KAPITTEL 10

Flerskala-analyse og kompresjon av lyd

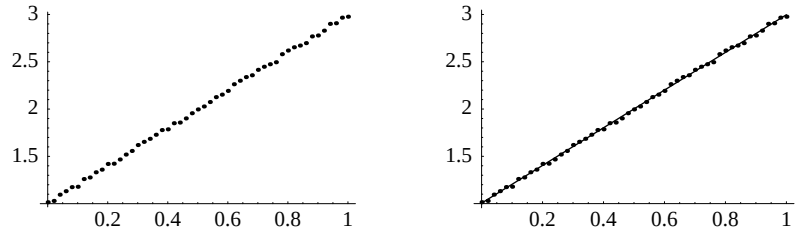
Vi kan lagre dokumenter av mange forskjellige typer på en datamaskin. Vi kan for eksempel ha en datafil der innholdet er tall, vi kan ha en tekstfil, en lydfil, en fil med bilder, en med video, og eventuelt kombinasjoner av disse. Et grunnleggende problem er at en del av disse dokumenttypene lett blir 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 kr-



(a) (b)

Figur 10.1. En samling av punkter i planet (a) og en rett linje som tilnærmer punktene.

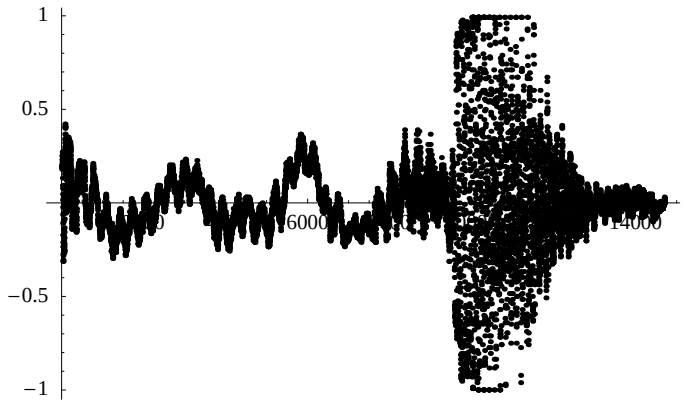
ever en bokstav 1 byte (8 bit) med lagerplass, så denne fila vil trenge en lagerplass på 1000 bytes. Hvis vi i steden lagrer informasjonen på en fil ved å angi at a'en skal gjentas tusen ganger kan vi tenke oss at det på fila står '1000a', altså fem tegn. Hvis hvert av disse fremdeles opptar 1 byte har vi klart å lagre teksten med 5 byte i steden for 1000 byte, en stor besparelse. Nå er det 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



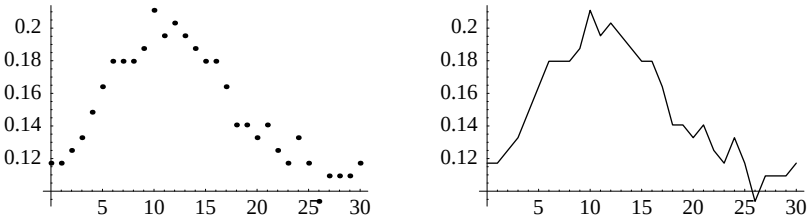
Figur 10.2. Et datasett bestående av 14 707 punkter jevnt fordelt i x-retningen. Datasettet representerer lyden produsert av en person som sier hei, samplet 8000 ganger i sekundet.

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 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.



(a) (b)

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.

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.

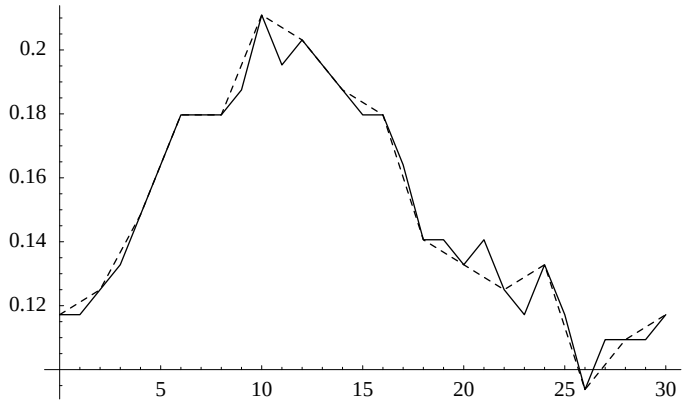
Vi trenger er en presis måte å skrive opp denne stykkevis lineære funksjonen på. I seksjon 10.5 viser vi en eksplisit måte å gjøre dette på der c_i'ene blir koeffisienter for noen enkle, stykkevis, lineære funksjoner. Men for vårt formål kan vi klare oss med en enklere og mer intuitiv representasjon. 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



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.

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

$$z_i = x_{2i}, \quad d_i = c_{2i}, \qquad i = 0,\, 1,\, \ldots,\, n.$$

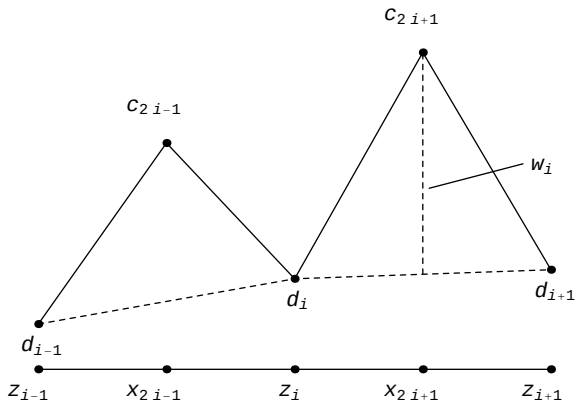
Legg merke til at avstanden mellom z_i'ene er $2h$, det dobbelte av avstanden mellom x_i'ene. Vi kan nå danne den stykkevis lineære funksjonen som interpolerer disse verdiene

$$g(x) = L_{2h}(d_0, d_1, d_2, \ldots, d_n)(x) = L_{2h}(c_0, c_2, c_4, \ldots, c_{2n})(x).$$

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.

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 $i = 0,\, 1,$



Figur 10.5. Forstørret bilde av de to tilnærmingene f og g (stiplet).

$\ldots$, 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*

$$v_{2i} = e(x_{2i}) = e(z_i) = 0, \quad \textit{for } i = 0,\, 1,\, \ldots,\, n;$$
$$v_{2i+1} = e(x_{2i+1}) = c_{2i+1} - \frac{1}{2}(d_i + d_{i+1}), \quad \textit{for } i = 0,\, 1,\, \ldots,\, n-1.$$

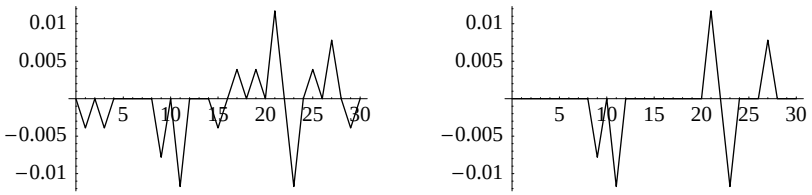
Bevis. Et detaljbilde av situasjonen er vist i figur 10.5. Vi ser at de to tilnærmingene er like i z_i'ene slik at $v_{2i} = e(z_i) = 0$. Feilen i x_{2i+1} er gitt ved forskjellen $f(x_{2i+1}) - g(x_{2i+1})$. Vi vet at $f(x_{2i+1}) = c_{2i+1}$, mens g er en rett linje på intervallet $[z_i, z_{i+1}]$ og verdien midt 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.



(a) (b)

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.

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* $f(x) = L_h(c_0, c_1, \ldots, c_{2n})(x)$ *være gitt. Da kan vi skrive* f *som* $f = g + e$ *der*

$$g(x) = L_{2h}(d_0, d_1, \ldots, d_n)(x), \tag{10.1}$$

$$e(x) = L_h(0, w_1, 0, w_2, \ldots, 0, w_n, 0)(x) \tag{10.2}$$

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

$$c_{2i} = d_i, \qquad \textit{for } i = 0, 1, \ldots, n; \tag{10.5}$$

$$c_{2i+1} = w_i + \frac{d_i + d_{i+1}}{2}, \qquad \textit{for } i = 0, 1, \ldots, n-1. \tag{10.6}$$

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.

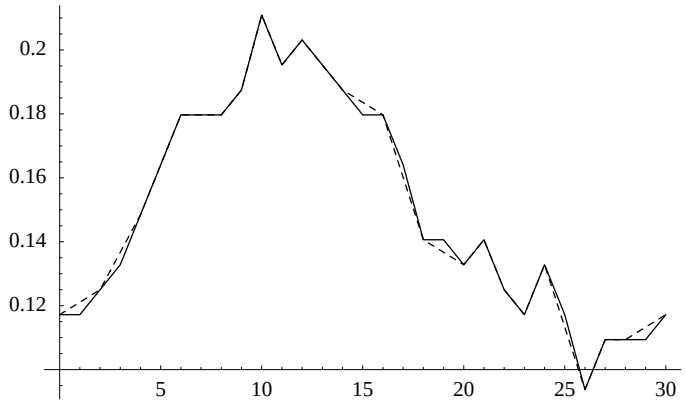
10.2.4 Kompresjon

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 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



Figur 10.7. Summen av den grove tilnærmingen g og den justerte feilen $\tilde{e}$.

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.

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

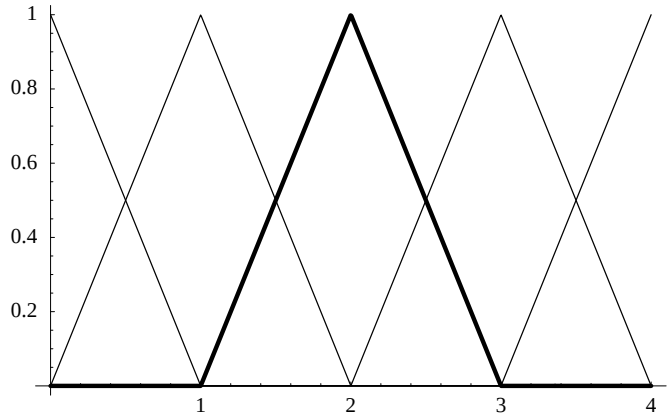
$$\phi_0(x) = \begin{cases} (x_1 - x)/h, & \text{for } x \in [0, x_1); \\ 0, & \text{ellers} \end{cases}$$

$$\phi_{2n}(x) = \begin{cases} (x - x_{2n-1})/h, & \text{for } x \in [x_{2n-1}, x_{2n}]; \\ 0, & \text{ellers.} \end{cases}$$

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



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 $x = 4$.

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 $1, x, \ldots, x^n$ (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 $f(x) = K_h(c_0, c_1, \ldots, c_{2n})(x)$. 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?