NTNU
Institutt for matematiske fag

TMA4135 Matematikk 4D, programmeringsøving 8, høst 2013

En anvendelse av Fourieranalyse

Denne programmeringsøvingen er ment til å illustrere én av svært mange anvendelsene av Fourieranaylse, nemlig *bildekomprimering*. Der vil gis mye informasjon, og øvingen vil forhåpentligvis være nyttig også om du ikke får til programmeringsoppgavene.

Denne øvingen begynner og slutter med litt informasjon om hvordan bildekomprimering fungerer generelt, noe som selvsagt ikke er pensum i Matematikk 4D. Denne informasjonen er kun til orientering, men anbefales likevel som en forankring av pensum.

Det bør presisereres at ingen i fagstaben har formell kompetanse inne bildebehandling, og at det som presenteres derfor må ses på som "approksimasjoner til virkeligheten".

Komprimering av informasjon

Komprimering handler om å representere informasjon på en kompakt måte. Dersom du blir bedt om å huske tekststrengen "acdbaaaaaaaaaaaaaaaaaaaamk", vil du kanskje heller huske "a, c, d, b, 20 a-er, m, k". Du har oppnådd en kortere streng å måtte huske, og du kjenner til reglene som må til for fullstendig å gjenopprette den opprinnelige strengen fra den du går og husker på.

Komprimeringen over er et eksempel på *tapsfri* ("lossless") komperimering, altså hvor den opprinnelige informasjonen kan gjenskapes perfekt. Du kjenner sikkert til mange bruksområder for tapsfri komprimering innen IT, som for eksempel zip-filer for generelle data, eller PNG-formatet for grafikk.

Tapsfri komprimering er nyttig, men i praksis er det begrenset hvor mye slik komprimering kan krympe data. For å oppnå enda mindre filstørrelser, kan *komprimering med tap* ("lossy") benyttes. Med tap vil vi ikke kunne perfekt gjenskape de opprinnelige data, men snarere en approksimasjon. Komprimering med tap er svært mye brukt i lyd- og bildebehandling, for eksempel i kjente formater som MP3 og JPEG. Det viser seg at i mange tilfeller er en approksimasjon mer enn godt nok, spesielt når menneskelige sanser som hørsel og syn er involvert.

Eksempelbildet til venstre i figur 1 er på omtrent 0,3 megapiksler, og opptar ca. 900 kB i opprinnelig form. PNG-komprimert er bildet 100% identisk, og opptar omtrent 460 kB. Med JPEG-komprimering kan bildefilen gjøres betraktelig mindre; med en relativt aggressiv innstilling opptar bildet kun 20 kB, men da er noe informasjon gått tapt. Bildet til høyre i figur 1 illustrerer imidlertid at dette ikke er umiddelbart åpenbart for det menneskelige øye. (Studerer man derimot bildene nøye, ser en snart mangler, spesielt på bladet bak froskens høyre øye.)

Figur 1: Rødøyet trefrosk *(Agalychnis callidryas)*. Bildet til venstre er 0,3 megapiksler stort, og opptar da en knapp megabyte. Bildet til høyre er JPEG-komprimert til 20 kilobyte, ca. 1/45 av originalen, og noe informasjon er gått tapt. Wikimedia Commons, Carey James Balboa. Frafalt opphavsrett.

1 Vårt mål: bildeformatet "JPAG"

Vi skal i denne øvingen forsøke å lage vår egen, svært forenklede, bildekomprimeringsalgoritme med tap. Fouriertransformasjon vil stå sentralt (som den også faktisk gjør i ordentlig JPEG).

Da JPEG står for *Joint Photographic Experts Group*, foreslår jeg at vi døper vår bildekomprimering *JPAG*; vi kan vel sies å være nettopp en gruppe amatører.

Fra Fourier-rekker vet vi at vi kan representere en periodisk, stykkevis kontinuerlig funksjon ved uendelig mange tall, nemlig Fourier-koeffisientene $a_0, a_1, \ldots$ og $b_1, \ldots$. Kjenner vi alle disse, kjenner vi også funksjonen fullstendig, bortsett fra i eventuelle diskontinuitetspunkter. Vi har også sett at ved å beholde kun *noen* av leddene i Fourier-rekken, får vi en approksimasjon til funksjonen vår. Dette vil være hovedingrediensen i vår fattigmanns-algoritme for bildekomprimering: vi finner "Fourier-rekken til bildet", og tar vare på *noen* av Fourier-koeffisientene. Jo færre vi tar vare på, jo kraftigere komprimering.

2 Digital representasjon av bilder

For å forenkle det datatekniske, ser vi på såkalte *gråtonebilder* (ofte kalt "sort-hvitt" i dagligtalen). Et gråtonebilde n piksler bredt og m piksler høyt kan i vårt tilfelle representeres som en $m \times n$-matrise B med elementer som er heltall mellom 0 og 255. Element $B_{i,j}$ representerer en piksel i rad i og kolonne j i bildet, og verdien angir gråtonen ("fargen", om du vil): en verdi på 0 indikerer svart, og en verdi på 255 indikerer hvitt. Verdier mellom er forskjellige nyanser av grått. Siden det krever 1 byte for nøyaktig å representere et heltall mellom 0 og 255, vil et slikt gråtonebilde oppta mn byte ukomprimert. For å oppsummere: Et bilde n piksler bredt og m piksler høyt er for oss en $m \times n$-matrise med elementer som er heltall mellom 0 og 255. Du vil få ferdigskrevet kode for å lese inn bildefiler til slike matriser.

Bilder er *diskret* informasjon, og *diskret Fourier-transformasjon* er *egentlig* riktig verktøy å benytte. Siden dette ikke er pensum i Matematikk 4D, vil vi nå overføre situasjonen til et kjent tilfelle.[1][2]

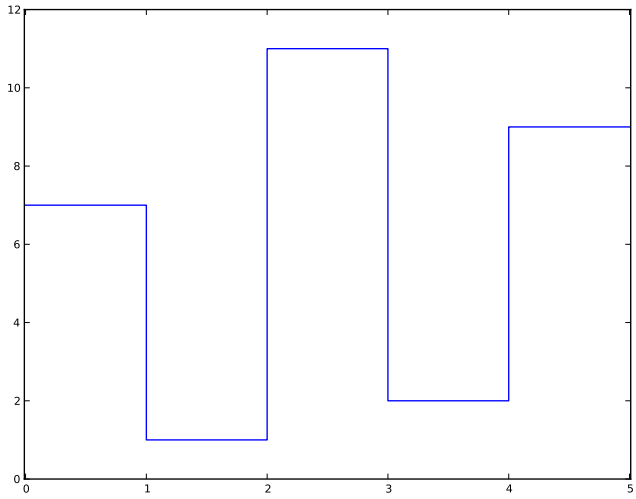
For hver rad i (nummerert fra 0) i en $m \times n$-bildematrise B definerer vi "radfunksjonen" $f_i : [0, n) \to \mathbb{R}$ ved

$$f_i(x) = B_{i,\lfloor x \rfloor} \quad \text{hvor } \lfloor x \rfloor \text{ betyr } x \text{ rundet nedover til nærmeste heltall.}$$

Forvirret? Fatt mot, disse funksjonene illustreres best ved et eksempel: Se på

$$B = \begin{pmatrix} 5 & 4 & 3 & 1 & 10 \\ 7 & 1 & 11 & 2 & 9 \\ 0 & 3 & 6 & 6 & 8 \end{pmatrix}, \tag{1}$$

som godt kan komme fra et svært lite (5 piksler bredt, 3 høyt) og mørkt (alle verdiene er nær 0) bilde. Figur 2 viser radfunksjonen f_1 for denne matrisen/bildet. Funksjonen plukker altså bare ut rad nummer 1 (den *andre* raden, siden vi teller fra 0) fra B. Sjekk at du forstår hva som foregår!



Figur 2: Radfunksjonen f_1 tilhørende matrisen B i ligning (1).

[1] Se også kommentarer i avsnitt 4.

[2] Du kan lese mer om DFT/FFT til sist i avsnitt 11.9 i Kreyszig.

Det er klart at hvis vi kjenner f_i for alle $0 \le i < m$, så kjenner vi også B (og vice versa).

2.1 Fourier-rekke-koeffisienter for stykkevis konstante funksjoner

Målet er nå innen rekkevidde: vi kan finne Fourier-rekke-koeffisientene til hver av funksjonene f_i ($0 \le i < m$), og ta vare på kun så mange som vi ønsker. Jo færre, jo mer komprimering. Som vi vet har vi da approksimasjoner for hver f_i, og vi kan derfor per diskusjonen i forrige avsnitt bygge opp en approksimasjon til bildematrisen B.

I øving 6 skrev vi kode for å finne Fourier-rekke-koeffisientene til en funksjon ved hjelp av numerisk integrasjon. Vi kan i prinsippet gjenbruke denne koden, men kjøretiden vil bli *meget* lang grunnet alle integrasjonene som inngår. Vi reddes av følgende: I vårt tilfelle er f_i spesielle funksjoner – de er stykkevis konstante (se figur 2). For slike funksjoner kan vi lett finne Fourier-rekke-koeffisientene for hånd.

Vi kommer til å se på den *jevne* periodiske utvidelsen til f_i-ene, så alle $b_k = 0$ (vi jobber altså med Fourier-*cosinus*-rekker).

Oppgave 1 La $L > 0$ være et heltall, og $h : [0, L) \to \mathbb{R}$ en stykkevis konstant funksjon som på hvert intervall på formen $[l, l+1)$, l heltall, er konstant (med verdi $h(l)$). (Radfunksjonene f_i, som illustrert i figur 2, er eksempler på slike funksjoner.) Vis at koeffisientene $a_0, a_1, \ldots, a_k, \ldots$ i Fourier-cosinus-rekken til h er

$$a_0 = \frac{1}{L}\sum_{l=0}^{L-1} h(l)$$
$$a_k = \frac{2}{k\pi}\sum_{l=0}^{L-1} h(l)\left(\sin\left(\frac{k\pi}{L}(l+1)\right) - \sin\frac{k\pi l}{L}\right).$$

(Hint: Del opp integralene i summer av integraler som hver går fra l til $l+1$ for heltallige l).

Oppgave 2 Det er klart at en funksjon som $h : [0, L) \to \mathbb{R}$ som i oppgave 1 (eller f_i fra forrige avsnitt) kan representeres som en vektor av lengde L (element nummer l er verdien $h(l)$). Skriv en Python-funksjon `coscoefficients` som tar inn en slik funksjon *i form av en vektor*, samt et heltall n, og beregner de n første Fourier-cosinus-koeffisientene til funksjonen ved hjelp av formlene fra oppgave 1. `coscoefficients` skal returnere en vektor av lengde n med disse koeffisientene, altså $(a_0, a_1, \ldots, a_{n-1})$.

Oppgave 3 Bruk det du kan om Fourier-rekker til å lage en Python-funksjon `cosseries` som approksimerer en funksjon basert på output fra `coscoefficients`. Med andre ord: beregn Fourier-cosinus-rekken til en stykkevis konstant funksjon basert på Fourier-cosinus-koeffisientene `coscoefficients` gir. Test ut `cosseries` ved å mate `coscoefficients` med en stykkevis konstant funksjon, og sammenlign denne med hva du får ved å gi output fra `coscoefficients` til `cosseries`.

2.2 Bilder i Python

Vi skal som tidligere nevnt styre unna det datatekniske, så ferdiglaget kode for å lese bilder til matriser i Python er tilgjengelig som `http://www.math.ntnu.no/emner/TMA4135/2013h/%C3%B8vinger/8/stuff/imagestuff.py`. Koden kan kun lese bildefiler i PGM-format (et ukomprimert gråtoneformat), så noen testbilder i dette formatet er klargjort og kan lastes ned:

- Frosken er fremdeles med oss: `http://www.math.ntnu.no/emner/TMA4135/2013h/%C3%B8vinger/8/stuff/frog.pgm`
- En mye mindre utgave av frosken kan være nyttig for testing senere: `http://www.math.ntnu.no/emner/TMA4135/2013h/%C3%B8vinger/8/stuff/frog-tiny.pgm`
- *Lena* er et legendarisk testobjekt innen bildebehandling[3]: `http://www.math.ntnu.no/emner/TMA4135/2013h/%C3%B8vinger/8/stuff/lena.pgm`

[3] De spesielt interesserte kan lese bildets historie på `http://ndevilla.free.fr/lena/` og `http://www.cs.cmu.edu/~chuck/lennapg/`.

- ...og Lena i mindre utgave for testing: `http://www.math.ntnu.no/emner/TMA4135/2013h/%C3%B8vinger/8/stuff/lena-tiny.pgm`

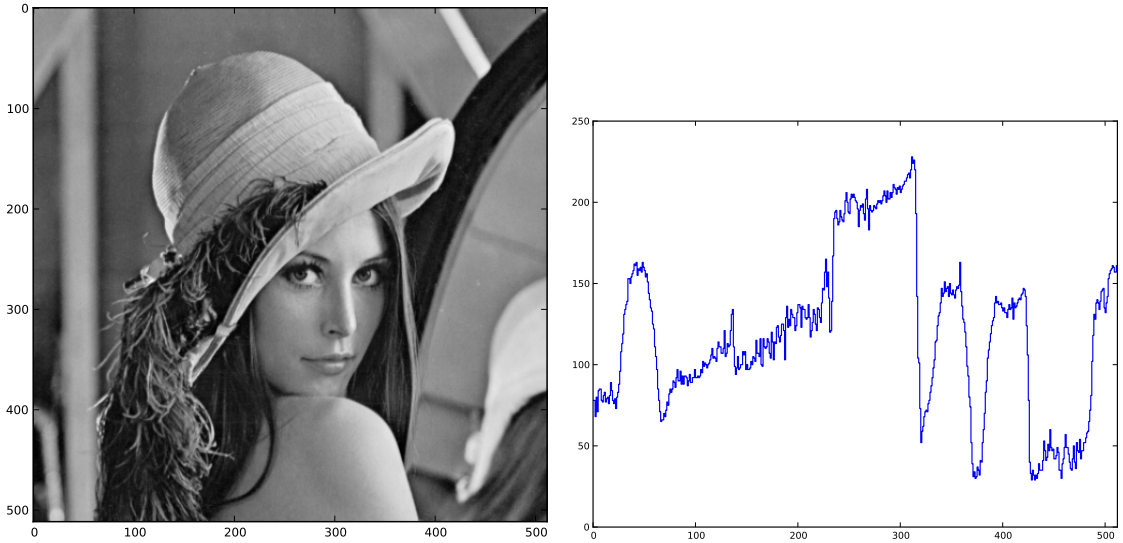
Du kan selvfølgelig også bruke dine egne bilder istedet, men disse må altså konverteres til PGM.[4]

Ved å plassere `imagestuff.py` i katalogen du skriver koden din i, kan modulen `imagestuff` importeres med `import imagestuff`.

Oppgave 4 Test ut `imagestuff` ved å laste inn et av eksemplene og vise bildet:

```
import imagestuff
img = imagestuff.readpgm('lena.pgm')
imagestuff.showimage(img)
```

Eksempelet over viser bildet av Lena, som til venstre i figur 3. Plott også f_{100} (radfunksjon nummer 100) for bildet, som til høyre i figur 3, ved å hente ut rad nummer 100 i matrisen `img`. Legg merke til hvordan f_{100} vokser i verdi omtrent midt i definisjonsmrådet – dette er det hvite i hatten til Lena.



Figur 3: **Venstre:** Eksempelbilde "lena". **Høyre:** Radfunksjonen f_{100} for dette bildet (det er på bildet vanskelig å se at denne er stykkevis konstant, men det er den altså!).

3 Vår komprimering

La B være en $m \times n$-bildematrise. For hver i $(0 \le i < m)$ kjenner vi den stykkevis konstante funksjonen $f_i : [0, n) \to \mathbb{R}$. For en fiksert i kan vi beregne Fourier-cosinus-koeffisientene til f_i; kall dem $a_0^i, a_1^i, \ldots, a_k^i, \ldots$. La oss si vi kun er interessert i $N < n$ av dem. Disse danner en vektor $(a_0^i, a_1^i, \ldots, a_{N-1}^i)$. Hver i gir oss en slik vektor, så la oss sette dem inn i en $m \times N$-matrise

$$A = \begin{pmatrix} a_0^0 & a_1^0 & a_2^0 & \ldots & a_{N-1}^0 \\ a_0^1 & a_1^1 & a_2^1 & \ldots & a_{N-1}^1 \\ \vdots & \vdots & \vdots & & \vdots \\ a_0^{m-1} & a_1^{m-1} & a_2^{m-1} & \ldots & a_{N-1}^{m-1} \end{pmatrix}.$$

Denne matrisen er jo også en "bildematrise" (selv om det langt ifra er klart hva for et bilde den representerer). For hver kolonne j $(0 \le j < N)$ i A kan vi lage "kolonnefunksjoner" $g_j : [0, m) \to \mathbb{R}$ på akkurat samme måte som radfunksjonene:

$$g_j(x) = A_{i,j} = a_j^i \quad \text{hvor } i \text{ er } x \text{ rundet nedover til nærmeste heltall.}$$

[4] GIMP og Imagemagick er eksempler på programmer som kan gjøre dette.

Hver g_j er en stykkevis konstant funksjon, så vi kan gjenta prosessen over! For hver j ($0 \le j < N$) kan vi beregne Fourier-cosinus-koeffisientene til g_j; kall dem $\tilde{a}_0^j, \tilde{a}_1^j, \ldots, \tilde{a}_k^j, \ldots$. La oss si vi kun er interessert i $M < m$ av dem. Disse danner en vektor $(\tilde{a}_0^j, \tilde{a}_1^j, \ldots, \tilde{a}_{M-1}^j)$. Vi setter vektorene inn i en $M \times N$-matrise

$$C = \begin{pmatrix} \tilde{a}_0^0 & \tilde{a}_0^1 & \ldots & \tilde{a}_0^{N-1} \\ \tilde{a}_1^0 & \tilde{a}_1^1 & \ldots & \tilde{a}_1^{N-1} \\ \vdots & \vdots & & \vdots \\ \tilde{a}_{M-1}^0 & \tilde{a}_{M-1}^1 & \ldots & \tilde{a}_{M-1}^{N-1} \end{pmatrix}$$

Vi har redusert $m \times n$-bildematrisen til en ny matrise C av størrelse $M \times N$. Det opprinnelige bildet krevde mn byte lagringsplass, mens informasjonen i matrisen C kun opptar $MN < mn$ byte![5]

Dersom vi kun har lagret C, sammen med dimensjonene m og n til det opprinnelige bildet, kan vi gjenskape en approksimasjon til bildet ved å utføre prosessen i "revers": Ved å beregne Fourier-cosinusrekken til hver kolonne i C får vi en approksimasjon til g_j'ene, som kan skrives som en ny matrise. Beregner vi Fourier-cosinusrekken til hver rad i denne, finner vi en approksimasjon til f_i'ene. Som diskutert tidligere har vi da oppnådd en approksimasjon til B selv.

Oppgave 5 La B være en bildematrise av størrelse $m \times n$. La $0 < r < 1$ være gitt, og sett $M = \lfloor rm \rfloor$ og $N = \lfloor rn \rfloor$. Uttrykt ved r, hvor mye mindre plass tar C-matrisen enn B-matrisen (under antagelse om at hvert element i C også kan lagres med 1 byte)? Størrelsen du kommer frem til kalles *komprimeringsraten* til algoritmen vår.

Oppgave 6 Bruk det vi kom frem til i dette avsnittet til å skrive en Python-funksjon `compress` som tar en bildematrise B, samt heltall M og N, og returnerer den *komprimerte* bildematrisen C, som definert over. Husk at funksjonene f_i og g_j er stykkevis konstante, og at du derfor kan bruke funksjonen `coscoefficients` du skrev i oppgave 2.

Oppgave 7 Skriv også en funksjon `decompress` som tar inn en komprimert bildematrise, og ved å beregne Fourier-cosinusrekken først til kolonnene og deretter radene, gir en approksimasjon til det opprinnelige bildet. Funksjonen må også få vite dimensjonene til det opprinnelige bildet.

Test ut `compress` og `decompress` på bildefilene fra avsnitt 2.2. Det kan lønne seg å bruke de små filene først, da algoritmen vår er særdeles lite effektiv, og kan bruke *lang* tid på store bilder.

De to siste oppgavene er ganske krevende. Dersom du ikke får dem til, ta gjerne en titt på figur 4 og figur 5 for å se resultatet av vår "JPAG"-komprimering. Ved kraftig komprimering (liten r) ser en mye "bølgete" forstyrrelser nær brå fargeoverganger i bildet; dette er den samme *Gibbs-effekten* som vi har sett ved trunkerte Fourier-rekker og Fourier-integral i forelesningene. (Effekten kan viskes litt ut etter printing, så se bildene på skjerm om mulig).

4 Noen kommentarer

Du har sikkert merket at algoritmen vår er *ekstremt* treg, og ikke særlig bra sammenlignet med JPEG.[6]. Hva er forskjellene? Her er noen:

- JPEG bruker diskret Fourier-transformasjon, implementert ved hjelp av lynrask *Fast Fourier Transform* (FFT). De spesielt interesserte som leser Kreyszigs stoff om DFT/FFT vil se at det vi har gjort er en ganske klønete omvei.
- JPEG deler opp bildet i 8×8-blokker og behandler disse hver for seg. Dette er hensiktsmessig.
- Farger er ikke vanskelig: et fargebilde er bare tre gråtonebilder, for eksempel et for rødt, et for grønt, og et for blått. JPEG kan komprimere disse fargekomponentene i forskjellig grad for å dra fordel av måten synet vårt fungerer.

[5] Vel…det er en sannhet med modifikasjoner. Den opprinnelige bildematrisen inneholdt *mn heltall mellom* 0 *og* 255 (som hver opptar en byte), mens den nye inneholder *MN flyttall*. Disse lar seg imidlertid approksimere greit til 1 byte store heltall. Vi vil ikke beskjeftige oss med prosessen for dette her, men tenker du deg om finner du nok ut hva som må gjøres.

[6] Men så ble ikke JPEG utformet iløpet av en øving i et grunnleggende matematikk-kurs, heller…

Figur 4: **Øverst:** Original. **Deretter:** $r = 1/2$ og $r = 1/4$.

Figur 5: **Øverst:** Original. **Deretter:** $r = 1/2$ og $r = 1/4$.

- Viktigst av alt: Vi tar med alle Fourier-koeffisienter opp til en bestemt en, og deretter kaster vi alle de gjenværende. JPEG definerer en glidende overgang; de første tas med fullstendig, og deretter legges avtagende vekt på koeffisientene. Dette sørger for at feilene som oppstår i bildet på grunn av komprimeringen er mindre synlig for det menneskelige øye.