom{n}{i}$ er definert ved

$$\binom{n}{i} = \frac{n!}{i!\,(n-i)!} \tag{2.10}$$

der $n \geq 0$ er et heltall og i er heltall i intervallet $0 \leq i \leq n$. Binomialkoeffisientene forekommer i mange formler og må derfor derfor ofte beregnes på datamaskin, og siden alle binomialkoeffisienter er heltallige (divisjonen i (2.10) må altså alltid gå opp) er det rimelig å bruke heltallsvariable til slike beregninger. For små verdier av n og i går dette greit, men for litt større verdier får vi fort problemer fordi teller og nevner i (2.10) da kan bli større enn de største heltall som kan representeres på maskinen selv om binomialkoeffisienten i seg selv ikke er så stor. Ved å bruke flyttallsvariable i steden kan vi håndtere litt større tall, men igjen vil vi kunne få for store tall underveis selv om sluttresultatet ikke er for stort. I tillegg vil få avrundingsfeil når vi bruker flyttall.

Det som er uheldig med formelen (2.10) er at selv om binomialkoeffisienten vi er ute etter ikke nødvendigvis er så stor vil teller og nevner allikevel kunne være svært store. Slikt er generelt uheldig for numeriske beregninger og bør unngås så sant det er mulig og ikke koster for mye i form av ekstra regnetid. Hvis vi ser litt nøyere på formelen (2.10) så ser vi at vi kan forkorte slik at

$$\binom{n}{i} = \frac{1 \cdot 2 \cdots i \cdot (i+1) \cdots n}{1 \cdot 2 \cdots i \cdot 1 \cdot 2 \cdots (n-i)} = \frac{i+1}{1} \cdot \frac{i+2}{2} \cdots \frac{n}{n-i}.$$

Med andre ord kan vi hjelp av produktnotasjon skrive $\binom{n}{i}$ som

$$\binom{n}{i} = \prod_{j=1}^{n-i} \frac{i+j}{j}.$$

a) Programmer en metode for å beregne binomialkoeffisienter basert på denne formelen. Hvorfor må du bruke flyttall?

b) Test metoden din på binomialkoeffisientene

$$\binom{10000}{3} = 166616670000,$$
$$\binom{100000}{80} = 1.353770149276343 \cdot 10^{281},$$
$$\binom{1000}{500} = 2.702882409454366 \cdot 10^{299}.$$

c) Er det nå mulig å få for store tall underveis i beregningene dersom binomialkoeffisienten vi skal beregne ikke er større enn det største flyttallet vi kan representere.

d) Når vi utledet metoden vår forkortet vi $i!$ mot $n!$ i (2.10) og forenklet på den måten uttrykket for $\binom{n}{i}$. Vi kan utlede en annen metode ved i steden å forkorte $(n-i)!$ mot $n!$. Utled denne metoden på samme måte som over, og diskuter når de to metodene bør brukes.

2.6 Skriv et program for å regne ut den eksakte verdien av 500! ved hjelp av Java-klassen for å håndtere vilkårlig store heltall (som fasit kan du bruke at $500! \approx 1.220136825991110 \cdot 10^{1134}$).

2.7 Skriv et program for å beregne e^{π} med 100 siffer (her er e grunntallet for naturlige logaritmer) ved hjelp av Java-klassen for å håndtere flyttall med vilkårlig mange siffer.

2.8 Reelle tall kan betraktes som desimaltall, og flyttall framkommer når desimaltallene gjøres 'endelige'. Men reelle tall kan også ses på som grenser for rasjonale tall slik at vi kan alltid finne en vilkårlig god tilnærming til et reelt tall ved hjelp av et

rasjonalt tall. Et alternativ til maskinrepresentasjon av reelle tall med flyttall kan derfor være representasjon ved hjelp av rasjonale tall. Diskuter fordeler og ulemper med en slik representasjon (hvordan kommer begrensingene med endelige ressurser til uttrykk, vil vi få avrundingsfeil etc.).

2.9 I denne oppgaven skal du utvide Java med en klasse for å håndtere komplekse tall.

a) Skriv en klasse for å representere komplekse tall i Java. Klassen skal inneholde metoder som implementerer addisjon, multiplikasjon, divisjon og kompleks konjugering, og lag også en metode for å skrive ut komplekse tall på en naturlig måte.

b) Skriv en metode som implementerer den komplekse eksponensialfunksjonen definert ved

$$e^z = e^{x+iy} = e^x(\cos y + i \sin y).$$

Test metoden på de tre funksjonsverdiene

$$e^{i\pi} = -1, \qquad e^{i\frac{\pi}{2}} = i, \qquad e^{\frac{\ln 2}{2}+i\frac{\pi}{4}} = 1 + i.$$

2.10 I denne oppgaven skal vi se hvordan overflow og verdien `NaN` oppfører seg i Java.

a) Skriv et program som forsøker å utføre de to udefinerte divisjonene 1.0/0.0 og 1/0, lagrer resultatet i henholdsvis en `double`-variabel og en `long`-variabel og til slutt skriver ut resulatet.

b) Bruk variable av typen `long` og multipliser sammen de to tallene 2^{50} og 2^{30}. Sjekk om svaret stemmer med den riktige verdien som er

$$2^{80} = 1208925819614629174706176.$$

c) Bruk `double`-variable, multipliser sammen de to tallene 10^{200} og 10^{300} og skriv ut svaret. Blir det 10^{500}?

d) Adder 1 og ta kvadratroten til svaret i foregående oppgave. Hva blir resultatet?

2.11 Skriv et program der du deklarer en variabel `n` av typen `long` og en variabel `x` av typen `float`. Gi først `n` verdien 10^{18}, sett `x` lik `n` og skriv ut det som nå er innholdet i `x`. Forklar resultatet.

2.12 Anta at $\tilde{a}$ er en tilnærming til a og regn ut den relative feilen i tilfellene under. Sammenlign også den relative feilen med antall riktige siffer.

a) $a = 1$ og $\tilde{a} = 0.9994$.

b) $a = 24$ og $\tilde{a} = 23.56$.

c) $a = -1267$ og $\tilde{a} = -1267.345$.

d) $a = 124$ og $\tilde{a} = 7$.

KAPITTEL 3

Litt logikk og noen andre småting

Logikk er sentralt både i matematikk og programmering, og en innføring i de enkleste delene av logikken er hovedtema i dette kapitlet. I tillegg ser vi litt på sammenhengen mellom induksjonsbevis og det som kalles rekursiv programmering.

3.1 Logikk

I dagligtale kommer vi med mange slags utsagn, og en del er forholdsvis presise slik at vi kan vurdere om de er sanne eller ikke. For eksempel er det nokså universell enighet om at utsagnene 'Oslo er hovedstaden i Norge' og 'Jorda er en del av universet' er sanne, mens det er noe større uenighet om utsagnet 'Pannekaker er det beste som fins' er sant.

Matematikken er bygd opp av presise logiske utsagn som er satt sammen i logiske resonnementer. I matematikk er et logisk utsagn presist, og som regel bare interessant dersom vi er i stand til å avgjøre om det er riktig eller ikke.[1] Hvis vi begrenser oss til utsagn som opplagt er sanne eller gale kan vi angi sannhetsgehalten med to verdier, for eksempel 0 (gal) og 1 (sann). Dessuten kan logiske utsagn kombineres med logiske operatorer som 'og' og 'eller'. For eksempel kan de to sanne utsagnene 'π er større enn 3' og 'π er mindre enn 4' kombineres med 'og' til det sanne utsagnet 'π er større enn 3 og π er mindre enn 4' som forenklet kan skrives 'π ligger mellom 3 og 4'.

Litt kunnskap om logikk er også nyttig i forbindelse med programmering. I programmeringsspråk har vi tilgang til if-tester og ulike slags løkker for å styre programutviklingen. En if-test er basert på et logisk utsagn som det kan avgjøres om er sant eller galt, og en løkke vil bli utført inntil et logisk utsagn ikke lenger er sant.

3.1.1 Logiske variable og sammenligninger

En grunnleggende funksjonalitet i datamaskinens elektronikk består i å kunne avgjøre om enkle utsagn om heltall og flyttall er sanne, og støtte for dette er bygd inn i så og si

[1] En viktig milepæl innen matematisk logikk var oppdagelsen til den tyske matematikeren Kurt Gödel (1906–1978) i 1931 at det fins utsagn innen aritmetikken (læren om tallene) som det er umulig å avgjøre om er sanne eller ikke. Slike utsagn kan sammenlignes med at en nordmann uttaler at 'Alle nordmenn lyver'.

alle programmeringsspråk. Hvis a og b er to tall kan maskinen kjapt svare ja eller nei på om relasjoner som

$$a < b, \quad a \leq b, \quad a = b, \quad a \geq b, \quad a > b,$$

holder, og ved hjelp av logiske variable kan vi ta vare på resultatet av en slik test. I Java deklareres logiske variable som `boolean`[2], mens 'sann' og 'gal' angis med de to logiske konstantene `true` og `false`. Hvis vi lar `p` være en variabel av type `boolean` kan vi gjøre tilordningen `p = 2<3`. Siden det er sant at 2 er mindre enn 3 vil `p` få verdien `true`, mens tilordningen `p = 2==3` vil føre til at `p` får verdien `false` (i Java er det operatoren `==` som tester likhet, mens operatoren `!=` tester ulikhet mellom tall.)

Direkte sammenligning av to tall slik som i `p = 2<3` er sjelden aktuelt. Siden vi vet at 2 er mindre enn 3 er dette det samme som å sette `p = true`. Det vanlige er å teste på relasjoner mellom variable. Hvis `a` og `b` er to variable som inneholder tall vil tilordningen `p = a<b` føre til at `p` får verdien `true` hvis tallet som er lagret i `a` er mindre enn tallet som er lagret i `b`, mens `p` vil få verdien `false` i alle andre tilfeller.

Ofte lagres ikke resultatet av et evaluert logisk uttrykk i en variabel, men brukes i stedet til å bestemme valg i programmet i forbindelse med `if`-tester og `while`-løkker. I Java vil for eksempel kodebiten

```
maxab = if (a<b) b
           else a;
```

føre til at det største av de to tallene lagret i `a` og `b` lagres i `maxab`, mens kodebiten

```
for (n=0; n<10; n++) {
      .
      .
      .
  }
```

gir en løkke som vil bli utført inntil det logiske uttrykket `n<10` ikke lenger er sant, altså inntil `n` blir 10. Dette betyr at løkka vil bli utført med `n=0, 1, 2, ..., 9`.

I diskusjonene som kommer under er det viktig å være klar over at et uttrykk som $a > 0$ i logisk sammenheng bare gir mening når a erstattes med et tall. Siden a kan anta alle mulige reelle verdier betegner derfor ulikheten $a > 0$ uendelig mange logiske uttrykk. Noen av disse uttrykkene er sanne, for eksempel de vi får når a erstattes med 1 eller π, mens noen ikke er sanne, for eksempel de vi får når a erstattes med 0 eller -1. Lar vi derimot bare a være et symbol kan vi ikke avgjøre om $a > 0$ er sant eller ikke. Det er også verdt å legge merke til at uansett hva a er vil alltid utsagnet $a \leq 0$ være det motsatte av $a > 0$ i den forstand at hvis det ene er sant vil det andre ikke være sant og omvendt.

3.1.2 Grunnleggende logiske operatorer

Som vi nevnte i innledningen setter vi ofte sammen logiske utsagn ved hjelp av 'og ' og 'eller', både i dagligtale og matematikk. Dette kan vi også gjøre når vi programmerer. De

[2]George Boole (1815–1864) var en engelsk matematiker som i 1854 publiserte en bok om logikk og hvordan man kan gjøre beregninger med logiske utsagn.

tre grunnleggende operasjonene som vi kan anvende på logiske uttrykk er 'ikke' (NOT), 'og' (AND) samt 'eller' (OR), de er *logiske operatorer*. Disse operatorene uttrykkes på forksjellige måter i ulike programmeringsspråk. En del språk bruker de engelske termene i parentes mens Java bruker `!` (NOT), `&` (AND) og `|` (OR). Når vi ikke eksplisitt beskriver programkode vil vi bruke de vanlige matematiske betgnelsene som er $\neg$ (NOT), $\wedge$ (AND) og $\vee$ (OR).

Operatoren $\wedge$ kombinerer to logiske utsagn p og q,

$$p \wedge q,$$

og kombinasjonen er sann hvis både p og q er sanne, i alle andre tilfeller er kombinasjonen ikke sann. Dette tilsvarer bruken av 'og' som logisk sammenbindingsord i dagligtale. Hvis a er et reelt tall er for eksempel uttrykket

$$(a > 0) \wedge (a < 1) \tag{3.1}$$

sant hvis a er et tall som ligger mellom 0 og 1, men ikke sant ellers.

Som ellers i matematikk har vi brukt parenteser i (3.1) for å gjøre betydningen klar. Når vi programmerer kan vi med fordel bruke parenteser på samme måte, både for å gjøre betydningen klar for oss selv og andre som skal lese programmet, og for å være sikre på at maskinen tolker programmet på riktig måte.

Den andre grunnoperatoren er $\vee$. Kombinasjonen

$$p \vee q$$

er ikke sann hvis hverken p eller q er sanne, i alle andre tilfelle er kombinasjonen sann. Med andre ord er det nok at en av p eller q er sanne for at $p \vee q$ skal være sann. Et eksempel er

$$(a > 2) \vee (a < 1),$$

der a er et reelt tall. Dette uttrykket vil ikke være sant hvis a ligger mellom 1 og 2, altså hvis $1 \leq a \leq 2$; i alle andre tilfeller vil minst en av de to ulikhetene være sanne og derfor gjøre hele uttrykket sant. Et annet eksempel er

$$(a > 0) \vee (a^2 > 0). \tag{3.2}$$

Vi ser at minst en av de to ulikhetene alltid vil være sanne, bortsett fra når a er null. Kombinasjonen er derfor sann for alle verdier av a, unntatt $a = 0$.

Denne tolkningen av 'eller' er litt annerledes enn den vi bruker i dagligtale. Hvis noen sier at 'huset var gult eller hvitt' vil det vanligvis ikke kunne bli tolket som at huset hadde begge farver. I vår matematiske språkbruk derimot, er $p \vee q$ sann hvis både p og q er sanne. I uttrykket i (3.2) er for eksempel begge ulikhetene tilfredstilt for $a > 0$ slik at kombinasjonen er sann. Med tolkningen fra dagligtale vil noen muligens synes at uttrykket burde være usant når $a > 0$.

Det fins en annen logisk operator med akkurat denne tolkningen, nemlig 'eksklusiv eller' (exclusive or, XOR). Denne operatoren har ingen standard betegnelse i matematikk,

men vi skal bruke notasjonen $\hat{\vee}$, mens Java bruker notasjonen `^`. Uttrykket $p \,\hat{\vee}\, q$ (som altså skrives `p^q` i Java) vil dermed være sant når en av p og q er sanne, men ikke sant hvis p og q er like (enten begge sanne eller begge ikke sanne).

Som et eksempel på 'eksklusiv eller' kan vi se på uttrykket i (3.2) med $\vee$ erstattet med $\hat{\vee}$,

$$(a > 0) \,\hat{\vee}\, (a^2 > 0).$$

Dette vil være sant hvis $a < 0$ for da er høyresiden av $\hat{\vee}$ sann mens venstresiden ikke er sann. For $a < 0$ er begge sidene sanne så uttrykket er ikke sant, mens for $a = 0$ er ingen av sidene sanne så uttrykket er ikke sant.

Den fullstendige oppførselen til de tre logiske operatorene AND, OR og XOR er oppsummert i tabell 3.1 som kalles en *sannhetstabell*. Merk at verdien 'sann' her er angitt med 1 mens 'usann' er angitt med 0.

p	q	$p \wedge q$	$p \vee q$	$p \,\hat{\vee}\, q$
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	1	0

Tabell 3.1. Fullstendig beskrivelse av de tre logiske operatorene AND, OR og XOR.

Den siste logiske operatoren vi skal ta for oss er negasjon. Når denne operatoren settes foran et utsagn reverseres betydningen til det motsatte. For eksempel er negasjonen av 'bilen er rød' utsagnet 'bilen er ikke rød', men det er verd å merke seg at hvis vi vet at vi bare har med røde og blå biler å gjøre så vil det siste utsagnet kunne skrives om til 'bilen er blå'.

I matematikk er negasjon ofte angitt med $\neg$, mens negasjon i Java angis med `!`. Uansett hva a er så er utsagnet $\neg(a = 0)$ det samme som $a \neq 0$. Hvis a kan være et vilkårlig reelt tall er $\neg(a > 0)$ det samme som $a \leq 0$, men legg merke til at hvis vi begrenser oss til bare å se på positive heltall er $\neg(a > 0)$ det samme som å si at 'a eksisterer ikke'.

Så langt har vi bare sett på logiske uttrykk som kombinerer to utsagn slik som i $p \wedge q$. Vi kan kombinere flere utsagn slik som i

$$(p \wedge q) \wedge r, \tag{3.3}$$

$$(p \vee q) \vee (r \vee s), \tag{3.4}$$

der p, q, r og s alle er logiske utsagn (vi skal se på blandede uttrykk senere). Her har vi satt inn parenteser siden vi så langt bare har klare regler for hvordan vi kan kombinere to logiske utsagn. Men det er ikke vanskelig å se at verdien av uttrykkene i (3.3) og (3.4) er uavhengig av hvordan parentesene er plassert. Det første uttrykket (3.3) er sant bare når p, q og r alle er sanne, uansett hvordan uttrykket utstyres med parenteser, mens det andre uttrykket (3.4) alltid er sant bortsett fra i det ene tilfellet der ingen av utsagnene

p, q, r og s er sanne. Vi kan derfor fjerne parentesene og skrive

$$p \wedge q \wedge r,$$
$$p \vee q \vee r \vee s,$$

uten noen fare for misforståelse. Det er mange matematiske operasjoner som har denne egenskapen, for eksempel addisjon og multiplikasjon, og den har derfor fått et eget navn; vi sier at operatorene $\wedge$ og $\vee$ er *assosiative*. Assosiativiteten gjelder også når vi har lengre lenker med logiske uttrykk: hvis alle operatorene er enten $\wedge$ eller $\vee$ blir resultatet uavhengig av rekkefølgen vi bruker operatorene.

Hva med den tredje logiske operatoren $\hat{\vee}$, er den assosiativ? Hvis vi ser på de to uttrykkene

$$(p \,\hat{\vee}\, q) \,\hat{\vee}\, r, \qquad p \,\hat{\vee}\, (q \,\hat{\vee}\, r),$$

så er spørsmålet om de alltid er like. Ved å sjekke alle mulige kombinasjoner og sette opp en sannhetstabell vil vi se at de to uttrykkene er like og derfor at 'eksklusiv eller' også er en assosiativ logisk operator. Assosiativiteten gjelder også for lengre lenker med $\hat{\vee}$, men en generell beskrivelse av $\hat{\vee}$ er litt mer komplisert enn for $\wedge$ og $\vee$. Det viser seg at hvis vi har en lang rekke med logiske utsagn lenket sammen med $\hat{\vee}$ så er verdien sann hvis antall sanne utsagn er et odde tall og ikke sann hvis antall sanne utsagn er et partall.

La oss til slutt i denne seksjonen nevne en opplagt egenskap ved alle de tre logiske operatorene $\wedge$, $\vee$ og $\hat{\vee}$, de er alle *kommutative*. Dette betyr at rekkefølgen av de logiske utsagnene som bindes sammen med operatorene er likegyldig. For eksempel har vi helt opplagt $p \wedge q = q \wedge p$ og $p \vee q \vee r = q \vee r \vee p$. Denne egenskapen er det også mange andre matematiske operatorer som har, for eksempel multiplikasjon og addisjon, mens divisjon ikke er kommutativ.

3.1.3 Logisk aritmetikk

Det er ofte behov for å kombinere logiske operatorer. Anta for eksempel at vi skal sjekke om heltallet n er et primtall. En enkel måte å gjøre dette på er å teste om n er delelig med noe tall mindre enn seg selv. I mange programmeringsspråk fins det en egen operator som gir resten ved divisjon av to heltall, og i Java er denne operatoren gitt ved `%` slik at resten i divisjonen n/i kan finnes ved operasjonen `n % i`. I matematikk finnes det ingen standard notasjon for resten ved heltallsdivisjon så vi stjeler fra Java og skriver $n \,\%\, i$. For å løse problemet lar vi en variabel i løpe gjennom alle tall fra 2 opp til n og sjekke om resten ved divisjon av n med i blir 0, noe vi kan gjøre med testen $n \,\%\, i = 0$. Hvis resten blir 0 for en i kan vi stoppe, siden n da er delelig med i og følgelig ikke kan være noe primtall. Dessuten trenger vi ikke sjekke resten for de i som tilfredstiller $i^2 > n$ (prøv med $n = 100$ så ser du hvorfor). Vi skal altså teste stadig økende verdier av i inntil en eller begge de to testene $n \,\%\, i = 0$ og $i^2 > n$ slår til. Men dette svarer til at vi skal forsøke nye verdier av i inntil det logiske uttrykket

$$(n \,\%\, i = 0) \vee (i^2 > n) \tag{3.5}$$

slår til. Når dette skjer kan vi så ved en enkel if-test finne ut hvilken av de to testene i (3.5) det var som slo til og dermed om n er et primtall eller ikke.

Den naturlige måten å implementere det ovenstående er ved en løkke, for eksempel en while-løkke,

```
i=2;
while (?)  i += 1;
if (n%i != 0) then println(n + "er et primtall");
```

(Husk at i 'ekte' Javakode må du fortelle hvilken klasse `println` hører til slik at den fullstendige utskriftssetningen vil være

```
system.out.println(n + "er et primtall")
```

men slike detaljer tar vi ikke med i vår kode.) Men hva skal spørsmålstegnet erstattes med? Det logiske uttrykket i (3.5) gir beskjed om når vi skal stoppe, mens spørsmålstegnet skal erstattes med et logisk uttrykk som forteller oss når vi skal fortsette, altså negasjonen av uttrykket i (3.5). Litt ettertanke viser at denne negasjonen, $\neg\big((n \,\%\, i = 0) \vee (i^2 > n)\big)$, er det samme som

$$(n \,\%\, i \neq 0) \wedge (i^2 \leq n).$$

Kodebiten som sjekker om n er et primtall kan da uttrykkes som

```
i=2;
while ((n%i != 0) & (i*i <= n))
  i += 1;
if (n%i != 0) then
  println(n + " er et primtall")
else
  println("Den minste faktoren i " + n + " er " +i);
```

Som vi skal se etterhvert er det mange matematiske problemer som kan løses med løkker på denne formen, og ofte er det mer naturlig å formulere et stoppkriterium enn en betingelse for når løkka skal fortsette. Men siden løkka skal fortsette akkurat når den ikke skal stoppe er den ene betingelsen negasjonen av den andre. Og siden det fins presise regler for hvordan negasjonen av et logisk uttrykk kan bestemmes er det som regel enkelt å komme fra stoppbetingelsen til løkkebetingelsen.

Det logiske uttrykket (3.5) er på formen $p \vee q$, mens det vi trengte i while-løkka var negasjonen $\neg(p \vee q)$. Hvis vi ser på kodebiten over så brukte uttrykket $\neg p \wedge \neg q$ for å styre for-løkka, så hvis dette skal være riktig må

$$\neg(p \vee q) = (\neg p) \wedge (\neg q). \tag{3.6}$$

Den enkleste måten å sjekke dette er ved en sannhetstabell som i tabell 3.1, men det er heller ikke så vanskelig å se direkte.

I andre sammenhenger kan det det være aktuelt å ta negasjonen av et uttrykk som $p \wedge q$. Vi finner på samme måte at

$$\neg(p \wedge q) = (\neg p) \vee (\neg q). \tag{3.7}$$

De to reglene (3.6) og (3.7) kalles de Morgans[3] lover.

Det fins en masse andre regneregler for logiske operatorer som vi ikke kan komme inn på her. Men hvis du i en eller annen sammenheng blir sittende å manipulere større logiske uttrykk kan det være lurt å konsultere en 'logisk formelsamling'.

3.1.4 Logikk og mengdelære

Det er en nær sammenheng mellom logikk og mengdelære som vi bare kort skal minne om her. Anta at vi har to mengder A og B, for å være konkrete kan vi tenke oss to delmengder av planet. Unionen av A og B som skrives $A \cup B$, er mengden vi får ved å ta med alle elementer som ligger i enten A eller B eller begge steder. En vanlig måte å beskrive denne unionen på er ved notasjonen

$$A \cup B = \{x \mid x \in A \vee x \in B\}.$$

Vi ser at det er en klar sammenheng mellom den logiske operatoren 'or' og mengdeoperasjonen 'union'.

Mengdeoperasjonen 'snitt' og den logiske operatoren 'og' hører sammen på samme måte. Snittet mellom A og B er mengden som inneholder de elementene som ligger i både A og B,

$$A \cap B = \{x \mid x \in A \wedge x \in B\}.$$

Mengdeoperasjonen 'komplement' er relatert til den logiske operatoren 'negasjon' slik at komplementet til A kan skrives

$$\overline{A} = \{x \mid \neg(x \in A)\}.$$

Uttrykket $\neg(x \in A)$ skrives som regel som $x \notin A$. Legg merke til at denne operasjonen ikke er helt opplagt. Siden mengden A er gitt er det klart hva den inneholder, men det er ikke opplagt hva som ikke ligger i A. Vi har vært konkrete og sagt at A er en del av planet, så da må komplementet til A være den delen av planet som ikke ligger i A. Hvis A betegner alle de reelle tallene i intervallet $[0, 1]$ vil vanligvis $\overline{A}$ betegne de reelle tallene som ikke ligger i dette intervallet. Men hvis vi ikke er interessert i negative tall vil komplementet til A betegne alle reelle tall som er større enn 1. Moralen er at i mengdelære er det viktig å spesifisere hva 'universet' er, en 'supermengde' som inneholder alle objektene som er interessante i den sammenhengen vi opererer i.

Den logiske operatoren $\hat{\vee}$ gir også opphav til en mengdeoperasjon, nemlig mengden av de elementene som ligger i en av mengdene A eller B, men ikke i begge,

$$\{x \mid x \in A \,\hat{\vee}\, x \in B\}.$$

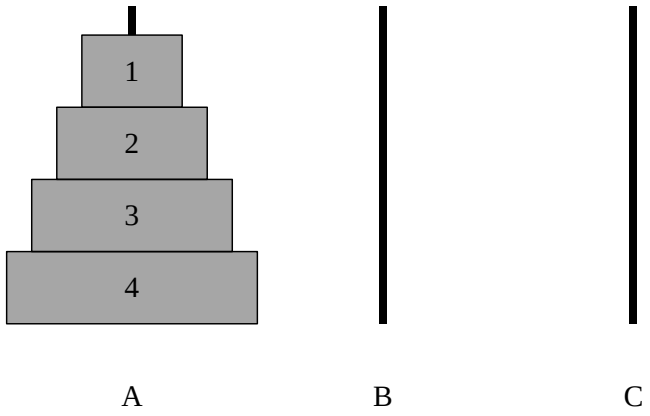
Denne operatoren har ingen standard notasjon i matematikk.

[3] Augustus De Morgan var (1806–1871) var en englesk matematiker. De Morgan var også den som først brukte begrepet *matematisk induksjon* og la en rigorøs basis for denne bevismetoden.

3.2 Induksjonsbevis og rekursjon

Til å begynne med kan matematiske bevis synes ugjennomtrengelige og mystiske, og for mange er det mest mystiske av alt induksjonsbevis. På den annen side er det kravet om at alt skal bevises som gjør matematikken uangripelig, og induksjonsbevis er i mange tilfeller både elegante og kortfattede. Her skal vi se litt på en nær slektning av induksjonsbeviset, nemlig rekursiv programmering.[4] Dette kan være en meget slagkraftig og elegant måte å løse et programmeringsproblem på, men til å begynne med kan rekursjon, som induksjon, virke svært mystisk. Men et rekursivt program er, i motsetning til et induksjonsbevis, noe konkret som det går an å se hvordan oppfører seg. Forhåpentligvis er det derfor ikke så vanskelig å se hvordan rekursjon fungerer, og denne forståelsen bør også kunne kaste lys over induksjonsprinsippet.

Vi skal se på ett eksempel på rekursiv programmering. Problemet vi skal løse er en gammel nøtt kalt *Hanois tårn*. Vi har gitt tre tårn A, B, og C, og på tårn A ligger det n ringer, alle av ulik størrelse, med den største nederst og så i avtagende størrelse oppover med den minste øverst. Problemet består i å flytte ringene fra tårn A til tårn B ved å bruke tårn C til mellomlagring, men uten noen gang å legge en større ring oppå en mindre. Utgangsposisjonen er vist i figur 3.1 for $n = 4$.



Figur 3.1. Utgangsposisjonen i Hanois tårn med 4 ringer.

Vi skal skrive en prosedyre `hanoi(n,X,Y,Z)` for å vise hvordan problemet kan løses. Her har vi kalt tårnene X, Y og Z siden problemet er det samme uansett hvilke tre tårn vi flytter mellom. Vi tenker oss altså at `hanoi(n,X,Y,Z)` løser problemet med å flytte n ringer fra tårn X til tårn Y med Z som hjelpetårn. Under løsningsprosessen vil X, Y og Z være forskjellige ordninger av tårnene A, B og C, og under skal vi se at for å løse problemet med n ringer må vi flytte $n - 1$ ringer fra A til C med B som hjelpetårn og fra C til B med A som hjelpetårn.

La oss forsøke å løse problemet ved å tenke induktivt. Problemet er lett å løse hvis vi

[4]Merk at rekursiv programmering *ikke* gjennomgås i grunnkurset i programmering ved Institutt for informatikk, men er tema i et videregående informatikkurs.

ikke har noen ringer, da gjør vi ingen ting. La oss nå tenke oss at vi har løst problemet med $n-1$ ringer med en prosedyre `hanoi(n-1,X,Y,Z)` som flytter ringene fra tårn X til tårn Y ved å bruke tårn Z til mellomlagring. Utfordringen er nå å utnytte at vi kan løse problemet med $n-1$ ringer til å løse problemet med n ringer. Dette er helt i tråd med induksjonsbevis: Skal vi vise påstanden $P(n)$ sjekker vi først at $P(1)$ er riktig. Vi antar så at $P(k)$ gjelder for $k \leq n-1$ og viser ved hjelp av dette at $P(n)$ også er riktig.

For å være konkrete setter vi $n = 4$ slik at utgangspunktet er som i figur 3.1 med (X,Y,Z)=(A,B,C). Induksjonshypotesen vår er at vi vet hvordan vi skal løse problemet med 3 ringer. Dette utnytter vi på følgende måte: For å løse problemet med 4 ringer kan vi først flytte de 3 øverste ringene fra A til C med B som hjelpetårn (vi har jo antatt at vi vet hvordan dette skal gjøres). Den siste ringen på tårn A flytter vi så til tårn B, og så flytter vi til slutt de 3 ringene på tårn C til tårn B med tårn A som hjelpetårn. Denne framgangsmåten virker fornuftig også for generell n, og algoritmen ser da ut som følger, med X, Y og Z som navn på tårnene.

Algoritme 3.1. *Koden under gir løsningen på problemet med Hanois tårn og flytter n ringer fra tårn X til tårn Y med tårn Z som hjelpetårn:*

```
hanoi(n,X,Y,Z)
  int n;
  char X, Y, Z;
  if (n>0) {
    hanoi(n-1,X,Z,Y);
    println("Ring " + n + " fra " + X + " til " + Y);
    hanoi(n-1,Z,Y,X);
  }
```

For å løse problemet med 4 ringer kan vi nå i hovedprogrammet gjøre metodekallet `hanoi(4,´A´,´B´,´C´)`. Dette vil produsere følgende løsning:

```
Ring 1 fra A til C                          Ring 1 fra C til B
Ring 2 fra A til B                          Ring 2 fra C til A
Ring 1 fra C til B                          Ring 1 fra B til A
Ring 3 fra A til C     Ring 4 fra A til B   Ring 3 fra C til B
Ring 1 fra B til A                          Ring 1 fra A til C
Ring 2 fra B til C                          Ring 2 fra A til B
Ring 1 fra A til C                          Ring 1 fra C til B
```

Den første kolonnen med trekk er resultatet av `hanoi(3,A,C,B)` som svarer til kallet `hanoi(n-1,X,Z,Y)` i algoritme 3.1 i dette tilfellet, mens trekkene i den siste kolonnen er resultatet av `hanoi(3,C,B,A)` som svarer til kallet `hanoi(n-1,Z,Y,X)` i algoritme 3.1. Trekket i midten er det eksplisitte trekket som er gitt i algoritme 3.1.

Algoritme 3.1 løser problemet med Hanois tårn, men ser noe underlig ut siden prosedyren kaller seg selv; vi har jo ikke skrevet `hanoi(n-1,X,Y,Z)`. Men det er nettopp det vi har gjort! Ved å skrive `hanoi(n,X,Y,Z)` har vi også skrevet `hanoi(n-1,X,Y,Z)` siden

`n`, `X`, `Y` og `Z` er variable som kan anta vilkårlige verdier. Starten på induksjonen er implementert ved at vi ikke gjør noen ting om ikke n er positiv, mens vi for positive verdier av n utnytter induksjonshypotesen til å løse problemet. Vi kan derfor faktisk bevise at kodebiten over gir oss løsningen på problemet med Hanois tårn: Prosedyren gir opplagt løsningen på problemet når $n = 0$, og hvis vi antar at den gir løsningen med $n - 1$ ringer (induksjonshypotesen) gir den også løsningen med n ringer.

En prosedyre som kaller seg selv på denne måten sies å være *rekursiv*, og rekursiv programmering er tillatt i de fleste språk, inklusiv Java (bortsett fra at vi selvsagt må legge til noen deklarasjoner og annen kosmetikk).

Rekursjon kan være en slagkraftig og elegant måte å løse problemer på, men det er på langt nær alle problemer som lar seg løse på denne måten. For at rekursjon skal fungere må vi ha en hel familie av problemer slik at hver heltallig verdi av en parameter k gir opphav til et problem $P(k)$ som vi tenker oss har en løsning $L(k)$. I problemet med Hanois tårn er da $P(k)$ problemet med k ringer, mens $L(k)$ er løsningen i form av prosedyren vi skrev. Typisk vil $P(0)$ være et enkelt problem slik at løsningen $L(0)$ også er enkel, mens kompleksiteten i problemene så øker raskt med k. Det 'magiske trikset' består i å utnytte de tidligere løsningene $L(0)$, $L(1)$, ..., $L(k-1)$ til å løse $P(k)$ med $L(k)$. I problemet med Hanois tårn bruker vi for eksempel $L(k-1)$ to ganger for å løse $L(k)$. Hvis vi skulle fylt inn den eksplisitte løsningen med $n - 1$ ringer på de to stedene i løsningen av Hanois tårn ville `hanoi(n,X,Y,Z)` blitt kompleks og uoversiktlig, mens det blir svært enkelt når vi kan bruke 'forkortelsene' `hanoi(n-1,X,Z,Y)` og `hanoi(n-1,Z,Y,X)`. Det flotte er at de fleste programmeringsspråk tillater slike konstruksjoner, som altså er kjent som rekursiv programmering.

Oppgaver

3.1 Bruk de Morgans lover og ekspander ut uttrykkene under.

a) $\neg((i > 0) \vee (i < 10))$ når i er heltallig.

b) $\neg((x > 0) \wedge (x < 1) \wedge (i = 10))$ når x er reell og i er heltallig.

c) $\neg((n = 1) \vee (n = 2) \vee (n = 3))$ når n er et naturlig tall som er mindre enn 4.

d) $\neg((x > 0) \wedge (x < 2) \wedge (x = 1))$ når x er et reelt tall.

3.2 Etabler de Morgans lover (3.6) og (3.7) ved å sette opp sannhetstabeller tilsvarende tabell 3.1.

3.3 Skriv en rekursiv prosedyre for å beregne $n! = 1 \cdot 2 \cdot 3 \cdots n$. Merk at i dette tilfellet vil en rekursiv prosedyre bruke mye lenger tid enn den naturlige løsningen med en enkel løkke.

3.4 I denne oppgaven skal vi se litt nærmere på den rekursive prosedyren for å løse problemet med Hanois tårn.

a) Finn løsningen på problemet med Hanois tårn når $n = 3$. Forsøk først å finne

løsningen ved prøving og feiling og sammenlign etterpå ved å gå i gjennom stegene som kallet `hanoi(3,A,B,C)` vil generere.

b) Programmer prosedyren `hanoi` og generer løsningen på problemet for $n = 4$ og $n = 5$.

3.5 I denne oppgaven skal vi se på noen egenskaper ved $\hat{\vee}$ operatoren.

a) Først skal vi verifisere at $\hat{\vee}$ er assosiativ når den binder sammen tre logiske utsagn. Gjør dette ved å sette opp en sannhetstabell for de to utrykkene $(p \,\hat{\vee}\, q) \,\hat{\vee}\, r$ og $p \,\hat{\vee}\, (q \,\hat{\vee}\, r)$ tabell 3.1,

p	q	r	$(p \,\hat{\vee}\, q) \,\hat{\vee}\, r$	$p \,\hat{\vee}\, (q \,\hat{\vee}\, r)$
0	0	0	0	0
1	0	0	1	1
⋮	⋮	⋮	⋮	⋮

Fyll inn de linjene som mangler.

b) Med fire logiske utsagn må vi sjekke om de fire uttrykkene

$$(p \,\hat{\vee}\, q) \,\hat{\vee}\, r \,\hat{\vee}\, s, \qquad p \,\hat{\vee}\, (q \,\hat{\vee}\, r) \,\hat{\vee}\, s, \qquad p \,\hat{\vee}\, q \,\hat{\vee}\, (r \,\hat{\vee}\, s) \tag{3.8}$$

er like. Dette kan vi gjøre ved å utnytte assosiativiteten når vi har to eller tre utsagn. For eksempel har vi

$$(p \,\hat{\vee}\, q) \,\hat{\vee}\, r \,\hat{\vee}\, s = (p \,\hat{\vee}\, q) \,\hat{\vee}\, (r \,\hat{\vee}\, s) = p \,\hat{\vee}\, q \,\hat{\vee}\, (r \,\hat{\vee}\, s).$$

Bruk et lignende resonnement til å vise at alle de tre utrykkene i (3.8) er like.

c) Forklar hvordan argumentet over kan utnyttes til å etablere assosiativiteten i det generelle tilfellet.

d) Som nevnt i teksten er verdien av et logisk uttrykk med en rekke av utsagn bundet sammen av $\hat{\vee}$ avhengig av antall utsagn som er sanne: Hvis antall sanne utsagn er et partal er det totale logiske utsagnet ikke sant mens hvis antall sanne utsagn er et odde tall er det totale utsagnet sant. For å vise dette er det lurt å utnytte at $\hat{\vee}$ er kommutativ og sette alle utsagnene som ikke er sanne først. Gjør dette og sjekk at den generelle regelen gjelder.

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
      .
      .
      .
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

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

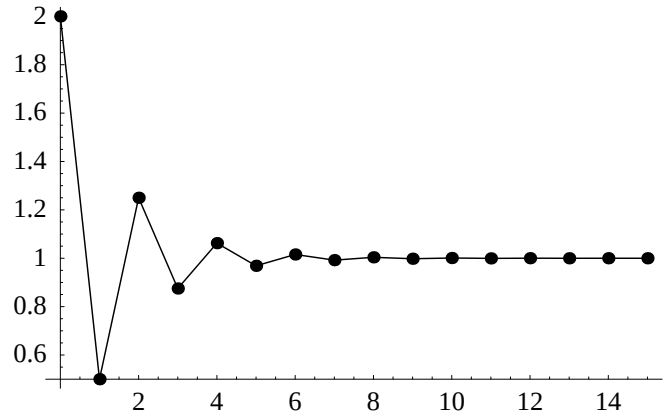
og vi ser at de beregnede verdiene stemmer godt med disse eksakte verdiene. Løsningen skal altså konvergere mot 1 når n går mot uendelig. Beregner vi x_{30} og skriver ut verdien med 15 desimaler får vi

```
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

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



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

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)

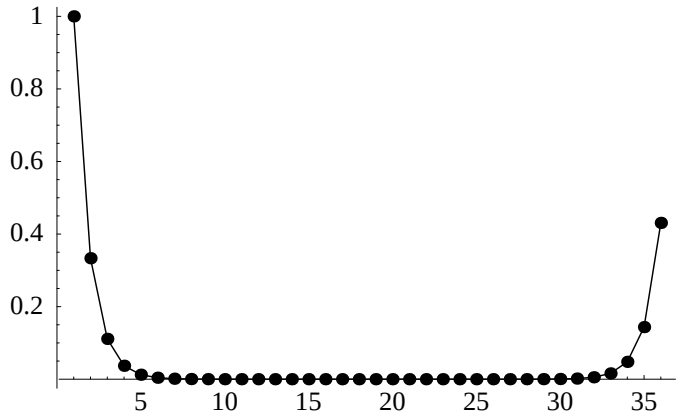
```
x[0]=1.0000000000
x[1]=0.3333333333
x[2]=0.1111111111
x[3]=0.0370370370
      .
      .
      .
x[20]=0.0000169351
```

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 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



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.

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

[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.

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 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

[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).

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 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 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

[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.

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

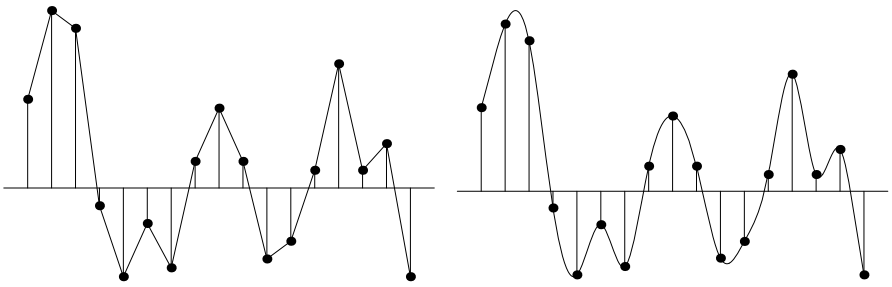
Ut fra vår diskusjon så langt ser vi at musikken på en CD kan betraktes som to endelige tallfølger $\{a_n\}$ og $\{b_n\}$, en følge for hver kanal (musikken er i stereo). For å illustrere de enkleste prinsippene bak digital lydbehandling er det tilstrekkelig å se på en kanal, så vi vil anta at et digitalt 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 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



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.

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. 2 ms[5]) gir et naturlig ekko. Hvis vi skal måle forsinkelsen i millisekunder må vi kjenne samplingsraten. Tidsintervallet mellom to sampler er 0.125 ms ved en samplingsrate på 8000. En forsinkelse på 2.5 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 $n < 20$,

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

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 2.5 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 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

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

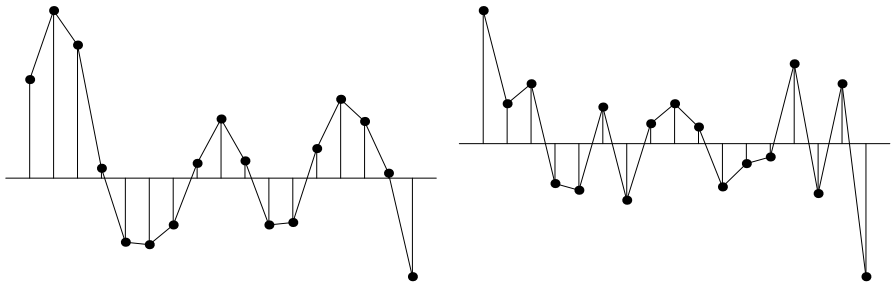
Mer generelt ser vi at hvis vi vil ha ν svingninger pr. sekund når samplingsraten er s må vi velge

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

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

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



(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.

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.

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.

KAPITTEL 5

Funksjoner og kontinuitet

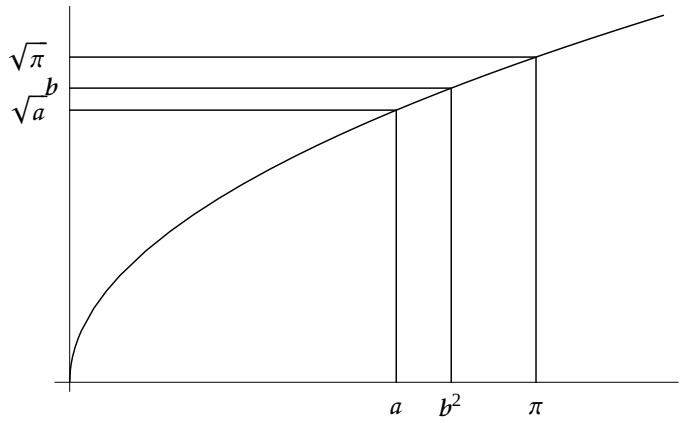
I dette kapitlet skal vi se litt på hva kontinuitet betyr for numeriske beregninger og plotting av funksjoner, og vi skal se at skjæringssetningen gir opphav til en enkel og robust metode for å finne numeriske tilnærminger til nullpunkter i funksjoner, nemlig halveringsmetoden. Vi skal dessuten se at 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

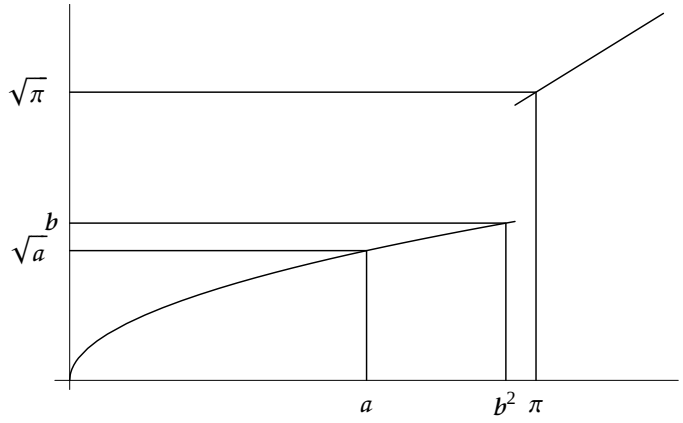
I kapittel 2 så vi at flyttallsberegninger som regel vil være befengt med avrundingsfeil, og vi så i kapittel 4 at dette kan få katastrofale følger for numerisk simulering av differensligninger. Store deler av matematikken gjør utstrakt brukt av funksjonsbegrepet, og programmeringsspråk har konstruksjoner 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 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



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

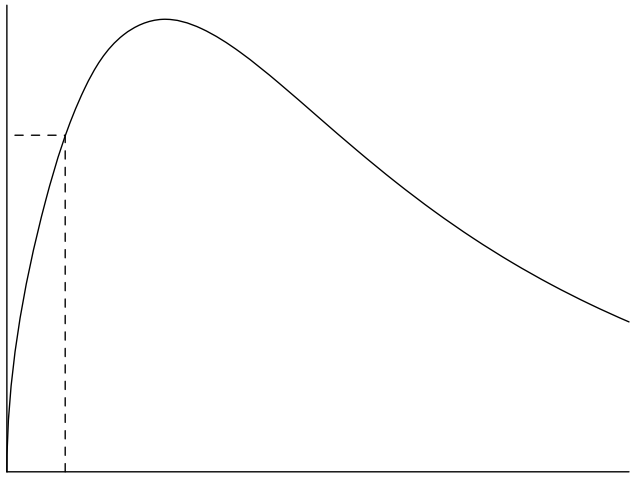


Figur 5.2. Beregning av et punkt på en diskontinuerlig funksjon.

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 $\sqrt{a}$ er større enn $\sqrt{a}$ slik at b^2 er større enn a. De to feilkildene trekker derfor i hver sin retning i vårt eksempel, men i andre situasjoner kan de trekke samme 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.



Figur 5.3. Beregning av et punkt på en bratt 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 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 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.

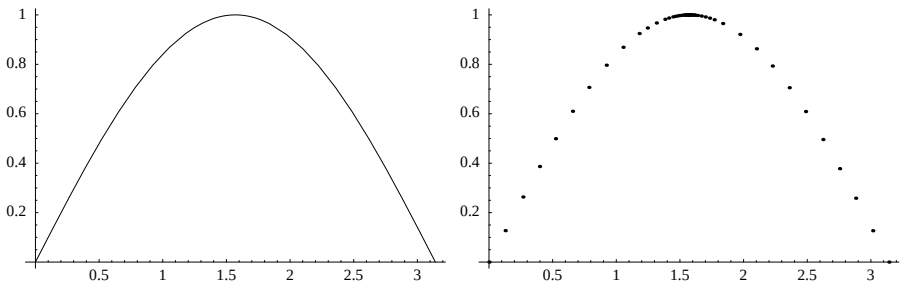
På laveste nivå opererer datamaskinen med et heltallig koordinatsystem som reflekterer den rektangulære punktstrukturen. Origo er i øverste venstre hjørne, og x-koordinatene øker med 1 når vi beveger oss et piksel mot høyre, mens y-koordinatene øker med 1 når vi beveger oss et piksel nedover på skjermen. 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 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.

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



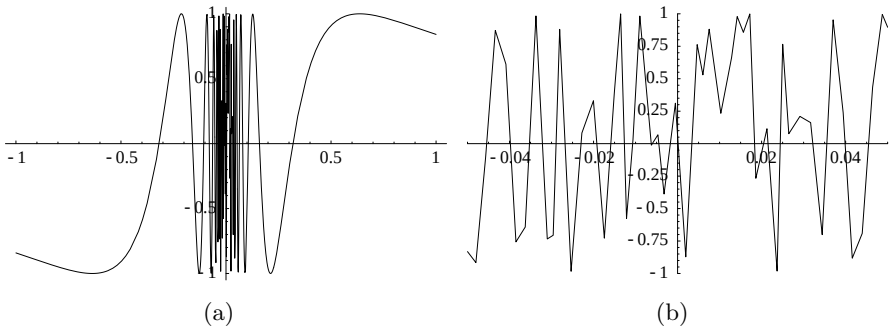
Figur 5.4. Plott av $\sin x$. Plottet i (a) er produsert ved å trekke rette linjer mellom punktene som er vist i (b).

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 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]

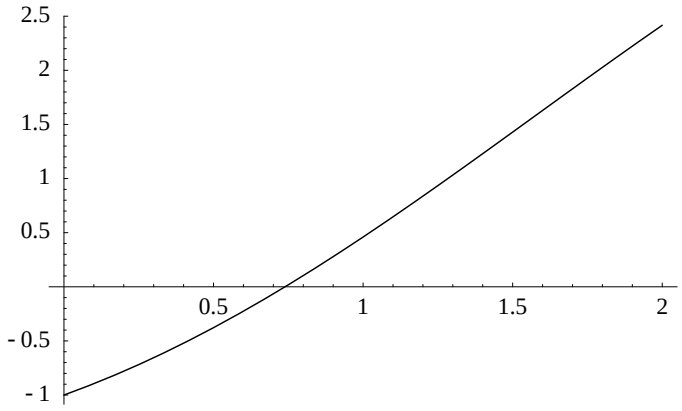
Siden denne plotteteknikken alltid vil gi en sammenhengende kurve er det en underliggende forutsetning at funksjonen er kontinuerlig, og helst bør grafen ligne en rett linje når vi 'zoomer' inn og ser på grafen i detalj. Hvis grafen ikke har denne oppførselen vil resultatet ved første øyekast kunne se 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 selv om funksjonene var helt forskjellige.

[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 5.5. 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 $[-0.05, 0.05]$.



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

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 maskimale antall halveringer av intervallet. Følgende kodebit vil produsere et tall m som er en tilnærming til et nullpunkt c for f:*

```
double a1, fa1, b1, fb1, m, fm;
int i;
a1 = a; fa1 = f(a1);
b1 = b; fb1 = f(b1);
i = 0;
m = (a1+b1)/2;
fm = f(m);
while (i<n && fm != 0) {
  if (sign(fa1*fm) < 0)
  {
      b1 = m; fb1 = f(b1);
  }
  else
  {
      a1 = m;
      fa1 = f(a1);
  }
```

```
  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.

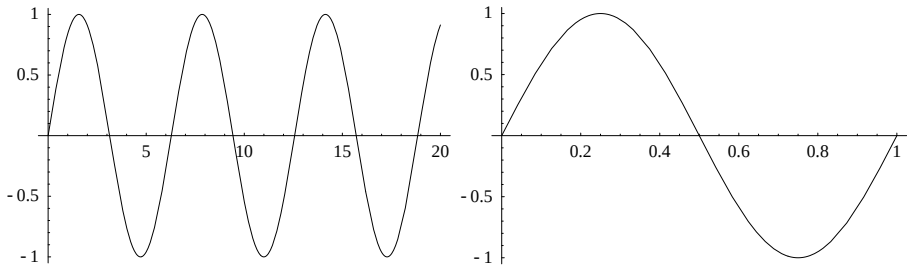
Halveringsmetoden er en svært robust metode for å finnne numeriske tilnærminger til nullpunkter siden alt vi trenger av antagelser er at f er kontinuerlig og har motsatt fortegn i de to endene av intervallet vi starter med. Under disse forutsetningene vil midtpunktene som metoden genererer alltid konvergere 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 et utsnitt av $\sin x$



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

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.

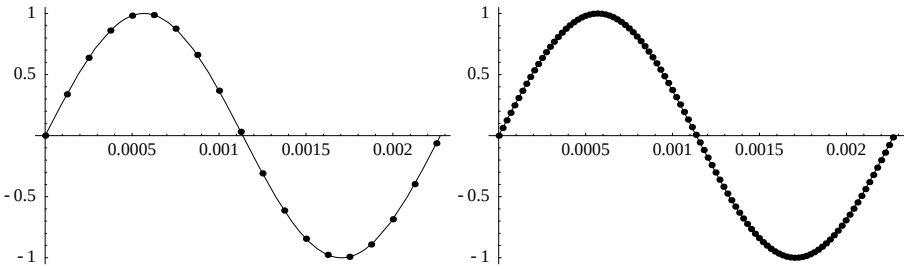
En datamaskin kan bare håndtere digital lyd så vi må lage en følge fra funksjonen vår. Men det er enkelt. La oss anta at vi bruker en samplingsrate på 8000. Da plukker vi bare 8000 jevnt fordelte verdier fra funksjonen $\sin 2\pi 440t$ og gir disse til `Play`-funksjonen i vår programmeringsomgivelse.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

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

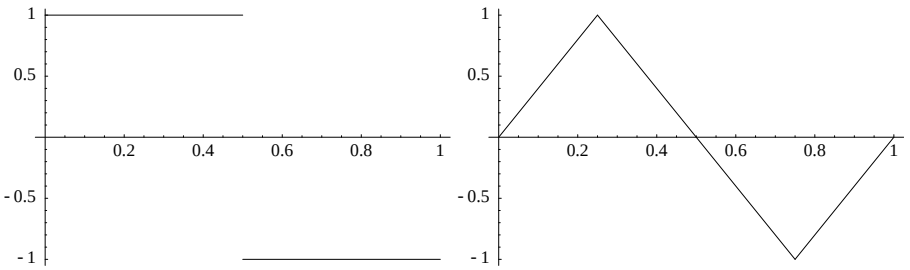
for $i = 0, 1, \ldots, 7999$. Når denne følgen avspilles med samplingsrate 8000 vil vi høre en lyd med frekvens 440 Hz. Denne tonen kalles *kammertonen* i musikk og har samme tonehøyde som summetonen i telefonen.

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

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



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.



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

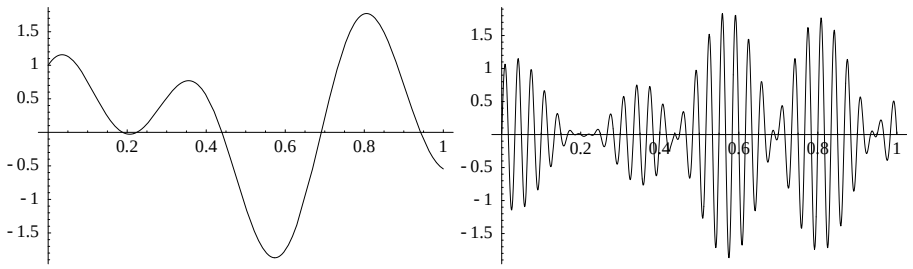
for $i = 0, 1, \ldots, 44099$. Effekten av å øke samplingsraten er altså at vi tar med flere verdier og på den måten får en bedre representasjon av funksjonen. I figur 5.8 har vi vist hvor tett samplene ligger på en periode av funksjonen når samplingsraten er 8000 (i (a)) og 44100 (i (b)).

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



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

over og presser 440 perioder inn på et sekund. Det oppnår vi ved å 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

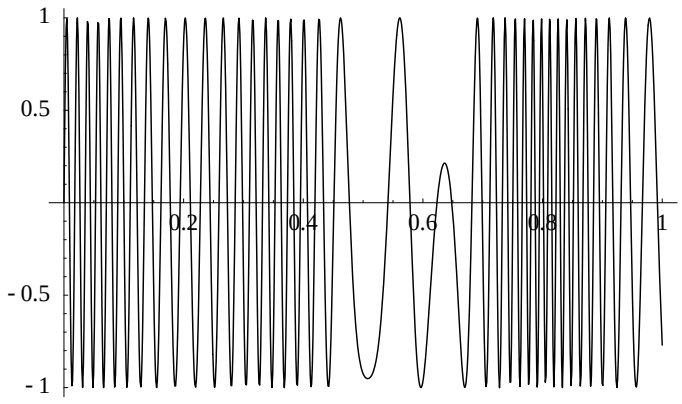
$$P_1(t) = \begin{cases} 4t, & \text{når } 0 \le t < 1/4, \\ 2 - 4t, & \text{når } 1/4 \le t < 3/4, \\ 4t - 4, & \text{når } 3/4 \le t < 1. \end{cases}$$

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 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*.



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

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] Lydsignalet $g(t)$ 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.

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

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.

KAPITTEL 6

Derivasjon

Derivasjon er et av de sentrale begrepene i reell analyse, og den deriverte har en mengde anvendelser i ulike sammenhenger. Vi husker at den deriverte av funksjonen f i et punkt x er definert ved

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

Hvis vi i første omgang overser at vi skal ta en grense så ser vi at den deriverte er den gjennomsnittlige endringen av funksjonen f på intervallet $[x, x+h]$. Som et eksempel kan vi tenke oss at vi er ute og kjører bil og at $f(x)$ måler hvor langt vi har kjørt ved tidspunktet x (her ville det vært mer naturlig å bruke t som variabel). Mellom de to tidspunktene x og $x+h$ tilbakelegger vi da en avstand $f(x+h) - f(x)$ slik at gjennomsnittshastigheten i tidsrommet fra x til $x+h$ blir $\big(f(x+h) - f(x)\big)/h$. Hvis vi måler gjennomsnittshastigheten over stadig mindre tidsintervaller så ser vi at i grensen når h går mot null går gjennomsnittshastigheten mot den deriverte $f'(x)$.

Som et annet eksempel tenker vi oss at vi har en lang tynn metallstav som vi stikker inn i en flamme. Den delen som er i nærheten av flammen vil da bli oppvarmet slik at temperaturen vil variere langs staven. Hvis $f(x)$ nå angir temperaturen i punktet x på staven så ser vi at uttrykket $\big(f(x+h) - f(x)\big)/h$ angir gjennomsnittlig endring i temperatur fra posisjon x til posisjon $x+h$. Den deriverte er grensen for dette uttrykket når h går mot 0 og gir dermed temperaturendringen langs staven akkurat i punktet x.

Den deriverte representerer endringen av en underliggende størrelse slik at begreper som angir vekst typisk kan angis som deriverte, for eksempel befolkningsvekst og kapitalvekst. I dette kapittelet skal vi se på et eksempel på kapitalvekst, men vi skal også se helt andre anvendelser av den deriverte, nemlig som mål på avrundingsfeil og som en måte å angi frekvensen til et lydsignal. Dessuten skal vi se at den deriverte kan utnyttes for å finne nullpunkter for funksjoner.

6.1 Avrundingsfeil og den deriverte

I kapittel 5 så vi litt på sammenhengen mellom kontinuitet av en funksjon i et punkt a og størrelsen på avrundingsfeilen når vi forsøker å beregne $f(a)$. Vi konkluderte med at

når vi beregner $f(a)$ blir en feil i a forstørret opp etter hvor bratt grafen til f er. Ved hjelp av den deriverte til f er vi i stand til å gjøre denne observasjonen mer presis. Vi husker fra kapittel 2 at feil kan måles på to måter, som absolutt feil og relativ feil. Vi begynner med å se på den absolutte feilen.

6.1.1 Absolutt feil

Utgangspunktet er at vi har en funksjon f som vi antar er deriverbar med kontinuerlig derivert, samt et tall a i definisjonsområdet til f; vi ønsker å beregne verdien $f(a)$. På grunn av avrundingsfeil (og kanskje andre feilkilder) blir a tilnærmet med et tall z slik at vi beregner $f(z)$ i steden for $f(a)$, og nå lurer vi på om det er noen sammenheng mellom feilen $z-a$ i a og feilen $f(z)-f(a)$ i $f(a)$. La oss se på forholdet mellom de to feilene. Dette forholdet er gitt ved uttrykket $\big(f(z)-f(a)\big)/(z-a)$, og fra middelverdisetningen vet vi at dette er lik den deriverte i et punkt c som ligger mellom a og z,

$$\frac{f(z)-f(a)}{z-a} = f'(c).$$

Dette kan vi omskrive til

$$f(z)-f(a) = f'(c)(z-a). \tag{6.1}$$

Vi ser derfor at feilen $f(z)-f(a)$ i $f(a)$ er lik feilen $z-a$ i a multiplisert med en proporsjonalitetsfaktor som er $f'(c)$. I grensen når z nærmer seg a, vil også c nærme seg a slik at $f'(c)$ nærmer seg $f'(a)$ (husk at vi antar at f' er kontinuerlig). For de 'vanlige' funksjonene som vi møter er det derfor vanligvis akseptabelt å si at $f'(c) \approx f'(a)$, i allefall så lenge forskjellen mellom a og z er av størrelsesorden med avrundingsfeilen på maskinen.

Ved å være litt mer presise kan vi finne en øvre grense for feilen $f(z)-f(a)$ ved å finne den største verdien av $f'(c)$ på intervallet $[a,z]$ (egentlig på intervallet (a,z), men det er ikke sikkert vi kan finne et maksimum på et åpent intervall). For å slippe problemer med fortegn tar vi tallverdier i (6.1) og ender opp med at

$$\big|f(z)-f(a)\big| \le \max_{c\in[a,z]} \big|f'(c)\big| |z-a|. \tag{6.2}$$

Husk at det godt kan hende at z er mindre enn a; i så fall må intervallet $[a,z]$ tolkes som $[z,a]$. Legg også merke til at det er greit å ta maksimum av $|f'(c)|$ over det aktuelle intervallet siden vi vet fra ekstremalverdisetningen at dette eksisterer når f' er kontinuerlig.

La oss teste dette på noen eksempler. Vi begynner enkelt med $f(x)=3x$ og $a=4$. Det er ofte vanskelig å vite nøyaktig hva avrundingsfeilen på en datamaskin eller lommeregner er, så for å ha godt kontrollerbare eksperimenter setter vi $z = a + 10^{-10}$. Ved hjelp av lommeregner kan vi nå regne ut at

$$f(z)-f(a) = f(4+10^{-10}) - f(4) = 3\cdot 10^{-10}.$$

Siden $f'(x)=3$ for alle x stemmer dette eksakt med (6.1) i dette tilfellet, noe som ikke er så overraskende siden f er en lineær funksjon.

La oss gå over til et litt mer interessant eksempel. Vi prøver oss med $f(x) = \sqrt{x}$ og $a = 2$ og setter igjen $z = a + 10^{-10}$. Vi beregner så $f(z)$ og $f(a)$ og finner at $f(z) - f(a) \approx 3.536 \cdot 10^{-11}$ slik at

$$\frac{f(z) - f(a)}{z - a} \approx 0.3536$$

På den annen side vet vi at $f'(x) = 1/\big(2\sqrt{x}\big)$. Siden a og z ligger såpass nær hverandre og f er en glatt funksjon (alle dens deriverte er kontinuerlige i nærheten av $x = 2$) bruker vi $c = a$ og finner

$$f'(c) \approx f'(a) = f'(2) = \frac{1}{2\sqrt{2}} \approx 0.3536.$$

Med andre ord ser vi at relasjonen (6.1) stemmer svært godt når vi regner med fire sifre i differansene.

La oss se på et tredje eksempel der $f(x) = \tan(x^3/2 - 14)$ og $a = \pi$. Som før setter vi $z = a + 10^{-10}$. Denne gangen finner vi

$$f(z) - f(a) \approx 3.239 \cdot 10^{-7}.$$

Vi ser altså at feilen har blitt kraftig forstørret opp. I dette tilfellet er den deriverte av f i a gitt ved

$$f'(a) = \frac{3a^2}{2\cos^2(a^3/2 - 14)} \approx 3239,$$

slik at feilen som (6.1) gir i $f(a)$ er

$$f'(c)(z - a) \approx f'(a)(z - a) \approx 3239 \cdot 10^{-10} = 3.239 \cdot 10^{-7},$$

altså fullstendig overenstemmelse med den virkelige feilen når vi regner differansene med firesifret nøyaktighet.

La oss se på dette siste eksempelet en gang til og forsøke å estimere feilen vi får hvis vi beregner $f(a) = f(\pi)$ med 64-bits flyttall. Hvis vi kjenner feilen $|z - \pi|$ er dette enkelt siden vi da kan finne $|f(z) - f(\pi)|$ ved å multiplisere med $f'(\pi)$. Vi vet at z er den beste tilnærmingen til π med 64-bits flyttall. Og siden 64-bits flyttall har en mantisse på 53 bits der et bit brukes til fortegn, kan vi regne med at de 52 første binære sifrene i z stemmer med π, mens det neste binære sifferet kan være feil. For tall nær 1 betyr dette at avrundingsfeilen vil være omtrent 2^{-53}. Men nå er π litt større enn 1 så vi må regne med at avrundingsfeilen multipliseres opp tilsvarende og er π ganger større enn dette,

$$|z - \pi| \leq \pi \cdot 2^{-53} \approx 3.488 \cdot 10^{-16}.$$

Som før har vi $f'(\pi) \approx 3239$ slik at

$$|f(z) - f(\pi)| \leq 3239\pi \cdot 2^{-53} \approx 1.130 \cdot 10^{-12}.$$

Legg merke til at dette argumentet viser at en øvre grense for avrundingsfeilen $|z - a|$ er $|a|2^{-53}$ når vi bruker 64-bits flyttall, uansett hva $|a|$ er. På denne måten kan vi derfor estimere avrundingsfeilen i alle funksjoner når vi kjenner den deriverte.

I de fleste tilfeller kan vi ved hjelp av denne teknikken holde god kontroll på avrundingsfeilen. Anta for eksempel at vi skal beregne verdier på funksjonen $\sin x$. Siden $D\big[\sin x\big] = \cos x$ og $|\cos x| \leq 1$ for alle x ser vi at

$$\big|f(z) - f(a)\big| = |\cos c||z - a| \leq |z - a|$$

uansett hva a og z. Når vi beregner verdier på $\sin x$ bør altså aldri avrundingsfeilen i $\sin a$ bli større enn avrundingsfeilen i a, og et lignende argument viser at det samme gjelder for $\cos x$. Men legg merke til at dette gjelder den absolutte feilen $\big|f(z) - f(a)\big|$. Denne feilen sier ikke noe om hvor mange riktige siffer vi har — for å uttale oss om dét må vi se på den relative feilen.[1]

6.1.2 Relativ feil

Vi husker fra kapittel 2 at den absolutte feilen er litt misvisende siden den ikke tar hensyn til størrelsen på tallene vi arbeider med. En absolutt feil på 10^{-2} i et tall a som er nær 1 betyr at vi har 2–3 riktige siffer, mens hvis feilen er den samme og tallet er av størrelsesorden 10^{10} så har vi 12–13 riktige siffer. Når vi dividerer den absolutte feilen med størrelsen på tallet vi arbeider med får vi relativ feil og denne angir omtrent hvor mange riktige siffer vi har. Hvis den relative feilen er 10^{-10} regner vi at vi har omtrent 10 riktige siffer. I forbindelse med beregning av verdier for en funksjon f er det derfor ofte nyttigere å relatere de relative feilene enn de asbolutte feilene, slik vi gjorde i (6.1).

Hvis z er en tilnærming til a så er den relative feilen gitt ved

$$\frac{|z - a|}{|a|},$$

mens den relative feilen i $f(z)$ er

$$\frac{\big|f(z) - f(a)\big|}{\big|f(a)\big|}.$$

Fra relasjonen (6.1) ser vi da at

$$\frac{\big|f(z) - f(a)\big|}{\big|f(a)\big|} = \frac{\big|af'(c)\big|}{\big|f(a)\big|} \cdot \frac{|z - a|}{|a|}, \tag{6.3}$$

der c ligger mellom a og z, som før. Som i (6.2) kan vi kvitte oss med c ved å bruke maksimumsverdien til f',

$$\begin{aligned} \frac{\big|f(z) - f(a)\big|}{\big|f(a)\big|} &\leq \frac{|a| \max\limits_{c\in[a,z]} \big|f'(c)\big|}{\big|f(a)\big|} \cdot \frac{|z - a|}{|a|} \\ &\approx \frac{\big|af'(a)\big|}{\big|f(a)\big|} \cdot \frac{|z - a|}{|a|} \end{aligned} \tag{6.4}$$

[1] Selv om vi ikke *bør* få stor avrundingsfeil er det allikevel *mulig* at vi kan få en stor feil. Det er nemlig ikke sikkert at algoritmen som brukes er god nok til at avrundingsfeilen blir liten. Det kan særlig være et problem når x er svært stor.

(vi antar igjen at f' er kontinuerlig). Tallet

$$\kappa(f;a) = \frac{\big|af'(a)\big|}{\big|f(a)\big|} \tag{6.5}$$

kalles *kondisjonstallet* til f i a og forteller hvor mye den relative feilen i a forstørres opp når vi beregner $f(a)$. Hvis kondisjonstallet er 100 er altså den relative feilen i $f(a)$ blitt 100 ganger større enn den relative feilen i a, så med andre ord er antall riktige siffer i $f(a)$ to mindre enn antall riktige siffer i a.

La oss regne ut kondisjonstallet i de eksemplene vi så på tidligere. Når $f(x) = 3x$ så får vi $\kappa(f;a) = 1$ slik at den relative feilen i a og $f(a)$ er den samme uansett hva a er, noe som stemmer godt med de numeriske verdiene vi regnet ut.

Det neste eksempelet var $f(x) = \sqrt{x}$. Vi finner da at $\kappa(f;a) = 1/2$ for alle verdier av a. Hvis vi regner ut de to relative feilene med $a = 2$ og $z = 2 + 10^{-10}$ så får vi

$$\frac{\big|f(z) - f(a)\big|}{\big|f(a)\big|} = 2.500 \cdot 10^{-11} \qquad \frac{|z-a|}{|a|} = 5 \cdot 10^{-11},$$

som vi ser stemmer med et kondisjonstall på $1/2$.

I det tredje eksempelet var $f(x) = \tan(x^3/2 - 14)$ og vi finner da ved derivasjon at

$$\kappa(f;a) = \left|\frac{3a^3}{2\sin(a^3/2-14)\cos(a^3/2-14)}\right|.$$

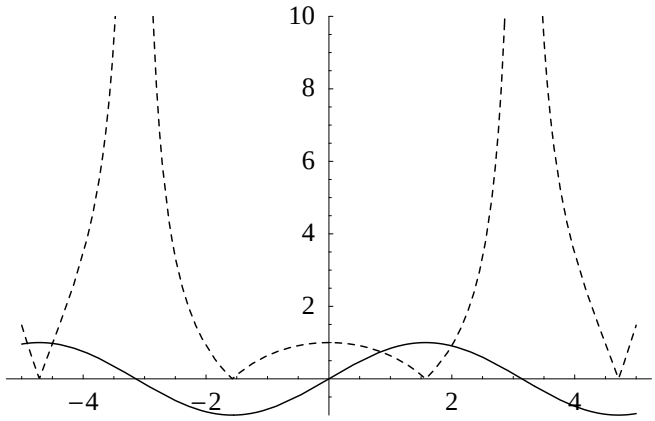
Med $a = \pi$ får vi da $\kappa(f;\pi)) \approx 689.5$, mens de to relative feilene er gitt ved

$$\frac{\big|f(z) - f(a)\big|}{\big|f(a)\big|} = 2.195 \cdot 10^{-8} \qquad \frac{|z-a|}{|a|} = 3.183 \cdot 10^{-11}.$$

Beregner vi forholdet mellom de to relative feilene ser vi at dette stemmer godt overens med kondisjonstallet. Vi ser altså at den relative feilen i a blir multiplisert med en faktor på nesten 1000 når vi beregner $f(a)$ i dette tilfellet. Med andre ord mister vi nesten 3 desimale siffer i denne beregningen.

Det er viktig å være klar over at kondisjonstallet er uavhengig av antall siffer vi regner med, det sier bare noe om hvordan antall riktige siffer *endrer* seg når vi beregner $f(a)$. Et kondisjonstall på 100 er derfor katastrofalt hvis vi starter med 2 riktige siffer, mens det er helt uproblematisk hvis vi bruker 64-bits flyttall og dermed starter med 16 riktige siffer. Men dersom kondisjonstallet blir så stort som 10^{10} begynner det også å bli dramatisk selv om vi bruker 64-bits flyttall, siden det betyr at antall riktige siffer blir redusert fra 16 til 6. I det siste eksempelet over var kondisjonstallet i underkant av 1000, slik at hvis vi regner med 64 bits flyttall vil verdien av $f(\pi)$ i dette tilfellet bare inneholde omtrent $16 - 3 = 13$ riktige desimale siffer.

Et annet vesentlig poeng med kondisjonstallet er at vi trenger ikke nødvendigvis å kjenne dets nøyaktige verdi. Vi bruker det til å få en indikasjon på hvor mange siffer vi kan risikere å miste når vi beregner $f(a)$, og da er det bare viktig å vite hvilken størrelsesorden kondisjonstallet har. Er kondisjonstallet omtrent 10^n vil den relative feilen



Figur 6.1. Funksjonen $\sin x$ og dens kondisjonstall (stiplet) på intervallet $[-5, 5]$.

i a bli multiplisert med 10^n slik at vi vil miste omtrent n desimale sifre. Vi trenger altså ikke kjenne det første sifferet i kondisjonstallet en gang, og litt grovt kan vi si at det holder om relativ feil i kondisjonstallet er mindre enn 500%! Dette betyr også at tilnærmingen vi gjorde i (6.4) da vi erstattet c med a vanligvis ikke vil være kritisk siden $f'(c)$ og $f'(a)$ i det minste vil være av samme størrelsesorden når c er nær a. I praksis vil som regel c være så nær a at den tilnærmede likheten i (6.4) kan regnes som likhet når vi diskuterer avrundingsfeil i en datamaskin. Med andre ord vil vi ut fra kondisjonstallet slik det er definert i (6.5) som regel kunne gi et svært godt estimat på hvor mange siffer vi vil miste når vi forsøker å beregne $f(a)$.

Kondisjonstallet ser altså ut til å gi et godt mål på hvor mye nøyaktigheten kan reduseres i numeriske beregninger. Men hvis vi ser litt nærmere på uttrykket (6.5) så inneholder det et faretruende element, nemlig divisjon med $f(a)$. Vi ser fra (6.4) at denne divisjonen arves direkte fra uttrykket for den relative feilen i $f(a)$, og når $f(a) = 0$ er kondisjonstallet uendelig, akkurat som den relative feilen. Når $f(a) = 0$ bryter altså hele konseptet 'relativ feil' sammen. Men da er det god grunn til å være på vakt når begrepet nesten bryter sammen, gir kondisjonstallet fornuftig informasjon om antall riktige siffer når $f(a)$ er liten?

I figur 6.1 has vi plottet $f(x) = \sin x$ sammen med kondisjonstallet på intervallet $[-5, 5]$. På dette intervallet har $\sin x$ tre nullpunkter, nemlig i $-\pi$, 0 og π. Dersom kondisjonstallet er til å stole på så ser vi at nullpunktet i $x = 0$ er uproblematisk, mens nullpunktene i $-\pi$ og π er problematiske i den forstand at antall riktige siffer i $\sin x$ reduseres betraktelig når vi nærmer oss disse nullpunktene, inntil kondisjonstallet blir udefinert i $x = \pm\pi$. Dette står i skarp kontrast til det vi fant gjelder for den absolutte feilen. Når vi regner ut $\sin a$ er den absolutte feilen i $\sin a$ i verste fall like stor som den absolutte feilen i a.

For å se hva som skjer forsøker vi å beregne $\sin a$ i $a = \pi(1 - 10^{-15})$, med 64-bits flyttall. Dette punktet er et irrasjonalt tall så maskinen vil erstatte a med det nærmeste

flyttallet z. I følge Mathematica er forskjellen mellom z og a med 4 siffers nøyaktighet

$$|z - a| \approx 4.031 \cdot 10^{-16}.$$

Siden den deriverte til $\sin x$ er $\cos x$ og $\cos a \approx 1$ forteller (6.2) oss at den absolutte feilen i $\sin a$ er

$$|\sin z - \sin a| \approx \cos a|z - a| \approx 4.031 \cdot 10^{-16}. \tag{6.6}$$

Beregner vi de to verdiene av sinus får vi[2]

$$\sin a = 3.142 \cdot 10^{-15}, \qquad \sin z = 2.738 \cdot 10^{-15}.$$

Venstresiden i (6.6) er derfor $4.031 \cdot 10^{-16}$, så (6.2) stemmer bra i dette tilfellet.

De relative feilene i a og $f(a)$ kan vi nå lett regne ut,

$$\frac{|z - a|}{|a|} \approx 1.283 \cdot 10^{-16}, \qquad \frac{\big|f(z) - f(a)\big|}{\big|f(a)\big|} = 0.1283,$$

mens kondisjonstallet er gitt ved

$$\kappa(\sin; x) = a|\cot a| = 1.000 \cdot 10^{15}.$$

Dette ser vi stemmer svært bra med forholdet mellom de to relative feilene, slik at (6.4) holder med likhet hvis vi regner med 4 siffer.

Kondisjonstallet er 10^{15} i dette tilfellet, så i følge vår analyse mister vi 15 desimale siffer når vi regner ut $\sin a$. Beregningene våre viser at denne dramatiske effekten faktisk er en realitet. Vi vet at a er kjent med en absolutt feil på omtrent 10^{-16}, og den absolutte feilen i $\sin a$ er omtrent like stor. Men når $\sin a$ er av størrelsesorden 10^{-15} betyr dette at avrundingsfeilen nesten er av samme størrelsesorden som $\sin a$. Følgen av dette er at av de 16 desimale sifrene i $\sin a$ som maskinen serverer oss er bare det første til å stole på. Hvis vi forsøker å beregne $\sin x$ i et punkt x som ligger enda nærmere π vil kondisjonstallet bli enda større og svaret meningsløst siden det ikke vil inneholde noen riktige siffer.

Konklusjonen er at vi ut fra kondisjonstallet kan si nokså presist hvor mange riktige siffer vi har i en beregnet funksjonsverdi. Spesielt bør vi være skeptiske til nøyaktigheten i svært små funksjonsverdier (bortsett fra når argumentet a også er lite) og funksjonsverdier i nærheten av områder der den deriverte nærmer seg uendelig siden kondisjonstallet i begge tilfeller vil kunne bli svært stort.

Det er viktig å understreke at funksjonsberegninger for det aller meste er uproblematiske. Hvis vi går tilbake til figur 6.1 er det bare svært små områder rundt nullpunktene som skaper problemer, og det er sjelden vi trenger å beregne akkurat disse verdiene med ekstrem nøyaktighet. På samme måte vil en ved å plotte en del kondisjonstall se at stort sett så kan funksjoner beregnes med god nøyaktighet. Det gjelder bare å kjenne til problemene som kan oppstå slik at en forstår hva som foregår de få gangene det er nødvendig å beregne problematiske funksjonsverdier. Ellers kan en lett bli sittende å lete etter det en tror er feil i programkoden, men som faktisk har sin årsak i problematiske flyttallsberegninger.

[2]I disse beregningene har vi brukt Mathematica og regnet med 30 siffers nøyaktighet — de verdiene som vi får på en lommeregner eller med 64-bits flyttall blir i dette tilfellet for unøyaktige.

6.2 Numerisk derivasjon

For en funksjon gitt ved en eksplisitt formel er det i prinsippet enkelt å finne en formel for den deriverte også. Dermed kan vi på en grei måte beregne verdien av den deriverte i forskjellige punkter etter behov. Men i mange sammenhenger har vi ingen enkel formel for funksjonen vi arbeider med. For eksempel er det ofte slik at verdien av en funksjon bare er kjent i noen isolerte punkter. Dette skjer når funksjonen framkommer ved måling av en eller annen størrelse, eller når funksjonen er resultatet av numeriske beregninger der vi bare produserer funksjonsverdier i noen utvalgte punkter. Når vi arbeider med digital lyd, for eksempel, er funksjonen som beskriver hvordan lufttrykket varierer bare kjent i samplingspunktene. Vi skal også se senere at mange differensialligninger (en differensialligning er en ligning som involverer en ukjent funksjon (som skal bestemmes) og enkelte av funksjonens deriverte) bare kan løses med numeriske metoder som produserer en tilnærming til løsningen i et endelig antall punkter.

Funksjoner som bare er kjent i isolerte punkter kan åpenbart ikke deriveres på vanlig måte, men det fins et rikt utvalg av numeriske metoder for å finne numeriske tilnærminger til den deriverte i slike situasjoner. Her skal vi bare se på den enkleste av disse metodene. Den deriverte er definert ved grenseprosessen

$$f'(a) = \lim_{x \to a} \frac{f(x) - f(a)}{x - a}.$$

Hvis vi bare kjenner verdien til funksjonen f i punktene x_1 og x_2 er det derfor naturlig å bruke brøken

$$\frac{f(x_2) - f(x_1)}{x_2 - x_1} \tag{6.7}$$

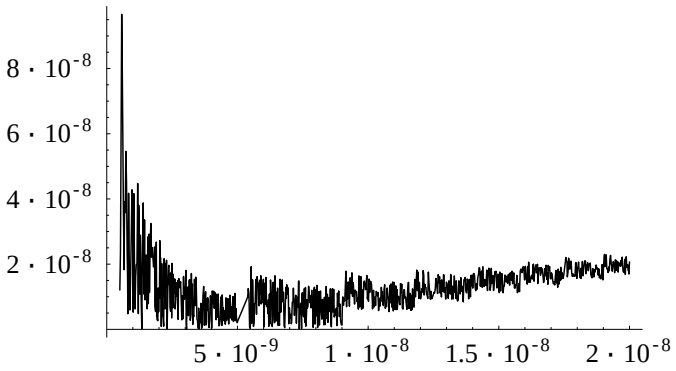
som en tilnærming til $f'(x)$ når x er nær x_1 og x_2. Hvor god denne tilnærmingen er avhenger både av f og av avstanden mellom de tre punktene x, x_1 og x_2.

For å se hvor godt dette fungerer i praksis tester vi med en kjent funksjon, nemlig $f(x) = x^2$. Utfordringen er å beregne $f'(1/2)$ bare ved hjelp av funksjonsverdier i nærheten av $x = 1/2$ — vi har altså ikke lov til å bruke formelen $f'(x) = 2x$. La oss anta at vi kjenner verdiene $f(0.5) = 0.25$ og $f(0.6) = 0.36$. Vi kan da bruke formelen (6.7) som gir

$$f'(0.5) \approx \frac{f(0.6) - f(0.5)}{0.1} = \frac{0.36 - 0.25}{0.1} = 1.1.$$

Nå vet vi at i dette tilfellet er $f'(x) = 2x$ slik at den riktige verdien er $f'(0.5) = 1$. Som vi ser er ikke 1.1 'helt på jordet', men feilen er ganske stor.

Siden den deriverte er grenseverdien for $\big(f(0.5 + h) - f(0.5)\big)/h$ når h går mot 0 er det rimelig å anta at vi kan få feilen mindre ved å kombinere $f(0.5)$ med et punkt som ligger nærmere 0.5. Prøver vi med $h = 10^{-5}$, 10^{-10}, 10^{-13}, 10^{-16} og 10^{-17} får vi med 64



Figur 6.2. Den totale feilen ved numerisk derivasjon for verdier av h i intervallet $[10^{-9}, 2 \cdot 10^{-8}]$.

bits flytttall

$$
\begin{aligned}
f'(0.5) &\approx \frac{f(0.5+10^{-5}) - f(0.5)}{10^{-10}} \approx 1.00001, \\
f'(0.5) &\approx \frac{f(0.5+10^{-10}) - f(0.5)}{10^{-10}} \approx 1.000000083, \\
f'(0.5) &\approx \frac{f(0.5+10^{-13}) - f(0.5)}{10^{-13}} \approx 1.000310945, \\
f'(0.5) &\approx \frac{f(0.5+10^{-16}) - f(0.5)}{10^{-16}} \approx 1.110223025, \\
f'(0.5) &\approx \frac{f(0.5+10^{-17}) - f(0.5)}{10^{-17}} \approx 0.0
\end{aligned}
$$

Vi ser at feilen først avtar, men så øker igjen når h avtar. Årsaken er, ikke uventet, avrundingsfeil. Hvis vi husker tilbake til kapittel 2 så er den mest kritiske flyttallsoperasjonen vi kan gjøre å trekke fra hverandre to tall som er omtrent like store, og det er nettopp det vi må gjøre for å utføre numerisk derivasjon. Jo mindre h er, dess nærmere vil $f(0.5+h)$ være $f(0.5)$, og når $h = 10^{-17}$ blir de to funksjonsverdiene like når vi regner med 64-bits flyttall slik at vi får tilnærmingen 0.0 til $f'(0.5)$.

Vi ser altså ut til å ha to feilkilder som trekker i hver sin retning. Den matematiske feilen som kommer av at vi erstatter $f'(0.5)$ med uttrykket $\big(f(0.5+h) - f(0.5)\big)/h$ blir mindre når h blir liten, mens avrundingsfeilen øker når h avtar siden de to tallene som skal subtraheres i telleren da blir stadig likere. Dette tyder på at det må finnes en 'beste' verdi av h som gjør den totale feilen minst mulig. Noen numeriske eksperimenter viser at den beste h-verdien er nær 10^{-8}. I figur 6.2 har vi plottet ut absoluttverdien av forskjellen mellom $f'(0.5)$ og den numeriske tilnærmingen, for verdier av h mellom 10^{-9} og $2 \cdot 10^{-8}$. Vi ser at feilen varierer forholdsvis mye, men det er tydelig at den har et minimum for

$h \approx 5 \cdot 10^{-9}$. For denne verdien av h får vi $f'(0.5) \approx 0.999999994$ som vi ser er litt bedre enn tilnærmingen vi hadde for $h = 10^{-10}$.

Fenomenet som vi har observert over er et generelt problem som gjelder ved numerisk derivasjon av de fleste funksjoner. Det er lett å tenke som så at vi bør la h bli så liten som mulig slik at vi kan få en nøyaktig verdi for den deriverte, men det er altså ikke så lurt. Det bør også nevnes at det fins andre metoder der den matematiske feilen er mindre slik at vi kan få bedre nøyaktighet med samme verdi av h (og mindre avrundingsfeil). Endelig må vi også huske på at beregningene vi gjorde her var et numerisk ekseperiment. I en praktisk situasjon er det ikke sikkert vi har særlige valgmuligheter når det gjelder størrelsen på h.

6.3 Frekvens som derivert

I kapittel 5 så vi at en funksjon som $\sin 2\pi 440t$ svarer til en lyd med frekvens 440 Hz. Ved å bytte ut sinus med en annen funksjon som ossilerer regelmessig fikk vi fram andre lyder med samme frekvens, og vi så også hvordan vi kunne generere mer spennende lyder ved modulasjon. Ved FM-modulasjon brukte vi funksjoner på formen

$$\sin \theta(t), \tag{6.8}$$

og det viste seg at slike funksjoner gir opphav til lydsignaler med varierende frekvens. Spørsmålet vi stiller oss nå er om det går an å gi en enkel matematisk formel for hvilken frekvens $f(t)$ vi hører ved tidspunktet t når vi avspiller funksjonen (6.8). En slik formel fins og det viser seg at frekvensen vi hører er gitt ved funksjonen

$$f(t) = \frac{\theta'(t)}{2\pi}.$$

Vi ser at dette stemmer godt med tilfellet der $\theta(t) = 2\pi 440t$ siden vi da får konstant frekvens 440. Hvorfor denne formelen gjelder skal vi komme tilbake til senere. Ved å kombinere denne frekvensformelen med integrasjon kan vi foreskrive hvordan frekvensen skal variere og deretter beregne lydsignalet ved integrasjon.

6.4 Newtons metode

Som vi nevnte i forbindelse med halveringsmetoden i kapittel 5 er det bare unntaksvis at vi kan finne en eksakt formel for nullpunktet i en ligning, mens tilnærminger ved hjelp av numeriske metoder kan beregnes i de aller fleste tilfeller. Halveringsmetoden er en robust metode for å finne slike tilnærminger, men den konvergerer ikke så raskt. Newtons metode, som er beskrevet i seksjon 7.3 i *Kalkulus*, er ikke så robust og konvergerer ikke alltid, men når den konvergerer går det som regel meget raskt. Metoden kan dessuten generaliseres og brukes i mange ulike sammenhenger. Her skal vi se på noen viktige egenskaper og begrensninger ved denne metoden.

Utgangspunktet for Newtons metode er at vi har en funksjon f som er deriverbar og som vi vet har et nullpunkt som vi kaller r. På en eller annen måte (for eksempel ved å se på et plott) gjetter vi på at et tall x_0 er en god tilnærming til nullpunktet. For å finne en bedre tilnærming konstruerer vi tangenten $l(x) = f'(x_0)(x - x_0) + f(x_0)$ til f i

x_0, finner nullpunktet x_1 til $l(x)$ og satser på at det er en bedre tilnærming til r. Enkel regning viser at x_1 er gitt ved

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

Denne prosessen kan vi åpenbart fortsette, og på den måten genererer vi stadig nye tilnærminger til nullpunktet.

Det er ikke på noen måte sikkert at denne prosessen vil konvergere, og det er ikke så enkelt å gi gode kriterier for hvordan vi kan avgjøre om vi har konvergens eller ikke. Det beste er å først forsikre seg om at det er et nullpunkt i nærheten av x_0 ved hjelp av for eksempel skjæringssetningen eller et plott. Det er da gode muligheter for at metoden vil konvergere, og en vanlig måte å teste dette på er ved å se om avstanden mellom x_{n+1} og x_n blir liten når n vokser. Hvis forskjellen mellom x_n og x_{n+1} for en n er mindre enn for eksempel 10^{-10} (eller et annet passende, lite tall), stopper vi iterasjonene og sier at x_{n+1} er et godt estimat for nullpunktet r (eventuelt kan vi bruke et relativt feilestimat som $|x_{n+1} - x_n|/|x_{n+1}$, men dette vil skape problemer hvis $\lim x_n = 0$). Det er viktig å være klar over at dette ikke er idiotsikkert, men det er vanskelig å finne noe bedre konvergenskriterium.

Som sagt kan det godt hende at Newtons metode ikke konvergerer, så det er ingenting i veien for at forskjellen mellom x_n og x_{n+1} aldri blir liten, uansett hvor stor n blir. For å hindre at vi ender opp med en evig løkke bør vi derfor ikke tillate at antall iterasjoner overstiger en forhåndsbestemt øvre grense. Ut fra disse betraktningene kan vi formulere Newtons metode som en algoritme.

Algoritme 6.1 (Newtons metode)**.** *La f være en funksjon som er deriverbar med derivert f', la* `x0` *være en initiell tilnærming til et nullpunkt for f, la* `eps` *være en gitt toleranse og la* `nmax` *være et heltall som angir det maksimale antall iterasjoner som tillates. Koden under vil enten konvergere mot et av f's nullpunkter eller stoppe etter* `nmax` *iterasjoner.*

```
double x0, xp, x, e;
int n;
x = x0; e = maxint; n = 1;
while (e>eps & n<=nmax) {
  xp = x;
  x = xp - f(xp)/Df(xp);
  e = abs(x-xp);
  n = n + 1;
}
```

Her er det antatt at f og f' er tilgjengelig som to metoder `f` *og* `DF`*. Hvis* `e <= eps` *ved utgangen av koden antar vi at* `x` *er en tilnærming til nullpunktet med feil mindre enn* `eps`*. Hvis ikke konvergerer ikke Newtons metode etter* `nmax` *iterasjoner.*

Det mest hensiktsmessige er som regel å implementere en kodebit som dette som en metode, for eksempel `newton(f, Df, x0, eps, nmax)`.

Det er viktig å huske på at selv om koden i algoritmen stopper ved at feilen `e` har blitt mindre enn (eller lik) `eps` er det ikke helt sikkert at `x` er nær et nullpunkt. Det kan for eksempel være lurt å sjekke på et plott om `x` virkelig ligger nær et nullpunkt. Eventuelt kan vi i tillegg til `e<=eps` også kreve `f(x)<=delta`, der `delta` er en passende toleranse, før vi stopper iterasjonene.

For at Newtons metode skal konvergere er det visse betingelser som bør være tilfredstilt. De viktigste er:

- Hvis nullpunktet vi skal finne er r bør vi ha $f'(r) \neq 0$ (ikke strengt nødvendig).
- Den andrederiverte til f bør være kontinuerlig i en omegn om nullpunktet r.
- Startverdien x_0 må være tilstrekkelig nær nullpunktet r.

Den første betingelsen er faktisk ikke essensiell, men hvis $f'(r) = 0$ vil dette redusere konvergenshastigheten og når vi nærmer oss nullpunktet vil vi få divisjon med stadig mindre tall som kan skape problemer. Det tryggeste er derfor å anta at $f'(r) \neq 0$.

Den andre betingelsen er nødvendig for å garantere konvergens når vi starter nær nullpunktet. Et eksempel som illustrerer dette er funksjonen definert ved

$$f(x) = \begin{cases} \sqrt{x}, & \text{når } x \geq 0, \\ -\sqrt{-x}, & \text{når } x < 0, \end{cases}$$

slik som vist i *Kalkulus*. Uansett hvilken verdi x_0 vi starter med vil Newtons metode bare hoppe mellom de to verdiene x_0 og $-x_0$, se figur 6.3 (c).

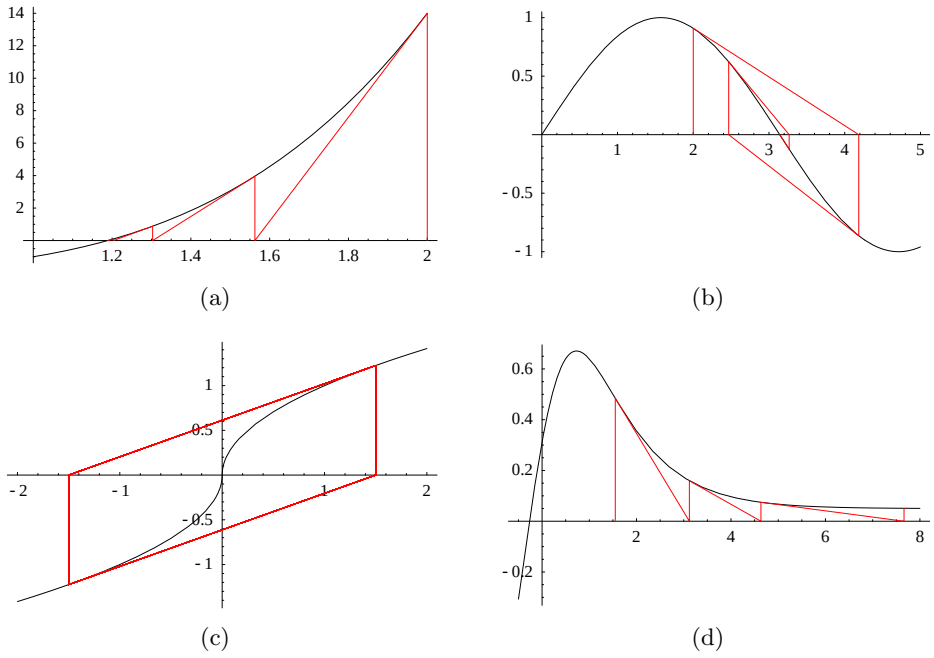
Den tredje betingelsen er vanskelig å håndtere siden den ikke sier noe presist om hvor nær et nullpunkt vi må starte, bare at vi må starte 'tilstrekkelig nær'. Det er mulig å gjøre betingelsen mer presis og gi en verdi for δ, men som regel vil denne verdien være mye mindre enn det som er nødvendig. Et eksempel på hva som kan skje hvis vi ikke starter nær nok et nullpunkt er vist i figur 6.3 (d).

Kanskje det mest karakteristiske med Newtons metode er konvergenshastigheten når den først konvergerer. Det er mulig å vise at når metoden konvergerer mot et nullpunkt så vil antall riktige siffer i x_{n+1} være det dobbelte av antall riktige siffer i x_n når vi kommer næR nullpunktet. Med 64-bits flyttall har vi totalt omtrent 16 desimale siffer, så hvis vi klarer å finne en x_0 som ligger nær nullpunktet trenger vi ikke mange iterasjoner for å få 16 riktige siffer — ofte vil det være nok med 3 eller 4 iterasjoner.

Newtons metode og halveringsmetoden komplementerer hverandre på en god måte. En mulig strategi er å bruke halveringsmetoden til å begynne med for å forsikre seg om at en er rimelig nær et nullpunkt og så skifte til Newtons metode for å finne nullpunktet med god nøyaktighet.

6.5 Diskret og kontinuerlig kapitalvekst

I kapittel 4 i *Kalkulus* så vi et par eksempler på hvordan kapitalvekst kan modelleres ved hjelp av differensligninger. I denne seksjonen skal vi se litt nærmere på slike modeller. Modellering med differensligninger i denne sammenhengen betyr at vi bare måler kapitalen ved isolerte tidspunkter, for eksempel en gang i året, men vi skal også se at vi kan



Figur 6.3. Fire eksempler på iterasjoner med Newtons metode. I (a) konvergerer $\{x_n\}$ pent mot nullpunktet fra høyre, mens verdiene i (b) ligger til høyre og venstre for nullpunktet annenhver gang. I eksempelet i (c) har vi ikke konvergens, metoden vil bare hoppe mellom de to verdiene x_0 og $-x_0$. Vi har heller ikke konvergens i (d); her vil $\{x_n\}$ divergere mot uendelig.

få en kontinuerlig modell ved å foreta en passende grenseovergang i differensligningen. Framstillingen her er noe knapp i og med at stoffet suppleres med oppgaver.

6.5.1 Diskret kapitalvekst

Den enkleste modellen for kapitalvekst har vi når vi setter en sum penger K_0 i banken som betaler oss en rente r pr. tidsenhet. Etter n tidsenheter har da kapitalen vokst til K_n, der K_n er gitt ved

$$K_n = (1+r)K_{n-1}, \qquad n = 1, 2, \ldots \tag{6.9}$$

Det er lett å vise at løsningen av denne differensligningen er gitt ved

$$K_n = (1+r)^n K_0. \tag{6.10}$$

Tidsenheten er her ikke spesifisert direkte og kan være dager, uker, år eller et annet fast tidsintervall. Men uansett hva tidsenheten er så er det viktig at renten r gis i forhold til riktig tidsenhet. Hvis renta er 10% pr. år og vi oppgir r som $r = 0.1$ betyr det implisitt at n teller antall år.

I virkeligheten vet vi at renta varierer og dette kan vi modellere ved å endre differensligningen til

$$K_n = (1+r_n)K_{n-1}, \qquad n = 1, 2, \ldots \tag{6.11}$$

Som før antar vi at startkapitalen er gitt som K_0. I denne ligningen kan vi angi renta for hver n, og en mulig modell for variabel rente er gitt ved formelen

$$r_n = r + \frac{r}{2}\cos(2\pi n/T) \tag{6.12}$$

der T er et helt tall. I dette tilfellet vil renta variere regelmessig mellom $3r/2$ og $r/2$, noe som kan virke nokså kunstig, men denne modellen fanger likevel opp vesentlige trekk ved et svingende rentenivå.

I kapittel 4 i *Kalkulus* så vi også hvordan vi kan inkludere inn- eller utbetalinger i modellen. Hvis vi endrer differensligningen til

$$K_n = (1+r)K_{n-1} - b, \qquad n = 1, 2, \ldots \tag{6.13}$$

har vi en modell der vi i tillegg til å opparbeide renter tar ut et beløp b i hver tidsperiode. Vi kan også angi innbetalinger ved å la b være negativ. Ved induksjon kan vi vise at løsningen av denne inhomogene differensligningen er gitt ved

$$K_n = (1+r)^n K_0 - \frac{b}{r}\big((1+r)^n - 1\big), \tag{6.14}$$

(eller vi kan løse ligningen ved hjelp av metodene i seksjon 4.2 i *Kalkulus*).

Med ligningen (6.13) kan vi også modellere tilbakebetaling av lån. Anta at vi tar opp et lån på K_0 kroner ved tid $n = 0$ med rente r og at vi betaler tilbake et beløp b pr. tidsenhet som dekker både renter og avdrag. Når det tilbakebetalte beløpet pr. tidsenhet er konstant slik som her kalles lånet et *annuitetslån*. Fra formelen (6.14) kan vi regne ut hvor stor b

må være for at hele lånet skal være nedbetalt etter for eksempel 10 eller 20 år, eller mer generelt etter N år, se oppgave 4.

Ligningen (6.13) kan også brukes til å modellere pensjonssparing. Da er K_0 oppspart kapital når pensjonsutbetalingene begynner, mens r er renta og b det fast utbetalte pensjonsbeløpet.

6.5.2 Kontinuerlig kapitalvekst

I forbindelse med banksparing er vi vant til at renter beregnes og legges til kapitalen ved slutten av hvert år. Ved renteberegninger på lån derimot er det vanlig med flere renteberegninger pr. år, og i aksjemarkedet og andre mer avanserte finansmarkeder er det vanlig å regne med kontinuerlig rente. La oss se hva dette innebærer.

Hvis vi starter med et beløp K_0 og har en rente r så vil kapitalen øke til $(1+r)K_0$ etter én tidsperiode. Alternativt kan vi tenke oss at vi har to renteutbetalinger, men at renta da er redusert til $r/2$ ved hver utbetaling. Da vil vi etter en hel tidsperiode ha beløpet

$$(1+r/2)(1+r/2)K_0 = (1+r/2)^2 K_0.$$

Hvis vi fordeler renta med $r/3$ over tre utbetalinger får vi på samme måte at beløpet etter en hel tidsperiode har vokst til $(1+r/3)^3 K_0$. Generelt kan vi fordele renta med r/n over n utbetalinger. Kapitalen etter en hel tidsperiode vil da være

$$(1+r/n)^n K_0. \tag{6.15}$$

Et naturlig spørsmål er nå hva som skjer hvis vi stadig øker antall renteutbetalinger og bruker en tilsvarende liten rente hver gang. Med andre ord, hva skjer når vi lar n vokse over alle grenser i (6.15)? Ved hjelp av l'Hopitals regel er det ikke vanskelig å vise at

$$\lim_{n\to\infty}(1+r/n)^n = e^{\lim_{n\to\infty} n\ln(1+r/n)} = e^r.$$

Hvis vi tenker oss at antall renteutbetalinger går mot uendelig blir derfor beløpet etter én tidsperiode $e^r K_0$.

Hvis vi fordeler renta over mange små tidsperioder på denne måten bør vi også studere kapitalveksten over mindre tidsperioder. Anta at vi deler året inn i m like lange tidsperioder, hver av lengde h slik at $mh = 1$. Tida vil altså løpe i steg på h og etter m tidssteg har det gått ett år. Vi antar fremdeles at renta er r pr. år. Med fast rente blir renta da $r/m = rh$ pr. tidsperiode av lengde h. La oss betegne tida etter j tidssteg med $t = jh$ og kapitalen ved dette tidspunktet med $K^h(t)$. Fra hva vi vet om diskret kapitalvekst har vi da at

$$K^h(t) = (1+rh)^j K_0 = (1+rh)^{t/h} K_0. \tag{6.16}$$

Lar vi nå h gå mot 0 får vi

$$K(t) = K_0 e^{rt}. \tag{6.17}$$

Dette kan vi også vise på en annen måte. Differensligningen (6.13) kan skrives

$$K^h(t) - K^h(t-h) = \frac{r}{m}K^h(t) = rhK^h(t).$$

Altså har vi

$$\frac{K^h(t) - K^h(t-h)}{h} = rK^h(t).$$

Lar vi h gå mot null i denne ligningen og setter $K(t) = K^0(t)$ ser vi at

$$K'(t) = rK(t). \tag{6.18}$$

Det er lett å sjekke at alle funksjoner på formen $K(t) = Ce^{rt}$ passer i denne ligningen og hvis vi velger $C = K_0$ får vi en løsning som tilfredstiller initialbetingelsen $K(0) = K_0$, nemlig $K(t) = K_0e^{rt}$ som vi ser stemmer med (6.17).

Oppgaver

6.1 Beregn kondisjonstallet til følgende funksjoner og lokaliser problematiske områder.

a) $f(x) = x^p$ der p er et reelt tall, og $x \geq 0$.

b) $f(x) = x^2 - 2$ på intervallet $[0, 2]$.

c) $f(x) = e^x$ på hele tallinjen.

d) $f(x) = \ln x$ på intervallet $(0, \infty)$.

e) $f(x) = \tan x$ på intervallet $(-\pi/2, \pi/2)$.

6.2 Gjenta oppgave 5.1, men bruk Newtons metode i steden for halveringsmetoden. Lag plott som gir et visuelt bilde av iterasjonene, slik som i figur 6.3.

6.3 I denne oppgaven skal vi se litt nærmere på modellen for diskret kapitalvekst.

a) Anta at renta er konstant 8 % i modellen (6.9) og at tiden regnes i år. Hvor lang tid tar det da før kapitalen er fordoblet?

b) La oss nå se hva som skjer hvis vi bruker rentemodellen (6.12).
Hvilken tolkning har parameteren T i (6.12)? Lag et plott av r_n som funksjon av n når $T = 6$ og $r = 0.08$ (la n variere fra 0 til 30).

c) Gjør en numerisk simulering av kapitalveksten for $n = 0, 1, \ldots 30$ når startkapitalen K_0 er 1 (vi måler i millioner), renta er gitt ved (6.12) med $r = 0.08$ og $T = 6$. Lag et plott av resultatet. Plott ut kapitalveksten ved konstant rente $r = 0.08$ i samme plott. Prøv å forklare det du ser.

d) En annen mulig rentemodell er gitt ved

$$r_n = r + \frac{r}{2}\cos(2\pi n/T + \pi).$$

Hvordan vil kapitalutviklingen bli med en slik modell i forhold til de to andre modellene vi har sett på? Legg inn denne modellen også i det tidligere plottet og se om dine betraktninger stemmer.

6.4 Denne oppgaven er en fortsettelse av oppgave 3 der vi inkluderer inn- og utbetalinger i modellen. Utgangspunktet er modellen gitt ved differensligningen (6.13) som har løsningen (6.14).

a) Hvor lang tid tar det før kapitalen fordobler seg hvis $b = 0.1K_0$ og $r = 0.08$?

b) Hvis vi tenker oss (6.13) som en modell for tilbakebetaling av lån (annuitetslån), hvor stor må da b være for at lånet skal være nedbetalt etter N år?

c) Anta at det opprinnelige lånebeløpet er 1 million kroner, at renten er 8 % og at lånet skal være nedbetalt etter 20 år. Hvor stort blir da det årlige avdraget b? Lag et plott av lånesaldoen K_n.

6.5 I oppgave 3 studerte vi en enkel form for diskret kapitalvekst; i denne oppgaven skal vi se på kontinuerlig kapitalvekst.

a) Forklar hvordan uttrykket for $K^h(t)$ gitt ved (6.16) framkommer.

b) Vis at $K^h(t)$ vokser for en gitt t når h blir mindre. Hvorfor er dette rimelig?

c) Anta at $r = 0.08$ og at vi har 12 tidssteg pr. år slik at $h = 1/12$. Lag et plott av kapitalveksten gitt ved (6.16) over 10 år sammen med et plott av kapitalveksten gitt ved (6.10) (bruk $K_0 = 1$).

d) Verifiser at funksjonen $K(t) = K_0 e^{rt}$ er en løsning av differensialligningen (6.18) med initialverdi $K(0) = K_0$ og legg denne funksjonen inn i plottet i deloppgave (c). (For å plotte denne funksjonen kan du beregne 10 funksjonsverdier pr. år og plotte ut ved å trekke en rett linje mellom verdiene $\big((t_i, K(t_i)\big)$.)

6.6 Vis at kondisjonstallet tilfredstiller likheten

$$\kappa(f \circ g; a) = \kappa(g; a)\kappa\big(f; g(a)\big)$$

og gi en tolkning av denne relasjonen (notasjonen $f \circ g$ angir den sammensatte funksjonen som har verdien $f\big(g(x)\big)$ i x).

KAPITTEL 7

Integrasjon

Integrasjon er et helt sentralt begrep i store deler av matematikken, samtidig som integralet har uttallige anvendelser i ulike praktiske situasjoner. Her skal vi ta for oss et par aspekter av integrasjonsbegrepet som ikke er dekket i *Kalkulus*. I seksjonene 7.1 og 7.2 skal vi se litt på hvordan datamaskiner kan brukes til å beregne integraler, både numerisk og symbolsk. Dette er viktig siden mange integraler bare kan beregnes numerisk, mens de som kan beregnes symbolsk ofte er så tidkrevende å regne ut for hånd at det er svært fordelaktig å la en datamaskin gjøre jobben. Til slutt skal vi i seksjon 7.3 se at integrasjon er et grunnleggende verktøy i sannsynlighetsregning.

7.1 Symbolsk integrasjon

I *Kalkulus*, som i alle andre bøker i grunnleggende, reell analyse, er det beskrevet en del teknikker og triks for å løse ubestemte integraler. Dette er ofte både komplisert og svært regnekrevende, og det er lett å få forståelse for sitatet av Viggo Brun som sier at *Derivasjon er et håndverk, men integrasjon er en kunst!*

Siden integrasjon ofte krever mye regning er det ikke så rart at en tidlig begynte å bruke datamaskiner for å beregne integraler, med metoder basert på de teknikkene vi kjenner fra håndregningen. Med en programmeringsomgivelse som kan håndtere funksjoner og symbolske beregninger (husk på hva vi skrev om objektorientering i seksjon 2.5) er det ikke så vanskelig å implementere mange av disse metodene, siden de er klart algoritmiske av natur. Utover på 1960-tallet fikk en på denne måten utviklet gode integrasjonsprogrammer, basert på de tradisjonelle integrasjonsteknikkene. På samme tid begynte en etterhvert å se etter helt andre måter å angripe integrasjonsproblemet på, og i 1968 publiserte R. Risch en overraskende rapport der han viste at det fins en algoritme som gir løsningen på ubestemte integraler.

Utgangspunktet er at vi har en funksjon f som vi ønsker å integrere, og en klasse $\mathbb{S}$ av funksjoner der vi ønsker å lete etter løsninger. Risch ga en algoritme som avgjør om f har en antiderivert i $\mathbb{S}$ eller ikke, og hvis det fins en antiderivert i $\mathbb{S}$ så vil algoritmen også finne denne funksjonen. Algoritmen er såpass komplisert at det bare er ganske nylig at de vanlige programsystemene for symbolsk matematikk, så som Maple og Mathematica,

har fått en rimelig fullstendig implementasjon av Risch-algoritmen.

Et grunnleggende problem med det å bruke datamaskinen til å beregne ubestemte integraler er at løsningene, om de eksisterer, ofte er så kompliserte at de er uinteressante. Selv om en løsning er komplisert kan det jo hende at den kan skrives på en ekvivalent form som er enkel, men da har vi kommet til et annet problem innen dataalgebra, nemlig forenkling av uttrykk. Dette problemet er enda vanskeligere enn integrasjonsproblemet, ikke minst fordi det er svært vanskelig å definere presist hva en forenkling av et uttrykk er.

7.2 Numerisk integrasjon

Selv om vi tar datamaskinen til hjelp er det mange ubestemte integraler som enten ikke har noen løsning som kan uttrykkes ved kjente funksjoner, eller så er løsningen så komplisert at den ikke har noen praktisk interesse. Dette betyr at det å beregne bestemte integraler bare i de enkleste tilfellene gjøres ved å finne en antiderivert og så sette inn integrasjonsgrensene. Når dette ikke fungerer er alternativet å bruke numerisk integrasjon. I *Kalkulus* er de to vanligste metodene for numerisk integrasjon beskrevet, trapesmetoden og Simpsons metode. Her skal vi se litt nærmere på hvordan disse kan implementeres effektivt på datamaskin. Dette vil illustrere noen viktige prinsipper ved numerisk programmering.

7.2.1 Implementasjon av trapesmetoden

Utgangspunktet for trapesmetoden er at vi skal finne det bestemte integralet fra a til b av den kontinuerlige funksjonen f. Som i definisjonen av integralet ved trappesummer deler vi intervallet $[a, b]$ opp i delintervaller ved hjelp av en partisjon gitt ved

$$a = x_0 < x_1 < x_2 < \cdots < x_{n-1} < x_n = b.$$

Men i steden for å tilnærme f med en funksjon som er konstant på hvert delintervall bruker vi nå en tilnærming som er en rett linje på hvert delintervall, og som på intervallet $[x_{i-1}, x_i]$ forbinder punktene $\big(x_{i-1}, f(x_{i-1})\big)$ og $\big(x_i, f(x_i)\big)$, se seksjon 8.7 i *Kalkulus*. Generelt kan delintervallene godt ha varierende bredde, men vi skal bare se på tilfellet der vi deler $[a, b]$ i n like deler, hver med bredde $\Delta x = h = (b - a)/n$. Da er $x_i = a + ih$ slik at tilnærmingen til det bestemte integralet er gitt ved

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\Big(f(a) + f(b) + 2\sum_{i=1}^{n-1} f(a + ih)\Big). \tag{7.1}$$

Spørsmålet nå er bare om om vi kan stole på denne tilnærmingen? I *Kalkulus* er det gitt et feilestimat for trapesmetoden, men denne involverer den andrederiverte til f som ikke nødvendigvis er så enkel å beregne. Den vanlige teknikken består i å beregne en følge av tilnærminger med økende n og avtagende h. Denne følgen vil da konvergere mot verdien av integralet slik at når n blir stor nok kan vi bruke den aktuelle høyresiden i (7.1) som tilnærming til integralet. For å finne ut hvor stor n trenger å være er det vanlig å bruke samme teknikk som ved Newtons metode: vi stopper når to påfølgende verdier adskiller seg med mindre enn en gitt toleranse.

Dette gir oss den generelle oppskriften på det vi skal gjøre, men det er en del detaljer som må avklares. Aller først må vi bestemme oss for hvordan n skal økes fra gang til gang. I steden for å øke n med én hver gang, øker vi n slik at bredden av delintervallene halveres hver gang. Dette betyr at vi første gang bruker $h_0 = b - a$ og 2 punkter, neste gang er $h_1 = h_0/2 = (b-a)/2$ slik at vi får 3 punkter, deretter setter vi $h_2 = h_1/2 = (b-a)/4$ slik at vi får 5 punkter også videre. Etter m steg setter vi

$$h_m = h_{m-1}/2 = \frac{b-a}{2^m}$$

og bruker de $2^m + 1$ punktene

$$x_i = a + ih_m, \qquad \text{for } i = 0,\, 1,\, \ldots,\, 2^m.$$

Vi kan nå sette inn disse verdiene i (7.1) å regne ut tilnærmingen til integralet. Men for å gjøre dette på en effektiv måte må vi se litt nøyere på hva som foregår. Hvis vi kaller tilnærmingen til integralet med $2^m + 1$ punkter for T_m, så ser vi fra (7.1) at T_m er gitt ved

$$T_m = \frac{h_m}{2}\Big(f(a) + f(b) + 2\sum_{i=1}^{2^m-1} f(a + ih_m)\Big). \tag{7.2}$$

Den neste tilnærmingen T_{m+1} er gitt ved

$$T_{m+1} = \frac{h_{m+1}}{2}\Big(f(a) + f(b) + 2\sum_{i=1}^{2^{m+1}-1} f(a + ih_{m+1})\Big). \tag{7.3}$$

Hvis vi sammenligner T_m og T_{m+1} så ser vi at det er mye som er felles. Alle x_i'ene der vi beregner verdier av f i T_m er også med i uttrykket for T_{m+1}, men i tillegg får vi en ny x_i mellom hvert par av x_i'er som inngår i T_m (husk at h_{m+1} er halvparten av h_m). Dette betyr at i summen på høyre side av (7.3) er det bare de funksjonsverdiene som svarer til en odde i-verdi som er nye; alle de andre har vi beregnet før. Vi ser derfor at

$$T_{m+1} = \frac{1}{2}T_m + h_{m+1}\sum_{\substack{i=1\\ i \text{ odde}}}^{2^{m+1}-1} f(a + ih_{m+1}).$$

Denne siste summen kan vi skrive litt tydeligere. Siden vi bare skal ha odde verdier av i betyr det at det fins en j slik at $i = 2j - 1$, og når i skal variere fra 1 til $2^{m+1} - 1$ ser vi at j må variere fra 1 til 2^m. Vi har derfor

$$T_{m+1} = \frac{1}{2}T_m + h_{m+1}\sum_{j=1}^{2^m} f\big(a + (2j-1)h_{m+1}\big) \tag{7.4}$$

$$= \frac{1}{2}T_m + \frac{h_m}{2}\sum_{j=1}^{2^m} f\big(a - h_{m+1} + jh_m\big) \tag{7.5}$$

siden $h_m = 2h_{m+1}$. Basert på dette kan vi sette opp følgende kode for trapesmetoden.

Algoritme 7.1 (Trapesmetoden)**.** *La f være en funksjon som er kontinuerlig på et intervall $[a, b]$. Følgende kode vil beregne en tilnærming til integralet av f over $[a, b]$ ved hjelp av trapesmetoden. Den relative feilen vil vanligvis være omtrent* `eps` *så sant beregningene ikke stoppes av at antall iterasjoner overstiger det gitte heltallet* `nmax`*.*

```
int jmax=1, n=0, j;
double h=b-a, hg, T, Tg, xj, nyf, e;

T = 0.5*h*(f(a)+f(b));
while (n <= nmax & e > eps) {
  n = n + 1; nyf = 0.0;
  hg = h; h = 0.5*h;
  xj = a + h;
  for (j=1; j<= jmax; j++) {
    nyf = nyf + f(xj);
    xj = xj + hg;
  }
  Tg = T;
  T = 0.5*(Tg + hg*nyf);
  e = abs(Tg-T)/abs(T);
  jmax = 2*jmax;
}
```

Som tidligere har vi her brukt en Java-lignende syntaks, men hoppet over alle mulige prefiks og lignende krimskrams. Det beste er å legge koden inn i en metode som har `f`, `a`, `b`, `nmax` og `eps` som inngangsparametre og gir ut den endelige verdien av `T` (hvis vi ikke stopper fordi `n` blir større enn `nmax`) som estimat for integralet.

Som vi ser er det en del detaljer som må på plass for å få fram en rimelig presis kode. Vi begynner med å beregne en tilnærming til integralet der vi tilnærmer f med en rett linje på hele intervallet $[a, b]$ (dette svarer til tilnærmingen T_0 i diskusjonen før algoritme 7.1) slik at $h = b - a$, og lagrer denne i T. Når dette er gjort kan vi starte løkka og suksessivt halvere bredden på delintervallene. Inne i `while`-løkka begynner med å regne ut den nye verdien av h og lagre den gamle verdien av h i `hg`. Deretter summerer vi opp de funksjonsverdiene som er nye i den nye oppdelingen. Siden disse verdiene beregnes i annenhver x_i, ligger de med avstand lik den gamle intervallbredden `hg`, med start i `a+h`. Antall verdier er 1 første gang og dobles så hver gang; dette heltallet lagrer vi i `jmax`. Når verdiene er summert opp kan vi så beregne den nye tilnærmingen til integralet ved hjelp av formelen (7.5). Men før vi gjør det passer vi på å ta vare på den gamle tilnærmingen i `Tg`. Vi beregner deretter et estimat for den relative feilen fra de to siste tilnærmingene før vi oppdaterer `jmax`.

Tidligere har vi bekymret oss mye for avrundingsfeil. Når det gjelder numerisk integrasjon har vi gode nyheter i så henseende: avrundingsfeil er vanligvis ikke et problem. Den sentrale operasjonen er å summere opp verdier som for de vanligste funksjonene ikke varierer så veldig mye, det er derfor liten risiko for at vi må legge sammen to omtrent like tall med motsatt fortegn. Det som kan skape problemer med avrundingsfeil er selve

beregningen av $f(x)$, men det er et problem som ikke har noe med numerisk integrasjon å gjøre.

Vi har valgt å estimere den relative feilen, og ikke den asbolutte feilen, siden den, som vi har sett tidligere, er uavhengig av størrelsen på tallet vi beregner. Hvis vi ønsker å beregne integralet med 10 riktige siffer kan vi derfor bruke `eps` $= 0.5 \cdot 10^{-10}$. Legg merke til at vi strengt tatt bør ha med en test på om `T==0.0` før vi regner ut den relative feilen.

Det er viktig å huske på at det å estimere feilen slik vi gjør her ikke er idiotsikkert. Anta for eksempel at funksjonen vi skal integrere er

$$f(x) = 1 + (x-a)(x-c)(x-b)$$

der $c = (a+b)/2$. Da har f verdien 1 i alle de tre punktene a, c og b og det er lett å se at begge de to første estimatene for integralet blir $b-a$. Vi får derfor `e` lik `0.0` første gang vi kommer inn i det indre av while-løkka og vil derfor stoppe etter én gjennomgang med beskjed om at integralet er $b-a$, noe som åpenbart er feil. Dersom vi har mistanke om at slike funksjoner kan forekomme bør vi starte iterasjonene med m noe større enn 0 slik at sannsynligheten blir mindre for at vi åpner beregningene med å beregne 'spesielle' verdier av f.

7.2.2 Implementasjon av Simpsons metode

Trapesmetoden er basert på at vi tilnærmer f med en rett linje på hvert delintervall. Simpsons metode er litt mer raffinert og tar utgangspunkt i at vi tilnærmer f med en parabel på hvert delintervall, slik som forklart i seksjon 8.7 i *Kalkulus*. Men to punkter er ikke nok for å bestemme en parabel, vi bruker derfor også midtpunktet i hvert delintervall for å bestemme parabelen. I praksis betyr dette at vi deler opp intervallet en ekstra gang. Med Simpsons metode må vi derfor begynne med å dele opp $[a,b]$ i to delintervaller og tilnærme f med parabelen som går gjennom de tre punktene $\big(a, f(a)\big)$, $\big((a+b)/2, f((a+b)/2\big)$ og $\big(b, f(b)\big)$. Deretter halverer vi intervallene slik som for trapesmetoden.

Hvis vi gir tilnærmingen når $h = h_m = (b-a)/2^m$ navnet S_m, så har vi fra *Kalkulus* at

$$S_m = \frac{h_m}{3}\Big(f(a) + f(b) + 4\sum_{j=1}^{2^{m-1}} f\big(a + (2j-1)h_m\big) + 2\sum_{j=1}^{2^{m-1}-1} f(a+2jh_m)\Big) \qquad (7.6)$$

for $m = 1, 2, \ldots$. Estimatet S_m er altså basert på $2^m + 1$ funksjonsverdier. Vi har her spaltet opp summen ved å gruppere sammen x_i'er med odde og like verdi for i ($i = 2j-1$ og $i = 2j$), siden disse skal multipliseres med forskjellige konstanter, henholdsvis 4 og 2. Hvis vi nå halverer delintervallene og setter $h_{m+1} = h_m/2$ får vi at den neste tilnærmingen er gitt ved

$$S_{m+1} = \frac{h_{m+1}}{3}\Big(f(a) + f(b) + 4\sum_{j=1}^{2^m} f\big(a + (2j-1)h_{m+1}\big) + 2\sum_{j=1}^{2^m-1} f(a+2jh_{m+1})\Big). \qquad (7.7)$$

Funksjonsverdiene i den første summen er nye i den forstand at de ikke inngår i beregningene av S_m, mens de andre er gamle siden de også inngår i S_m. Vi legger også merke til at de 'nye' verdiene multipliseres med konstanten 4 når vi bruker dem i beregningen av S_{m+1}, mens de i senere beregninger alltid vil bli multiplisert med 2 siden de da vil være 'gamle'. Hvis vi lar r_{m+1} betegne summen av alle funksjonsverdier med odde faktor foran h_{m+1} (summen av de 'nye' funksjonsverdiene),

$$r_{m+1} = \sum_{j=1}^{2^m} f\big(a + (2j-1)h_{m+1}\big),$$

så ser vi at S_m og S_{m+1} er relatert gjennom

$$S_{m+1} = \frac{1}{2}S_m - \frac{h_m}{3}r_m + 4\frac{h_{m+1}}{3}r_{m+1}.$$

Siden $h_{m+1} = h_m/2$ kan denne formelen forenkles til

$$S_{m+1} = \frac{1}{2}S_m + \frac{h_m}{3}(2r_{m+1} - r_m). \tag{7.8}$$

På bakgrunn av dette kan vi sette opp en detaljert algoritme for Simpsons metode.

Algoritme 7.2 (Simpsons metode)**.** *La f være en funksjon som er kontinuerlig på et intervall $[a, b]$. Følgende kode vil beregne en tilnærming til integralet av f over $[a, b]$ ved hjelp av Simpsons metode. Den relative feilen vil vanligvis være omtrent* `eps` *så sant beregningene ikke stoppes av at antall iterasjoner overstiger det gitte heltallet* `nmax`*.*

```
int jmax=2, n=0, j;
double h, hg, S, Sg, xj, e, r, rg;

h=0.5*(b-a);
xj = a + h; r = f(xj);
S = h*(f(a)+4*r+f(b))/3;
while (n <= nmax & e > eps) {
  n = n + 1;
  rg = r; r = 0.0;
  hg = h; h = 0.5*h;
  xj = a + h;
  for (j=1; j<= jmax; j++) {
    r = r + f(xj);
    xj = xj + hg;
  }
  Sg = S;
  S = 0.5*Sg + hg*(2*r-rg)/3;
  e = abs(Sg-S)/abs(S);
  jmax = 2*jmax;
}
```

Kommentarene i forbindelse med trapesmetoden er også aktuelle her, men legg merke til at vi nå må starte med `jmax=2`. De observante vil kanskje synes at det ser litt skummelt ut med differansen `r-rg` i uttrykket for `S`, men husk at summen som definerer `r` er dobbelt så lang som summen som definerer `rg` så det skal mye til at `r` og `rg` er omtrent like store og dermed gir kansellering og tap av nøyaktighet.

7.2.3 Valg av metode

Med to metoder tilgjengelig for numerisk integrasjon er spørsmålet hvilken vi skal velge? Som vi vet fra *Kalkulus* er Simpsons metode mer nøyaktig enn trapesmetoden når funksjonen vi skal integrere har kontinuerlig fjerdederivert. Selv om denne metoden er litt mer regnekrevende faller derfor valget som regel på Simpsons metode. Unntaket er om funksjonen vi skal integrere ikke har så mange som 4 kontinuerlige deriverte; da er det som oftest bedre å bruke trapesmetoden.

Det bør også nevnes at det fins metoder som er enda mer nøyaktige enn Simpsons metode. Disse er basert på at vi tilnærmer f lokalt med polynomer av høyere grad enn 2. Vi kan for eksempel bruke 4 funksjonsverdier og tilnærme f med et tredjegrads polynom. Generelt kan vi bruke $k+1$ funksjonsverdier og tilnærme f med et polynom av grad k og få en metode som er svært nøyaktig for funksjoner som har $2k$ kontinuerlige deriverte. Rombergintegrasjon er en generell metode som starter med trapesmetoden og deretter på en enkel måte beregner tilnærminger basert på polynomer av stadig høyere grad, inntil det ikke er flere funksjonsverdier tilgjengelig.

7.3 Integrasjon og sannsynlighet

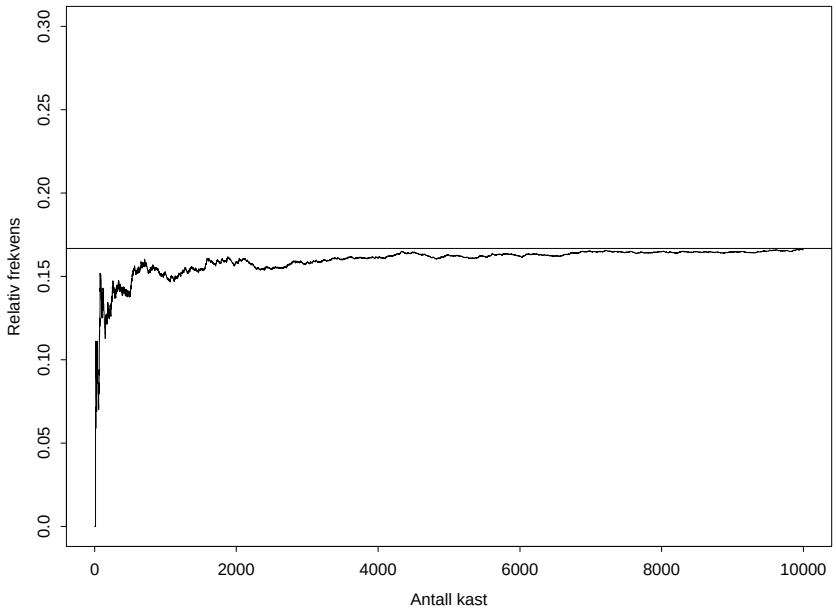
I denne seksjonen skal vi se hvordan integralregningen kan brukes til å beregne sannsynligheter, og vi skal også se nærmere på stokastisk simulering. Men aller først tar vi en rask repetisjon av sannsynlighetsbegrepet, og hva vi mener med en stokastisk (tilfeldig) variabel.

7.3.1 Hva er sannsynlighet?

På kalendere er tidspunktene for fullmåne ofte angitt, men hvordan er det mulig å vite dette på forhånd, lenge før året har begynt? Bakgrunnen er at astronomene ved hjelp av matematiske ligninger kan gi en nøyaktig beskrivelse av himmellegemenes bevegelser. Dermed kan de også regne ut nøyaktig når vi får fullmåne eller neste solformørkelse. Hendelser som fullmåne og solformørkelse kan med andre ord forutsies — de er *deterministiske*.

Når vi kaster en terning, vet vi ikke på forhånd hvor mange øyne vi får. Vi sier derfor at terningkast er et *stokastisk* (eller tilfeldig) forsøk. Et annet stokastisk forsøk er det når vi ser om et nyfødt barn er en gutt eller en jente. For heller ikke her vet vi resultatet på forhånd (hvis ikke barnets kjønn er blitt avklart i løpet av svangerskapet ved en kromosomtest eller en ultralydundersøkelse). Et kjennetegn på et stokastisk forsøk[1], er

[1]Merk at vi bruker ordet 'forsøk' i en videre betydning enn det som er vanlig ved laboratorieøvelser i biologi, fysikk og kjemi.



Figur 7.1. Relativ frekvens av seksere i 10000 terningkast.

altså at vi ikke på forhånd kan si hva resultatet vil bli, vi vet bare hvilke resultater som *kan* forekomme.

En terning har seks sider. På grunn av symmetrien til terningen har alle disse sidene like stor sjanse for å vende opp etter et terningkast. Sannsynligheten er derfor 1/6 for å få en sekser. Men hva betyr egentlig dette? Det betyr *ikke* at hvis vi kaster en terning 6 ganger, så vil vi få nøyaktig én sekser. Det *betyr* at hvis vi kaster mange ganger, vil vi få sekser i omtrent en sjettedel eller 16.7% av kastene.

For å illustrere dette har vi (ved hjelp av datamaskin) kastet en terning 10000 ganger. Etter N kast er den *relative frekvensen* av seksere gitt som antall seksere i de N første kastene dividert med N. Figur 7.1 viser den relative frekvensen som funksjon av N. Vi ser at variasjonen i den relative frekvensen er ganske stor til å begynne med, men etterhvert stabiliserer den seg nær $1/6 = 0.167$.

Hva er sannsynligheten for at et nyfødt barn er en jente? Noen vil kanskje tro at det blir født like mange gutter som jenter slik at sannsynligheten er 50%. Hvis du ser etter i Statistisk årbok (`www.ssb.no/aarbok/`) vil du se at dette ikke er tilfelle. Hvert år blir det født litt færre jenter enn gutter, og fordelingen mellom kjønnene er forholdsvis konstant fra år til år. For perioden 1994–1998 varierte andelen jenter hvert år mellom 48.3% og 48.8%. At variasjonen er så liten skyldes at det er mange fødsler — omtrent 60000 — hvert eneste år. I hele perioden 1994–1998 ble det født 299464 barn i Norge, og av disse var 145438 jenter. Den relative frekvensen av jentefødsler i femårsperioden var

derfor $145438/299464 = 0.486$.

Den relative frekvensen av seksere vil være omtrent $1/6$ når vi kaster en terning mange ganger, og den relative frekvensen av jenter blant alle nyfødte er hvert år omtrent 48.6%. Grunnlaget for begge disse utsagnene er at vi har gjentatt 'forsøkene' (kaste trening, observere kjønn til nyfødt barn) mange ganger. Denne muligheten for å gjøre mange gjentagelser av det samme 'forsøket' danner fundamentet for sannsynlighetsbegrepet, og følgende er en enkel og uformell definisjon[2] av sannsynlighet som er tilstrekkelig for våre formål:

Vi er interessert i en begivenhet (eller hendelse) A som er knyttet til et stokastisk forsøk som gjentas under like betingelser. Den relative frekvensen av begivenheten vil nærme seg en grenseverdi når forsøket gjentas mange ganger, og denne grenseverdien er sannsynligheten $\mathrm{P}(A)$ *for begivenheten A.*

Fra definisjonen av relativ frekvens ser vi at sannnsynligheten $\mathrm{P}(A)$ alltid vil være et tall i intervallet $[0, 1]$.

Rent språklig er ordet sannsynlighet knyttet til ett forsøk. Vi sier at sannsynligheten for å få sekser ved ett terningkast er $1/6$ og at sannsynligheten for at et nyfødt barn skal være en jente er 0.486. Det vi egentlig uttaler oss om er imidlertid ikke et enkelt kast eller en enkelt fødsel, men hva som vil skje i 'det lange løp'. I det lange løp vil 16.7% av terningkastene gi sekser og 48.6% av de nyfødte vil være jenter.

7.3.2 Stokastiske variable

Når vi kaster to terninger, spiller det ofte ingen rolle om vi får en firer og en femmer eller om vi får en treer og en sekser. Det som betyr noe, er at summen av antall øyne er ni. På lignende måte kan vi for en trebarns familie være interessert i hvor mange gutter og jenter det er i søskenflokken, uten at vi er interessert i kjønnet til den eldste, nesteldste eller yngste.

Det som kjennetegner disse to situasjonene, er at vi er interessert i en tallstørrelse knyttet til resultatet av et stokastisk forsøk. En slik tallstørrelse kalles en *stokastisk variabel* (eller tilfeldig variabel). Stokastiske variable betegnes gjerne med store bokstaver fra slutten av (det engelske) alfabetet, for eksempel X og Y. Ved kast med to terninger er $X =$"sum antall øyne" en stokastisk variabel, og $Y =$"antall gutter" er en stokastisk variabel for forsøket som består i å se hvilke(t) kjønn barna har i en tilfeldig valgt trebarns familie.

For en stokastisk variabel X er vi ofte interessert i funksjonen

$$p(x) = \mathrm{P}(X = x). \tag{7.9}$$

Denne funksjonen er definert for de x-verdiene variabelen kan anta og kalles *punktsannsynligheten* til X. Vi har $0 \leq p(x) \leq 1$ for alle x og $\sum_x p(x) = 1$ (siden X alltid må anta

[2]Vi kan ikke bruke grenseverdien av den relative frekvensen som en matematisk definisjon av sannsynlighet. Den grenseverdien vi snakker om her er empirisk (eksperimentell) og er ikke en grenseverdi i samme forstand som for en matematisk tallfølge som $\{1/n\}$. Når vi skal gi en presis matematisk definisjon av sannsynlighet, gjøres derfor dette ved å sette opp noen aksiomer som sannsynlighetsbegrepet skal tilfredsstille. Men motivasjonen for disse aksiomene kommer blant annet fra fortolkningen av sannsynlighet som relativ frekvens ved mange forsøk.

en av de mulige verdiene).

Hvis vi går tilbake til forsøkene våre, så ser vi at hvis $X =$ "sum antall øyne" ved kast med to terninger, er punktsannsynligheten gitt ved tabellen (sjekk selv)

x	2	3	4	5	6	7	8	9	10	11	12
$p(x)$	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$

Hvis derimot $Y =$ "antall gutter" i en trebarns familie viser det seg at en rimelig modell er gitt ved den binomiske punktsannsynligheten

$$p(y) = \binom{3}{y} 0,514^y \, 0,486^{3-y}$$

for $y = 0$, 1, 2, 3.

De stokastiske variablene vi har sett på så langt, kan bare anta et endelig antall verdier og sies derfor å være *diskrete*. Men i mange sammenhenger har vi stokastiske variable som kan anta et kontinuerlig spekter av verdier — vi sier at vi har *kontinuerlige* stokastiske variable. Dette betyr at variabelen i prinsippet kan anta alle verdier i et intervall på tallinja (eventuelt på hele tallinja). Noen eksempler på dette er

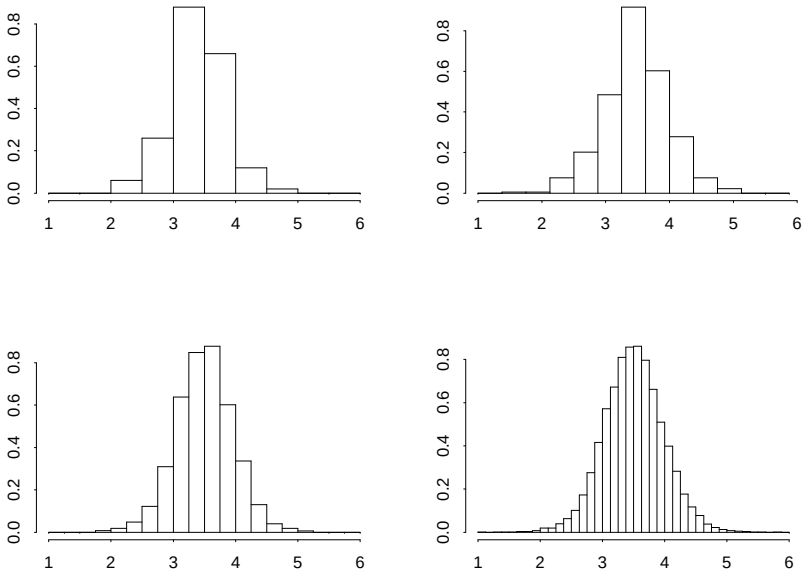
- vekten til en nyfødt jente,
- høyden til en norsk rekrutt,
- tiden mellom to oppringninger til en telefonsentral,
- endringen i en aksjekurs i løpet av en dag.

I praksis vil ikke vekten til en nyfødt jente bli målt mer nøyaktig enn til nærmeste tiende gram, og høyden til en rekrutt vil bare måles til nærmeste hele (eller halve) centimeter. Så selv om vekt og høyde i prinsippet kan anta alle verdier i et intervall, er det på grunn av avrunding bare et endelig antall forskjellige verdier en vil registrere. Vi kunne derfor valgt å se på vekt og høyde som diskrete stokastiske variable, men det viser seg å være mest hensiktsmessig å betrakte disse som kontinuerlige variable. Dette vil også være tilfellet for endringen i en aksjekurs, selv om denne ikke en gang i prinsippet kan anta alle verdier i et intervall. Dette er tilsvarende som i mange andre situasjoner ved matematisk modellering: om vi benytter en diskret eller en kontinuerlig modell er ofte et spørsmål om hva som er (matematisk og/eller numerisk) mest hensiktsmessig.

7.3.3 Sannsynlighetstetthet – et motiverende eksempel

For en diskret stokastisk variabel X kan vi angi fordelingen til variabelen ved å oppgi sannsynligheten $\mathrm{P}(X = x)$ for alle mulige verdier av x i en tabell eller ved en formel, slik vi gjorde i eksemplene over. Hvis X er en kontinuerlig stokastisk variabel er $\mathrm{P}(X = x) = 0$ for alle x, se (7.11) nedenfor. Vi må derfor angi fordelingen til X på en annen måte.

For å se hvordan vi kan angi fordelingen til en kontinuerlig stokastisk variabel bruker vi variabelen $V =$ "vekt til nyfødt jente" som eksempel, og benytter data fra Medisinsk



Figur 7.2. Histogram av fødselsvekter for "fullbårne" jenter født i Norge i 1980. Histogrammene er basert på ulike klassebredder og antall registreringer av fødselsvekter: øverst til venstre 100 vekter, øverst til høyre 500 vekter, nederst til venstre 2500 vekter og nederst til høyre 20000 vekter.

fødselsregister om fødselsvekter til jenter født i Norge i 1980. Vi vil bare se på "fullbårne" jenter, så vi begrenser oss til fødsler der svangerskapet varte mellom 37 og 43 uker.

Vi ser først på et tilfeldig utvalg av 100 nyfødte jenter. Et histogram av fødselsvektene til disse er gitt øverst til venstre i Figur 7.2. Vi bruker her klassebredde 0.5 kg, hvilket betyr at vi deler inn fødselsvektene i intervaller som er 0.5 kg brede når vi lager histogrammet. Histogrammet er normert slik at *arealet* av en søyle er lik den relative frekvensen av fødselsvekter i det intervallet søylen dekker. Merk at den relative frekvensen av fødselsvekter mellom for eksempel 2.0 kg og 3.5 kg er summen av de relative frekvensene av fødselsvekter mellom 2.0 kg og 2.5 kg, mellom 2.5 kg og 3.0 kg og mellom 3.0 kg og 3.5 kg, altså det totale arealet under histogrammet mellom 2.0 kg og 3.5 kg.

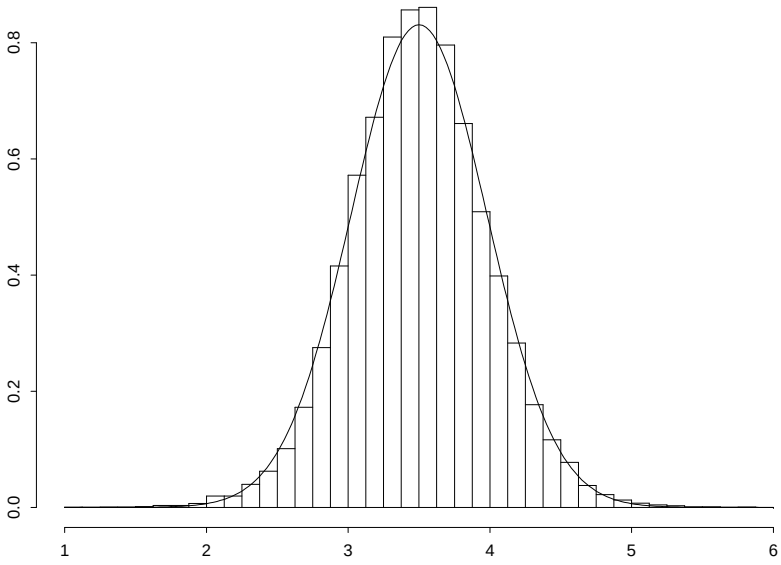
Vi ser så på et histogram til fødselsvektene av et tilfeldig utvalg av 500 jenter slik som vist øverst til høyre i Figur 7.2. Siden vi nå har flere fødselsvekter, reduserer vi klassebredden til 0.375 kg. Den relative frekvensen av fødselsvekter mellom 2.0 kg og 3.5 kg er nå summen av de relative frekvensene av fødselsvekter i intervallene 2.0–2.375 kg, 2.375–2.75 kg, 2.75–3.125 kg og 3.125–3.50 kg. Igjen svarer dette til arealet under histogrammet mellom 2.0 kg og 3.5 kg.

Nederst til venstre i Figur 7.2 har vi gitt et histogram til fødselsvektene av et tilfeldig utvalg på 2500 jenter. Her er klassebredden 0.25 kg. Vi merker at også nå er den relative frekvensen av fødselsvekter mellom 2.0 kg og 3.5 kg lik arealet under histogrammet mellom 2.0 kg og 3.5 kg.

Vi ser endelig på et histogram til fødselsvektene av 20000 jenter med klassebredde 0.125 kg. Dette er gitt nederst til høyre i Figur 7.2. Som i de andre tilfellene er den relative frekvensen av fødselsvekter mellom 2.0 kg og 3.5 kg lik arealet under histogrammet mellom 2.0 kg og 3.5 kg.

Vi ser fra histogrammene at når vi øker antall fødselsvekter så får vi en mer nøyaktig oversikt over de relative hyppighetene av ulike fødseksvekter, siden vi kan bruke mindre klassebredde når vi har mange observasjoner. Dessuten merker vi oss at histogrammene er laget slik at den relative frekvensen av fødselsvekter i et intervall er lik arealet under histogrammet over dette intervallet. Men kanskje det mest iøynefallende med plottene i figur 7.3 er hvordan histogrammene ser ut til å nærme seg en underliggende glatt funksjon. Det er rimelig å anta at dersom vi kunne legge til stadig nye fødselsvekter så ville histogrammet komme nærmere og nærmere denne funksjonen, men dette er det selvsagt ikke mulig å sjekke siden vi bare har et endelig antall fødselsvekter til rådighet. En statistiker vil allikevel, som en modell, tenke seg at når antall fødselsvekter øker, vil histogrammene nærme seg en funksjon $f(v)$. Figur 7.3 viser denne funksjonen sammen med histogrammet av de 20000 fødselsvektene. Funksjonen $f(v)$ kalles *sannsynlighetstettheten* til den stokastiske variabelen V = "vekt til nyfødt jente". Ved å erstatte histogrammene med sannsynlighetstettheten $f(v)$ går vi i en viss forstand til grensen og får et histogramm med klassebredde på 0 kg, basert på uendelig mange fødsler, helt analogt med hvordan vi definerer integralet ved hjelp av trappesummer over stadig mindre intervaller.

Når vi har mange fødselsvekter vil den relative frekvensen av vekter mellom 2.0 kg og 3.5 kg være nær sannsynligheten for at en jente skal ha en fødselsvekt i dette intervallet, i følge vår uformelle definisjon av sannsynlighet i avsnitt 7.3.1. Siden histogrammene våre vil være nær sannsynlighetstettheten $f(v)$ når vi har mange observasjoner, vil sannsyn-



Figur 7.3. Histogram av fødselsvekter for 20000 jenter med inntegnet sannsynlighetstetthet.

ligheten for en fødselsvekt mellom 2.0 kg og 3.5 kg være lik arealet under sannsynlighetstettheten over intervallet fra 2.0 kg til 3.5 kg. Men dette arealet vet vi er gitt ved integralet av $f(v)$ over dette intervallet. Konklusjonen er derfor at sannsynligheten for at en nyfødt jente skal veie mellom 2.0 kg og 3.5 kg er gitt ved

$$\mathrm{P}(2.0 \leq V \leq 3.5) = \int_{2.0}^{3.5} f(v)\, dv.$$

Hvis vi er interessert i sannsynligheten for en fødselsvekt mellom a og b kan vi finne denne ved å bruke a og b som integrasjonsgrenser i steden for 2.0 og 3.5.

7.3.4 Sannsynlighetstettheter og kumulative fordelinger

I foregående avsnitt så vi at sannsynligheten for at den stokastiske variabelen $V =$ "vekt til nyfødt jente" skal anta en verdi mellom a og b er lik integralet av sannsynlighetstettheten $f(v)$ over dette intervallet. Vi vil nå se litt mer generelt på kontinuerlige stokastiske variable og lar X være en kontinuerlig stokastisk variabel med sannsynlighetstetthet $f(x)$. For at en funksjon f skal kunne kalles en sannsynlighetstetthet må den tilfredstille et par betingelser. Siden sannsynligheter aldri kan bli negative kan ikke $f(x)$ være negativ for noen x. Dessuten må vi ha $\int_{-\infty}^{\infty} f(x)dx = 1$ siden vi med full sikkerhet kan si at den stokastiske variabelen alltid vil anta en eller annen verdi på tallinjen. Som nevnt over er det slik hvis $a < b$ så er sannsynligheten for at X skal anta en verdi mellom

a og b gitt ved

$$\mathrm{P}(a \leq X \leq b) = \int_a^b f(x)dx. \tag{7.10}$$

Spesielt har vi at

$$\mathrm{P}(X = a) = \int_a^a f(x)dx = 0 \tag{7.11}$$

for ethvert tall a. Med andre ord er sannsynligheten 0 for at X skal anta en på forhånd angitt verdi a. Dette kan virke litt underlig, men ved nærmere ettertanke er det ikke så rart. Husk at den stokastiske variabelen angir resultatet av et tilfeldig 'forsøk' der resultatet er et reelt tall. Et reelt tall kan vi tenke på som et desimaltall med uendelig mange siffer til høyre for desimalkommaet, og det synes helt utenkelig at vi i et forsøk skulle kunne matche alle de uendelig mange sifrene i a.

Den *kumulative fordelingsfunksjonen* til X er gitt ved den antideriverte til sannsynlighettettheten f,

$$F(x) = \mathrm{P}(X \leq x) = \int_{-\infty}^x f(u)du. \tag{7.12}$$

Legg merke til at siden en sannsynlighetstetthet er ikke-negativ så vil den kumulative fordelingsfunksjonen alltid være en voksende funksjon. Vi kan bruke den kumulative fordelingsfunksjonen til å finne sannsynligheten for at X ligger i et intervall siden vi fra egenskaper ved integralet har relasjonen

$$\mathrm{P}(a \leq X \leq b) = F(b) - F(a).$$

Formelen (7.12) viser hvordan vi kan finne den kumulative fordelingsfunksjonen fra sannsynlighetstettheten. Ved analysens fundamentalteorem (teorem 8.3.3 i *Kalkulus*) har vi omvendt at $f(x) = F'(x)$.

Enhver ikke-negativ funksjon f_0 gir opphav til en sannsynlighetstetthet f ved formelen

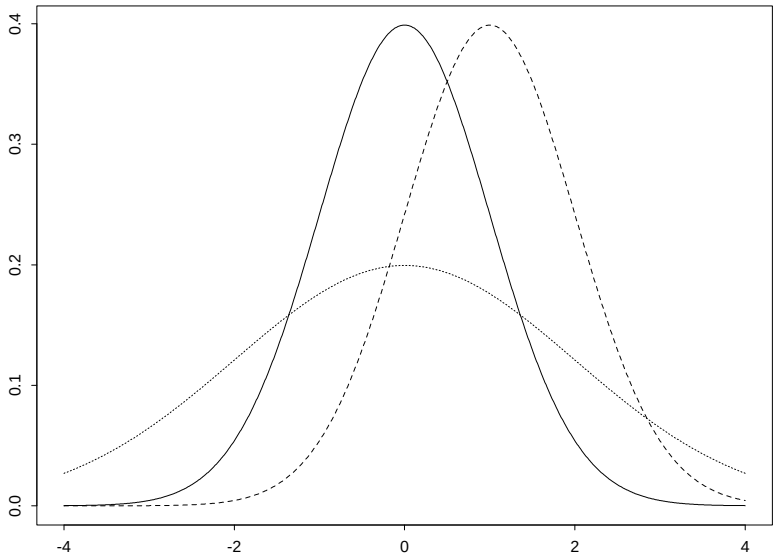
$$f(x) = \frac{f_0(x)}{\int_{-\infty}^{\infty} f_0(x)\,dx},$$

for vi ser at f, i tillegg til å være ikke-negativ slik som f_0, også har integral 1. Siden vi vet at det fins uendelig mange ikke-negative funksjoner har vi derfor et stort utvalg av sannsynlighetstettheter. På den annen side er det enkelte sannsynlighetstettheter som går igjen i mange ulike sammenhenger, og derfor er spesielt viktige. Vi skal se på tre av disse her.

Normalfordelingen Normalfordelingen spiller en sentral rolle i sannsynlighetsregningen. Den kalles også den gaussiske fordelingen etter Carl Friedrich Gauss som foreslo normalfordelingen som en modell for målefeil.

Vi sier at X er normalfordelt med forventning μ og standardavvik σ hvis sannsynlighetstettheten er gitt ved

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}}\, e^{-(x-\mu)^2/(2\sigma^2)}. \tag{7.13}$$



Figur 7.4. Tre normalfordelingstettheter: (i) $\mu = 0$, $\sigma = 1$ (heltrukket linje); (ii) $\mu = 1$, $\sigma = 1$ (stiplet linje); (iii) $\mu = 0$, $\sigma = 2$ (prikket linje).

Figur 7.4 viser normalfordelingstettheten for tre valg av μ og σ. Legg merke til at μ gir plasseringen av toppunktet til $f(x)$, mens σ er et mål for hvor "bred" sannsynlighetstettheten er. Dette betyr at tall i nærheten av μ er de mest sannsynlige verdiene for X, mens σ forteller oss hvor stor variasjon vi kan forvente om vi genererer mange verdier fra fordelingen. Mer presist går det an å vise at hvis vi trekker mange verdier i henhold til normalfordelingen (7.13) så vil omtrent 2/3 av disse ligge innfor intervallet $[\mu - \sigma, \mu + \sigma]$. Normalfordelingen er viktig fordi mange stokastiske variable, så som målefeil, er normalfordelt. Dessuten er normalfordelingen nyttig for å tilnærme andre fordelinger.

Den sannsynlighetstettheten vi brukte i forrige avsnitt for fødselsvekten til en fullbåren jente er gitt ved (7.13) med $\mu = 3.50$ og $\sigma = 0.48$. Fødselsvekten til en jente er altså normalfordelt med forventning 3.50 kg og standardavvik 0.48 kg.

Fra (7.10) og (7.13) har vi

$$\mathrm{P}(a \leq X \leq b) = \int_a^b \frac{1}{\sigma\sqrt{2\pi}} e^{-(u-\mu)^2/(2\sigma^2)} du. \tag{7.14}$$

Med $a = 2.00$, $b = 3.50$, $\mu = 3.50$ og $\sigma = 0.48$, blir (7.14) sannsynligheten for at en jente skal ha en fødselsvekt mellom 2.0 kg og 3.5 kg. Det er ikke mulig å bestemme integralet (7.14) analytisk, så vi må bruke numeriske metoder for å beregne slike sannsynligheter[3], se oppgave 3.

[3]Det har i lang tid vært utarbeidet tabeller over den kumulative normalfordelingen med $\mu = 0$ og $\sigma = 1$ (standard normalfordelingen). Slike tabeller kan brukes til å bestemme integralet (7.14) numerisk.

Den uniforme fordelingen I avsnitt 4.3 så vi hvordan vi kan generere tilfeldige tall mellom 0 og 1. Men hva betyr det egentlig å trekke tilfeldige tall?

Siden alle relle tall mellom 0 og 1 (i prinsippet) er mulige verdier når vi trekker et tilfeldig tall, er et tilfeldig tall en kontinuerlig stokastisk variabel U som tar verdier i intervallet $(0, 1)$. Men da er sannsynligheten null for at U for eksempel er *nøyaktig* lik $1/2$ eller $\pi/4$ (husk (7.11)). At vi trekker et tilfeldig tall mellom 0 og 1 betyr derfor *ikke* at alle tall mellom 0 og 1 er like sannsynlige verdier. Det *betyr* at sannsynligheten er δ for at U skal ligge i et intervall av lengde δ, uansett hvor mellom 0 og 1 dette intervallet er plassert.

Konklusjonen er dermed at et tilfeldig tall mellom 0 og 1 svarer til en kontinuerlig stokastisk variabel U som har sannsynlighetstettheten

$$g(u) = \begin{cases} 1, & \text{hvis } 0 < u < 1, \\ 0, & \text{ellers.} \end{cases}$$

Vi sier at U er uniformt fordelt over $(0, 1)$. Fra definisjonen (7.12) ser vi at den kumulative fordelingsfunksjonen i dette tilfellet er gitt ved

$$G(u) = \begin{cases} 0, & \text{hvis } u \leq 0, \\ u, & \text{hvis } 0 < u < 1, \\ 1, & \text{hvis } u \geq 1. \end{cases} \tag{7.15}$$

Når vi trekker tilfeldige tall på en datamaskin regner vi vanligvis med flyttall slik at det bare er et endelig antall mulige tall som kan trekkes. Sannsynligheten for å få en bestemt verdi er derfor ikke 0 i dette tilfellet, så fordelingen er strengt tatt ikke kontinuerlig. Hvis vi for eksempel regner med fire gjeldende siffer, vil alle tall i intervallet $[0.49995, 0.50005)$ blir rundet av til 0.5000. Sannsynligheten for at U er 0.5000 når vi regner med fire gjeldende siffer er derfor lik $\mathrm{P}(0.49995 \leq U < 0.50005) = 0.0001$.

Eksponensialfordelingen En kontinuerlig stokastisk variabel T med sannsynlighetstetthet gitt ved

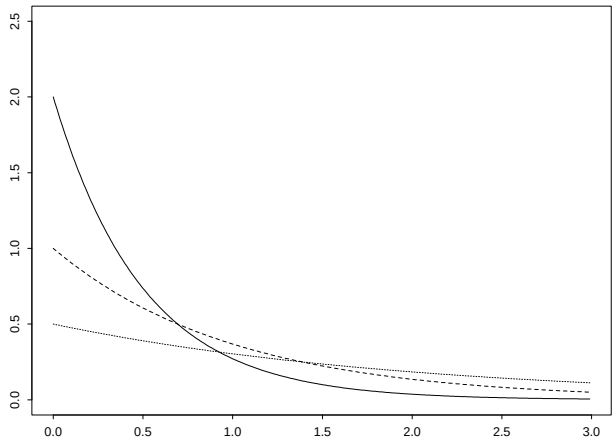
$$h(t) = \begin{cases} \lambda e^{-\lambda t} & \text{hvis } t > 0 \\ 0 & \text{hvis } t \leq 0 \end{cases}$$

sies å være eksponensialfordelt med parameter λ. Figur 7.5 viser dennne sannsynlighetstettheten for tre verdier av λ.

Eksponensialfordelingen brukes i studier av levetider for tekniske komponeneter. Den brukes også ved analyse av teletrafikk, for eksempel til å beskrive hvor lang tid det går mellom to oppringninger til en telefonsentral. I det siste tilfellet vil parameteren λ angi forventet antall oppringninger pr. tidsenhet.

7.3.5 Stokastisk simulering

I seksjon 4.3 så vi hvordan vi kan generere tilfeldige tall mellom 0 og 1. Mer presist var det vi gjorde å simulere uniformt fordelte stokastiske variable. Ofte vil vi være interessert



Figur 7.5. Tre forskjellige varianter av eksponensialfordelingen: $\lambda = 2$ (heltrukken linje), $\lambda = 1$ (grovstiplet linje), $\lambda = 0.5$ (finstiplet linje).

i å simulere stokastiske variable som har en annen fordeling. Hvis vi for eksempel ønsker å simulere trafikken til en telefonsentral, må vi generere eksponensialfordelte stokastiske variable, se oppgave 6.

Vi ønsker å generere en stokastisk variabel Y med en bestemt kumulativ fordeling $F(y)$ (for eksempel kan $F(y)$ være den kumulative eksponensialfordelingen). Vi antar at $F(y)$ er strengt voksende (bortsett fra muligens for verdier av y hvor $F(y) = 0$ eller $F(y) = 1$) slik at den omvendte funksjonen $F^{-1}(u)$ er definert for $0 < u < 1$.

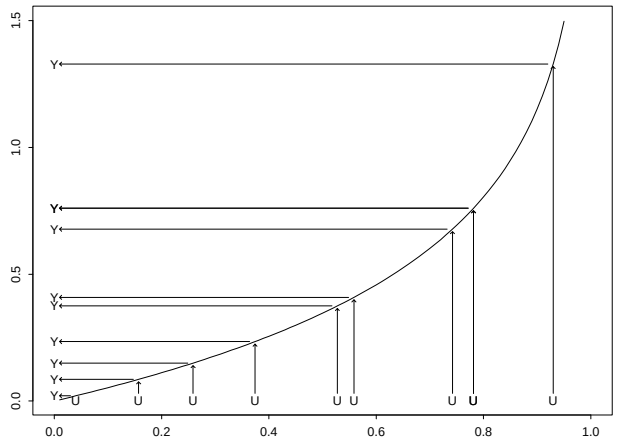
Vi antar at vi er i stand til å generere en stokastisk variabel U som er uniformt fordelt over intervallet $(0, 1)$. Vi kan da generere Y ved $Y = F^{-1}(U)$. (Merk at siden U er en stokastisk variabel, så vil også $Y = F^{-1}(U)$ være det.) For å se at dette stemmer må vi vise at Y har den ønskede kumulative fordelingen $F(y)$. Vi må altså vise at $\mathrm{P}(Y \leq y) = F(y)$. Men dette følger fra egenskaper for omvendte funksjoner,

$$\mathrm{P}(Y \leq y) = \mathrm{P}(F^{-1}(U) \leq y) = \mathrm{P}(U \leq F(y)).$$

Her er høyre side den kumulative fordelingen til U, regnet ut for $u = F(y)$. Dermed gir (7.15) at $\mathrm{P}\big(U \leq F(y)\big) = G\big(F(y)\big) = F(y)$. Altså er $\mathrm{P}(Y \leq y) = F(y)$, så Y har kumulativ fordeling $F(y)$ slik vi ønsket å vise.

For å generere kontinuerlige stokastiske variable med gitt fordeling, kan vi dermed bruke følgende framgangsmåte[4]: Først genererer vi tilfeldige tall mellom 0 og 1 slik det ble beskrevet i avsnitt 4.3 (eller med en annen metode). Deretter transformerer vi disse til den ønskede fordelingen ved hjelp av den omvendte funksjonen til den kumulative fordelingen. Figur 7.6 gir en illustrasjon av denne framgangsmåten.

[4]Denne måten å generere stokastiske variable på, fungerer fint for alle fordelinger hvor det er lett å finne den omvendte funksjonen til den kumulative fordelingen. Hvis dette ikke er tilfelle (slik det er for normalfordelingen) må en finne en tilnærmelse til den omvendte funksjonen eller benytte andre teknikker.



Figur 7.6. Simulering av 10 eksponensialfordelte variable med $\lambda = 2$.

Oppgaver

7.1 Programmer trapesmetoden og test den på integralet

$$\int_0^1 x^2\,dx = \frac{1}{3}. \tag{7.16}$$

7.2 Programmer Simpsons metode og test den på integralet (7.16).

7.3 Bruk (7.14) og numerisk integrasjon til å finne sannsynligheten for at en nyfødt jente skal veie mellom mellom 2.0 kg og 3.5 kg.

7.4 I denne oppgaven skal vi beregne den kumulative fordelingsfunksjonen $F(x)$ for normalfordelingen gitt ved sannsynlighetstettheten (7.13) når $\mu = 0$ og $\sigma = 1$ (standard normalfordeling).

a) Forklar hvorfor

$$F(x) = \frac{1}{2} + \int_0^x f(x)\,dx.$$

b) Bruk numerisk integrasjon med Simpsons metode til å bestemme den kumulative normalfordelingen $F(x)$ for $x = 0.5$, 1.0, 1.5 og 2.0. Sammenlign resultatene med det du får ved å bruke en lommeregner eller eksisterende tabeller.

c) Lag et plott av $F(x)$ på intervallet $[-3, 3]$.

7.5 En stokastisk variabel V er uniformt fordelt over (a, b) hvis den har sannsynlighetstetthet

$$f(v) = \begin{cases} 1/(b-a), & \text{hvis } a < v < b, \\ 0, & \text{ellers.} \end{cases}$$

a) Bestem den kumulative fordelingen $F(v)$ og den omvendte funksjonen til denne.

b) Hvordan kan du generere variable som er uniformt fordelt over (a, b)?

7.6 La T være eksponensialfordelt med parameter λ.

a) Bestem den kumulative fordelingen $F(t)$ og den omvendte funksjonen til denne.

b) Hvordan kan du generere eksponensialfordelte variable?

c) Skriv et program som genererer eksponensialfordelte stokastiske variable med parameter $\lambda = 2$. Ta utgangspunkt i generatoren (4.11) hvor a, c og M er gitt som i oppgave 4.6 eller en rutine i din programmeringsomgivelse som genererer uniformt fordelte tilfeldige tall i intervallet $(0, 1)$. Generer 10 eksponensialfordelte verdier.

Den eksponensialfordelte variabelen med $\lambda = 2$ kan vi tenke oss svarer til tiden mellom to oppringninger til et sentalbord med trafikkintensitet $\lambda = 2$ samtaler per minutt. De 10 verdiene $(t_i)_{i=1}^{10}$ gir da tidsintervallet mellom 10 telefonsamtaler slik at det totale tidsintervallet vi ser på blir $t_1 + t_2 + \cdots + t_{10}$.

KAPITTEL 8

Differensialligninger

Svært mange fenomener i våre omgivelser kan modelleres matematisk ved hjelp av differensialligninger. De vanligste og mest kjente eksemplene på dette finner vi i de tradisjonelle naturvitenskapene, og da særlig i fysikk. Men utbredelsen av datateknologi har gitt nye anvendelser, ofte i kombinasjon med naturvitenskapene. Anta for eksempel at du arbeider med et treningsprogram for kirurger der brukeren skal 'operere' på en 'pasient' som bare er representert grafisk (ved hjelp av matematikk) på skjermen. I de fleste kirurgiske inngrep er det behov for å sy med nål og tråd. En viktig del av et slikt program vil derfor være å gi brukeren tilgang til en virtuell ('kunstig') nål med tråd og muligheter for interaktiv bevegelse av nåla. Nåla styres altså av brukeren, og utfordingen er å vise hvordan sytråden beveger seg når nåla beveges. Fysikken bak dette er velkjent og kan beskrives med en passende differensialligning. Ved å la trådens bevegelse følge løsningen til denne ligningen vil vi få en oppførsel som brukeren oppfatter som naturlig. En viktig del av simuleringen av sytråden er derfor å løse differensialligningen som beskriver trådens bevegelse. Løsningen må være rimelig nøyaktig, men framfor alt rask å beregne, siden trådens bevegelse skal vises i sann tid på dataskjermen.

I eksempelet over er det viktig at tråden modelleres riktig rent fysisk, slik at oppførselen blir naturlig. Men en spennende mulighet når fenomener modelleres på en datamaskin er å endre naturlovene slik at vi ikke får helt naturlig oppførsel. Dette er som regel umulig i virkeligheten, men i en kunstig dataverden er det ingenting i veien for å gjøre slike eksperimenter.

Som ved integrasjon er det bare de færreste differensialligninger som kan løses eksakt ved at vi finner en enkel formel for løsningen. Og som for integraler er det slik at selv om vi kan finne en eksakt løsning er denne ofte så komplisert at den ikke er til særlig nytte. Det vanlige er derfor at differensialligninger løses numerisk på datamaskin ved å beregne en tilnærming til løsningen i en del punkter. Ut fra disse verdiene kan vi så beregne en tilnærming til løsningsfunksjonen ved å trekke en rett linje mellom de beregnede verdiene. Noen slike metoder er diskutert i seksjon 10.7 i *Kalkulus*, og i seksjon 8.1 skal vi se litt på hvordan de kan implementeres effektivt på en datamaskin. I tillegg skal vi i seksjon 8.2 studere fallskjermhopping ved hjelp av differensialligninger. I analysen av

fallskjermhopping får vi bruk for det å løse differensialligninger både eksakt (løsning ved formel) og numerisk.

8.1 Numerisk løsning av differensialligninger

Siden det er mange differensialligninger som ikke kan løses eksakt i den forstand at vi kan finne en enkel formel for løsningen, er det viktig med numeriske metoder som kan finne tilnærminger til løsningene. Disse metodene fungerer for et bredt spekter av differensialligninger og er den vanlige måten å løse differensialligninger på ved hjelp av datamaskiner. Flere metoder er nevnt i *Kalkulus*, men her skal vi bare se på to varianter av *Eulers metode*: Eulers metode for førsteordens ligninger (som også står i *Kalkulus*), og Eulers metode for andreordens ligninger (som ikke står i *Kalkulus*). Vi skal utlede Eulers metode på en litt annen måte enn i læreboka, og vi baserer oss på at Taylorutvikling av funksjoner er kjent.

8.1.1 Eulers metode for førsteordens differensialligninger

Standardversjonen av Eulers metode gir en algoritme for å løse førsteordens differensialligninger på formen

$$y' = f(x, y), \qquad y(a) = d, \tag{8.1}$$

der vi antar at f er en (stykkevis) kontinuerlig funksjon av de to variablene x og y. Løsningen av ligningen er en funksjon $y(x)$ som passer inn i ligningen for alle verdier av x. Legg merke til at denne ligningen er mer generell enn de ligningene vi kan løse eksakt ved hjelp av metodene i *Kalkulus*.

For å finne en tilnærmet løsning til ligningen (8.1) forsøker vi oss med en Taylorutvikling av løsningen. Taylorpolynomet til y om x er gitt ved

$$y(x+h) = y(x) + hy'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \cdots,$$

der h er et gitt tall. Hvis y er en 'pen' funksjon betyr likheten her at høyre side vil konvergere mot venstre side når vi tar med stadig flere ledd i Taylorpolynomet. For enkelhets skyld tar vi nå med to ledd. Da vil, i beste fall, høyre side bli en tilnærming til $y(x+h)$,

$$y(x+h) \approx y(x) + hy'(x).$$

Det fine med denne tilnærmingen er at differensialligningen (8.1) gir oss en formel for $y'(x)$. Bruker vi denne får vi

$$y(x+h) \approx y(x) + hf(x, y). \tag{8.2}$$

Hvis $y(x)$ er kjent gir dette oss derfor en metode for å finne en tilnærming til $y(x+h)$. Nå er ikke funksjonen $y(x)$ kjent, men legg merke til at initialbetingelsen $y(a) = d$ i (8.1) gir verdien av y i a. Vi kan derfor velge oss en h og bruke formelen (8.2) for $x = a$ til å regne ut en tilnærming $y^h(a+h)$ til $y(a+h)$,

$$y(a+h) \approx y^h(a+h) = y(a) + hf\big(a, y(a)\big) = d + hf(a, d).$$

Når vi nå har tilnærmingen $y^h(a+h)$ kan v i bruke denne til å beregne en tilnærming $y^h(a+2h)$ til $y(a+2h)$,

$$y(a+2h) \approx y^h(a+2h) = y^h(a+h) + hf\big(a+h, y^h(a+h)\big). \tag{8.3}$$

Nå begynner forhåpentligvis mønsteret å bli klart. Ved å bruke formelen (8.2) i punktet $x = a+2h$ kan vi beregne en tilnærming $y^h(a+3h)$ til $y(a+3h)$ siden vi nå har tilnærmingen $y^h(a+2h)$ til $y(a+2h)$, og slik kan vi fortsette. For å forenkle formlene setter vi $x_i = x + ih$ og $y_i^h = y^h(a+ih)$. Da kan prosessen oppsummeres ved

$$y_i^h = y_{i-1}^h + hf(x_{i-1}, y_{i-1}^h), \qquad i = 1,\, 2,\, 3,\, \ldots \tag{8.4}$$

Hvis vi er interessert i løsningen på intervallet $[a,b]$ er det rimelig å velge et naturlig tall n og dele opp intervallet i n delintervaller med bredde $h = (b-a)/n$. Vi lar så i løpe opptil n i (8.4) siden $x_n = a + nh = b$. Dette gir følgende algoritme.

Algoritme 8.1 (Eulers metode for førsteordens ligninger)**.** *Anta at høyresiden i differensialligningen $y' = f(x,y)$ er gitt som en metode* `f` *sammen med konstantene a, d, b og n som angir initialbetingelsen $y(a) = d$, det høyre endepunktet i intervallet $[a,b]$, og antall beregningspunkter i intervallet. Følgende kode vil generere en tilnærming til løsningen av differensialligningen på intervallet $[a,b]$:*

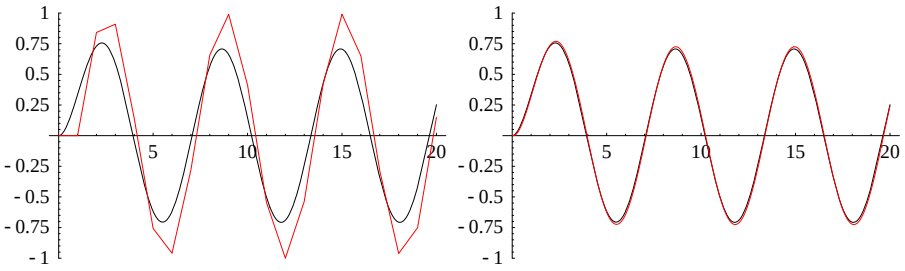
```
int i;
double h, x, yh[n];
h = (b-a)/n;
x = a; yh[0]=d;
for (i=1; i<n; i++) {
  yh[i] = yh[i-1] + h*f(x,yh[i-1]);
  x = a + i*h;
}
```

Tabellen `yh` *vil etter dette inneholde en tilnærming til løsningen av $y' = f(x,y)$ i den forstand at* `yh[i]` *er tilnærmet lik $y(a+ih)$.*

Her har vi lagret den tilnærmede løsningen i en tabell `yh`, og på den måten har vi tilgang til alle de beregnede verdiene etter at koden over er ferdig utført. I mange sammenhenger er dette ikke nødvendig, for eksempel hvis vi bare er interessert i verdien $y(b)$ eller hvis vi skal plotte løsningen og har en plotteomgivelse som tillater oss å bygge opp plottet etterhvert som vi gjør beregningene. I så fall kan vi erstatte tabellen `yh` med en eller to `double`-variable.

To eksempler på bruk av Eulers metode er vist i figur 8.1. I begge tilfeller er det den samme ligningen som er løst, men verdien av h i (b) er 1/10 av verdien i (a).

Eulers metode gir bare en tilnærming til løsningen av differensialligningen, og spørsmålet er om vi kan stole på de verdiene som algoritmen over beregner. Dette er det vanskelig å svare på generelt, men vi kan bruke den vanlige teknikken med å stadig halvere h inntil forskjellen på to tilnærminger blir mindre enn en gitt toleranse. Hvis vi har to tilnærminger $\big(y^h(i)\big)_{i=0}^{n}$ og $\big(y^{h/2}(j)\big)_{j=0}^{2n}$ så kan vi sammenligne $y^{h/2}(2i)$ med $y^h(i)$



Figur 8.1. Figuren viser numerisk løsning av differensialligningen $y' = \sin x - y$ med initialbetingelse $y(0) = 0$, på intervallet $[0, 20]$. I (a) er den eksakte løsningen vist sammen med den numeriske løsningen gitt ved Eulers metode med 20 steg. I (b) er Eulers metode med 200 steg brukt.

siden disse er tilnærminger til $y(x)$ i det samme punktet, nemlig $a + (2i)(h/2) = a + ih$. Hvis $\left|y^{h/2}(2i) - y^h(i)\right|$ er mindre enn en gitt toleranse ϵ for $i = 1, 2, \ldots, n$ så er det rimelig å si at feilen vi gjør ved å bruke $y^{h/2}$ som tilnærming til y er omtrent ϵ (men vi har ingen garanti for at dette er riktig).

Det er verdt å merke seg at *feil akkumuleres* i Eulers metode. Vi gjør først en feil når vi tilnærmer $y(a + h)$ med $y^h(a + h)$ og så bruker vi denne feilaktige verdien til å beregne en tilnærming $y^h(a + 2h)$ til $y(a + 2h)$. Når vi fortsetter denne prosessen ser vi at feilen generelt vil bli større ettersom beregningene utvikler seg, og vanligvis blir feilen størst i den siste verdien $y^h(b)$. Hvis h velges for stor kan derfor mye rart skje. Det mest problematiske er om svaret blir rimelig, men galt, siden slikt kan være vanskelig å oppdage. For å gardere seg mot dette er det viktig å ha såpass innsikt i fenomenet som simuleres at en er i stand til å vurdere om løsningen er rimelig. For å få slik innsikt er det nyttig å løse forenklede varianter av ligningen og se hvordan disse løsningene oppfører seg. Enkelte ganger kan det gå riktig galt om h er for stor, i den forstand at beregningene kan gi såkalt 'overflow', altså at tallene som dukker opp i beregningene blir større en det største tillatte tallet for datatypen som brukes (vanligvis `double`). Men dette er i og for seg en grei feilsituasjon siden den er så lett å oppdage. Og hvis h bare velges tilstrekkelig liten forsvinner problemet og vi vil kunne få tilnærmingen så god som vi måtte ønske.

8.1.2 Eulers metode for andreordens differensialligninger

Eulers metode, slik vi har beskrevet den over, fungerer bare for førsteordens differensialligninger, men hva kan vi gjøre hvis vi har en andreordens ligning

$$y'' = g(x, y, y'), \qquad y(a) = d, \quad y'(a) = e\,? \tag{8.5}$$

Her tillater vi at g er en funksjon av de tre variablene x, y og y' som gir en mye større klasse av andreordens differensialligninger enn de inhomogene, lineære, med konstante koeffisienter som er behandlet i *Kalkulus*. Et par eksempler på slike ligninger er $y'' = x + y + y'$ og $y'' = \sin(x + y + y')$. Den første er en inhomogen, lineær ligning med konstante koeffisienter mens den andre er et eksempel på en ikkelineær ligning.

Vi kan ikke godta helt vilkårlige funksjoner g i (8.5); for vårt formål er det greit å anta at g er (stykkevis) kontinuerlig.

La oss forsøke oss med Eulers metode for å beregne en tilnærmet løsning av (8.5) i $x = a + h$. Vi bruker to ledd i Taylorpolynomet og tilnærmer $y(a + h)$ med

$$y^h(a + h) = y(a) + hy'(a) = d + e.$$

Dette gikk jo riktig bra, så la oss nå forsøke å beregne en tilnærming til $y(a + 2h)$ på samme måte. En slik formel må ta utgangspunkt i tilnærmingen

$$y(a + 2h) \approx y(a + h) + hy'(a + h).$$

I denne formelen kan vi erstatte $y(a + h)$ med $y^h(a + h)$ som vi allerede har beregnet, men vi trenger også en tilnærming til $y'(a + h)$, og det har vi ikke. Løsningen er å også tilnærme $y'(a + h)$ med et lineært Taylorpolynom,

$$y'(a + h) \approx y'(a) + hy''(a) = e + hg\big(a, y(a), y'(a)\big) = e + hg(a, d, e).$$

Hvis vi betegner tilnærmingen til den deriverte med Dy^h så beregner vi altså $Dy^h(a+h)$ fra formelen

$$Dy^h(a + h) = e + hg(a, d, e).$$

Når vi har denne verdien kan vi beregne en tilnærming til $y(a + 2h)$ med

$$y^h(a + 2h) = y^h(a + h) + hDy^h(a + h).$$

For å være i stand til å beregne $y^h(a + 3h)$ trenger vi også å beregne

$$Dy^h(a + 2h) = Dy^h(a + h) + hg\big(a + h, y^h(a + h), Dy^h(a + h)\big)$$

(legg merke til at alle størrelsene på høyre side i den siste formelen er kjent). Hvis vi skal beregne en tilnærming på intervallet $[a, b]$ velger vi et naturlig tall n og beregner $h = (b - a)/n$. Vi bruker forkortelsene $y_i^h = y^h(a + ih)$ og $Dy_i^h = Dy^h(a + ih)$ og utfører beregningene

$$\left.\begin{aligned} Dy_i^h &= Dy_{i-1}^h + hg\big(x_{i-1}, y_{i-1}^h, Dy_{i-1}^h\big), \\ y_i^h &= y_{i-1}^h + hDy_{i-1}^h, \end{aligned}\right\} \qquad i = 1, 2, 3, \ldots, n. \tag{8.6}$$

Disse formlene leder til følgende algoritme.

Algoritme 8.2 (Eulers metode for andreordens ligninger)**.** *Anta at høyresiden i differensialligningen $y'' = g(x, y, y')$ er gitt som en metode* g *sammen med konstantene a, d, e, b og n som angir initialbetingelsene $y(a) = d$ og $y'(a) = e$, det høyre endepunktet i intervallet i $[a, b]$, og antall beregningspunkter i intervallet. Følgende kode vil generere en tilnærming til løsningen av differensialligningen på intervallet $[a, b]$:*

```
int i;
double h, x, yh[n], Dyh[n];
h = (b-a)/n;
x = a;
yh[0] = d;
Dyh[0] = e;
for (i=1; i<=n; i++) {
  Dyh[i] = Dyh[i-1] + h*g(x,yh[i-1],Dyh[i-1]);
  yh[i] = yh[i-1] + h*Dyh[i-1];
  x = a + i*h;
}
```

Tabellen `yh` *vil etter dette inneholde en tilnærming til løsningen av differensialligningen* $y'' = g(x, y, y')$ *i den forstand at* `yh[i]` *vil være tilnærmet lik* $y(a + ih)$*, mens* `Dyh[i]` *vil være tilnærmet lik* $y'(a + ih)$*.*

Legg merke til at om vi regner ut y_i^h før Dy_i^h i (8.6) så har vi mulighet for å erstatte y_{i-1}^h med y_i^h som andre argument til funksjonen g. Dette svarer ikke til hva vi brukte i utledningen, men det er ikke urimelig å tro at det vil kunne gi en bedre metode. Det å vise noe slikt går langt utover det vi skal befatte oss med her, så vi holder oss til Algoritme 8.2.

8.2 Simulering av fallskjermhopping

For en fallskjermhopper er det livsviktig å ha god kjennskap til hva som egentlig skjer når hun faller gjennom luften og åpner fallskjermen for å bremse fallet — i denne aktiviteten er det ikke rom for å eksperimentere med noe en ikke på forhånd vet utfallet av! Matematiske modeller kan på en enkel måte gi innsikt i fallforløpet og mulighet for å se i hvert fall den kvalitative effekten av ulike eksperimenter en hopper kan foreta seg under fallet. Mest av alt bidrar en matematisk modell til å øke *forståelsen* av hva som skjer i fallet. Denne forståelsen er av generisk natur; den kan overføres på en lang rekke andre problemstillinger der legemer faller i væsker og gasser.

Ordet dynamikk brukes gjerne om samspillet mellom krefter og bevegelse. At bevegelse må skyldes en kraft ble først presist etablert av Galileo og Newton for nesten 400 år siden. Før det hadde menneskene bare en løs forståelse av hvorfor ting beveget seg.

Newtons andre lov er et grunnleggende prinsipp i fysikk som relaterer summen av kreftene som virker på et legeme og legemets bevegelse. Kaller vi summen av kreftene F og legemets akselerasjon a (måles vanligvis i $\mathrm{m/s^2}$), sier Newtons andre lov at

$$F = ma, \tag{8.7}$$

der m er legemets masse (måles vanligvis i kg (kilogram)). I en typisk situasjon har vi et gitt legeme som påvirkes av krefter som setter det i bevegelse; utfordringen består i å beskrive legemets bevegelse. Det kan vi altså gjøre ved å finne fram til alle kreftene som virker på legemet. Summerer vi opp kreftene i F er da akselerasjonen gitt ved Newtons lov (8.7).

8.2.1 Utledning av ligningen for fallskjermhopp

La oss anvende Newtons andre lov på en fallskjermhopper. Vi kjenner hopperens masse m, men ikke akselerasjonen a. Kreftene som virker på en hopper faller i to kategorier:

1. Tyngdekraften F_g, som er lik mg, der $g = 9.81\text{m/s}^2$ er tyngdens akselerasjon.

2. Krefter som virker fra luften på hopperen langs overflaten av hopperen.

Kreftene i kategori 2 er svært kompliserte å beskrive og regne ut. Vi skal derfor gjøre betydelige forenklinger. Kreftene fra luften på hopperen består av en motstandskraft F_m, som skyldes en slags friksjon mellom hopperen og luften, og en oppdriftskraft F_o, som er det sasmme fenomenet som hindrer legemer med liten tetthet å synke i vann. Oppdriftskraften er så liten at den kan neglisjeres for en fallskjermhopper, men vi skal ta den med likevel for å illustrere hvordan vi mer presist kan begrunne hvorfor den kan neglisjeres.

Fysiske eksperimenter med legemer som beveger seg i luft eller i andre gasser og væsker viser at motstandskraften F_m er relatert til legemets fart. Hvis legemet beveger seg sakte, er F_m proporsjonal med farten v. Hvis legemet derimot beveger seg med høy fart, er F_m proporsjonal med v^2. Hva "sakte" og "høy fart" betyr må selvfølgelig presiseres nærmere før man kan vite hvilken variant av F_m som er relevant i et problem. Her nøyer vi oss med å si at en fallskjermhopper faller i kategorien "høy fart". Vi kan da skrive

$$F_m = -C_D v^2.$$

Her er C_D en *motstandskoeffisient* som avhenger av legemets form og lufttettheten, og som har benevning kg/m. For eksempel vil størrelsen på C_D være avhengig av om fallskjermen er utløst eller ikke, og i tilfellet fritt fall, vil hopperens kroppsform påvirke C_D. I et typisk hopp vil derfor C_D variere med tiden. Det knytter seg stor usikkerhet til verdien av C_D, og det er derfor akseptabelt at vi gjør en del forenklinger i andre deler av den matematiske modellen. Minustegnet foran $C_D v^2$ indikerer at *motstandskraften virker mot fartsretningen*, og at positiv bevegelsesretning er rettet nedover mot jorda i vår matematiske beskrivelse.

Oppdriftskraften er gitt av Arkimedes lov og har formen

$$F_o = -\varrho_L g V,$$

der ϱ_L er tettheten av mediet legemet er i (her luften), og V er volumet av legemet (her hopperen). Minustegnet i F_o indikerer at oppdriften virker oppover (når positiv retning er valgt nedover).

Setter vi inn alle tre kraftbidragene i Newtons andre lov, får vi

$$F_o + F_m + F_g = ma$$

eller

$$\varrho_L g V - C_D v^2 + mg = ma. \tag{8.8}$$

Hvilke størrelser er kjente og hvilke er ukjente i denne formelen? Akselerasjonen a og farten v har med bevegelsen å gjøre og er derfor ukjente, konstantene ϱ_L, g, og m er definitivt kjente størrelser, mens C_D er mer uklar. I utgangspunktet kjenner vi ikke C_D; den avhenger detaljert av strømningsforholdene rundt hopperen, som igjen avhenger av bevegelsen og hopperens geometri. Vi burde derfor introdusere en matematisk modell som kan beskrive C_D. Dette er prinsipielt mulig, men modellen er svært komplisert og vil for en enkelt beregning av C_D antakelig kreve mange dagers regning på de største datamaskinene i verden. Vi trenger dessuten mange forskjellige verdier av C_D for ulike fallformer, fallskjermer osv. I steden må vi forsøke å finne mer kvalitativ informasjon om C_D, og ut fra dette estimere noen fornuftige verdier til bruk i modellen vår.

Om vi antar at vi kan finne egnede verdier for C_D, er likning (8.8) nå én likning med to ukjente, akselerasjonen a og farten v. Vi trenger derfor en likning til for å få like mange likninger som ukjente. Denne ekstra likningen kommer fra sammenhengen mellom fart og akselerasjon. Både farten og akselerasjonen er funksjoner av tiden, og akselerasjon er definert som fartsendring pr. tidsenhet. Matematisk betyr dette at akselerasjon er den tidsderiverte av farten,

$$a = v' = \frac{dv}{dt}.$$

Dermed har vi to likninger med to ukjente. Som vanlig er det en fordel om vi kan redusere dette til én likning med én ukjent. Det er enkelt i dette tilfellet ettersom vi bare erstatter a i (8.8) med dv/dt,

$$\varrho_L g V - C_D v^2 + mg = mv'. \tag{8.9}$$

Vi har altså fått en førsteordens differensialligning for den ukjente funksjonen $v(t)$.

Det er verdt å merke seg at det ikke er noe i ligning (8.9) som direkte har med en fallskjermhopper å gjøre. Ligningen er gyldig for alle fysiske fenomener der legemer beveger seg rettlinjet i en væske eller gass, og der det er relevant å gjøre de samme forutsetningene som vi har gjort over (blant annet formen på motstandskraften).

Vi nevnte at oppdriftskraften F_o kan neglisjeres for en fallskjermhopper. La oss se hvordan vi kan argumentere mer presist for det. Vi ser at

$$F_o + F_g = -\varrho_L g V + mg.$$

Hopperens masse m er produktet av hopperens tetthet ϱ_h og volum V, altså er $m = \varrho_h V$. Dermed kan vi skrive

$$F_o + F_g = -\varrho_L g V + \varrho_h V g = gV(\varrho_h - \varrho_L).$$

Hvis ϱ_L er mye mindre enn ϱ_h, kan vi neglisjere oppdriftskraften fordi den vil være fullstendig dominert av tyngdekraften. Dette er tilfellet her. Hopperens tetthet er av samme størrelsesorden som vann (kroppen består for det meste av vann!) som har tetthet omtrent 1g/cm^3, mens luftens tetthet er av størrelsesorden $\varrho_L \sim 10^{-4}\text{g/cm}^3$. Som nevnt over er ligning (8.9) også en modell for legemer som beveger seg i vann. Siden en dykker har omtrent samme tetthet som vann vil oppdriftskraften gi viktige bidrag til bevegelsen i en slik sammenheng.

I det videre skal vi arbeide med en matematisk modell for fallskjermhopping der oppdriftskraften er neglisjert,

$$-C_D v^2 + mg = mv' \tag{8.10}$$

Løsningen av denne differensialligningen er en funksjon $v(t)$, og ved hvert tidspunkt skal (8.10) være tilfredstilt. Men for at løsningen skal være entydig bestemt må vi kjenne farten i det fallet starter. Her er det naturlig å anta at bevegelsen starter fra ro slik at initialbetingelsen er $v(0) = 0$. Av estetiske årsaker velger vi å skrive om likning (8.10) slik at det er færre parametre. Ved å sette $c = C_D/m$ får vi

$$v' = -cv^2 + g, \qquad v(0) = 0. \tag{8.11}$$

8.2.2 Enkle metoder for å lære om modellen

Det er mange måter å skaffe seg informasjon om løsningen av en differensialligning på, selv om det i mange tilfeller kan være vanskelig å finne en enkel formel for løsningen. Vanligvis har vi iallefall følgende muligheter til å skaffe informasjon:

1. Ved analytisk løsning. Dette er svært fordelaktig om funksjonsuttrykket for $v(t)$ er enkelt, men det er som oftest vanskelig å finne analytiske løsninger av alt annet enn de enkleste differensiallikningene. For ligning (8.11) har en analytisk løsning den fordelen at den gjør $v(t)$'s avhengighet av c tydelig.

2. Ved numerisk løsning. De aller fleste differensialligninger kan løses numerisk, men da må alle parametre spesifiseres. For ligning (8.11) betyr det at når c er gitt (tyngdekraften $g = 9.81$ er konstant og derfor uaktuell å eksperimentere med) kan ligningen løses numerisk. På denne måten får vi ikke ekspisitt informasjon om hvordan $v(t)$ avhenger av c, men kan bare finne numeriske tilnærminger til løsningen for forskjellige verdier av c. Ut fra dette må vi så forsøke å si noe om hvordan den generelle løsningen, der c er en parameter, avhenger av c.

3. Løse forenklede varianter av differensialligningen. Når ligningen forenkles er det ofte mulig å finne enkle løsninger som gir forståelse for dynamikken i modellen. Kunnskap om løsninger i spesialtilfeller er også nyttig for å teste dataprogrammer for numerisk løsning av den fullstendige likningen.

Analytisk og numerisk løsning av ligningen utsetter vi til oppgavene, men la oss se på noen forenklinger som vi kan gjøre.

Neglisjering av motstandskraften. La oss utelate motstandskraften $F_m = -cv^2$ i (8.11), noe som svarer til å sette konstanten $C_D = 0$ (husk at $c = C_D/m$). Da forenkles differensialligningen til

$$v' = g, \qquad v(0) = 0.$$

Denne likningen er enkel å løse siden det er snakk om å finne den antideriverte til konstanten g. Vi får

$$v = \int g\,dt = gt + C,$$

der C er en integrasjonskonstant som vi kan bestemme fra initialbetingelsen $v(0) = 0$. Vi ser lett at denne betingelsen gir $C = 0$, så når $c = 0$ er løsningen altså $v(t) = gt$. Dette betyr at farten øker lineært med tiden, slik at hopperen faller fortere og fortere, uten noen begrensninger. Dette er ikke en fysisk løsning; både med og uten fallskjerm vil motstandskraften påvirke bevegelsen betydelig. Men løsningen $v = gt$ når $c = 0$ er nyttig som en test på et dataprogram for numerisk løsning av (8.11).

Neglisjering av tyngdekraften. Tyngdekraften $F_g = mg$ gjenfinnes som leddet g i (8.11). Neglisjerer vi dette leddet (setter tygdekraften g til 0) får vi den forenklede ligningen

$$v' = -cv^2, \qquad v(0) = 0. \tag{8.12}$$

Denne ligningen kan vi løse analytisk siden den er en enkel separabel differensiallikning. Ved å dividere med v^2 på begge sider (vi antar at v er ulik 0) får vi ligningen $v'/v^2 = -c$, og den vanlige løsningsmetoden for separable ligninger gir da

$$\int \frac{dv}{v^2} = \int -c\,dt.$$

Regner vi ut integralene får vi

$$-v^{-1} = -ct + C$$

slik at den generelle løsningen blir

$$v(t) = 1/(ct - C), \tag{8.13}$$

der integrasjonskonstanten C er et vilkårlig reelt tall.

Initialbetingelsen $v(0) = 0$ er problematisk i dette tilfellet siden vi måtte anta at $v \neq 0$ da vi løste ligningen ved separasjon av variable. Vi bruker derfor en generell initialverdi $v(0) = v_0$ i steden, der vi antar at $v_0 \neq 0$. Hvis vi setter dette inn i løsningen finner vi $C = -1/v_0$ slik at løsningen er

$$v(t) = \frac{v_0}{1 + v_0 ct}. \tag{8.14}$$

Men i dette uttrykket er ikke $v_0 = 0$ noe problem, og siden $v(t) = 0$ er en løsning av (8.12) ser vi at (8.14) gir løsningen av (8.12) for alle mulige intialbetingelser $v(0) = v_0$.

Hvis $v_0 \neq 0$ så ser vi at $v(t)$ til avta mot null ettersom tiden går. Dette er naturlig siden motstandskraften er den eneste kraften som virker, og den virker mot bevegelsen. Vi får derfor en oppbremsing av bevegelsen.

Konstant fart. Tyngdekraften vil akselerere hopperen mot jorda mens motstandskraften vil bremse bevegelsen, og oppbremsingen blir kraftigere etterhvert som farten øker. Når de to kreftene blir like store i absoluttverdi, er summen av kreftene null, og dermed akselerasjonen null. Farten forblir da konstant. Når farten ikke endrer seg er $v' = 0$, så ved å sette dette inn i (8.11) kan vi finne denne konstante farten. Dette gir ligningen

$$0 = -cv^2 + g$$

som har løsningen

$$v_c = \sqrt{g/c} \tag{8.15}$$

Farten gitt ved v_c er en øvre grense for hvor fort hopperen kan falle. Som vi ser, avhenger maksimalfarten av c, som igjen avhenger av hopperens geometri, for eksempel om fallskjermen er utløst eller ikke, kroppens form og lignende.

8.2.3 Varierende motstandskoeffisient

I praksis vil ikke motstandskoeffisienten være konstant. Under fallet beveger hopperen seg, og dette betyr at c varierer. Den største forandringen kommer i det fallskjermen utløses. Da vil c, i løpet av et svært kort tidsrom, endres fra en forholdsvis liten verdi til en stor verdi. I tillegg øker lufttettheten, og dermed c, ettersom hopperen nærmer seg jordoverflaten. Alt dette taler for at c må avhenge av t for å få en realistisk modell.

Spørsmålet er derfor hvordan vi kan finne c? Som nevnt tidligere er dette et svært komplisert problem, men vi kan gjøre noen eksperimenter for å finne omtrentlige verdier. Ved 'normale' fall vil farten før fallskjermen utløses stabilisere seg på omtrent 180 km/t. Vi finner da fra (8.15) at $c = g/v_c^2 = 0.003924\text{m}^{-1}$. På samme måte kan vi finne en passende verdi for c etter at fallskjermen er utløst hvis vi vet hva den asymptotiske farten med fallskjerm er.

8.2.4 Beregning av hopperens posisjon

Så langt har vi konsentrert oss om å bestemme farten til fallskjermhopperen, men en viktig forutsetning for et vellykket hopp er at hopperen ikke treffer bakken før fallskjermen er utløst. I tillegg til å kjenne hastigheten er det derfor også viktig å vite hvor langt hopperen har falt ved tidspunktet t. Hvis vi kaller denne avstanden $z(t)$ vet vi fra fysikk at farten er den deriverte av $z(t)$. Vi måler fallhøyden ut fra hvor langt hopperen har falt fra uthoppsstedet slik at $z(0) = 0$. Fallhøyden er derfor gitt ved differensialligningen

$$z' = v, \qquad z(0) = 0. \tag{8.16}$$

Når hastigheten er kjent som funksjon av t kan vi derfor bestemme z ved å løse denne enkle differensialligningen. Vi ser at z er integralet av v slik at

$$z(t) = \int_0^t v(s)\, ds. \tag{8.17}$$

I praksis er det sjelden mulig å regne ut dette integralet eksakt så vi må finne en numerisk tilnærming. Dette kan vi gjøre ved å bruke numerisk integrasjon i (8.17) eller løse differensialligningen (8.16) numerisk. I begge tilfeller er alt vi trenger verdien til v (eller en tilnærming) i noen isolerte punkter, noe vi kan få både fra en eksakt løsning og en numerisk løsning av den opprinnelige ligningen (8.11).

En annen måte å finne z på er å sette inn z i (8.16). Da kan differensialligningen (8.11) omskrives som

$$z''(t) = -cz'(t)^2 + g, \qquad z(0) = 0, \quad z'(0) = v(0) = 0. \tag{8.18}$$

Ved å løse denne ligningen finner vi z direkte, og når z er kjent kan vi finne v ved derivasjon. Bruker vi Eulers metode for andreordens ligninger må både z og $z' = v$ beregnes underveis slik at vi automatisk får tilgang til både høyde og fart.

Oppgaver

8.1 Programmer Eulers metode for førsteordens differensialligninger (algoritme 8.1) på formen

$$y' = f(x, y), \qquad y(a) = d,$$

der funksjonen f og tallene a og d er gitt. Bruk $h = (b-a)/n$ der n er et gitt naturlig tall, og finn en tilnærmet løsning på intervallet $[a, b]$ der b også er et gitt tall. Test programmet på ligningen $y' = y$ med $y(0) = 1$ som har løsningen $y(x) = e^x$. Bruk $b = 1$ og eksperimenter med forksjellige verdier av n.

8.2 Programmer Eulers metode for andreordens differensialligninger (algoritme 8.2) på formen

$$y'' = g(x, y, y'), \qquad y(a) = d,\ y'(a) = e,$$

der funksjonen f og tallene a, d og e er gitt. Bruk $h = (b - a)/n$ der n er et gitt naturlig tall, og finn en tilnærmet løsning på intervallet $[a, b]$ der b også er et gitt tall. Test programmet på ligningen $y'' = -y$ med $y(0) = 0$ og $y'(0) = 1$ som har løsningen $y(x) = \sin x$. Bruk $b = 1$ og eksperimenter med forksjellige verdier av n.

8.3 I denne oppgaven skal vi studere fallskjermhopping ved hjelp av den matematiske modellen i seksjon 8.2. Husk at $g = 9.81\text{m/s}^2$, mens hastighetene under er gitt i km/t. Dette betyr at enten må g omregnes til km/t^2 eller hastighetene til m/s.

a) Ved normale hopp regner vi med at farten kommer opp i omtrent 180–200 km/t før fallskjermen utløses. Bestem ved hjelp av ligning (8.15) en verdi c_1 for motstandskoeffisienten slik at 200 km/t blir den asymptotiske farten til hopperen.

b) Løs ligningen

$$y' = -cv^2 + g, \qquad v(0) = 0 \tag{8.19}$$

med $g = 9.81$ og c lik verdien c_1 som du fant i (a), og plott hastigheten som funksjon av t. Hvor lang tid tar det før hopperen oppnår en hastighet på 180 km/t?

c) En vanlig landingshastighet er omtrent 30 km/t. Finn hvilken verdi c_2 av motstandskoeffisienten som gir en asymptotisk hastighet på 28 km/t, og gjør en simulering for å finne ut hvor lang tid det tar å redusere hastigheten fra 180 km/t til 30 km/t med denne verdien av c. Lag et plott som viser hvordan farten avtar.

Hint: Løs (8.19), men med initialbetingelse $v(0) = 180$ km/t.

d) Så langt har vi bare sett på hastigheten som hopperen faller med. Men det er åpenbart også viktig å vite hvor mange meter hopperen faller slik at hun kan være sikker på at hun hopper ut høyt nok oppe til at farten er dempet tilstrekkelig før landing. Utvid beregningene i (b) og (c) med å beregne fallhøyden ved å løse differensialligningen (8.16) ved hjelp av Eulers metode. Lag et plott som viser hvordan høyden avtar.

Legg merke til at når en numerisk tilnærming til v er beregnet ved hjelp av Eulers metode med steglengde h så er dette akkurat de verdiene du trenger for å løse (8.16) ved hjelp av Eulers metode med den samme steglengden h.

e) Vi skal nå gjøre en simulering av et fullstendig hopp ved å bruke variabel c. Definer $c(t)$ ved

$$c(t) = \begin{cases} c_1, & \text{når } 0 \le t \le t_1, \\ c_2, & \text{når } t_1 < t \le t_2, \end{cases} \tag{8.20}$$

der c_1 og c_2 er verdiene du fant i (a) og (c). Verdiene t_1 og t_2 må du selv bestemme slik at du får et 'pent' hopp med et 'passende' innslag av fritt fall og fall med åpen skjerm.

Gjør en simulering og lag plott som viser hvordan hastigheten og høyden varierer.

8.4 Tidligere i høst (2000) stod det å lese i avisene om en fransk fallskjermhopper som ønsker å sette hastighetsrekord ved å oppnå en hastighet på oppimot 1800 km/t i fritt fall før fallskjermen løses ut. For å få til dette må han hoppe ut fra stor høyde slik at lufta er tynn (gir lite friksjon) og han har stor fallhøyde. I tillegg må han stupe med hodet først for å redusere luftmotstanden til et minimum. I denne oppgaven skal vi gjøre en forenklet simulering av et slikt hopp.

Gjenta oppgave 3, men erstatt farten 200 km/t med 2000 km/t. I den endelige simuleringen antar vi at hoppet har tre faser: først fritt fall der målet er å oppnå en hastighet på 1800 km/t, så en fase med 'vanlig' fritt fall (bruk den verdien av c som du fant i oppgave 3 (a)) og til slutt en fase med åpen skjerm (bruk verdien av c fra oppgave 3 (c)). Lag en stykkevis konstant c slik som i (8.20), men med tre grener svarende til de tre forskjellige verdiene av c. Bestem selv hvor lange de ulike fasene av hoppet skal være og lag plott som viser hvordan hastigheten og høyden utvikler seg. Hvor høyt må hopperen starte for å få oppnå 1800 km/t og lande forsvarlig?

8.5 Gjenta oppgave (3), men ta utgangspunkt i den andreordens ligningen (8.18), og bruk Eulers metode for andreordens ligninger til å beregne den numeriske løsningen.

8.6 En populær sport i enkelte kretser er såkalt 'basehopping'. Dette går ut på å gjøre fallskjermhopp fra kjente bygninger og tårn, helst så lave som mulig, men ikke lavere enn at hoppet er forsvarlig. Gjør simuleringer for å undersøke hva som er det laveste punktet det går an å hoppe fra og allikevel oppnå en landingshastighet på omtrent 30 km/t. Vi er ikke interessert i hopp der fallskjermen kan utelates fordi høyden er lav. Vurder selv hvor lenge hopperen minst bør vente før skjermen utløses (for

å være sikker på at skjermen fyller seg skikkelig og kjapt med luft). Bruk verdiene av c fra oppgave 3. Lag plott av hastighets- og høydeforløpet og finn en omtrentlig minimumshøyde.

8.7 Ligningen $y' = -cy^2 + g$ er separabel, og integralene som dukker opp ved den vanlige løsningsmetoden for slike ligninger er løsbare. Finn den eksakte løsningen.

KAPITTEL 9

Approksimasjon av funksjoner

En grunnleggende teknikk som ofte brukes i ulike deler av matematikk og anvendelser er å tilnærme eller approksimere et objekt med et annet. Som regel er objektet som skal tilnærmes mer komplisert enn objektet som det tilnærmes med. På denne måten får vi et enklere objekt å håndtere på bekostning av at vi må akseptere en, vanligvis liten, feil. I dette kapitlet skal vi se litt på approksimasjon av funksjoner, for det meste med polynomer. Polynomer er populære som approksimasjoner siden verdien av et polynom i et punkt kan bestemmes eksakt (hvis vi ser bort fra avrundingsfeil) ved hjelp av de elementære operasjonene addisjon og multiplikasjon av tall. Dette gjør at polynomer er godt tilpasset beregninger på datamaskin. Mot slutten av kapitlet skal vi også se litt på approksimasjon med stykkevise polynomer. Dette er funksjoner som består av polynombiter som er lenket sammen til mer kompliserte funksjoner. Stykkevise polynomer egner seg også godt til beregninger på datamaskin siden den eneste operasjonen vi trenger i tillegg til polynomoperasjonene er å avgjøre hvilket polynombit vi er på. Dette gjør vi med en test som datamaskiner også håndterer effektivt og greit.

Vi skal først se litt raskt på Taylor polynomer og noen aspekter ved dem som ikke vanligvis dekkes i tradisjonelle matematikkbøker. Deretter skal vi så vidt ta for oss en annen tilnærmingsmetode, nemlig interpolasjon. Et viktig poeng i dette kapitlet er at det er viktig hvordan vi skriver polynomene våre, og vi skal se at det kan være spesielt hendig å skrive polynomer ved hjelp av de såkalte Bernstein-polynomene. Dette leder oss over i såkalte Bezier-kurver som er mye brukt i grafikk. Mot slutten skal vi se mer på stykkevise polynomer som egner seg svært godt til å spalte opp en funksjon i komponenter med forskjellig oppløsning eller skala, såkalt *flerskalaoppløsning*. Som vi skal se egner dette seg godt for representasjon og kompresjon av lyd.

9.1 Taylor polynomer

Taylor[1]-polynomer er polynomer som tilnærmer en gitt funksjon godt i nærheten av et punkt $x = a$. Vi oppnår dette ved å kreve at Taylor-polynomet skal reprodusere så mange som mulig av funksjonens deriverte i a.

[1]Etter Brooke Taylor (1685–1731), engelsk matematiker.

9.1.1 Konstruksjon av Taylor-polynomer

Funksjonen vi skal tilnærmere kaller vi f, og vi antar at f kan deriveres så mange ganger vi ønsker; en kort og vanlig måte å beskrive denne antagelsen på er å si at f skal være glatt. For å kunne bestemme Taylor-polynomer trenger vi et tall a i definisjonsområdet til f. Det enkleste Taylor-polynomet er av grad 0, det er altså konstant, og framkommer ved at vi konstruerer det enklest mulige polynomet som reproduserer verdien til f i a. Hvis vi betegner dette polynomet med p_0 så har vi altså

$$p_0(x) = f(a).$$

Dette er en svært god tilnærming til f i $x = a$, men for de fleste funksjoner blir kvaliteten fort dårlig når vi fjerner oss fra dette punktet.

Neste steg er å forsøke å finne en enkel tilnærming til f som tar med seg både verdien til f og dens førstederiverte f' i a. Siden vi har to betingelser, er det rimelig å tro at vi trenger et polynom med to frie koeffisienter for å få til dette. Vi prøver derfor med et førstegradspolynom $p_1(x) = b_0 + b_1x$. De to betingelsene gir oss to ligninger i de to ukjente b_0 og b_1,

$$\begin{aligned} f(a) &= p_1(a) = b_0 + b_1a, \\ f'(a) &= p_1'(a) = b_1. \end{aligned}$$

Dette gir $b_1 = f'(a)$ og $b_0 = f(a) - af'(a)$ så p_1 kan skrives som

$$p_1(x) = b_0 + b_1x = f(a) - af'(a) + xf'(a) = f(a) + (x - a)f'(a).$$

Siden p_1 er en rett linje som har samme verdi og derivert som f i a er p_1 tangenten til f i dette punktet.

La oss også ta bryderiet med å konstruere $p_2 = b_0+b_1x+b_2x^2$, et andregradspolynom som tilnærmer f best mulig i a. Siden vi har tre frie koeffisienter er det rimelig å tvinge p_2 til å ha samme verdi, førstederivert og andrederivert som f i a. Dette gir betingelsene

$$f(a) = p_2(a) = b_0 + b_1a + b_2a^2, \tag{9.1}$$
$$f'(a) = p_2'(a) = b_1 + 2b_2a, \tag{9.2}$$
$$f''(a) = p_2''(a) = 2b_2. \tag{9.3}$$

Vi legger merke til at dette ligningssystemet har en spesiell struktur som gjør det lett å løse. Fra (9.3) kan vi finne b_2, setter vi resultatet inn i (9.2) kan vi finne b_1 og setter vi begge disse resultatene inn i (9.1) kan vi finne b_0. Løsningen vi finner er

$$\begin{aligned} b_2 &= \frac{f''(a)}{2}, \\ b_1 &= f'(a) - af''(a), \\ b_0 &= f(a) - af'(a) + \frac{a^2f''(a)}{2}. \end{aligned}$$

Setter vi dette inn i uttrykket for p_2 og forenkler så får vi

$$p_2(x) = f(a) + (x-a)f'(a) + \frac{1}{2}(x-a)^2 f''(a). \tag{9.4}$$

Fra dette er det ikke så vanskelig å gjette den generelle formen på Taylor-polynomet av grad n. Men før vi skriver opp dette, la oss reflektere litt over utledningene våre. Hvis vi ser på andregradstilfellet så begynte vi med å skrive opp det generelle polynomet av grad 2

$$p_2(x) = b_0 + b_1 x + b_2 x^2. \tag{9.5}$$

Vi skrev deretter opp betingelsene våre som ga oss et ligningssystem som var lett å løse. Etter noen forenklinger kom vi så fram til (9.4). Legg merke til at om vi til å begynne med hadde antatt at p_2 var på formen

$$p_2(x) = c_0 + c_1(x-a) + c_2(x-a)^2 \tag{9.6}$$

så ville ligningssystemet vårt blitt

$$f(a) = p(a) = c_0, \quad f'(a) = p'(a) = c_1, \quad f''(a) = p''(a) = 2c_2$$

som har løsningen $c_0 = f(a)$, $c_1 = f'(a)$ og $c_2 = f''(a)/2$. Dette gir oss kjapt og greit (9.4) uten behov for noen forenklinger.

Dette illustrerer et viktig poeng: Et polynom kan skrives på mange ekvivalente måter, og ved å velge en form tilpasset problemet vi skal løse kan vi ofte forenkle beregingene våre. Et annet aspekt av dette er at ulike polynomformer kan være mer eller mindre følsomme for avrundingsfeil. Dessverre er det slik at (9.5), som er den vanligste måten å skrive polynomer på, kan være svært følsom for avrundingsfeil om ikke x ligger nær 0, særlig når graden er høy. Vi skal se nærmere på dette i seksjon 9.3 under.

Når vi nå har funnet en måte å skrive polynomer på som er godt tilpasset problemet vi skal løse, så er det på tide å angripe det generelle tilfellet. Vi ønsker å danne et Taylor-polynom p_n av grad n som er en best mulig tilnærming til f i punktet $x = a$ og skriver

$$p_n(x) = c_0 + c_1(x-a) + \cdots c_n(x-a)^n.$$

Siden p_n har $n+1$ frie koeffisienter så forventer vi å kunne legge $n+1$ betingelser på f, nemlig betingelsene $f(a) = p(a)$, $f'(a) = p_n'(a)$, $\ldots$, $f^{(n)}(a) = p_n^n(n)(a)$. Setter vi inn uttrykket for p_n får vi ligningssystemet

$$f(a) = c_0, \quad f'(a) = c_1, \quad f''(a) = 2\,c_2, \quad f'''(a) = 6\,c_3, \quad \ldots, \quad f^{(n)}(a) = n!\,c_n$$

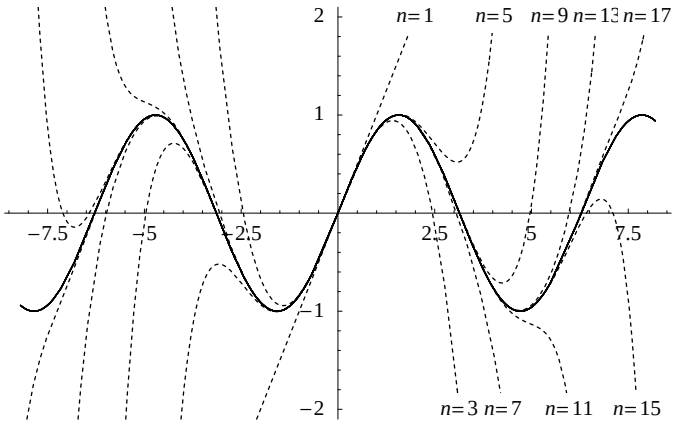
som har løsningen

$$c_0 = f(a), \quad c_1 = f'(a), \quad c_2 = \frac{f''(a)}{2}, \quad c_3 = \frac{f'''(a)}{6}, \quad \ldots, \quad c_n = \frac{f^{(n)}(a)}{n!}.$$

Taylor-polynomet av grad n er dermed

$$p_n(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2}f''(a) + \cdots + \frac{(x-a)^n}{n!}f^{(n)}(a),$$

og for å understreke avhengigheten av f skriver vi ofte $p_n(x) = T_n f(x)$ eventuelt $p_n(x) = T_n f(x;a)$ hvis vi også trenger å ta med avhengigheten av a.



Figur 9.1. Funksjonen $\sin x$ og dens 9 første Taylor-polynomer.

9.1.2 Feilleddet

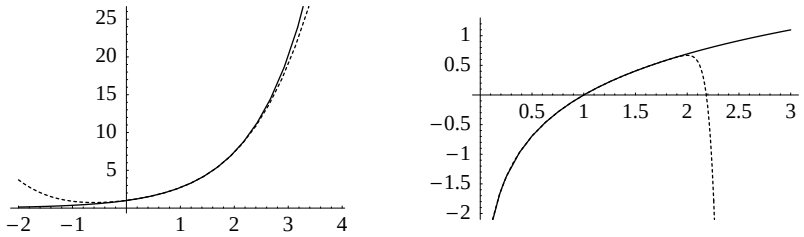
Det er viktig å huske på at Taylor-polynomet $T_n f(x)$ er en tilnærming til f, og skal vi bruke denne tilnærmingen er det som regel viktig å ha en formening om hvor stor feilen er. Dette kan vi lese ut fra restleddet $R_n f(x) = f(x) - T_n f(x)$ som kan skrives på formen

$$R_n f(x) = \frac{1}{(n+1)!}(x-a)^{n+1} f^{(n+1)}(c)$$

der c er et tall i intervallet (a, x) (intervallet (x, a) hvis $x < a$).

Vi legger merke til at når x nærmer seg a så vil feilen $R_n f(x)$ gå mot null. Dette er ikke så overaskende siden vi vet at feilen er 0 i $x = a$. Det er også interessant å se hva som skjer når n går mot uendelig og x holdes fast. Uttrykket $1/(n+1)!$ vil ganske raskt bli svært lite mens $(x-a)^{n+1}$ vil gå mot null, en eller uendelig, avhengig av om $|x-a|$ er mindre enn en, lik en eller større enn en. Den siste faktoren $f^{(n+1)}(c)$ kan essensielt oppføre seg på to måter. De peneste funksjonene er de som er slik at alle de deriverte kan begrenses av en konstant som er uavhengig av derivasjonsordenen. For eksempel vet vi at tallverdien til alle de deriverte til både $\sin x$ og $\cos x$ kan begrenses av konstanten 1. Siden uttrykket $(x-a)^{n+1}/(n+1)!$ går mot null når n går mot uendelig, uansett hva x og a er, så ser vi at for slike funksjoner vil verdien av Taylor-polynomet i x konvergere pent mot $f(x)$. De ikke så pene funksjonene er de som har deriverte som ikke kan begrenses av en konstant uavhengig av derivasjonsordenen. Slike funksjoner skal vi ikke beskjeftige oss med her.

Figurene 9.1 og 9.2 viser noen kjente funksjoner med noen tilhørende Taylor-polynomer. I figur 9.1 ser vi hvordan Taylor-polynomene får med seg flere og flere av svingningene til $\sin x$ når graden øker. Det viser seg at når n går mot uendelig så vil Taylor-polynomene konvergere mot $\sin x$ for alle verdier av x. Det samme gjelder for eksponensialfunksjonen e^x som er vist i figur 9.2 (a). For logaritmefunksjonen $\log x$ er situasjonen imidlertid annerledes. Som vi ser i figur 9.2 (b) så tilnærmes $\log x$ godt av sitt Taylor-polynom av grad 20 på intervallet $(0, 2]$, men når x blir større enn 2 bli avviket fort stort. Det viser



(a) (b)

Figur 9.2. Funksjonen e^x og dens Taylor-polynom av grad 4 (a) og funksjonen $\log x$ og dens Taylor-polynom av grad 20.

seg at dette gjelder uansett hvor høy grad Taylor-polynomet har. Studiet av når følger av polynomer som disse konvergerer mot en fast funksjon er en viktig del av matematikken som du vil møte i senere matematikkurs.

9.1.3 Beregning av verdier på polynomer

Til slutt i denne delen skal vi ta for oss noe som ikke har spesielt med Taylor-polynomer å gjøre, men som vi trenger om vi skal bruke Taylor-polynomer på en datamaskin. Vi skal utlede en effektiv algoritme for å beregne verdien i x av polynomet

$$p(x) = c_0 + c_1 x + \cdots + c_n x. \tag{9.7}$$

Legg merke til at om vi har en slik algoritme kan vi også beregne verdier på polynomer på formen $q(x) = c_0 + c_1(x-a) + \cdots + c_n(x-a)^n$. For hvis vi setter $u = x - a$ kan vi skrive q som $c_0 + c_1 u + \cdots + c_n u^n$, altså formen i (9.7), men med u som variabel.

Den opplagte måten å beregne $p(x)$ på er å regne ut de forskjellige produktene på formen $x \cdot x \cdots x$, multiplisere med koeffisientene og addere sammen. En enklere måte er lettest å vise ved et eksempel. Anta at $n = 3$; da kan vi skrive p som

$$p(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 = c_0 + x\big(c_1 + x(c_2 + x c_3)\big).$$

Dette kan vi gjøre for generell grad,

$$p(x) = \sum_{i=0}^{n} c_i x^i = c_0 + x\Big(c_1 + x\big(c_2 + \cdots + x(c_{n-1} + x c_n)\cdots\big)\Big).$$

For å regne ut dette begynner vi innerst og arbeider oss utover og bruker $(r_i)_{i=0}^n$ for å ta vare på delresultatene. Vi setter først $r_n = c_n$ og regner deretter ut

$$r_i = c_i + x\, r_{i+1}, \qquad \text{for } i = n-1,\, n-2,\, \ldots,\, 0.$$

Etter dette har vi multiplisert ut alle parentesene slik at r_0 vil ha verdien $p(x)$. Siden vi ikke trenger å ta vare på verdien av alle r_i'ene kan vi klare oss med en r. Dette gir oss følgende algoritme.

Algoritme 9.1 (Nestet multiplikasjon)**.** *La polynomet* $p(x) = c_0 + c_1x + \cdots c_nx^n$ *være gitt ved at koeffisientene er lagret i arrayen* `c` *og* x *i variabelen* `x`*. Etter beregningene*

```
r = c[n];
for (i=n-1; i>=0; i--)
   {
      r = c[i] + x*r;
   }
```

vil variabelen `r` *inneholde verdien* $p(x)$*.*

9.2 Interpolasjon og andre approksimasjonsmetoder

Når vi nå er kjent med Taylor-polynomer er det naturlig å se seg om etter andre metoder for å tilnærme funksjoner. Før vi tar for oss interpolasjon kan det være på sin plass med noen overordnede betraktninger omkring approksimasjon av funksjoner.

La oss aller først minne om at matematiske metoder og teknikker generelt kan brukes på minst to måter. Taylor-polynomer er et eksempel på en teknikk som ofte brukes for å få dypere innsikt i en funksjon, og ofte trenger vi bare papir og blyant for å gjøre beregningene. Det er derfor viktig å øve seg i bruk av Taylor-polynomer ved å regne oppgaver (med papir og blyant). Taylor-polynomer er også et nyttig verktøy for å utlede andre metoder som Newtons metode for å finne nullpunkter og Eulers metode for å løse differensialligninger. Taylor-polynomer brukes sjeldnere på datamaskin som en praktisk approksimasjonsmetode fordi vi som regel trenger en approksimasjon som ikke bare gir liten feil i nærheten av et punkt, men over et intervall.

De andre approksimasjonsmetodene vi skal se på her er for det meste praktiske metoder som brukes på datamaskiner for å tilpasse måledata med glatte kurver. Dette betyr at det ikke er så viktig å kunne bruke metodene på papiret. Derimot er det viktig å vite om det er noen begrensninger på når metoden kan brukes, hvor komplisert er algoritmen som implementerer metoden, hvilke egenskaper vil tilnærmingen ha, hvor nøyaktig er tilnærmingen, hvor følsom er metoden for avrundingsfeil også videre. Dette betyr at det er viktigere med en god forståelse av metodene mer enn fingerferdighet med å bruke dem på mindre problemer med papir og blyant (selv om det også kan være nyttig i blant). En slik grundig forståelse får du neppe av å lese det som står her, men forhåpentligvis får du en liten smakebit på hvordan metodene framkommer og hva de kan brukes til.

Når vi skal velge oss en approksimasjonsmetode er det to hovedvalg vi må gjøre. Det første er å bestemme oss for hvilken form approksimasjonen skal ha. Skal den være et polynom, en trigonometrisk funksjon, en rasjonal funksjon, en eksponensialfunksjon eller noe annet? Det fins gode metoder basert på alle disse funksjonsklassene, men det vanligste er å bruke approksimasjonsmetoder basert på polynomer eller trigonometriske funksjoner. Og av disse to klassene er nok polynomer det vanligste. Dette har sammenheng med det vi før har vært inne på, at polynomer egner seg særdeles godt til beregninger på datamaskin.

Det andre valget dreier seg om hvilken metode vi velger for å bestemme approksimasjonen når klassen av funksjoner vi skal bruke er valgt. Vi skal se litt nærmere på dette når vi først har tatt for oss et alternativ til Taylor-polynomer.

9.2.1 Interpolasjon med polynomer

Når vi tilnærmer en funksjon f med Taylor-polynomer må vi kunne beregne deriverte av f. Det er mange situasjoner der dette ikke er mulig, den vanligste er kanskje når f bare er kjent ved måleverdier, slik som når vi måler temperaturen ved ulike tidspunkter eller når vi sampler et lydsignal. I kapittel 6 så vi litt på hvordan den deriverte kan tilnærmes ut fra kjente funksjonsverdier, men det er som regel vel så bra å beregne approksimasjonen direkte fra funksjonsverdiene uten å gå veien om deriverte.

Naturlig nok er tilnærmingene våre polynomer når vi bruker polynominterpolasjon, og tilnærmingen bestemmes ved at vi krever at feilen skal være null i noen isolerte punkter. Et velkjent eksempel er sekanten til en funksjon f. Da velger vi to forskjellige punkter x_0 og x_1 og lar tilnærmingen p_1 være den rette linja (det polynomet av grad 1) som har samme verdi som f i disse punktene. Med andre ord så krever vi at

$$f(x_0) = p_1(x_0), \quad f(x_1) = p_1(x_1).$$

Hvis vi skriver p_1 på standardformen $p_1(x) = b_0 + b_1 x$ får vi et ligningssystem som består av to ligninger med de to ukjente b_0 og b_1 som har løsningen

$$b_0 = \frac{f(x_0)x_1 - f(x_1)x_0}{x_1 - x_0}, \quad b_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0}.$$

Siden vi har antatt at $x_0 \neq x_1$ gir disse formlene alltid mening. Dette betyr at p_1 er gitt ved

$$p_1(x) = \frac{f(x_1)x_0 - f(x_0)x_1}{x_1 - x_0} + \frac{f(x_1) - f(x_0)}{x_1 - x_0}x. \tag{9.8}$$

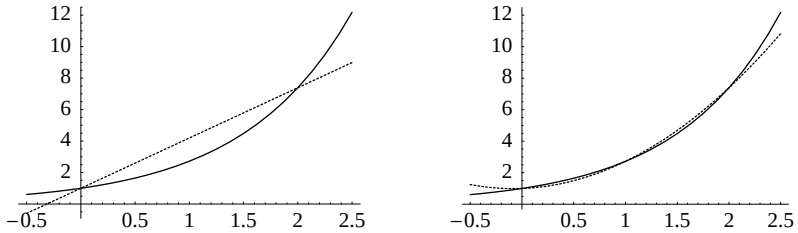
Dette polynomet kan også skrives som

$$p_1(x) = f(x_0)\frac{x_1 - x}{x_1 - x_0} + f(x_1)\frac{x - x_0}{x_1 - x_0}. \tag{9.9}$$

At dette er riktig ser vi ved at høyresiden er et førstegradspolynom som har verdien $f(x_0)$ når $x = x_0$ og verdien $f(x_1)$ når $x = x_1$. Dette illustrerer igjen fenomenet med at et polynom kan skrives på flere måter. Formen (9.9) er spesielt hendig siden vi der kan bruke de to funksjonsverdiene $f(x_0)$ og $f(x_1)$ som koeffisienter. Dette kommer av at polynomet $\ell_0(x) = (x_1 - x)/(x_1 - x_0)$ tilfredstiller betingelsene $\ell_0(x_0) = 1$ og $\ell_0(x_1) = 0$ mens $\ell_1(x) = (x - x_0)/(x_1 - x_0)$ tilfredstiller betingelsene $\ell_1(x_0) = 0$ og $\ell_1(x_1) = 1$. Dette skriver vi ofte som $\ell_i(x_j) = \delta_{i,j}$ for $i, j = 0, 1$ der symbolet $\delta_{i,j}$ er definert som

$$\delta_{i,j} = \begin{cases} 1, & \text{hvis } i = j; \\ 0, & \text{ellers.} \end{cases}$$

La oss nå forsøke å øke graden. Siden et andregradspolynom har tre frie koeffisienter er det rimelig å håpe på at vi kan tvinge et slikt polynom gjennom tre gitte punkter x_0, x_1 og x_2 med tilhørende funksjonsverdier $f(x_0)$, $f(x_1)$ og $f(x_2)$. Hvis vi skriver polynomet som $p_2(x) = b_0 + b_1 x + b_2 x^2$ så vil de tre interpolasjonsbetingelsene gi tre ligninger for de tre ukjente b_0, b_1 og b_2. Dette ligningssystemet har ingen spesiell struktur som vi



Figur 9.3. Funksjonen e^x og med sekanten som interpolerer i $x = 0$ og $x = 2$ i (a), og parabelen som interpolerer i $x = 0$, $x = 1$ og $x = 2$ i (b) .

kan utnytte for å forenkle løsningsprosedyren. La oss derfor se om det lar seg gjøre å generalisere den spesielle formen for sekanten som vi fant i (9.9).

Trikset består i finne andregradspolynomer som er 0 i alle punkter unntatt ett, der de har verdien 1. Det er lett å finne et polynom som er 0 i x_1 og x_2. Polynomet $q(x) = c(x - x_1)(x - x_2)$ har åpenbart denne egenskapen uansett hva konstanten c er. Det vil vanligvis ikke ha verdien 1 i x_0, men det kan vi fort ordne ved å velge c på riktig måte. Betingelsen

$$q(x_0) = c(x_0 - x_1)(x_0 - x_2) = 1$$

gir $c = 1/\big((x_0 - x_1)(x_0 - x_2)\big)$. Vi ser derfor at polynomet

$$\ell_{0,2}(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}$$

tar verdien 1 i x_0 og verdien 0 i x_1 og x_2. Denne konstruksjonen kan vi gjenta med de andre punktene. Dette gir oss de to polynomene $\ell_{1,2}$ og $\ell_{2,2}$,

$$\ell_{1,2}(x) = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)},$$
$$\ell_{2,2}(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}.$$

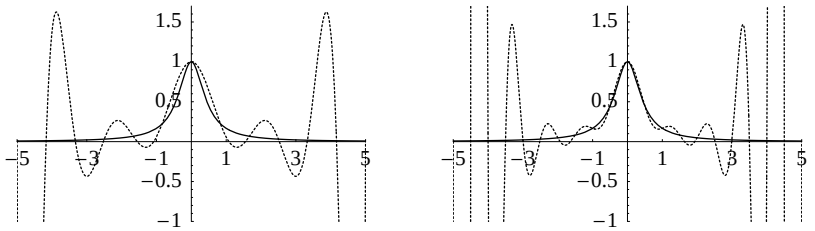
Vi ser at $\ell_{1,2}$ har verdien 1 i x_1 og 0 i x_0 og x_2, mens $\ell_{2,2}$ har verdien 1 i x_2 og verdien 0 i x_0 og x_1. Men dette betyr at polynomet

$$p_2(x) = f(x_0)\ell_{0,0}(x) + f(x_1)\ell_{1,2}(x) + f(x_2)\ell_{2,2}(x)$$

er av grad 2 og har samme verdi som f i punktene x_0, x_1 og x_2 siden hvert av de tre leddene bare er ulik 0 i ett punkt, og i dette punktet har de riktig funksjonsverdi. Når vi adderer de tre leddene sammen "ødelegger de derfor ikke for hverandre".

Figur 9.3 viser eksempler på interpolasjon av eksponensialfunksjonen med en rett linje og med en parabel (polynomene er stiplet). Vi ser at parabelen er en så god tilnærming at det er vanskelig å skille den fra funksjonen den tilnærmer. Hvis vi øker graden og interpolerer e^x i flere uniformt spredte punkter kan vi få feilen så liten vi måtte ønske.

Konstruksjonen av interpolerende polynomer kan generaliseres til vilkårlig grad.



Figur 9.4. Funksjonen $1/(1+5x^2)$ med polynomet av grad 11 som interpolerer 12 uniformt spredte punkter i intervallet $[-5,5]$ (a) og interpolanten av grad 19 som interpolerer 20 uniformt spredte punkter i det samme intervallet (b) (polynomene er stiplet).

Setning 9.2. *La funksjonen f være gitt sammen med de $n+1$ punktene $x_0, x_1, \ldots, x_n$, og definer de $n+1$ polynomene $\{\ell_{i,n}\}_{i=0}^n$ ved*

$$\ell_{i,n}(x) = \prod_{\substack{k=0 \\ k\neq i}}^{n} \frac{(x-x_k)}{(x_i-x_k)}.$$

Da er polynomet $p_n(x) = \sum_{i=0}^n f(x_i)\ell_{i,n}(x)$ det eneste polynomet av grad n som tilfredstiller interpolasjonsbetingelsene $p_n(x_j) = f(x_j)$ for $j = 0, 1, \ldots, n$.

Som i det kvadratiske tilfellet er det ikke så vanskelig å sjekke at $\ell_{i,n}$ er 1 i x_i og 0 i alle de andre gitte punktene. Ved å multiplisere med $f(x_i)$ får vi derfor et polynom som har verdien $f(x_i)$ i x_i og verdien 0 i alle de andre punktene. Når vi så adderer opp alle leddene får vi polynomet av grad d som tilfredstiller interpolasjonsbetingelsene.

Det som gjenstår er å vise at det bare fins ett polynom som tilfredstiller interpolasjonsbetingelsene. Vi skal anta at det fins et polynom r som også har denne egenskapen og vise at da må r være lik p. La oss se på differansen $e = p-r$ mellom de to polynomene. Siden både p og r har verdien $f(x_i)$ i x_i for $i = 0, \ldots, n$ ser vi at e er 0 i alle de $n+1$ punktene. Dette polynomet av grad n har altså $n+1$ nullpunkter. Men fra algebraens fundamentalteorem vet vi at dette er umulig for et polynom av grad n så da er eneste mulighet at e er identisk null (alle koeffisientene er null). Men hvis $e = p - r = 0$ må vi ha $r = p$. Altså har vi bare ett polynom som tilfredstiller interpolasjonsbetingelsene.

Figur 9.4 viser polynomer av forholdsvis høy grad (11 og 19) som interpolerer funksjonen $1/(1+5x^2)$. Noe overaskende så ser vi at feilen er forholdsvis stor, særlig nær endene av intervallet, og den blir større ettersom graden øker (plottene er klippet i y-retningen). I midten av intervallet er imidlertid tilnærmingen bra og ser ut til å bedres med økende grad. Forøvrig er kanskje det mest iøynefallende de voldsomme svingningene i de interpolerende polynomene. Selv når feilen er liten vil et interpolerende polynom vanligvis svinge rundt funksjonen det interpolerer, det ligger i interpolasjonens natur. Du vil se det samme om du plasserer en bøyelig linjal på et bord og forsøker å tvinge den til å gå gjennom noen faste punkter.

Selv om figur 9.4 kan virke litt deprimerende med tanke på om interpolasjon er en god approksimasjonsmetode så fins det en del positive resultater. Resultatene forutsetter

at funksjonen f som vi interpolerer er glatt (en funksjon som kan deriveres mange ganger uten problemer). Dersom vi lar interpolasjonspunktene bevege seg mot hverandre og bli konsentrert i et stadig mindre intervall vil feilen på dette intervallet gå mot null når intervallbredden går mot null. I grensen, når alle punktene blir like, vil det interpolerende polynomet falle sammen med Taylor-polynomet av samme grad. Det går også an å vise at for en glatt funksjon definert på et intervall $[a, b]$ så går det for hvert heltall $n \geq 0$ an å finne en samling av $n + 1$ interpolasjonspunkter slik at hvis vi interpolerer i disse punktene med et polynom av grad n vil feilen gå mot null når graden n går mot uendelig.

9.2.2 Andre metoder

De to approksimasjonsmetodene vi har sett på så langt er begge basert på å konstruere polynomer som reproduserer informasjon om f i diskrete punkter. Taylor-polynomer reproduserer verdien og de første deriverte til f i ett punkt, mens vi ved interpolasjon reproduserer funksjonsverdiene til f i flere punkter. Det er imidlertid andre måter å finne tilnærminger på.

En vanlig metode er å finne det polynomet som gjør at feilen i approksimasjonen blir minst mulig. Dette kan høres greit ut, men innebærer noen uventede utfordringer. Spørsmålet er nemlig hvordan vi kan måle feilen når vi tilnærmer f med et polynom p. Feilfunksjonen $e = f - p$ er jo en funksjon, og det er dermed ikke uten videre så lett å se hva det vil si å gjøre denne feilen minst mulig. Nøkkelen ligger i hvordan vi kan måle størrelsen på en funksjon.

La oss anta at funksjonen f vi skal tilnærme er definert på et interval $[a, b]$. Et mål for størrelsen til feilen har vi ved uttrykket

$$\max_{x\in[a,b]} \big|f(x) - p(x)\big|. \tag{9.10}$$

Dette svarer til at vi benytter verdien av feilfunksjonen i det punktet der den er størst som et mål på feilen. Det å minimere feilen betyr da å finne et polynom p slik at dette maksimale avviket blir minst mulig. Hvis vi bruker polynomer av grad n og for enkelhets skyld skriver dem på formen $p(x) = c_0 + c_1x + \cdots c_nx^n$ så ser vi at feilen i (9.10) kan skrives som

$$\epsilon_1(c_0, \ldots, c_n) = \max_{x\in[a,b]} \big|f(x) - (c_0 + c_1x + \cdots + c_nx^n)\big|.$$

Å finne polynomet av grad n som gjør feilen minst mulig innebærer derfor å minimere funksjonen ϵ_1 med hensyn på variablene c_0, c_1, …, c_n.

En annen måte å måle feilen på er ved integralet

$$\int_a^b \big|f(x) - p(x)\big|\, dx,$$

altså arealet av forskjellen mellom de to funksjonene. I dette tilfellet blir funksjonen vi skal minimere

$$\epsilon_2(c_0, \ldots, c_n) = \int_a^b \big|f(x) - (c_0 + c_1x + \cdots + c_nx^n)\big|\, dx.$$

Igjen kan vi bestemme en tilnærming til f ved å gjøre dette uttrykket minst mulig. Vi ser da at det er viktig å ha med tallverditegnet, for hvis ikke kan vi få feilen til å bli veldig liten ved å velge p slik at arealet mellom funksjonene er stort, men med negativt fortegn, noe som ikke svarer til at p er nær f.

Ulempen med disse to metodene er at det for begge er forholdsvis vanskelig å bestemme polynomene som gjør feilen minst mulig. Fra teorien for funksjoner av flere variable vet vi at vi kan finne ekstremalpunktene til en deriverbar funksjon ved å løse ligningene vi får ved å sette gradienten (de partielle deriverte) til null. Dessverre inneholder uttrykkene for ϵ_1 og ϵ_2 tallverdifunksjonen som ikke er deriverbar overalt ($|x|$ har en knekk i $x = 0$). Det er fremdeles mulig å bestemme minimumspunktene til ϵ_1 og ϵ_2, men den eneste muligheten er å gjøre det ved iterative algoritmer som først gjetter på en løsning og så stadig forsøker å forbedre denne i håp om at prosessen vil konvergere mot polynomet som gjør feilen minst mulig.

Et tredje feilmål er populært fordi en med det kan finne polynomet som gjør feilen minst mulig uten å gjøre slike iterasjoner. Vi måler da feilen ved

$$\int_a^b \big(f(x) - p(x)\big)^2 \, dx.$$

Siden vi kvadrerer feilfunksjonen vil uttrykket under integraltegnet aldri bli negativt så vi slipper unna tallverditegnet. Setter vi inn for p får vi

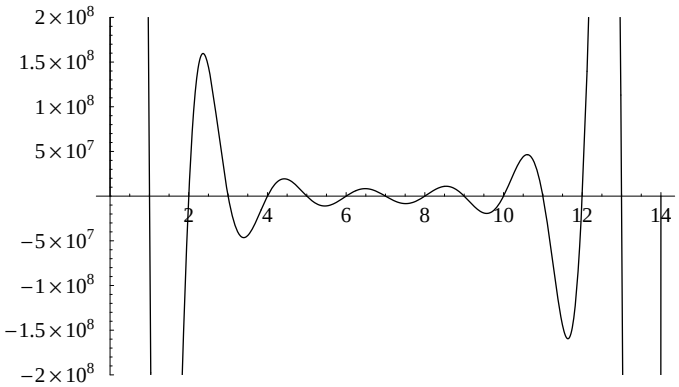
$$\epsilon_3(c_0, \ldots, c_n) = \int_a^b \big(f(x) - (c_0 + c_1 x + \cdots + c_n x^n)\big)^2 \, dx.$$

Dette uttrykket kan vi derivere med hensyn på hver av de $n + 1$ koeffisientene c_0, …, c_n. Setter vi disse deriverte lik null får vi et lineært ligningssystem med $n + 1$ lineære ligninger i de $n + 1$ ukjente koeffisientene c_0, c_1, …, c_n. Det er ikke så vanskelig å vise at dette ligningssystemet har nøyaktig en løsning og at denne løsningen gir det eneste minimumspunktet til ϵ_3.

9.3 Valg av basis er viktig, Bernstein-basisen

Så langt i dette kapitlet har vi sett flere eksempler på at det kan være fordelaktig å ikke alltid skrive et polynom av grad n på standardformen $c_0 + c_1 x + \cdots + c_n x^n$. Ved å bruke andre former så vi at det ble mye enklere å finne både Taylor-polynomer og interpolerende polynomer. Ikke overaskende har de ulike måtene å skrive polynomer på ulik følsomhet for avrundingsfeil også. I denne seksjonen skal vi presentere en spesielt gunstig måte å skrive polynomer på, ikke minst med tanke på å gjøre avrundingsfeilen liten.

Men hvorfor trenger vi å velge oss en måte å skrive polynomer på, kan vi ikke bruke den formen som til en hver tid er mest hendig? Hvis vi arbeider med papir og blyant er det selvsagt riktig, da er det ingen som legger noen begrensninger på hvordan vi representerer polynomer. Arbeider vi på en datamaskin derimot blir det fort tungvint å bruke flere forskjellige polynomformer. Skal vi lagre et polynom er det mest naturlig å lagre koeffisientene i forhold til en eller annen polynomform. Hvis vi da skal bruke ulike polynomformer må vi si fra hvilken polynomform som er brukt hver gang vi lagrer et



Figur 9.5. Wilkinson-polynomet $w(x)$ (noe klipping i y-retningen).

polynom og vi må ha programvare for å oversette mellom de forskjellige formene. En ting er at det tar tid å skrive slik programvare, en annen at hver gang vi gjør en slik konvertering gjør vi en avrundingsfeil som dermed ødelegger noe av nøyaktigheten i vår representasjon. Det er derfor flere fordeler med å velge seg en fast representasjon når vi skal lage et programsystem.

9.3.1 Et problematisk polynom

Aller først trenger vil litt terminologi omkring ulike skrivemåter for polynomer. Når vi skriver polynomer på formen $c_0 + c_1x + \cdots + c_nx^n$ så er polynomet dannet ved *lineære kombinasjoner* av polynomene 1, x, ..., x^n. Disse enkle polynomene som vi bruker som byggeklosser kalles *basis-polynomer* og samlingen av alle disse basis-polynomene kalles *potens-basisen*. Som vi har sett fins det andre måter å skrive polynomer på, og dette svarer til å bruke andre basiser. Skriver vi polynomer som $c_0+c_1(x-a)+\cdots+c_n(x-a)^n$ bruker vi den skiftede potensbasisen 1, $x-a$, ..., $(x-a)^n$, mens basisen $\{\ell_{i,n}\}_{i=0}^n$ som vi brukte i forbindelse med interpolasjon kalles *Lagrange-basisen*[2].

Med tanke på avrundingsfeil er potensbasisen spesielt problematisk, særlig når vi beveger oss langt bort fra $x = 0$. Som et eksempel skal vi se på polynomet

$$w(x) = \prod_{i=0}^{14}(x - i)$$

som ofte kalles Wilkinson-polynomet[3], se figur 9.5. Hvis vi multipliserer ut parentesene

[2]Etter den franske matematikeren Joseph-Louis Lagrange (1736–1813).

[3]Etter James Wilkinson (1919–1986), engelsk matematiker. Wilkinson var innvolvert i arbeidet med den første engelske datamaskinen ACE og var en av pionerene innen analyse av avrundingsfeil.

og skriver polynomet på potensform finner vi at polynomet også kan skrives

$$\begin{aligned} w(x) = {} & 87178291200\,x - 283465647360\,x^2 + 392156797824\,x^3 - 310989260400\,x^4 \\ & + 159721605680\,x^5 - 56663366760\,x^6 + 14409322928\,x^7 \\ & - 2681453775\,x^8 + 368411615\,x^9 - 37312275\,x^{10} + 2749747\,x^{11} \\ & - 143325\,x^{12} + 5005\,x^{13} - 105\,x^{14} + x^{15}. \end{aligned}$$

Hvis vi representerer koeffisientene som 64-bits flyttall forteller Mathematica oss at $w(13) = 264$. Dette er selvsagt helt feil siden vi har konstruert polynomet slik at det er 0 i alle heltallene mellom 0 og 14. Spørsmålet er hva som er årsaken til problemet. Er det bare rutinene i Mathematica for å beregne verdien av et polynom på potensform som er dårlige, er det potensformen som skaper problemene eller er polynomet i seg selv så problematisk at vi ikke kan forvente et mer nøyaktig svar? Polynomet er helt klart problematisk i seg selv, men det viser seg at potensformen forsterker problemet. For å vise dette skal vi representere polynomet i en annen basis og se at om vi da regner ut $w(13)$ så får vi et bedre resultat.

Vår nye basis er spesielt tilpasset et intervall $[a, b]$ og kalles *Bernstein-basisen*[4]. Den er gitt ved

$$b_{i,n}(x) = \binom{n}{i} \Big(\frac{x-a}{b-a}\Big)^i \Big(\frac{b-x}{b-a}\Big)^{n-i}, \qquad i = 0, 1, \ldots, n.$$

Hvis vi setter $[a, b] = [0, 1]$ forenkler høyresiden seg og basisen blir

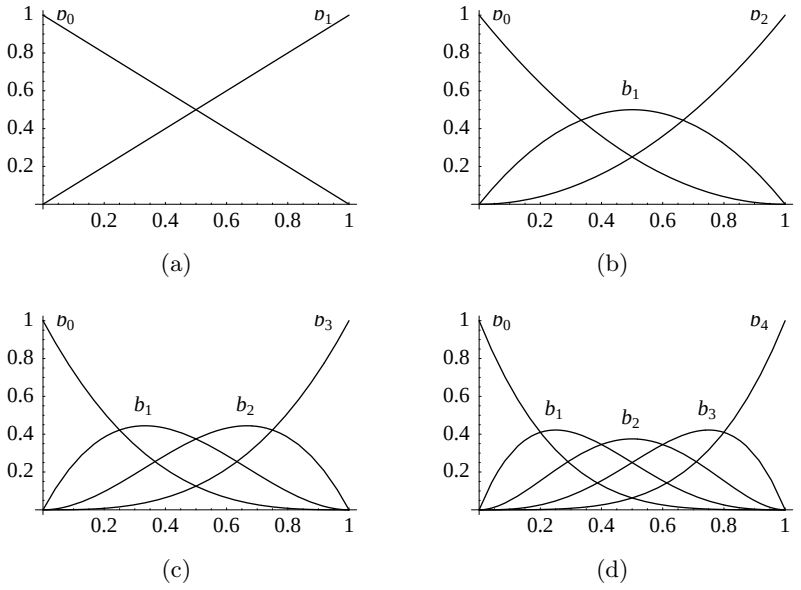
$$b_{i,n}(x) = \binom{n}{i} x^i (1-x)^{n-i}, \qquad i = 0, 1, \ldots, n.$$

De ulike polynomene i Bernstein-basisen av grad 1–4 på intervallet $[0, 1]$ er vist i figur 9.6. Når det er klart fra sammenhengen hvilken grad vi bruker vil vi ofte utelate den andre indeksen n i $b_{i,n}$.

Bernstein-basisen på et generelt intervall $[a, b]$ framkommer ved å flytte venstre ende til a og så strekke eller krympe funksjonene til høyre ende faller i b. Plott av basisen for et vilkårlig intervall ser derfor akkurat ut som plottene i figur 9.6 bortsett fra at tallene på x-aksen vil være annerledes.

Wilkinson-polynomet $w(x)$ kan også skrives som $w(x) = \sum_{i=0}^{15} c_i b_{i,15}(x)$ for passende koeffisienter (c_i). Gjør vi det og representerer koeffisientene som 64-bits flyttall kan vi regne ut $w(13)$ i denne representasjonen. Resultatet vi da får er $w(13) = -0.0000244541$. Selv om dette også er galt er det langt fra så galt som svaret vi fikk med potensbasisen. Det viser seg at Bernstein-basisen generelt oppfører seg bedre numerisk enn potensbasisen. Bernstein-basisen har også mange andre gode egenskaper som gjør at den ofte brukes ved representasjon av polynomer på datamaskiner. Ikke minst er Bernstein-basisen utgangspunktet for de såkalte Bezier-kurvene.

[4]Sergi Natanovich Bernstein (1880–1968) var født i Ukraina, men arbeidet i Russland (Sovjetunionen)



Figur 9.6. Polynomene i Bernsteinbasisen av grad 1 (a), 2 (b), 3 (c) og 4 (d).

9.3.2 Noen egenskaper ved Bernstein-basisen

Når polynombasisen er bestemt er all informasjon om et polynom gitt ved koeffisientene. Det burde derfor være mulig å lese polynomets oppførsel ut fra koeffisientene. For de fleste polynombasiser er dette ikke så enkelt, men den viktigste egenskapen til Bernstein-basisen er at koeffisientene modellerer polynomet på en intuitiv og naturlig måte. Før vi ser hva dette innebærer må vi ta for oss noen viktige egenskaper ved Bernstein-basisen. Disse egenskapene er stort sett ganske enkle å utlede, så vi overlater bevisene til oppgaver.

Lemma 9.3. *Bernstein-basisen har blant annet følgende egenskaper:*

1. *Alle polynomene i Bernstein-basisen er alltid større enn eller lik null på intervallet* $[a, b]$,
$$b_{i,n}(x) \geq 0 \qquad \text{for alle } x \in [a, b].$$

2. *Polynomene i Bernsteinbasisen summerer seg opp til 1,*
$$\sum_{i=0}^{n} b_{i,n}(x) = 1, \qquad \text{for alle } x \in \mathbb{R}.$$

3. *Basispolynomet* $b_{i,n}(x)$ *har sitt maksimum på intervallet* $[a, b]$ *i punktet* $x_i^* = a + i(b - a)/n$.

4. *I* $x = a$ *er* $b_{0,n}(a) = 1$ *mens alle de andre basispolynomene er 0. I* $x = b$ *er* $b_{n,n}(b) = 1$ *mens alle de andre basispolynomene er 0.*

5. *Den deriverte av $b_{i,n}$ er gitt ved*

$$b'_{i,n}(x) = \begin{cases} -n\, b_{0,n-1}(x), & \text{når } i = 0; \\ n\left(b_{i-1,n-1}(x) - b_{i,n-1}(x)\right), & \text{når } 1 \le i \le n-1; \\ n\, b_{n-1,n-1}(x), & \text{når } i = n. \end{cases}$$

6. *Med $u = (x-a)/(b-a)$ så er $1-u = (b-x)/(b-a)$ slik at*

$$b_{i,n}(x) = \binom{n}{i}\Big(\frac{x-a}{b-a}\Big)^i\Big(\frac{b-x}{b-a}\Big)^{n-i} = \binom{n}{i}u^i(1-u)^{n-i}.$$

Egenskap 6 er svært nyttig ved programmering av polynomer på Bernstein-form i og med at polynomer på et generelt intervall kan omskrives til polynomer på intervallet $[0,1]$. Ved hjelp av denne transformasjonen kan vi også oversette egenskaper som gjelder på intervallet $[0,1]$ til det mer generelle intervallet $[a,b]$.

9.3.3 Egenskaper ved polynomer på Bernstein-form

Fra egenskapene over kan vi nå utlede egenskaper for polynomer uttrykt i Bernstein-basisen. For enkelhets skyld gjør vi dette i det kubiske tilfellet ($n = 3$).

For å forenkle terminologien skal vi heretter kalle et polynom som er skrevet i Bernstein-basisen et *Bernstein-polynom*. La

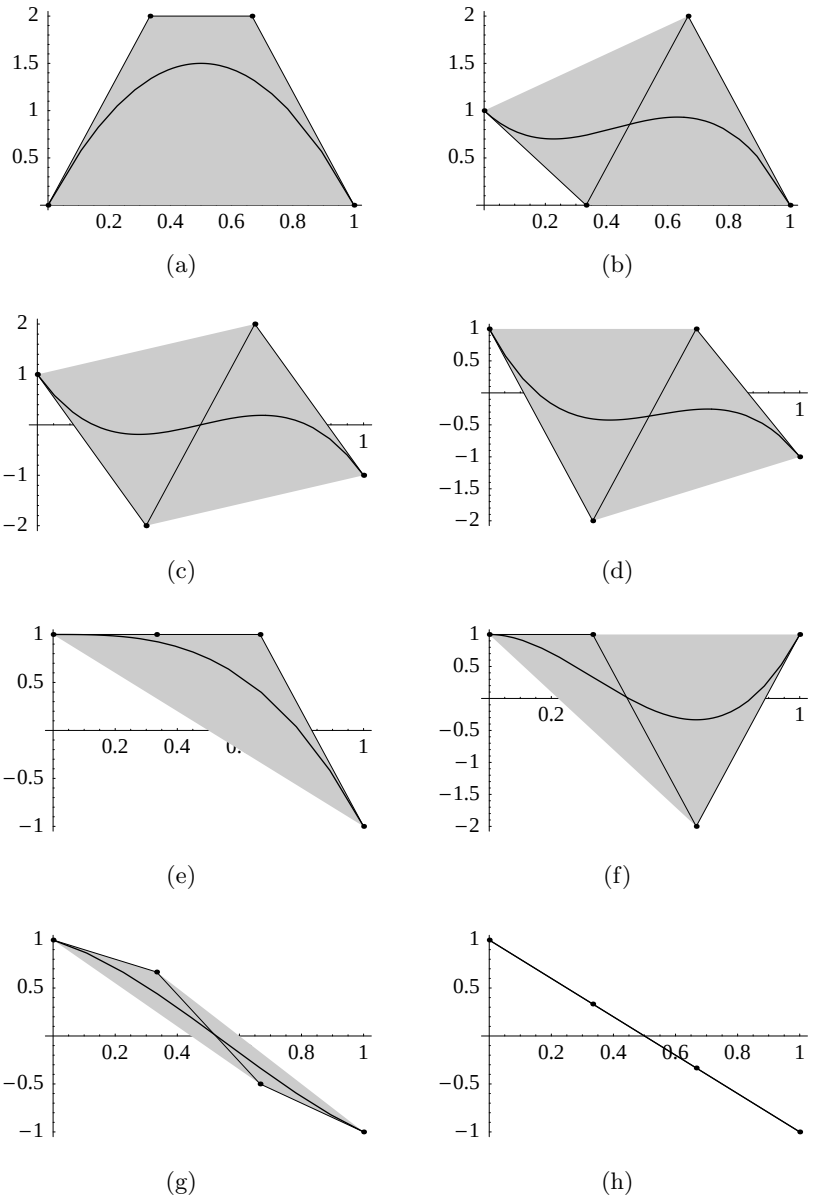
$$p(x) = c_0b_0(x) + c_1b_1(x) + c_2b_2(x) + c_3b_3(x)$$

være et kubisk Bernstein-polynom på intervallet $[a,b] = [0,1]$. Hvis x ligger i intervallet $[a,b]$ så vet vi fra egenskapene 1 og 2 at de fire basispolynomene er større enn eller lik null og summerer seg til 1. Dette betyr at $p(x)$ er et veiet gjennomsnitt av de fire koeffisientene $c_0, \ldots, c_3$. Siden et gjennomsnitt av en samling med tall alltid må være større enn det minste tallet og mindre enn det største tallet, gjelder det samme her — tallet $p(x)$ må ligge mellom $\min_i c_i$ og $\max_i c_i$ (hvis alle fire c'ene er like vil $p(x)$ være lik denne felles verdien).

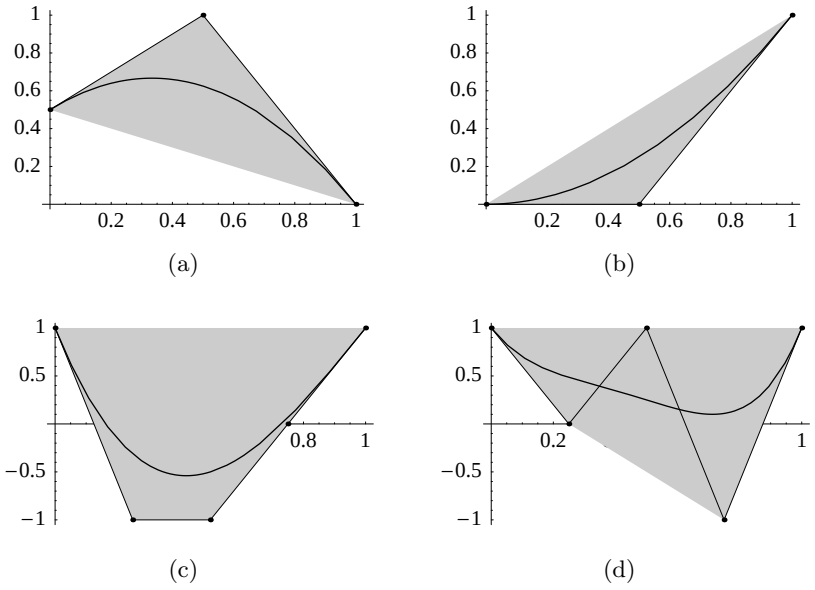
Hvis vi tar en titt på figur 9.6 (c) så ser vi at når vi beveger oss fra $x = 0$ til $x = 1$ så varierer størrelsen på basispolynomene. Spesielt så har alle basispolynomene et entydig maksimum i $[a,b]$, og fra egenskap 3 over vet vi at b_i har sitt maksimum i $x = i/3$. Dette betyr at koeffisienten c_i bidrar mest til $p(x)$ i $x = i/3$. Hvis vi skal tegne ut koffisienten c_i bør vi derfor tegne den med x-koordinat $i/3$, mens y-koordinaten er c_i. Punktet i planet gitt ved $(i/3, c_i)$ kalles det i'te *kontrollpunktet* til p slik at p til sammen har de fire kontrollpunktene $\{(i/3, c_i)\}_{i=0}^3$. Hvis vi forbinder kontrollpunktene med rette linjer får vi *kontrollpolygonet* til p.

Dette er motivasjonen bak den generelle definisjonen av kontrollpunkter og kontrollpolygon.

Definisjon 9.4. *La Bernstein-polynomet $p(x) = \sum_{i=0}^n c_ib_{i,n}(x)$, definert på intervallet $[a,b]$, være gitt. De $n+1$ punktene i planet gitt ved $c_i^p = \big(a + i(b-a)/n, c_i\big)$ for $i = 0$, $1, \ldots, n$, kalles kontrollpunktene til p mens den brudne linja som framkommer ved å forbinde kontrollpunktene med rette linjer kalles kontrollpolygonet til p.*



Figur 9.7. Åtte eksempler på kubiske Bernstein-polynomer med tilhørende kontrollpolygoner. Koeffisientene er $(0, 2, 2, 0)$ i (a), $(1, 0, 2, 0)$i (b), $(1, -2, 2, -1)$ i (c), $(1, -2, 1, -1)$ i (d), $(1, 1, 1, -1)$ i (e), $(1, 1, -2, 1)$ i (f), $(1, 2/3, -1/2, -1)$ i (g) og $(1, 1/3, -1/3, -1)$ i (h).



Figur 9.8. Eksempler på kvadratiske Bernstein-polynomer med tilhørende kontrollpolygoner ((a) og (b)) og Bernstein-polynomer av grad 4 ((c) og (d)).

Tolv eksempler på kubiske Bernstein-polynomer med tilhørende kontrollpunkter og kontrollpolygon er vist i figur 9.7 og 9.8. I alle tilfellene ser vi hvordan Bernstein-polynomet er en utglattet versjon av sitt kontrollpolygon. Det samlede visuelle inntrykket av sammenhengen mellom funksjon og kontrollpolygon har flere ingredienser.

Lemma 9.5. *Et Bernstein-polynom $p(x)$ definert på intervallet $[a, b]$ har blant annet følgende egenskaper:*

1. *Verdien av polynomet i $x = a$ faller sammen med det første kontrollpunktet, og verdien av polynomet i $x = b$ faller sammen med det siste kontrollpunktet.*

2. *Tangenten til polynomet i $x = a$ har samme retning som linja fra det første til det andre kontrollpunktet og tangenten i $x = b$ har samme retning som linja fra det nest siste til det siste kontrollpunktet.*

3. *Verdien av polynomet ligger alltid mellom verdien til minste og største koeffisient.*

Disse egenskapene følger direkte fra egenskaper ved basispolynomene $b_{i,n}$. For eksempel så følger egenskap 1 fra egenskap 4 i lemma 9.3 og egenskap 3 fra egenskap 1 og 2 i lemma 9.3, se oppgavene. Egenskap 2 følger fra egenskap 5 i lemma 9.3, men dette krever litt regning så vi tar med denne utledningen her.

Bevis for egenskap 2. La oss igjen holde oss til kubiske Bernstein-polynomer på in-

tervallet $[0,1]$. Den deriverte ser vi da er gitt ved

$$\begin{aligned} p'(x) &= \sum_{i=0}^{3} c_i b'_{i,3}(x) \\ &= 3\Big(-c_0 b_{0,2}(x) + c_1\big(b_{0,2}(x) - b_{1,2}(x)\big) + c_2\big(b_{1,2}(x) - b_{2,2}(x)\big) + c_3 b_{2,2}(x)\Big) \\ &= 3(c_1 - c_0)b_{0,2}(x) + 3(c_2 - c_1)b_{1,2}(x) + 3(c_3 - c_2)b_{2,2}(x). \end{aligned}$$

Med andre ord ser vi at den deriverte er et kvadratisk Bernstein-polynom med koeffisienter som er gitt ved differansen mellom to nabokoeffisienter til det opprinnelige polynomet, multiplisert med 3. Spesielt så får vi at

$$p'(0) = 3(c_1 - c_0), \qquad p'(1) = 3(c_3 - c_2),$$

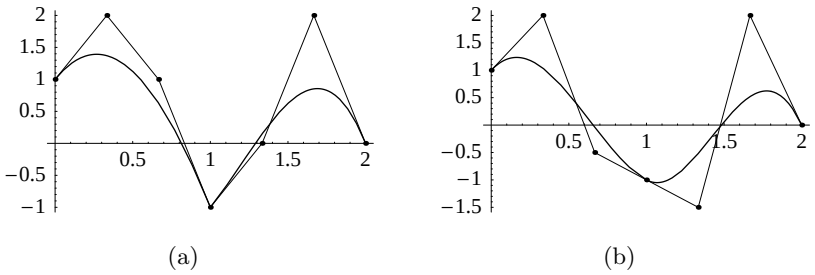
siden $b_{0,2}(x) = 1$ mens $b_{1,2}(0) = b_{2,2}(0) = 0$, og tilsvarende så er $b_{2,2}(1) = 1$ mens $b_{0,2}(1) = b_{1,2}(1) = 0$. Siden vi også har $p(0) = c_0$ fra egenskap 1 så vet vi dermed at tangenten til p i $x = 0$ har ligningen $h(x) = c_0 + 3(c_1 - c_0)x$. Vi har dessuten $h(1/3) = c_1$ så tangenten går gjennom de to første kontrollpunktene til p og faller dermed sammen med det første segmentet av kontrollpolygonet. ∎

Vi har skissert enda en egenskap ved Bernstein-polynomer i figurene: vi ser at Bernstein-polynomet i hvert tilfelle holder seg innenfor det grå området definert av kontrollpunktene. Før vi kan formulere dette mer presist må vi vite hvordan dette området framkommer. Det grå området $\mathbb{G}$ er gitt ved den *konvekse innhylningen* til kontrollpunktene. Det at området er *konvekst* betyr at om to punkter ligger i $\mathbb{G}$ så vil også alle punktene på linjesegmentet som forbinder punktene ligge i $\mathbb{G}$. Nå går det an å vise at den konvekse innhylningen til en samling punkter er den minste konvekse mengden som inneholder alle punktene i samlingen. Det grå området i figurene er dermed den minste mengden i planet som inneholder alle kontrollpunktene til det aktuelle Bernstein-polynomet. Med denne bakgrunnen kan vi formulere den siste egenskapen.

Lemma 9.6. *La $p = \sum_{i=0}^{n} c_i b_{i,n}(x)$ være et Bernstein-polynom. For enhver x i intervallet $[a,b]$ så ligger punktet $\big(x, p(x)\big)$ i den konvekse innhylningen til kontrollpunktene til p.*

Beviset for dette vil det føre for langt å ta med her, men det hele bunner i at den konvekse innhylningen av en samling punkter kan tolkes som samlingen av alle veide gjennomsnitt av punkter i samlingen. Vi ser derved sammenhengen med Bernstein-polynomer som jo er et veiet gjennomsnitt av sine koeffisienter.

Bernstein-polynomer har en mengde andre elegante egenskaper som vi ikke kan gå innpå her, men kjernen i de aller fleste av disse egenskapene er den nære sammenhengen mellom polynomet selv og dets kontrollpolygon. Det typiske er at en egenskap ved et Bernstein-polynom som regel kan karakteriseres ved en tilsvarende egenskap ved polynomets kontrollpolygon. Dette er særlig hendig ved beregninger siden vi i stedenfor å gjøre beregninger direkte med polynomet da kan gjøre tilsvarende beregninger med kontrollpolygonet. Fordelen med dette er at kontrollpolygonet matematisk sett er enkelt siden det bare består av n rette linjesegmenter.



Figur 9.9. Sammenlenking av to kubiske Bernstein-polynomer i $x = 1$. I (a) er resultatet kontinuerlig i skjøten, men den deriverte er diskontinuerlig, mens i (b) er både funksjonsverdi og derivert kontinuerlige i skjøten.

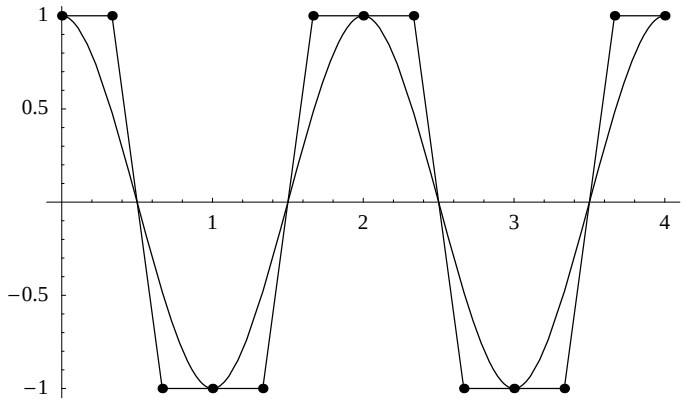
9.3.4 Sammenlenking av Bernstein-polynomer

De mange fine egenskapene ved Bernstein-basisen gjør at den ofte er å foretrekke når vi skal arbeide med polynomer på en datamaskin. Men hvor anvendelige er egentlige polynomer som approksimasjonsredskap? For å lage en tilnærming som skal være nøyaktig i nærheten av ett punkt fungerer Taylor-polynomer svært godt, men hva om vi skal tilnærme en komplisert funksjon over et langt intervall? Vi så jo faktisk i seksjon 9.2.1 at det ikke nødvendigvis er så lurt å interpolere i mange punkter med et polynom av høy grad. Og selv om polynomer av lav grad egner seg godt for datamaskinberegninger, øker både kompleksiteten og avrundingsfeilen raskt når graden øker, selv om vi bruker Bernstein-basisen.

For å kunne approksimere kompliserte funksjoner trenger vi mange frie parametre som kan justeres slik at tilnærmingen blir god. For polynomer er de frie parametrene koeffisientene, og polynomer med mange koeffisienter må derfor nødvendigvis ha høy grad, noe som altså ikke er så attraktivt fra et beregningsteknisk synspunkt. Heldigvis fins det en annen mulighet. Det er ikke bare ved å øke graden vi kan få tilgang på mange koeffisienter, vi kan også få det til med lav grad om vi bruker mange polynomer. Med en slik framgangsmåte kan vi dele opp det totale definisjonsintervallet til funksjonen vi skal approksimere i mindre delintervaller og så bruke et polynom på hvert delinterval. Det eneste vi må passe på er at vi hekter polynombitene pent sammen i skjøtene.

Figur 9.9 viser to eksempler på funksjoner som begge er sammensatt av to Bernstein-polynomer. I begge tilfellene er polynomene lenket sammen i $x = 1$. Siden Bernstein-polynomer interpolerer første og siste koeffisient er det lett å sikre at to sammenlenkede polynomer er kontinuerlige i skjøten. Dette vil være tilfelle om den siste koeffisienten i polynomet til venstre er lik den første koeffisienten i polynomet til høyre, slik som i figur 9.9.

Når det sammenlenkede polynomet bare er kontinuerlig i en skjøt vil det som regel ha en iøynefallende knekk, slik som i figuren. For å få en glattere overgang trenger vi at også den deriverte er kontinuerlig i skjøten. Vi vet at tangenten i det venstre endepunktet til et Bernstein-polynom faller sammen med det første linjesegmentet i kontrollpolygonet mens tangenten i det høyre endepunktet faller sammen med kontrollpolygonets siste linjesegment. Fra dette ser vi at både funksjonsverdi og derivert blir kontinuerlig i skjøten om det siste linjesegmentet i det første kontrollpolyognet ligger på samme rette linje som



Figur 9.10. En funksjon med kontinuerlig derivert satt sammen av 4 Bernstein-polynomer.

det første linjesegmentet i det andre kontrollpolyognet, slik som i figur 9.9 (b).

La oss oppsummere dette i en setning.

Setning 9.7. *La $p(x) = \sum_{i=0}^{n} c_i b_i(x)$ være et Bernstein-polynom av grad n på intervallet $[a, b]$, la $q(x) = \sum_{i=0}^{n} d_i b_i(x)$ være et Bernstein-polynom av samme grad på intervallet $[b, c]$ og la f betegne den sammensatt funksjonen gitt ved*

$$f(x) = \begin{cases} p(x), & \text{for } x \in [a, b); \\ q(x), & \text{for } x \in [b, c]. \end{cases}$$

Da er f kontinuerlig i $x = b$ hvis og bare hvis $c_n = d_0$, mens f er kontinuerlig med kontiuerlig derivert i $x = b$ hvis og bare hvis $c_n = d_0$ og de tre kontrollpunktene $(b - h_1, c_{n-1})$, (b, c_n) og $(b + h_2, d_1)$ ligger på en rett linje. Her er $h_1 = (b - a)/n$ og $h_2 = (c - b)/n$.

Ved å lenke sammen Bernstein-polynomer får vi en svært fleksibel klasse av funksjoner med gode approksimasjonsegenskaper. Et enkelt eksempel er vist i figur 9.10. For å beregne tilnærmingene kan vi bruke interpolasjon eller minimering av et passende feilmål, akkurat som for polynomer.

I en del sammenhenger kan det være ønskelig at både den andrederiverte og enda høyere deriverte skal være kontinuerlige over skjøtene mellom polynombitene. Dette kan vi få til ved å legge passende betingelser på koeffisientene, slik som i setning 9.7, men etterhvert blir disse betingelsene kompliserte å holde styr på. Det viser seg at det er mulig å konstruere enkle basisfunksjoner bestående av ulike polynomsegmenter som har denne kontinuiteten innebygd. Ved å multiplisere disse basisfunksjonene med koffisienter og så addere sammen kan vi danne generelle stykkevise polynomer. Bygger vi stykkevise polynomer på denne måten kalles de ofte *splinefunksjoner*[5].

[5]Ordet spline brukes på engelsk om en bøyelig linjal som kan tvinges til å gå gjennom noen faste punkter.

9.3.5 Beregning av verdier på Bernstein-polynomer

Vi har så langt ikke nevnt noen metode for å beregne verdien av et Bernstein-polynom. En mulighet er selvsagt å beregne verdien av hvert basispolynom, multiplisere med koeffisientene og så summere. Det fins flere andre (og bedre) metoder. Den mest brukte er basert på å beregne en sekvens av veide gjennomsnitt mellom to og to tall og er svært motstandsdyktig overfor avrundingsfeil. En enklere metode er å omskrive Bernstein-polynomet til en slags potensform, noe som i det kubiske tilfellet kan gjøres ved

$$\begin{aligned} p(x) &= c_0(1-x)^3 + 3c_1x(1-x)^2 + 3c_2x^2(1-x) + c_3x^3 \\ &= (1-x)^3\left(c_0 + 3c_1\frac{x}{1-x} + 3c_2\Big(\frac{x}{1-x}\Big)^2 + c_3\Big(\frac{x}{1-x}\Big)^3\right) \\ &= (1-x)^3(c_0 + 3c_1v + 3c_2v^2 + v^3, \end{aligned}$$

der vi har satt $v = x/(1-x)$ og antar at $x \neq 1$. I stedenfor å sette $(1-x)^3$ utenfor som faktor kan vi sette x^3 utenfor og få et polynom på potensform i $w = (1-x)/x$,

$$p(x) = x^3(c_0w^3 + 3c_1w^2 + 3c_2w + c_3).$$

Potensbasisen gir større avrundingsfeil jo lenger bort fra $x = 0$ vi kommer. Fra et beregningssynspunkt er det derfor best å bruke det første alternativet når $x \in [0, 1/2]$ og det andre når $x \in [1/2, 1]$. Ordner vi oss på denne måten vil vi alltid ha $v \leq 1$ eller $w \leq 1$, og på intervallet $[0, 1]$ oppfører potensbasisen seg rimelig pent med tanke på avrundingsfeil.

For generell grad får vi at $p(x)$ kan skrives på de to ekvivalente måtene

$$\begin{aligned} p(x) &= (1-x)^n \sum_{i=0}^{n} \binom{n}{i} c_i v^i \\ &= x^n \sum_{i=0}^{n} \binom{n}{i} c_{n-i} w^i \end{aligned} \tag{9.11}$$

der $v = x/(1-x)$ og $w = (1-x)/x$ som før (den første formelen gjelder ikke når $x = 1$ og den siste ikke når $x = 0$). På denne måten har vi dermed redusert det å beregne en funksjonsverdi for et Bernstein-polynom til det å beregne en verdi for et polynom skrevet på potensform.

Hvis vi arbeider på et generelt intervall $[a, b]$ husker vi fra egenskap 6 i lemma 9.3 at det er lurt å sette $u = (x-a)/(b-a)$. Da er også $1-u = (b-x)/(b-a)$ slik at

$$b_{i,n}(x) = \binom{n}{i}\Big(\frac{x-a}{b-a}\Big)^i\Big(\frac{b-x}{b-a}\Big)^{n-i} = \binom{n}{i}u^i(1-u)^{n-i}.$$

Dermed kan vi beregne verdier på Bernstein-polynomer på et vilkårlig intervall $[a, b]$ dersom vi kan gjøre slike beregninger på intervallet $[0, 1]$.

Vi er nå nesten klar til å gi en algoritme for å beregne $p(x)$, det eneste som mangler er beregning av binomialkoeffisientene. Ved å sette inn definisjonen av $\binom{n}{i-1}$ er det ikke så vanskelig å se at relasjonen

$$\binom{n}{i} = \frac{n-i+1}{i}\binom{n}{i-1}$$

holder for $i = 1, \ldots, n$. Når vi dessuten vet at $\binom{n}{0} = 1$ gir dette oss en grei algoritme for å beregne binomialkoeffisientene. Dermed har vi alle ingrediensene for å beregne verdien av et Bernstein-polynom.

Algoritme 9.8. *La Bernstein-polynomet $p(x) = \sum_{i=0}^{n} c_i b_{i,n}(x)$ på intervallet $[a, b]$ være gitt. Etter at koden under er utført vil variabelen r inneholde verdien $p(x)$.*

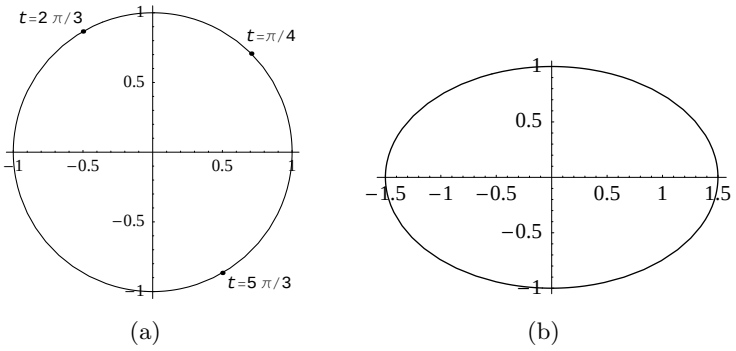
```
binom = 1; d[0] = c[0];
for (i=1; i<n+1; i++)
   {
      binom = binom*(n-i+1)/i;
      d[i] = c[i]*binom;
   }

u = (x-a)/(b-a);

if (u < 0.5)
   {
       v = u/(1-u);
       r = d[n];
       for (i=n-1; i>=0; i--)
           {
              r = d[i] + v*r;
           }
       r = Math.pow(1-u,n)*r;
   }
else
   {
       w = (1-u)/u;
       r = d[0];
       for (i=1; i<=n; i++)
           {
               r = d[i] + w*r;
           }
       r = Math.pow(u,n)*r;
   }
```

I den første delen av algoritmen multipliserer vi koeffisientene med binomialkoeffisienter og lagrer resultatet i en ny array `d`. Matematisk svarer dette til å utføre $d_i = \binom{n}{i} c_i$ for $i = 0, \ldots, n$. Legg merke til at dette kan gjøres på samme måte uansett hvilken av de to formlene i (9.11) vi bruker siden en linje i Pascals trekant er symmetrisk om midten (mer presist så har vi $\binom{n}{i} = \binom{n}{n-i}$).

Med tilordningen $u = (b - x)/(b - a)$ transformerer vi oss fra intervallet $[a, b]$ til intervallet $[0, 1]$. Den nye variabelen er dermed u, og avhengig av hvilken halvdel av $[0, 1]$ denne ligger i velger vi den varianten i (9.11) som gjør at den endelige variabelen (v eller



Figur 9.11. De to parametriske kurvene $\boldsymbol{r}(t) = (\cos t, \sin t)$ (i (a)) og $\boldsymbol{r}(t) = (2\cos t, \sin t)$ (i (b).

w) ligger i intervallet $[0, 1]$. Hvis $u < 0.5$ setter vi derfor $v = u/(1-u)$ og beregner $p(x)$ som $p(x) = (1-u)^n \sum_{i=0}^{n} d_i u^i$, hvis $u \geq 0.5$ bruker vi $p(x) = u^n \sum_{i=0}^{n} d_{n-i} w^n$.

9.4 Parametriske kurver

En grunnleggende egenskap ved funksjonsbegrepet er at til hver verdi av argumentet så svarer det nøyaktig en funksjonsverdi. Geometrisk betyr dette at en vertikal linje ikke kan skjære grafen til funksjonen mer enn en gang. Ved hjelp av parametriske kurver kan vi representere mer generelle, en-dimensjonale, geometriske former.

9.4.1 Definisjon av parametriske kurver

Fra skolematematikken vet vi at ligningen for en sirkel med radius 1 er gitt ved $x^2+y^2 = 1$. Hvis vi løser denne med hensyn på y er vi vant til å skrive $y = \pm\sqrt{1-x^2}$. Dette siste uttrykket gir ikke y som én funksjon av x, men som to funksjoner av x, en verdi på den øvre halvsirkelen og en på den nedre. Det er altså umulig å representere hele sirkelen ved hjelp av en funksjon.

Ved hjelp av *parametriske representasjoner* kommer vi unna denne begrensningen. En parametrisk representasjon av sirkelen er gitt ved uttrykket

$$\boldsymbol{r}(t) = (\cos t, \sin t), \qquad t \in [0, 2\pi].$$

Vi ser at som funksjoner av en variabel så trenger $\boldsymbol{r}$ et argument som er et reelt tall, men forskjellen er at $\boldsymbol{r}$ for hver slik t i definisjonsområdet $[0, 2\pi]$ produserer *to* reelle tall. Disse kan vi tenke på som et punkt i planet (eller ekvivalent, som en vektor i planet), og vi angir dette ved å skrive $\boldsymbol{r} : [0, 2\pi] \mapsto \mathbb{R}^2$.

Legg merke til at om vi setter $x = \cos t$ og $y = \sin t$ så kommer vi tilbake til den vanlige sirkelligningen via en velkjent identitet for trigonometriske funksjoner,

$$x^2 + y^2 = \cos^2 t + \sin^2 t = 1.$$

Et plott av $\boldsymbol{r}$ er vist i figur 9.11 (a) med verdien av t angitt for noen typiske punkter, og figur 9.11 (b) viser den relaterte parametriske representasjonen $\boldsymbol{r}(t) = (2\cos t, \sin t)$.

Hvis vi nå setter $x = 2\cos t$ og $y = \sin t$ så ser vi at

$$\frac{x^2}{4} + y^2 = \cos^2 t + \sin t = 1$$

som vi kjenner igjen som en typisk ellipseligning.

Noen flere eksempler på parametriske representasjoner er gitt i figur 9.12. Legg spesielt merke til at den parametriske representasjonen $\boldsymbol{r}(t) = \boldsymbol{a} + t(\boldsymbol{b} - \boldsymbol{a})$ gir den rette linja gjennom de to punktene i planet gitt ved $\boldsymbol{a}$ og $\boldsymbol{b}$. Med funksjoner har vi problemer med å representere vertikale linjer, men på parametrisk form skiller slike linjer seg ikke ut. Hvis $\boldsymbol{a}$ og $\boldsymbol{b}$ har samme x-koordinat får vi en vertikal linje uten problemer, se (b). I (c) ser vi at funksjoner kan representeres som parametriske kurver. I (d) har vi rotert kurven i (c) slik at den havner langs y-aksen i steden, noe som selvsagt er umulig med funksjoner. Funksjonene i (e)–(h) har det til felles at de alle er på formen $\boldsymbol{r}(t) = h(t)(\cos t, \sin t)$ for ulike valg av funksjonen h. Siden $(\cos t, \sin t)$ gir enhetssirkelen så ser vi at alle kurvene oppstår ved å trykke sammen eller trekke ut sirkelen på ulike måter.

Ut fra disse innledende eksemplene kan vi sette opp en generell definisjon av parametriske representasjoner.

Definisjon 9.9. *En plan, parametrisk representasjon eller parametrisk kurve definert på intervallet $[a, b]$, er et uttrykk på formen $\boldsymbol{r}(t) = \big(f(t), g(t)\big)$ der f og g er to funksjoner definert på intervallet $[a, b]$.*

En viktig observasjon er at ulike parametriske representasjoner kan gi opphav til samme geometriske bilde. For eksempel kan vi også få fram sirkelen ved representasjonen $\boldsymbol{r}(t) = (\cos 2\pi t, \sin 2\pi t)$, men nå holder det om t varierer i intervallet $[0, 1]$. En litt mer overaskende sirkelrepresentasjon er gitt ved

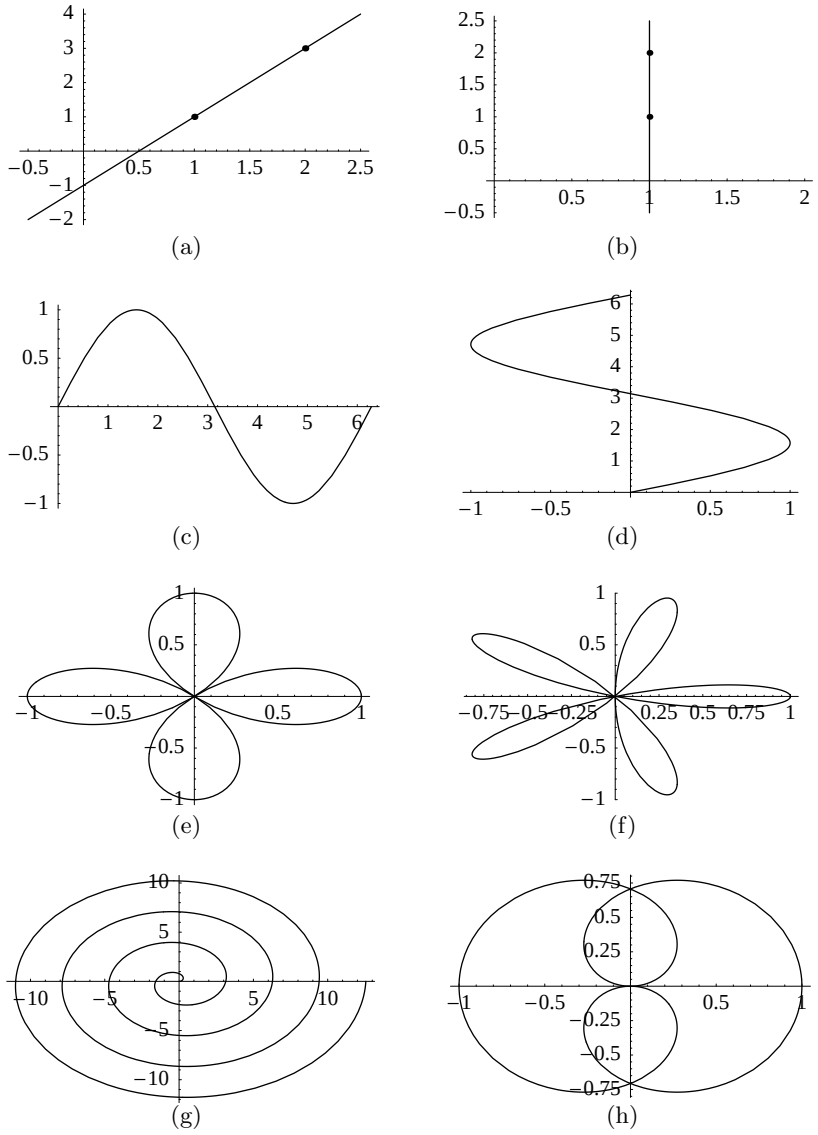
$$\boldsymbol{r}(t) = \left(\frac{t}{\sqrt{1+t^2}}, \frac{1}{\sqrt{1+t^2}}\right), \qquad t \in \mathbb{R},$$

men merk at denne bare gir halve sirkelen.

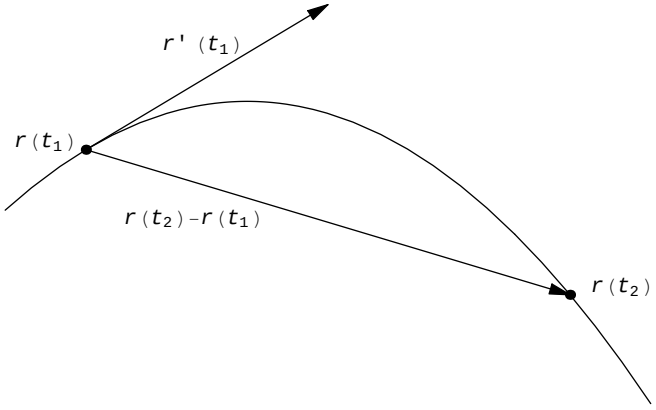
Generelt er det alltid mange parametriske representasjoner som gir det samme, geometriske bildet, og i mer formelle framstillinger er en parametrisk kurve definert som samlingen av alle parametriske representasjoner som gir opphav til det samme geometriske bildet. Vi skal ikke være så formelle og vil bruke begrepene om hverandre.

Vi skal holde oss til parametriske kurver i planet, men de kan enkelt generaliseres til høyere dimensjoner ved å legge til flere komponenter. For eksempel representerer den parametriske representasjonen $\boldsymbol{r}(t) = (\cos t, \sin t, t)$ en spiral i rommet som hever seg over xy-planet. Dette ser vi siden de to første koordinatene gir en sirkel i xy-planet, mens z-komponenten vokser med t slik at vi beveger oss oppover i rommet når t øker. Ved å legge på flere komponenter får vi parametriske representasjoner som kan representere kurver med så mange romdimensjoner som vi måtte trenge.

Parametriske kurver har en fysisk tolkning som ofte kan være nyttig. Representasjonen $\boldsymbol{r}$ definert på intervallet $[a, b]$ gir et geometrisk bilde som vi kan tenke på som en måte å kjøre bil fra $\boldsymbol{r}(a)$ til $\boldsymbol{r}(b)$ langs veien definert av $\boldsymbol{r}$. Uttrykket $\boldsymbol{r}(t)$ gir dermed posisjonen langs veien ved tidspunkt t. Ulike parametriske representasjoner svarer da til



Figur 9.12. Åtte parametriske kurver: (a) $\boldsymbol{r}(t) = \boldsymbol{a} + t(\boldsymbol{b} - \boldsymbol{a})$ med $\boldsymbol{a} = (1,1)$ og $\boldsymbol{b} = (2,3)$; (b) som (a), men med $\boldsymbol{a} = (1,1)$ og $\boldsymbol{b} = (1,2)$; (c) $\boldsymbol{r}(t) = (t, \sin t)$; (d) $\boldsymbol{r}(t) = (\sin t, t)$, (e) $\boldsymbol{r}(t) = \cos(2t)(\cos t, \sin t)$; (f) $\boldsymbol{r}(t) = \cos(5t)(\cos t, \sin t)$; (g) $\boldsymbol{r}(t) = t(\cos t, \sin t)/2$, (h) $\boldsymbol{r}(t) = \sin(t/2)(\cos t, \sin t)$.



Figur 9.13. Grenseovergang fra sekant til tangent.

ulike kjøremønstre langs veien. Vi kan for eksempel kjøre fort eller bruke lang tid, vi kan til og med kjøre et stykke, rygge litt og så kjøre framover igjen, eller vi kan kjøre med konstant fart.

9.4.2 Tangent, hastighet og derivert

Ved de to tidspunktene t_1 og t_2 har vi de to posisjonene $\boldsymbol{r}(t_1)$ og $\boldsymbol{r}(t_2)$ på kurven vår. Differansen $\boldsymbol{r}(t_2) - \boldsymbol{r}(t_1)$ gir dermed vektoren fra $\boldsymbol{r}(t_1)$ til $\boldsymbol{r}(t_2)$, altså sekanten mellom de to punktene, og lar vi t_2 nærme seg t_1 vil sekanten nærme seg tangenten til kurven i $\boldsymbol{r}(t)$. Det viser seg at det er lurt å skalere lengden med forskjellen i tid $t_2 - t_1$. Gjør vi dette får vi

$$\lim_{t_2 \to t_1} \frac{\boldsymbol{r}(t_2) - \boldsymbol{r}(t_1)}{t_2 - t_1} = \left(\lim_{t_2 \to t_1} \frac{f(t_2) - f(t_1)}{t_2 - t_1}, \lim_{t_2 \to t_1} \frac{g(t_2) - g(t_1)}{t_2 - t_1} \right) = \big(f'(t_1), g'(t_1) \big), \quad (9.12)$$

Dette er utgangspunktet for vår neste setning og er illustrert i figur 9.13.

Setning 9.10. *Anta at f og g er deriverbare funksjoner. Den deriverte av den parametriske kurven $\boldsymbol{r}(t) = \big(f(t), g(t)\big)$ er definert som $\boldsymbol{r}'(t) = \big(f'(t), g'(t)\big)$. Retningen til den deriverte faller sammen med tangentlinja til kurven i $\boldsymbol{r}(t)$ mens lengden av tangenten gir farten til bevegelsen ved dette tidspunktet.*

Bevis. Utsagnet om retningen til den deriverte følger fra kommentarene foran og relasjonene i (9.12), mens utsagnet om farten trenger litt forklaring. Fra (9.12) har vi

$$\big|\boldsymbol{r}'(t_1)\big| = \lim_{t_2 \to t_1} \frac{\big|\boldsymbol{r}(t_2) - \boldsymbol{r}(t_1)\big|}{t_2 - t_1}, \quad (9.13)$$

der notasjonen $|\boldsymbol{a}|$ generelt angir lengden til vektoren $\boldsymbol{a}$. Uttrykket $\big|\boldsymbol{r}(t_2) - \boldsymbol{r}(t_1)\big|$ er avstanden mellom de to punktene $\boldsymbol{r}(t_1)$ og $\boldsymbol{r}(t_2)$, målt i "luftlinje". Når t_2 nærmer seg t_1 vil avstanden etterhvert nærme seg avstanden langs veien gitt ved kurven. Dividerer vi med tiden $t_2 - t_1$ som det tar å bevege seg fra $\boldsymbol{r}(t_1)$ til $\boldsymbol{r}(t_2)$ får vi gjennomsnittsfarten

over dette tidsintervallet. Når så lengden av tidsintervallet går mot 0 som i (9.13) får vi farten ved tidspunktet t_1 som lengden av tangenten. ∎

Det er vanlig å referere til tangenten $\boldsymbol{r}'(t)$ som *hastigheten* til bevegelsen gitt ved parametriseringen $\boldsymbol{r}(t)$ mens lengden av tangenten er farten, det vi måler på speedometeret på en bil. Hastigheten gir altså mer informasjon enn farten og viser også hvilken retning vi beveger oss i. Informasjonen vi nå har om tangentretninger er vesentlig når vi skal hekte sammen kurvebiter til større kurver i seksjon 9.5.

9.4.3 Anvendelser av parametriske kurver

Parametriske kurver har mange mange anvendelser. De klassiske anvendelsene er innen fysikk, som ett skreddersydd verktøy for å angi partikkelbaner. Med datateknologien har mange nye anvendelser kommet til. Innen datagrafikk er det alltid behov for å kunne representere geometrien som skal vises fram på skjermen, og kurver representeres da som regel på parametrisk form som her. Etterhvert har det blitt vanlig også å lage film ved hjelp av datamaskin, der handlingen foregår i en virtuell verden som bare eksisterer inne i datamaskiner. Slike filmer filmes ikke på vanlig måte. I steden produseres hvert bilde digitalt og lagres på disk, basert på hvor det virtuelle kameraet er plassert i den virtuelle verdenen. I en slik sammenheng er det viktig å føre kameraet riktig, og kamerabaner representeres naturlig som parametriske kurver. Legg merke til at i en slik anvendelse er det ikke bare den geometrisk kamerabanen som er viktig, det er også av avgjørende betydning hvor fort kameraet beveger seg, med andre ord er det viktig hvilken parametrisk representasjon som er valgt blant de mange som alle representerer den samme geometriske banen.

En annen sentral anvendelse av parametriske kurver er innen font-teknologi (en font er en bestemt måte å tegne de ulike tegnene som er tilgjengelig ved trykking, i våre dager de tegnene som datamaskinen kan tegne). Omrisset av hver bokstav representeres ved en eller flere parametriske kurver, og når bokstaven skal tegnes på skjerm eller papir bestemmer maskinen hvilke punkter som skal skrus av eller på for å vise bokstaven på en pen måte. Vi skal se litt nærmere på dette i seksjonen om Bezier-kurver.

9.5 Bezier-kurver

Eksemplene i seksjon 9.4 illustrerte forhåpentligvis at vi kan representere et stort spekter av geometriske kurver ved hjelp av parametriske representasjoner. Ved å lagre formelen i en datamaskin med passende programvare kan vi alltid tegne den geometriske formen så pent som utskriftsmediet tillater. Spørsmålet er bare hvordan vi kan finne fram til en parametrisk representasjon som gir akkurat den formen vi er ute etter. Vi kan selvsagt prøve oss fram med formler av typen i figur 9.12, men det blir som regel fomling mer eller mindre i blinde.

Det vi trenger er en samling av enkle, men fleksible, parametriske representasjoner som inneholder frie koeffisienter som kan justeres slik at vi kan få fram ulike geometriske former. Vi kan da bruke teknikker som interpolasjon og minimering av feil til å beregne gode tilnærminger til gitte geometriske former. Hvis de frie koeffisientene har intuitive

geometriske tolkninger, kan vi også gjøre bruk av disse basisrepresentasjonene til interaktivt å bygge opp en ønsket kurve ved å manipulere koeffisienter. Det viser seg at en generalisering av Bernstein-polynomene har de ønskede egenskapene.

Definisjon 9.11. *En Bezier-kurve av grad n i planet er en parametrisk kurve på formen*

$$\boldsymbol{p}(t) = \boldsymbol{c}_0 b_{0,n}(t) + \boldsymbol{c}_1 b_{1,n}(t) + \cdots + \boldsymbol{c}_n b_{n,n}(t), \qquad t \in [a,b],$$

der koeffisientene $(\boldsymbol{c}_i)_{i=0}^n$ er punkter i planet og $\{b_{i,n}\}_{i=0}^n$ er Bernstein-basisen på intervallet $[a,b]$ gitt ved

$$b_{i,n}(t) = \binom{n}{i}\left(\frac{t-a}{b-a}\right)^i\left(\frac{b-t}{b-a}\right)^{n-i}.$$

Koeffisientene kalles også kontrollpunktene til $\boldsymbol{p}$ og den brudne linja som framkommer ved å forbinde kontrollpunktene med rette linjer kalles kontrolpolygonet til $\boldsymbol{p}$.

Legg merke til at polynomene som inngår er 'vanlige' funksjoner som til hver t vi putter inn gir ut et reelt tall. Grunnen til at dette blir kurver og ikke funksjoner er at koeffisientene vi multipliserer med ikke lenger er tall, men punkter i planet. Uttrykket $\boldsymbol{c}_i b_{i,n}(t)$ betyr vanlig skalar multiplikasjon av en vektor med et tall; hvis $\boldsymbol{c}_i = (x,y)$ er $\boldsymbol{c}_i b_{i,n}(t) = \big(x\, b_{i,n}(t), y\, b_{i,n}(t)\big)$. Dette betyr at det er lett å regne ut verdier på en Bezier-kurve. Hvis vi har en kvadratisk Bezier-kurve med kontrollpunkter $\boldsymbol{c}_i = (x_i, y_i)$ for $i = 0$, 1, 2 så har vi

$$\begin{aligned}\boldsymbol{p}(t) &= \boldsymbol{c}_0 b_0(t) + \boldsymbol{c}_1 b_1(t) + \boldsymbol{c}_2 b_2(t)\\ &= \big(x_0 b_0(t) + x_1 b_1(t) + x_2 b_2(t), y_0 b_0(t) + y_1 b_1(t) + y_2 b_2(t)\big).\end{aligned}$$

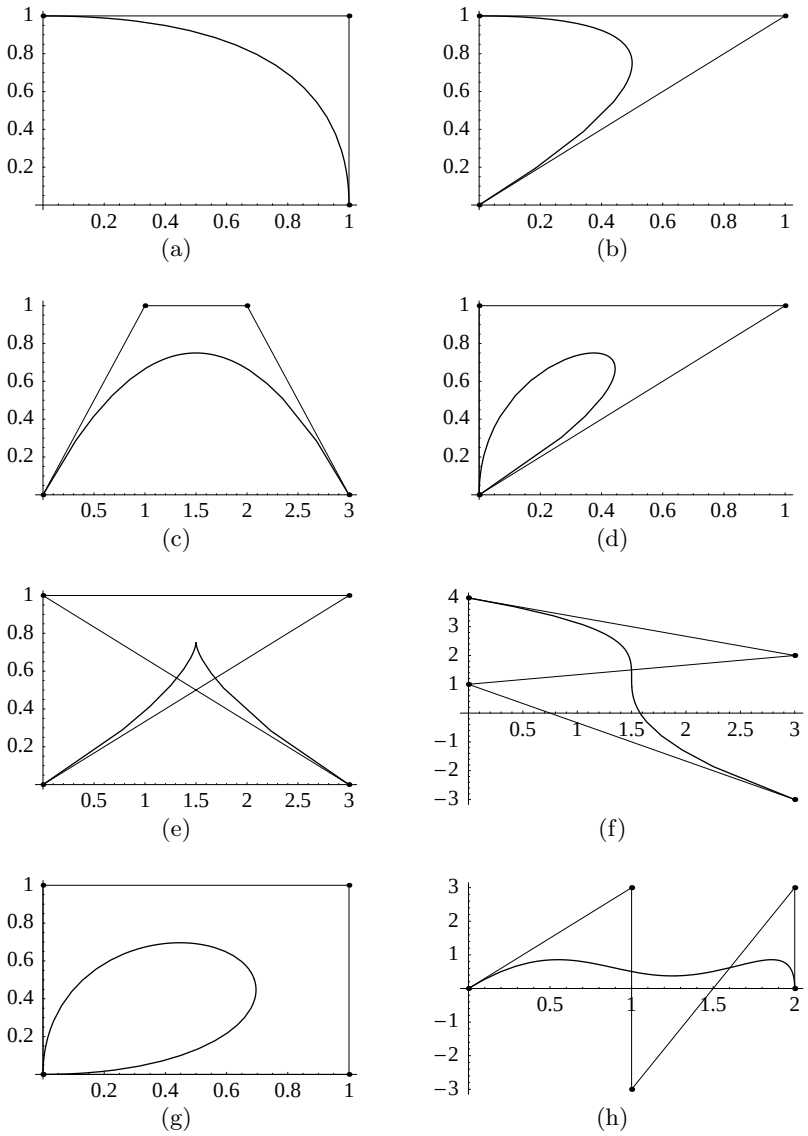
En slik oppspalting kan vi gjøre for generell grad, så det å beregne et punkt på en Bezier-kurve i planet er derfor ikke noe annet enn å beregne verdien av to Bernstein-polynomer, ett for x-koordinaten og ett for y-koordinaten.

Vi ser at definisjonen av kontrollpunkter er enklere for kurver enn for funksjoner. For funksjoner måtte vi finne en x-verdi å assosiere hver koeffisient med, det er ikke lenger nødvendig siden koeffisientene er punkter i planet.

Bezier-kurven av grad 1 gitt ved $\boldsymbol{p}(t) = \boldsymbol{c}_0 b_{0,1}(t) + \boldsymbol{c}_1 b_{1,1}(t) = (1-t)\boldsymbol{c}_0 + t\boldsymbol{c}_1$ gir linjesegmentet som starter i $\boldsymbol{c}_0$ og ender i $\boldsymbol{c}_1$. En del eksempler på Bezier-kurver av høyere grad er vist i figur 9.14, og ikke overaskende ser vi at disse også interpolerer sitt første og siste kontrollpunkt. Vi ser i tillegg at tangentretningene i endene er gitt ved retningen til kontrollpolygonet i endene. Dette følger av de grunnleggende egenskapene ved Bernstein-basisen i lemma 9.3.

Setning 9.12. *En Bezier-kurve $\boldsymbol{p}(t) = \sum_{i=0}^n \boldsymbol{c}_i b_i(t)$ har verdiene $\boldsymbol{p}(a) = \boldsymbol{c}_0$ og $\boldsymbol{p}(b) = \boldsymbol{c}_n$ i endepunktene. Tangenten til kurven i $t = a$ er parallell med linja gjennom $\boldsymbol{c}_0$ og $\boldsymbol{c}_1$, mens tangenten i $t = b$ er parallell med linja gjennom $\boldsymbol{c}_{n-1}$ og $\boldsymbol{c}_n$.*

Plottene i figur 9.14 viser at Bezier-kurvene oppfører seg helt tilsvarende Bernstein-polynomene, men vi har i tillegg fått den ekstra friheten som kurver gir i forhold til funksjoner. Samtidig er det slik at for å få mange frihetsgrader å spille på som vi kan utnytte til å få mer kompliserte geometriske former, trenger vi Bezier-kurver av høy grad. Som for funksjoner er dette ikke så lurt med tanke på effektivitet og avrundingsfeil, og vi



Figur 9.14. Åtte Bezier-kurver med kontrollpunkter. Kvadratiske ((a)og (b)), kubiske ((c)–(f)), grad 4 ((g) og (h)).

ser fra figurene at sammenhengen mellom kontrollpolygonet og kurven blir mindre tydelig når graden øker.

En god løsning å danne mer kompliserte former ved å hekte sammen flere Bezier-kurver, helt parallelt med konstruksjonen vi gjorde for Bernstein-polynomer.

Definisjon 9.13. *En sammensatt Bezier-kurve av grad n på intervallet $[0, m]$ er en parametrisk kurve på formen*

$$\boldsymbol{p}(t) = \begin{cases} \boldsymbol{p}_1(t), & \text{for } t \in [0,1); \\ \boldsymbol{p}_2(t), & \text{for } t \in [1,2); \\ \vdots & \\ \boldsymbol{p}_m(t), & \text{for } t \in [m-1, m]; \end{cases}$$

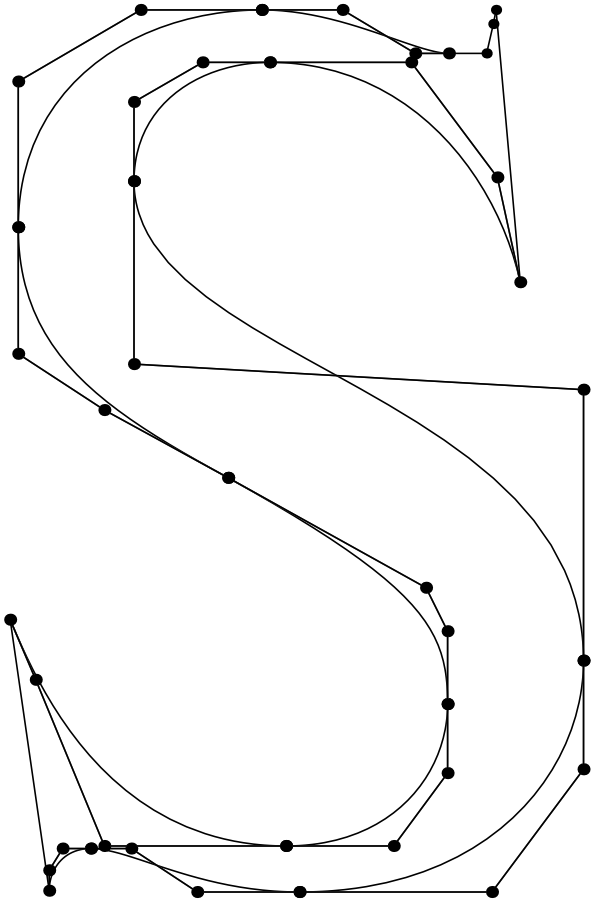
der $\boldsymbol{p}_i$ er en Bezier-kurve på intervallet $[i-1, i]$ for $i = 1, 2, \ldots, m$. En sammensatt Bezier-kurve kalles ofte bare en Bezier-kurve.

Her har vi antatt at alle delkurvene er definert på intervaller av bredde 1. Det er ingenting å tape på dette hvis vi bare er interessert i den geometriske formen som kurven representerer. Hvis vi for eksempel skal representere kamerabaner må vi være mer forsiktige siden bredden på intervallene sier noe om hvor lang tid vi bruker på å gjennomløpe hver kurvebit og dermed noe om hastigheten vi beveger kameraet med.

På grunn av den enkle sammenhengen mellom kontrollpolyogonet og kurven i endene er det enkelt å hekte sammen to Bezier-kurver slik at den sammensatte kurven blir både kontinuerlig og har kontinuerlig tangentretning.

Setning 9.14. *En sammensatt Bezier-kurve bestående av de to segmentene $\boldsymbol{p}(t) = \sum_{i=0}^{n} \boldsymbol{c}_i b_i(t)$ og $\boldsymbol{q}(t) = \sum_{i=0}^{n} \boldsymbol{d}_i b_i(t)$ definert på henholdsvis $[0, 1]$ og $[1, 2]$, vil være kontinuerlig i $t = 1$ hvis og bare hvis $\boldsymbol{c}_n = \boldsymbol{d}_0$. Hvis den er kontinuerlig i $t = 1$ vil tangentretningen være kontinuerlig i $t = 1$ hvis og bare hvis de tre punktene $\boldsymbol{c}_{n-1}$, $\boldsymbol{c}_n = \boldsymbol{d}_0$ og $\boldsymbol{d}_1$ ligger på en rett linje.*

Den matematiske bakgrunnen vi nå har om Bezier-kurver bør være nok til å gi en grov ide om hvordan denne kurvetypen kan utnyttes til å representere geometriske objekter i planet. Det kommersielle programmet *Adobe Illustrator* som ble lansert i 1988 var det første programmet som gjorde Bezier-kurver tilgjengelig for 'massene'. Slike kurver hadde blitt brukt i bil- og flyindustri siden slutten av 1950-tallet til å representere geometrisk form, men da var brukerne ingeniører. Selskapet Adobe sin forretningside var å lage et dataspråk som kunne beskrive alt innhold på en papirside, inklusive tegninger og bokstaver, ved hjelp av matematikk, og til det utviklet de språket *Postscript*. Det grunnleggende primitivet i Postscript er kubiske Bezier-kurver, og Postscript-fonter er bygget opp av slike kurver. Ett eksempel er vist i figur 9.15. Ideen i Adobe Illustrator er å utnytte kubiske Bezier-kurver til frihåndstegning på datamaskin, og programmet er i dag et av de viktigste verktøyene for grafiske designere. Mange skrivere (blant annet på Universitet i Oslo) er basert på Postscript. Dette betyr at hver gang du skriver ut et dokument på en slik skriver må den regne ut en Bezier-kurve for hver eneste bokstav i dokumentet ditt!



Figur 9.15. Bezier-kurve med kontrollpolygon som representerer bokstaven 'S' i Postscript-fonten Times-Roman. De ulike Bezier-segmentene ser du mellom kontrollpunktene som ligger på konturen til bokstaven. Legg merke til at den totale kurven også består av noen rette linjestykker.

Dataselskapet Apple utviklet tidlig på 1990-tallet en teknologi for å bryte Adobes monopol på font-området. Denne teknologien kalles *TrueType* og er basert på kvadratiske Bezier-kurver i steden for kubiske. TrueType er i dag svært utbredt og brukes også av Microsoft.

Oppgaver

9.1 Taylor-polynomene til e^x, $\cos x$ og $\sin x$ er

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} + \frac{x^7}{5040} + \cdots$$
$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \cdots$$
$$\sin x = x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \cdots$$

Regn ut Taylor-polynomet til e^{ix} og bruk dette til å forklare at Eulers formel $e^{ix} = \cos x + i \sin x$ er en rimelig definisjon av e^{ix}.

9.2 Utled egenskapene 1 og 2 ved Bernsteinbasisen gitt i lemma 9.3. Et nyttig redskap for å bevise egenskap 2 er binomialteoremet.

9.3 Utled egenskap 3 i lemma 9.3.

9.4 Utled egenskap 4 og 5 i lemma 9.3.

9.5 Utled egenskapene 1–3 i lemma 9.5 for Bernstein-polynomer av generell grad n på intervallet $[0, 1]$.

9.6 Sjekk at relasjonen

$$\binom{n}{i} = \frac{n-i+1}{i}\binom{n}{i-1}$$

stemmer.

9.7 a) Vis at polynomet x kan uttrykkes i Bernsteinbasisen som

$$x = \sum_{i=0}^{3} \frac{i}{3} b_{i,3}(x).$$

Hint: Sjekk at de deriverte på de to sidene av likhetstegnet er like og at de to sidene har samme verdi i ett punkt.

b) Bruk (a) til å vise relasjonen

$$\big(x, p(x)\big) = \sum_{i=0}^{3} (i/3, c_i) b_{i,3}(x).$$

(Dette må tolkes som en vektorrelasjon der venstresiden $\big(x, p(x)\big)$ og kontrollpunktene $(i/3, c_i)_{i=0}^3$ er punkter i planet, mens $b_{i,3}(x)$ for hver x er en skalar.) Denne relasjonen viser at grafen til $p(x)$ er et veiet gjennomsnitt av sine kontrollpunkter.

9.8 Programmer algoritme 9.8 og test programmet ved å plotte ut noen av Bernsteinpolynomene i figur 9.7 og 9.8.

KAPITTEL 10

Flerskala-analyse og kompresjon av lyd

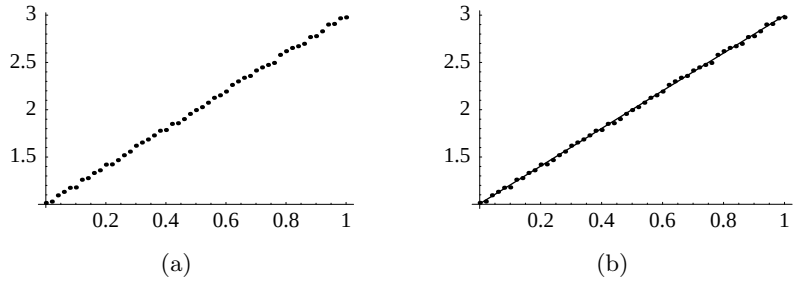
Vi kan lagre dokumenter av mange forskjellige typer på en datamaskin. Vi kan for eksempel ha en datafil der innholdet er tall, vi kan ha en tekstfil, en lydfil, en fil med bilder, en med video, og eventuelt kombinasjoner av disse. Et grunnleggende problem er at en del av disse dokumenttypene lett blir svært store og tunge å håndtere og krever mye lagerplass på datamaskinens harddisk. Dette gjelder særlig lyd-, bilde- og videofiler. For delvis å omgå dette problemet har det blitt utviklet teknikker for å komprimere informasjonen som er lagret i en fil. Slike metoder gjør utstrakt bruk av matematikk og i dette kapitlet skal vi ta for oss noen grunnleggende ideer som ligger bak kompresjonsteknikker og utvikle en rudimentær kompresjonsstrategi.

10.1 Litt generelt om kompresjon

Det kan kanskje høres underlig ut at filer av noe slag kan komprimeres uten at innholdet endres eller ødelegges. I denne innledende seksjonen skal vi se på noen overordnede prinsipper for å illustrere at dette er mulig. Vi skiller mellom to helt forskjellige kompresjonsteknikker: Feilfri kompresjon ('lossless compression' på engelsk) og toleransekompresjon ('lossy compression' på engelsk). Ideen bak feilfri kompresjon skal vi bare skissere meget grovt, mens vi skal studere en teknikk for toleransekompresjon i mer detalj.

10.1.1 Feilfri kompresjon

Som navnet sier er dette en kompresjonsteknikk der det komprimerte dokumentet inneholder nøyaktig den samme informasjonen som det opprinngelige dokumentet. Et enkelt eksempel illustrerer hvordan dette kan gjøres. Anta at vi har en tekstfil som består av bokstaven 'a' skrevet 1000 ganger. Vanligvis krever en bokstav 1 byte (8 bit) med lagerplass, så denne fila vil trenge en lagerplass på 1000 bytes. Hvis vi i steden lagrer informasjonen på en fil ved å angi at a'en skal gjentas tusen ganger kan vi tenke oss at det på fila står '1000a', altså fem tegn. Hvis hvert av disse fremdeles opptar 1 byte har vi klart å lagre teksten med 5 byte i steden for 1000 byte, en stor besparelse. Nå er det



Figur 10.1. En samling av punkter i planet (a) og en rett linje som tilnærmer punktene.

lett å se at dette er litt vel optimistisk for vi må forvisse oss om at betydningen av koden '1000a' blir rett tolket når den komprimerte fila leses igjen. Dette kan bety at vi må lagre en del tilleggsinformasjon, men prinsippet burde være forståelig fra dette eksempelet.

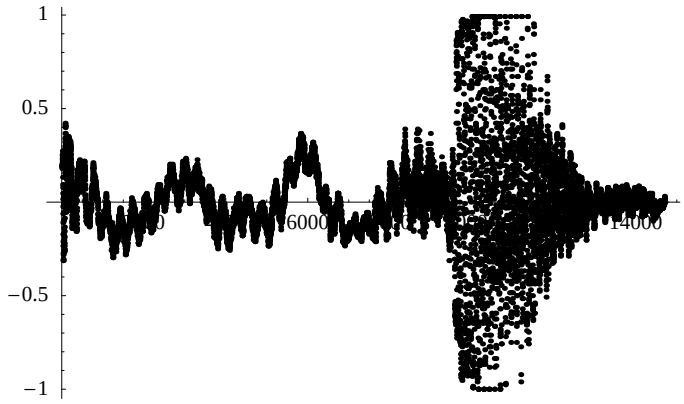
Moderne programmer for feilfri kompresjon ser typisk etter mønstre i fila som skal komprimeres. Hvis for eksempel tegnsekvensen 'ere' forekommer ofte kan den få en egen kode så som '1'. I steden for å skrive 'ere' på fila vår kan vi i steden skrive '1' hver gang denne kombinasjonen forekommer. Ved å bygge opp ordlister over forskjellige tegnkombinasjoner på fila vår, gi de vanligste kombinasjonene en kort kode og deretter erstatte kombinasjonene med koden sin, kan vi ofte få til en kraftig kompresjon av mange typer filer.

Detaljene omkring hvordan dette gjøres er langt fra enkelt og er basert på informasjonsteori og sannsynlighetsregning, men det fins gode gratisprogrammer tilgjengelig som gjør denne jobben bra. Det kanskje vanligste programmet har navnet `gzip`. Når en fil er komprimert er den kodet på en måte som er uleselig for det vanlige datasystemet. Før den kan brukes på normal måte må den derfor pakkes ut, og til det har `gzip` en kompanjong som heter `gunzip`.

10.1.2 Toleransekompresjon

Noen typer data kan tolerere at vi endrer dem lite grann før vi komprimerer dem med en eksakt kompresjonsteknikk. Anta for eksempel at vi har en fil bestående av mange punkter i planet slik som i figur 10.1 (a). I x-retningen ligger punktene med jevn avstand i intervallet $[0, 1]$, mens y-verdiene nesten ligger på en rett linje. I figur 10.1 (b) har vi funnet en rett linje som tilnærmer punktene våre godt. I en del anvendelser vil det være helt greit å flytte punktene slik at de ligger nøyaktig på den rette linja, og i så fall kan vi lagre formelen for den rette linja (sammen med en formel for hvordan x-verdiene skal beregnes) i steden for alle punktene i (a). Hvis vi etterpå har bruk for informasjonen på den opprinnelige fila kan vi ikke gjenskape den eksakt, men vi kan regne ut tilsvarende punkter på den rette linja. Hvis vi til å begynne med hadde mange punkter vil dette kunne redusere størrelsen på fila betraktelig. Etterpå kan vi eventuelt gjøre feilfri kompresjon av fila der vi lagret informasjonen om linja.

En bruker av et kompresjonsprogram slik som dette må kunne oppgi en toleranse til programmet for hvor stor feil som er akseptabel. Er det mulig å finne en rett linje som tilnærmer alle punktene med feil mindre enn toleransen erstatter vi punktene med den



Figur 10.2. Et datasett bestående av 14 707 punkter jevnt fordelt i x-retningen. Datasettet representerer lyden produsert av en person som sier hei, samplet 8000 ganger i sekundet.

rette linja, er det ikke mulig å finne en slik linje må vi la punktene være i fred.

10.2 Flerskala-analyse med stykkevise lineære funksjoner

Det er sjelden et sett punkter nesten ligger på en rett linje som her. Det betyr også at det er sjelden vi kan representere datasettet vårt med liten feil ved hjelp av en rett linje. Vi trenger en mer fleksibel klasse av funksjoner som kan erstatte klassen bestående av rette linjer. Dette er samme problemstilling som vi hadde i kapittel 9. Der fant vi ut at polynomer egner seg dårlig for å tilnærme mange punkter fordi vi da vil trenge polynomer av høy grad, og polynomer av høy grad er uhåndterlige på en datamaskin. Løsningen vi brukte var å hekte sammen flere polynomer av lav grad til sammenlenkede Bernstein-polynomer og sammensatte Bezier-kurver. Utgangspunktet for vår versjon av toleransekompresjon er et enkelt spesilatilfelle av denne ideen.

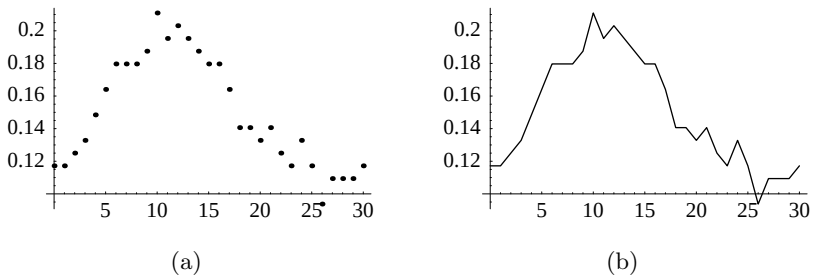
10.2.1 Stykkevis lineær representasjon av data

La oss anta at vi har en stor samling med punkter i planet. Vi tenker oss at punktenes x-koordinater er jevn fordelt i et intervall $[a, b]$ og at vi tilsammen har $2n + 1$ punkter for et passende naturlig tall n. Dette betyr at avstanden mellom to nabopunkter er $h = (b - a)/(2n)$ og at x-koordinatene er gitt ved

$$x_i = a + ih, \qquad \text{for } i = 0,\, 1,\, \ldots,\, 2n.$$

I hvert punkt har vi gitt en y-verdi, og vi kaller y-verdien i x_i for c_i. Et eksempel på et slikt datasett er vist i figur 10.2. Det vi er ute etter er en enkel, men fleksibel klasse av funksjoner som vi kan bruke til å gi en kontinuerlig representasjon av datasettet vårt. Noe av det enkleste vi kan gjøre er å forbinde nabopunkter med rette linjer, akkurat som når vi plotter. Et eksempel på en slik representasjon er vist i figur 10.3.

Vi trenger er en presis måte å skrive opp denne stykkevis lineære funksjonen på. I seksjon 10.5 viser vi en eksplisit måte å gjøre dette på der c_i'ene blir koeffisienter for noen enkle, stykkevis, lineære funksjoner. Men for vårt formål kan vi klare oss med en enklere



Figur 10.3. Et utsnitt av punktene i figur 10.2 er vist i (a) og i (b) er nabopunkter forbundet med en rett linje.

og mer intuitiv representasjon. Vi gir den stykkevis lineære funksjonen som interpolerer de $2n+1$ punktene $(x_i, c_i)_{i=0}^{2n}$ navnet f og skriver

$$f(x) = L_h(c_0, c_1, c_2, \ldots, c_{2n})(x)$$

der indeksen h angir avstanden mellom x-verdiene. Intervallet $[a, b]$ er ikke angitt siden det ligger fast hele tiden. Merk at interpolasjonen betyr at $f(x_i) = c_i$ for $i = 0, 1, \ldots, 2n$.

10.2.2 Oppspalting i en tilnærming og en feilfunksjon

I det enkle eksempelet med punktene som lå nær en rett linje brukte vi størrelsen på avviket fra den rette linja som kriterium på om det var greit å erstatte punktene med linja. Mer presist så måtte vi for hvert punkt sjekke den vertikale avstanden mellom punktet og linja. Denne ideen ønsker vi også å bruke her, men da trenger vi en generell metode for å tilnærme vår kontinuerlige representasjon av det opprinnelige datasettet. Den grunnleggende ideen vi skal bruke er enkel: *Lag en ny stykkevis lineær funksjon som bare bruker annethvert av de opprinnelige punktene, og se på forskjellen mellom f og denne tilnærmingen. Der forskjellen er liten kan vi bruke den nye tilnærmingen som representant for datasettet, der forskjellen er stor må vi passe på å få med mer informasjon.*

La oss se hvordan dette kan gjøres. Vi begynner med å plukke ut annenhver skjøt og setter

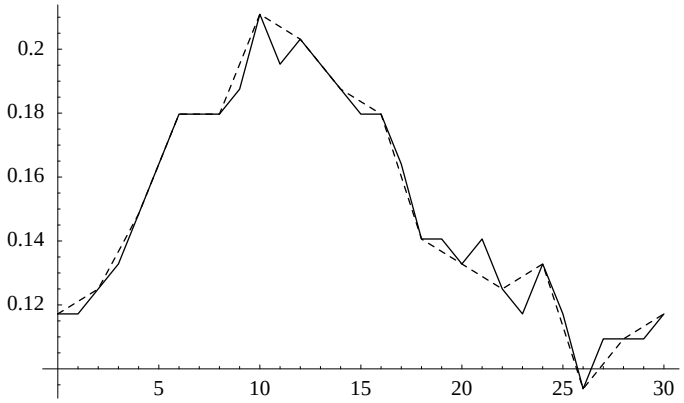
$$z_i = x_{2i}, \quad d_i = c_{2i}, \qquad i = 0, 1, \ldots, n.$$

Legg merke til at avstanden mellom z_i'ene er $2h$, det dobbelte av avstanden mellom x_i'ene. Vi kan nå danne den stykkevis lineære funksjonen som interpolerer disse verdiene

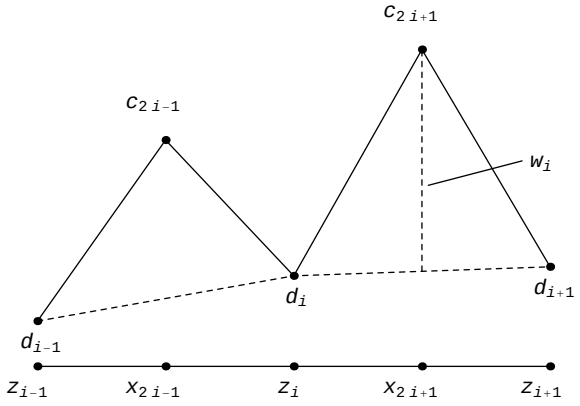
$$g(x) = L_{2h}(d_0, d_1, d_2, \ldots, d_n)(x) = L_{2h}(c_0, c_2, c_4, \ldots, c_{2n})(x).$$

Denne funksjonen tilfredstiller altså betingelsene $g(x_i) = d_i = c_{2i}$ for $i = 0, 1, \ldots, n$.

I figur 10.4 har vi plottet tilnærmingen g og den opprinnelige stykkevis lineære funksjonen f for datasettet i figur 10.3. Som ventet er de to funksjonene forholdsvis like de fleste steder, men i mindre områder er forskjellene ganske store. I områdene der de er like er det derfor rimelig å tro at vi godt kan bruke den grove tilnærmingen g som representant for datasettet, men det går ikke der forskjellen er stor.



Figur 10.4. Den heltrukne funksjonen er den stykkevis lineære tilnærminge til datasettet i figur 10.3 mens den stiplede funksjonen er den stykkevis lineære tilnærmingen til annethvert datapunkt.



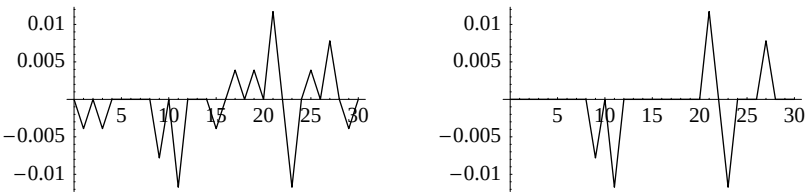
Figur 10.5. Forstørret bilde av de to tilnærmingene f og g (stiplet).

Heldigvis er det mulig på en enkel måte å kombinere f og g. Løsningen er å studere feilfunksjonen $e = f - g$ litt nøyere. Siden g interpolerer annenhver av verdiene som f interpolerer har vi $g(z_i) = g(x_{2i}) = f(x_{2i}) = c_{2i}$ for $i = 0, 1, \ldots, n$, så feilfunksjonen må være null i disse felles punktene. Det er heller ikke så vanskelig å finne verdien i de andre datapunktene.

Lemma 10.1. *Feilfunksjonen $e = f - g$ har egenskapene*

$$v_{2i} = e(x_{2i}) = e(z_i) = 0, \quad \textit{for } i = 0, 1, \ldots, n;$$
$$v_{2i+1} = e(x_{2i+1}) = c_{2i+1} - \frac{1}{2}(d_i + d_{i+1}), \quad \textit{for } i = 0, 1, \ldots, n-1.$$

Bevis. Et detaljbilde av situasjonen er vist i figur 10.5. Vi ser at de to tilnærmingene er like i z_i'ene slik at $v_{2i} = e(z_i) = 0$. Feilen i x_{2i+1} er gitt ved forskjellen $f(x_{2i+1}) - g(x_{2i+1})$. Vi vet at $f(x_{2i+1}) = c_{2i+1}$, mens g er en rett linje på intervallet $[z_i, z_{i+1}]$ og verdien midt



Figur 10.6. Plottet i (a) viser forskjellen mellom de to tilnærmingene i figur 10.4. I (b) har alle koeffisientene i (a) som i tallverdi er mindre enn 0.004 blitt satt lik 0.

mellom de to endepunktene er derfor lik gjennomsnitt av verdiene i endepunktene. Altså er $g(x_{2i+1}) = (d_i + d_{i+1})/2$ og $v_{2i+1} = c_{2i+1} - (d_i + d_{i+1})/2$. ∎

Lemma 10.1 gir oss verdien av feilfunksjonen i alle de opprinnelige skjøtene $(x_i)_{i=0}^{2n}$. I tillegg er feilfunksjonen en rett linje mellom skjøtene og dermed en stykkevis lineær funksjon som interpolerer verdiene gitt i lemma 10.1. Den kan derfor skrives

$$e(x) = L_h(0, v_1, 0, v_3, 0, v_5, \ldots, 0, v_{2n-1}, 0)(x).$$

Denne notasjonen indikerer at e er en funksjon av samme type som f; en stykkevis lineær funksjon med skjøter $\boldsymbol{x}$. Men det er viktig å understreke at vi også vet at i annenhver x_i er e null. I et program bør derfor e representeres annerledes enn f slik at vi bare lagrer v'ene med odde indeks og unngår å lagre alle nullene.

Figur 10.6 (a) viser feilfunksjonen for eksempelet gitt i figur 10.4. Verdiene til f og g varierte i dette eksempelet grovt sett i intervallet $[0.1, 0.2]$, mens vi ser at feilen varierer i intervallet $[-0.01, 0.01]$ og er betydelig mindre i enkelte områder.

Lemmaet under oppsummerer det vi har kommet fram til og her har vi gitt v'ene med oddetallig indeks et eget navn.

Lemma 10.2 (Dekomposisjon)**.** *La den stykkevis lineære funksjonen f være gitt ved $f(x) = L_h(c_0, c_1, \ldots, c_{2n})(x)$. Da kan f skrives som $f = g + e$ der*

$$g(x) = L_{2h}(d_0, d_1, \ldots, d_n)(x), \tag{10.1}$$

$$e(x) = L_h(0, w_0, 0, w_1, \ldots, 0, w_{n-1}, 0)(x) \tag{10.2}$$

og

$$d_i = c_{2i}, \qquad \text{for } i = 0, 1, \ldots, n; \tag{10.3}$$

$$w_i = c_{2i+1} - \frac{d_i + d_{i+1}}{2}, \qquad \text{for } i = 0, 1, \ldots, n-1. \tag{10.4}$$

La oss oppsummere hva vi har oppnådd så langt. Siden $e = f - g$ har vi også $f = g + e$. Vi har med andre ord skrevet vår opprinnelige funksjon f som en sum av en tilnærming g og en feilfunksjon e gitt ved (10.1) og (10.2); vi sier at f er dekomponert i g og e. Disse funksjonene er tilsammen gitt ved de $n + 1$ verdiene $(d_i)_{i=0}^{n}$ og de n verdiene $(w_i)_{i=0}^{n-1}$, totalt $2n + 1$ verdier. Dette er nøyaktig like mange verdier som de opprinnelige $(c_i)_{i=0}^{2n}$

som vi trenger for å definere f, så om vi erstatter f med g og e taper vi ingenting med tanke på lagerplass.

10.2.3 Rekonstruksjon fra tilnærming og feilfunksjon

Om vi skulle finne på å erstatte dekomponere f i g og e er det viktig å vite at vi kan regne oss tilbake til f igjen etterpå. Vi ser litt nøyere på formlene (10.3) og (10.4) for dekomposisjon så ser vi at de er lette å snu på hodet slik at vi kan finne alle c'ene om d'ene og w'ene er kjent.

Lemma 10.3 (Rekonstruksjon)**.** *Anta at de to stykkevis lineære funksjonene* $g(x) = L_{2h}(d_0, d_1, \ldots, d_n)(x)$ *og* $e(x) = L_h(0, w_1, 0, w_2, \ldots, 0, w_n)(x)$ *er gitt. Funksjonen* $f = g+e$ *er da på formen* $f(x) = L_h(c_0, c_1, \ldots, c_{2n})(x)$ *der verdiene* $(c_i)_{i=0}^{2n}$ *er gitt ved*

$$c_{2i} = d_i, \qquad \text{for } i = 0,\, 1,\, \ldots,\, n; \tag{10.5}$$

$$c_{2i+1} = w_i + \frac{d_i + d_{i+1}}{2}, \qquad \text{for } i = 0,\, 1,\, \ldots,\, n-1. \tag{10.6}$$

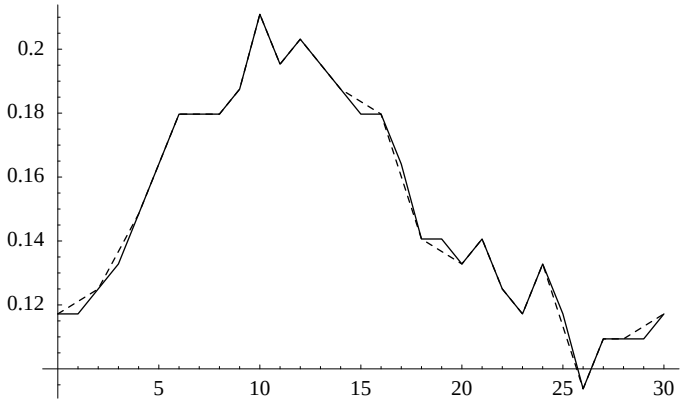
Dette viser at vi enkelt kan konvertere fra en representasjon ved f til en representasjon med g og e og tilbake til f igjen. Rent matematisk er det akkurat den samme funksjonen vi representerer i begge tilfeller. For en del formål er det enklest å manipulere f direkte, for eksempel hvis vi skal plotte ut datasettet eller spille av lyden det representerer. I andre sammenhenger er det bedre å representere f ved de to funksjonene g og e. Dette gjelder ikke minst hvis vi skal forsøke å lagre datasettet på en kompakt måte. Med konverteringsalgoritmene som ligger i lemmaene 10.2 og 10.3 er det en smal sak å veksle mellom de to reprentasjonsformene etter behov.

Det er verdt å legge merke til at dekomposisjonsformlene (10.3)–(10.4) og rekonstruksjonsformlene (10.5)–(10.6) ikke involverer h på noen måte. Denne størrelsen vet vi er dobbelt så stor for g som for f, og den er derfor lett å holde orden på utenfor løkkene gitt ved de fire formlene.

10.2.4 Kompresjon

La oss nå se hvordan vi kan legge forholdene til rette for kompresjon ved å skrive f som $g+e$. Hvis e er liten i noen områder kommer det åpenbart av at noen av w-verdiene i dette området er små. Vi kan derfor sette feilfunksjonen lik 0 i et slikt område ved å sette de aktuelle w-verdiene lik 0. I vårt eksempel blir resultatet som i figur 10.6 (b) om vi setter alle feilkoeffisienter som i absoluttverdi er mindre enn eller lik 0.004 til 0. Hvis vi kaller den nye feilfunksjonen som framkommer på denne måten $\tilde{e}$ så har vi altså $f \approx \tilde{f} = g + \tilde{e}$. Figur 10.7 viser den opprinnelige f'en sammen med $\tilde{f}$ som er stiplet. Sammenligner vi med figur 10.5 ser vi at $\tilde{f}$ er en klart bedre tilnærming til det opprinnelige datasettet enn om vi bruker g alene som tilnærming.

Hvordan får vi til kompresjon? Jo, i steden for å lagre det opprinnelige datasettet lagrer vi de to funksjonene g og $\tilde{e}$ på samme fil og bruker en eksakt kompresjonsmetode på denne fila. Dette bør gi gode muligheter for kompresjon siden de to funksjonene tilsammen inneholder like mange koeffisienter som den opprinnelige funksjonen f, men mange av koeffisientene til $\tilde{e}$ vil være null, noe som gir den eksakte kompresjonsmetoden



Figur 10.7. Summen av den grove tilnærmingen g og den justerte feilen $\tilde{e}$.

gode muligheter til å finne mønstre i fila som den kan utnytte til å redusere behovet for lagerplass.

10.2.5 Flere dekomposisjoner

Oppspaltingen av f i en grovere tilnærming g og en feilfunksjon e har en naturlig utvidelse. Den grove tilnærmingen g er jo en funksjon av akkurat samme type som f (bortsett fra at skjøtene har dobbelt så stor avstand i g som i f). Det er derfor naturlig å utsette g for samme behandling som f og spalte den opp i en enda grovere tilnærming og en ny feilfunksjon.

For å formulere dette er det greiest å gi g navnet g_1 og e navnet e_1. Vi kan da beregne en grovere tilnærming g_2 til g_1 som har dobbelt så stor skjøtavstand som g_1. Dette gir opphav til en ny feilfunksjon $e_2 = g_1 - g_2$ slik at $g_1 = g_2 + e_2$. Denne feilfunksjonen vil igjen ha noen små koeffisienter som vi kan sette lik 0, noe som gjør at vi får en ny feilfunksjon $\tilde{e}_2$. Resultatet av dette er at

$$f = g_1 + e_1 = g_2 + e_2 + e_1,$$

og når feilfunksjonene erstattes med $\tilde{e}_1$ og $\tilde{e}_2$ får vi en tilnærming $\tilde{\tilde{f}}$ til f gitt ved

$$\tilde{\tilde{f}} = g_2 + \tilde{e}_2 + \tilde{e}_1.$$

Hvis vi bruker $\tilde{\tilde{f}}$ som en tilnærming til f og lagrer g_2, $\tilde{e}_2$ og $\tilde{e}_1$ bør en eksakt kompresjonsmetode kunne utnytte dette til en enda mer kompakt lagring av f, med en feil som fremdeles er liten.

Om vi ønsker kan vi spalte opp g_2, deretter g_3 og så videre på samme måte. Siden vi ikke introduserte noen ekstra parametre ved å spalte f i $g + e$ vil dette heller ikke skje når vi fortsetter oppspaltingen. Hvis vi gjør k oppspaltinger er det vi oppnår å skrive f som

$$f = g_k + e_k + e_{k-1} + \cdots + e_1.$$

Siden antall skjøter grovt sett halveres hver gang vil g_k ha omtrent $(2n+1)/2^k$ skjøter. Vi kan maksimalt dele opp inntil vi sitter igjen med 2 skjøter, det vil si når $\frac{2n+1}{2^k} \approx 2$ eller

$$k \approx \log_2(2n+1) - 1.$$

Hvis vi begynner med 1000 punkter gir dette $k \approx 9$, med 10^6 punkter får vi $k \approx 19$ og med 10^9 punkter blir maksimal k omtrent 29.

I praksis er det neppe lurt å spalte opp f det maksimale antall ganger. Husk at første oppdeling gir oss en feilfunksjon som inneholder n verdier. Med tanke på kompresjon er da det beste vi kan oppnå at alle disse n verdiene er 0. Neste gang vi spalter opp får vi en feilfunksjon som inneholder omtrent $n/2$ verdier så nå er det beste som kan skje at vi får $n/2$ nuller i tillegg. Vi ser dermed at kompresjonspotensialet blir stadig mindre ettersom vi spalter opp de grove tilnærmingene våre.

10.3 Praktisk kompresjon av digital lyd

Beskrivelsen over er generell og kompresjonsmetoden som er skissert kan brukes på alle typer data som faller innenfor rammene som er satt. Hvis datasettet representerer et lydsignal er det enkelte sider ved metoden som kan presiseres noe.

10.3.1 Komprimering og avspilling

Kompresjonsmetoden består i å dekomponere f til $f = g_k + e_k + e_{k-1} + \cdots + e_1$, sette alle verdier som beskriver $\{e_j\}_{j=1}^k$ som er mindre enn en gitt toleranse til null, lagre g_k og alle feilfunksjonene på en fil og så bruke en metode for eksakt kompresjon på denne fila. Dette vil gi en ny fil som forhåpentligvis tar opp mindre plass enn den første vi skrev.

Hvis vi skal spille av lyden må vi kunne gjenskape f (egentlig en tilnærming til f siden vi har satt en del koeffisienter til å være null). Da må vi først pakke ut informasjonen om g_k og feilfunksjonene fra den lagrede informasjonen og så rekonstruere tilnærmingen til f som en vanlig stykkevis lineær funksjon som så kan avspilles.

10.3.2 Samplingsrate

For et digitalt lydsignal er det tiden som løper langs x-aksen, men vi holder oss til tidligere notasjon og bruker x som variabel likevel. Imidlertid er det ikke så vanlig å angi avstanden h mellom nabopunkter når vi arbeider med lyd, men heller antall verdier pr. sekund, altså samplingsraten. Hvis samplingsraten er s er avstanden mellom samplene gitt ved $h = 1/s$. For lyd er det også vanlig å tenke seg at tiden begynner ved første datapunkt, altså er $a = 0$.

10.3.3 Lyd krever 16 bits heltall

Digital lyd opererer som regel med dataverdier som er 16 bits heltall, i Java kalles denne data-typen `short`. I tidligere kapitler der vi har diskutert digital lyd har vi antatt at sample-verdiene (c_i'ene her) har vært flyttall, typisk av typen `float`, normalisert til å ligge mellom -1 og 1. Når vi skal forsøke oss på kompresjon er dette ikke så lurt. En `short` tar opp 2 bytes (16 bit) med lagerplass mens en `float` opptar 4 bytes (32 bit). Dette betyr at om vi konverterer alle verdiene fra `short` til `float` dobler vi automatisk lagerplassen!

Nå kunne det jo tenkes at denne overflødige informasjonen ble fjernet igjen når vi bruker vår feilfrie kompresjonsmetode, men dessverre er så ikke alltid tilfelle. Vi bør derfor holde oss til datatypen `short` når vi implementerer dekomposisjon og rekonstruksjon som beskrevet i lemma 10.2 og 10.3 med tanke på kompresjon.

Hvis vi ser på operasjonene vi skal utføre så er det mest kompliserte vi gjør i dekomposisjonen å regne ut uttrykket

$$w_i = c_{2i+1} - \frac{d_i + d_{i+1}}{2} \tag{10.7}$$

der c_{2i+1}, d_i og d_{i+1} er heltall av type `short`. Hvis divisjonen ikke går opp vil svaret bli trunkert, for eksempel vil 3/2 bli regnet ut til 1. Her gjør vi altså en feil, men denne vil vanligvis være ubetydelig i forhold til de endringene vi gjør når alle verdier mindre enn toleransen settes til 0. I rekonstruksjonen har vi det tilsvarende uttrykket

$$c_{2i+1} = w_i + \frac{d_i + d_{i+1}}{2}. \tag{10.8}$$

Siden akkurat de sammen tallene er involvert for hver i vil resultatet av $(d_i + d_{i+1})/2$ bli regnet ut på samme måte som i (10.7) slik at effekten av å gjøre disse to operasjonene i sekvens er akkurat det vi ønsker, de opphever hverandre. Dette sikrer at det å dekomponere f til $g+e$ og så rekonstruere igjen (uten å sette noen verdier til null) gir oss nøyaktig f. Og husk, vi bruker `short` så det er ingen avrundingsfeil!

Et noe mer alvorlig problem er overflow. Det kan tenkes at c_{2i+1} har en verdi som er lik det største positive heltallet som kan lagres i en `short` og at d_i og d_{i+1} begge er negative. Da vil høyresiden i (10.7) bli et heltall som er for stort for en `short`. Vi vet at i slike situasjoner gir ikke Java feilmeldinger, men kaster de mest signifikante sifrene slik at resultatet blir tilsynelatende tilfeldig. Dette kan unngås ved å huske på at Java faktisk vil oversette alle variable av type `short` til `int` og gjøre alle heltallsberegninger med tall av typen `int`. En `int` består som vi husker av 4 bytes slik at vi aldri vil overskride grensen for største heltall av type `int` i operasjonene over. Vi kan derfor sjekke om resultatet i (10.7) eller (10.8) blir for stort og i såfall gi som resultat det største heltall som passer i en `short` (eventuelt det minste negative tallet som passer i en `short` hvis vi får overflow andre veien). I praksis vil slik overflow forhåpentligvis forekomme så sjelden at det vanligvis ikke vil være hørbart i dårlige PC-høytalere eller hodetelefoner om vi ignorerer det.

10.3.4 Datasettet inneholder et like antall punkter

Det siste potensielle problemet vi kan få er mer grunnleggende. Vi har antatt at vi starter med et datasett som består av $2n+1$ punkter, altså et odde antall. Grunnen til dette er at da kan vi på en pen måte plukke ut annenhver verdi når vi skal beregne vår grovere tilnærming g. Men hva om lydsignalet vårt består av et like antall punkter, hva kan vi da gjøre? Anta at vi har $2n$ punkter med x-koordinater $(x_i)_{i=0}^{2n-1}$ og at vi begynner med det første punktet i venstre ende og plukker ut annethvert punkt. I høyre ende vil da endepunktet for g bli (x_{2n-2}, c_{2n-2}), mens datapunktet (x_{2n-1}, c_{2n-1}) ligger til høyre for dette punktet. Siden g ikke har noen datapunkter lenger til høyre er det rimelig å si at g er

konstant lik c_{2n-2} til høyre for x_{2n-2}. Dermed er $w_n = e(x_{2n-1}) = f(x_{2n-1}) - g(x_{2n-1}) = c_{2n-1} - c_{2n-2}$.

I rekonstruksjonen må vi passe på at dette blir gjort i motsatt rekkefølge. Vi går gjennom algoritmen på vanlig måte, men til slutt må vi regne ut $c_{2n-1} = w_n + c_{2n-2}$.

Et enklere, men ikke fullt så godt alternativ er å utvide datasettet ved å gjenta verdien lengst til høyre slik at vi får ønsket lengde. Hvis antall punkter er et partall gjentar vi bare siste verdi en ekstra gang. Husk at et lydsignal innholder mange tusen verdier for hvert sekund så en slik liten endring vil ikke bli merkbar så lenge verdien ikke er merkbart annerledes enn verdien foran.

10.3.5 Dekomposisjon flere ganger

Anta nå at vi skal dekomponere flere ganger. Hvis vi begynner med $2n + 1$ punkter vil den første tilnærmingen g_1 være basert på $n + 1$ punkter. For at vi skal kunne gjenta dekomposisjonen uten problemer må $n + 1$ også være et oddetall, hvilket betyr at n må være et partall, $n = 2n_2$ for et passende tall n_2. Den nye tilnærmingen g_2 vi da beregner vil være basert på de n_2+1 punktene vi får ved å plukke annethvert punkt fra g_1, og for å kunne dekomponere g_2 må $n_2 = 2n_3$ for et passende tall n_3. Gjentar vi dette ser vi at om vi skal dekomponere k ganger uten å få problemer må det opprinnelige datasettet inneholde $N = 2^k n_k + 1$ punkter for et passende naturlig tall n_k. Om dette ikke er tilfellet kan vi enten bruke teknikken med å spesialbehandle et eventuelt overskytende ekstra punkt på høyre side ved hver dekomposisjon, eller duplisere siste punkt så mange ganger at det totale antall punkter blir slik vi trenger det. Legg merke til at om vi velger siste løsning vil dette ikke ødelegge for kompresjonsmulighetene siden alle verdiene til feilfunksjonene i dette området vil bli null og dermed gi godt potensiale for eksakt kompresjon.

10.3.6 Valg av toleranse

En viktig ingrediens i en kompresjonsstrategi basert på flerskala-analyse er valg av toleranse. Stor toleranse vil tillate at $\tilde{e}$ avviker mye fra e og dermed gi stor feil og mye støy i signalet vi faktisk lagrer med tanke på kompresjon. Samtidig kan det godt hende at det er greit med relativt mye støy. Skal det lagrede lydsignalet kun avspilles over en telefonlinje er det klart at kravet til kvalitet er et helt annet enn om signalet representerer musikk som skal spilles på et dyrt lydanlegg. Det kan også tenkes at kapasiteten i avspillersystemet setter begrensninger på hvor mye informasjon som kan overføres pr. tidsenhet og dermed tvinger fram en relativt stor toleranse.

10.3.7 Store datasett

Hvis antall punkter er svært stort vil arrayene vi bruker til dekomposisjon og rekonstruksjon bli tilsvarende store, og blir de så store at vi ikke får plass til alle data i maskinens hukommelse vil beregningene gå svært sakte. Løsningen på dette problemet er å dele datasettet i passelig store biter som ikke er større enn at hver bit kan håndteres raskt og effektivt med algoritmene vi har skissert her.

10.4 Andre anvendelser av flerskala-analyse

Oppspaltingsteknikken vi har beskrevet over har fått navnet 'flerskala-analyse'. Navnet er ikke så overaskende om vi tenker litt over hva vi har oppnådd. La oss tenke oss at vårt opprinnelige datasett representerer et lydsignal samplet slik at høyeste frekvens vi kan få med er ν. For at dette skal være mulig husker vi fra tidligere kapitler om lyd at vi trenger 2ν sampler pr. sekund. Når vi spalter opp f i $g_1 + e_1$ vet vi at g_1 inneholder halvparten så mange verdier som f. Dermed kan ikke g_1 inneholde høyere frekvenser enn $\nu/2$, så feilfunksjonen e_1 må inneholde de resterende frekvensene i f som ligger mellom $\nu/2$ og ν. Spalter vi opp g_1 i $g_2 + e_2$ kan vi bruke det samme argumentet igjen. Vi ser at g_2 ikke kan inneholde høyere frekvenser enn $\nu/4$, og at e_2 derfor må inneholde frekvensene i f som ligger i intervallet $[\nu/4, \nu/2]$.

Hvis vi gjør k dekomponeringer slik at $f = g_k + e_k + e_{k-1} + \cdots + e_1$ får vi spaltet f i en komponent som ikke innholder frekvenser over $\nu/2^k$ og så k feilsignaler som innholder frekvenser i intervallene

$$[\nu/2^k, \nu/2^{k-1}], [\nu/2^{k-1}, \nu/2^{k-2}], \ldots, [\nu/4, \nu/2], [\nu/2, \nu].$$

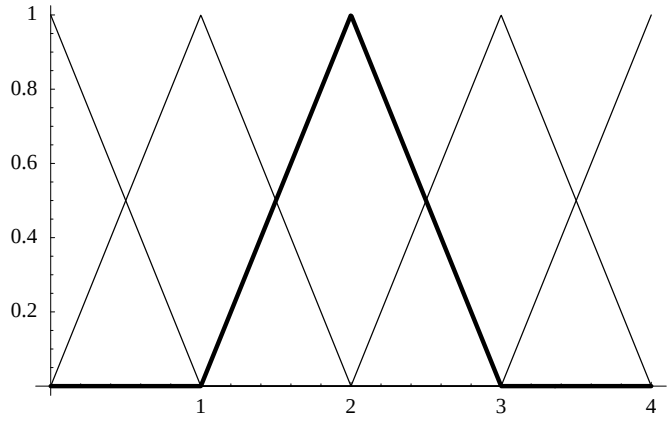
Vi har med andre ord fått delt opp signalet vårt i delsignaler som innholder svingninger på forskjellig skala, derav navnet flerskala-analyse.

Vi ser at et dekomposisjonen gir et glimrende utgangspunkt for å justere de ulike frekvensene i signalet vårt. Setter vi for eksempel e_1 til å være identisk null har vi fjernet alle frekvenser mellom $\nu/2$ og ν, med andre ord fjernet mesteparten av diskanten, mens hvis vi setter $g_k = 0$ forsvinner bassen. Poenget er at oppspaltingen i ulike frekvenser gir tilgang på informasjon om signalet som ikke er så lett å se når signalet er i sin opprinnelige form. Ved hjep av denne informasjonen kan vi for eksempel fjerne støy fra signalet, lage spesielle effekter, gjenkjenne ulike typer fenomener og veldig mye annet.

Når det ovenstående er sagt er det på sin plass med en liten demper på entusiasmen. Frekvensbegrepet er definert ut fra hvor mange ganger en sinus eller cosinus svinger på et sekund og ikke hvor mange ganger en stykkevis lineær funksjon svinger. Det viser seg at dette betyr at selv om vi setter $e_1 = 0$ vil restsignalet g fremdeles kunne inneholde høye frekvenser. Løsningen på dette problemet er å bruke mer raffinerte funksjoner enn de stykkevis lineære til å representere og tilnærme datasettet. For eksempel er det vanlig å bygge flerskala-analyse basert på stykkevise polynomer av høyere grad som er limt sammen på en glatt måte, og jo glattere sammenlimingen er jo mindre blir frekvensproblemet nevnt over. Og på tross av kantene er det svært mye som kan gjøres med flerskala-analysen vi har skissert i dette kapitlet. Den matematiske teorien for denne type dekomposisjoner er fokusert rundt noen spesielle funksjoner som kalles *wavelets* (hattefunksjonene i seksjon 10.5 kan anses som en type wavelets).

10.5 Representasjon av stykkevis lineære funksjoner

Foran har vi skrevet f som $f(x) = L_h(c_0, c_1, \ldots, c_{2n})(x)$, og g på en tilsvarende måte. Det fins en annen skrivemåte der avhengigheten av $(c_i)_{i=0}^n$ blir tydeligere.



Figur 10.8. De 5 ϕ-funksjonene svarende til $x_i = i$ for $i = 0, 1, \ldots, 4$. Funksjonen ϕ_2 har blitt plottet med tykkere strek for å understreke at hver funksjon er en slik hattefunksjon, bortsett fra den første som starter med verdi 1 i $x = 0$ og faller til 0 i $x = 1$ og så er null opp til 4, og den siste som er null opp til $x = 3$ og så vokser til verdien 1 i $x = 4$.

Setning 10.4. *Den stykkevis lineære funksjonen f som tilfredstiller betingelsene $f(x_i) = c_i$ for $i = 0, 1, \ldots, 2n$ kan skrives som*

$$f(x) = \sum_{i=0}^{2n} c_i \phi_i(x)$$

der ϕ_i er funksjonen gitt ved

$$\phi_i(x) = \begin{cases} (x - x_{i-1})/h, & \text{for } x \in [x_{i-1}, x_i); \\ (x_{i+1} - x)/h, & \text{for } x \in [x_i, x_{i+1}); \\ 0, & \text{ellers.} \end{cases}$$

for $i = 1, \ldots, 2n-1$ mens

$$\phi_0(x) = \begin{cases} (x_1 - x)/h, & \text{for } x \in [0, x_1); \\ 0, & \text{ellers} \end{cases}$$

$$\phi_{2n}(x) = \begin{cases} (x - x_{2n-1})/h, & \text{for } x \in [x_{2n-1}, x_{2n}]; \\ 0, & \text{ellers.} \end{cases}$$

Tallene $\boldsymbol{x} = (x_i)_{i=0}^{2n}$ kalles skjøtene til de stykkevis lineære funksjonene. Hvis det er nødvendig å understreke ϕ_i's avhengighet av $\boldsymbol{x}$ brukes notasjonen $\phi_{i,\boldsymbol{x}}$.

Et eksempel på en samling av slike ϕ_i'er er vist i figur 10.8. Når $1 \leq i \leq 2n-1$ er funksjonen ϕ_i en 'hattefunksjon' som har verdien 1 i x_i og faller lineært til begge sider slik at den får verdien 0 i x_{i-1} og x_{i+1}. Utenfor intervallet $[x_{i-1}, x_{i+1}]$ er funksjonen identisk null.

Bevis. La oss først sjekke at f har riktig verdi i skjøtene. Fra konstruksjonen vet vi at hver av ϕ_i'ene har verdien 1 i en skjøt og verdien 0 i alle de andre. Hvis vi skal sjekke verdien til f i x_j vet vi altså at den eneste hattefunksjonen som har en verdi i x_j er ϕ_j som har verdien 1 der. Dette gir

$$f(x_j) = \sum_{i=0}^{2n} c_i \phi_i(x_j) = c_j \phi_j(x_j) = c_j.$$

Hvis vi i tillegg klarer å vise at f er en rett linje mellom skjøtene er det klart at f må være den stykkevis lineære funksjonen som interpolerer de gitte verdiene. Men husk at mellom skjøtene er alle ϕ_i'ene polynomer av grad 1, og en sum av polynomer av grad 1, multiplisert med tall, er igjen et polynom av grad 1, altså en rett linje. ∎

Vi kan selvsagt skrive opp g med samme konstruksjon og får da

$$g(x) = \sum_{i=0}^{n} d_i \phi_{i,\boldsymbol{z}}(x)$$

(merk at hattefunksjonene nå er konstruert ut fra skjøtene $\boldsymbol{z}$). Feilfunksjonen e kan vi skrive som

$$e(x) = \sum_{i=1}^{n} w_i \phi_{2i-1,\boldsymbol{x}}(x).$$

Funksjonene $\{\phi_{i,\boldsymbol{x}}\}_{i=0}^{2n}$ spiller rollen som byggeklosser for de stykkevis lineære funksjonene med skjøter $\boldsymbol{x}$, og har samme funksjon for stykkevis lineære polynomer som basispolynomene 1, x, …, x^n (eller en annen basis) har for polynomer av grad n. I begge tilfeller får vi fram en funksjon i den aktuelle klassen ved å multiplisere med koeffisienter og summere opp. Dette er en svært generell konstruksjon som er helt analog med det å skrive vektorer ved hjelp av basisvektorer, og som studeres i lineær algebra.

Oppgaver

10.1 Flerskala-analysen vi utviklet i teksten er ikke den aller enkleste som kan konstrueres. Den enkleste framkommer om vi sier at funksjonen f som interpolerer de gitte verdiene $(x_i, c_i)_{i=0}^{2n}$ er konstant mellom hver verdi,

$$f(x) = \begin{cases} c_i, & \text{for } x \in [x_i, x_{i+1}) \text{ og } i = 0, 1, \ldots, 2n-1; \\ c_{2n}, & \text{for } x = x_{2n}. \end{cases}$$

Hvis vi bruker en notasjon tilsvarende den i teksten kan vi skrive dette litt mer kompakt som $f(x) = K_h(c_0, c_1, \ldots, c_{2n})(x)$. Tilnærmingen med dobbel avstand mellom verdiene er nå gitt ved den stykkevis konstante funksjonen som kan skrives $g(x) = K_{2h}(c_0, c_2, \ldots, c_{2n})(x)$, og feilfunksjonen er $e(x) = f(x) - g(x)$.

a) Studer feilfunksjonen og utled en analog til lemma 10.1.

b) Bruk resultatet i (a) til å utlede formler tilsvarende (10.3)–(10.4) og (10.5)–(10.6) som kan danne basis for en alternativ dekomposisjons- og rekonstruksjonsstrategi.

c) Legg merke til at denne konstruksjonen er litt usymmetrisk siden den første verdien brukes over et helt intervall, men den siste bare i et punkt. Kan du tenke deg en måte å justere konstruksjonen på som blir mer symmetrisk?