NTNU
Institutt for matematiske fag

TMA4135 Matematikk 4D, programmeringsoppgave til øving 8, høst 2014

Denne programmeringsøvingen er ment til å illustrere én av de svært mange anvendelsene av Fourieranaylse, nemlig *bildekomprimering*. Der vil gis mye informasjon, men oppgavene er i seg selv ikke lange. Øvingen vil forhåpentligvis være nyttig også om du ikke får til programmeringsoppgavene (noe som er helt greit)!

Vi 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 i 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». Dette er kortere og lettere å huske på, og du kjenner til reglene som må til for fullstendig å gjenopprette den opprinnelige strengen fra det du må huske.

Dette er et eksempel på *tapsfri* («lossless») komprimering, 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 det er 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 (for lyd) og JPEG (for fotografier). 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 (tapsfri komprimering) er bildet fullstendig 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.

Figur 1: Rødøyet trefrosk *(Agalychnis callidryas)*. Originalbildet, vist 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. Offentlighetens eiendom.

1 Vårt mål: en enkel bildekomprimeringsalgoritme

Nøkkelen til god komprimering med tap for bilder kommer frem over: Hvis vi må kaste bort informasjon, la oss velge å kaste bort informasjon som mennesker har vanskelig for å legge merke til uansett! Her kommer Fourier-analyse inn.

Som vi har sett i kurset, er den Fourier-transformerte til en funksjon en alternativ representasjon ved hjelp av trigonometriske funksjoner, eller, som man ofte sier, en *frekvensrepresentasjon*. Det viser seg at når mennesker hører på lyder eller ser på bilder, er enkelte frekvenser viktigere enn andre (merk: for bilder er det her snakk om *romlige frekvenser*, mens det for lyd er snakk om frekvenser til oscillasjoner i *tid*). Ved å forflytte oss til en frekvensrepresentasjon av bildet eller lyden vi skal komprimere, kan vi velge å overse «mindre viktige» frekvenser for å oppnå besparelse. Å bestemme hva som er «mindre viktige viktige frekvenser» er svært vanskelig, og et helt forskningsområde i seg selv. Et sted å begynne er derimot det velkjente faktum at *lave frekvenser er viktigere enn høye* for menneskers oppfattelse. I frekvensrepresentasjonen blir det da klart hvilke data vi skal beholde, og hvilke vi skal kaste (behold de lave frekvenskomponentene, kast de høye).

2 Digital representasjon av bilder

For å forenkle det datatekniske, ser vi på såkalte *gråtonebilder*[1] (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.

En slik *bildematrise* kan ses på som en funksjon $\mathbb{R}^2 \to \mathbb{R}$ som er samplet i m punkter i én dimensjon og n punkter i den andre dimensjonen. I Kreyszigs 11.9 lærer vi om diskret Fourier-transformasjon (heretter DFT) for samplede funksjoner $\mathbb{R} \to \mathbb{R}$. Dette skal vi utvide til å behandle bildematriser.

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

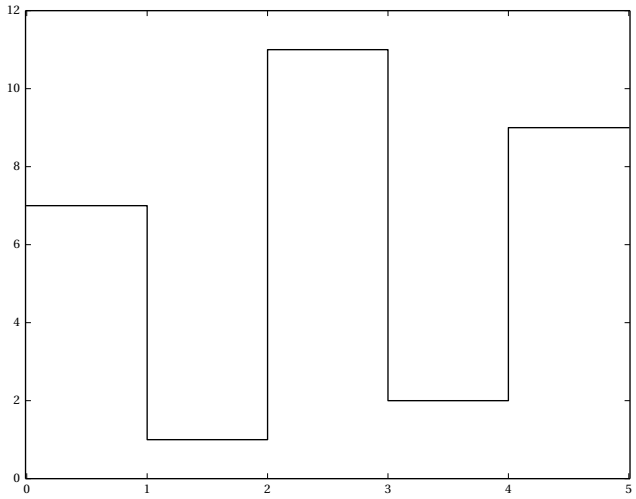
$$R^i(x) = B_{i,\lfloor x \rfloor} \quad \text{hvor } \lfloor x \rfloor \text{ betyr } x \text{ rundet nedover til nærmeste heltall.}$$

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 R^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!

[1] Veien til fargebilder er imidlertid kort: Et fargebilde kan for eksempel representeres som tre gråtonebilder – ett for de røde fargekomponentene, ett for de grønne, og ett for de blå.



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

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

2.1 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/TMA4135/2014h/ov/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/TMA4135/2014h/ov/8/stuff/frog.pgm`
- *Lena* er blitt et standard testobjekt i bildebehandling[2]: `http://www.math.ntnu.no/emner/TMA4135/2014h/ov/8/stuff/lena.pgm`

Du kan selvfølgelig også bruke dine egne bilder istedet, men disse må altså konverteres til PGM.[3]

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

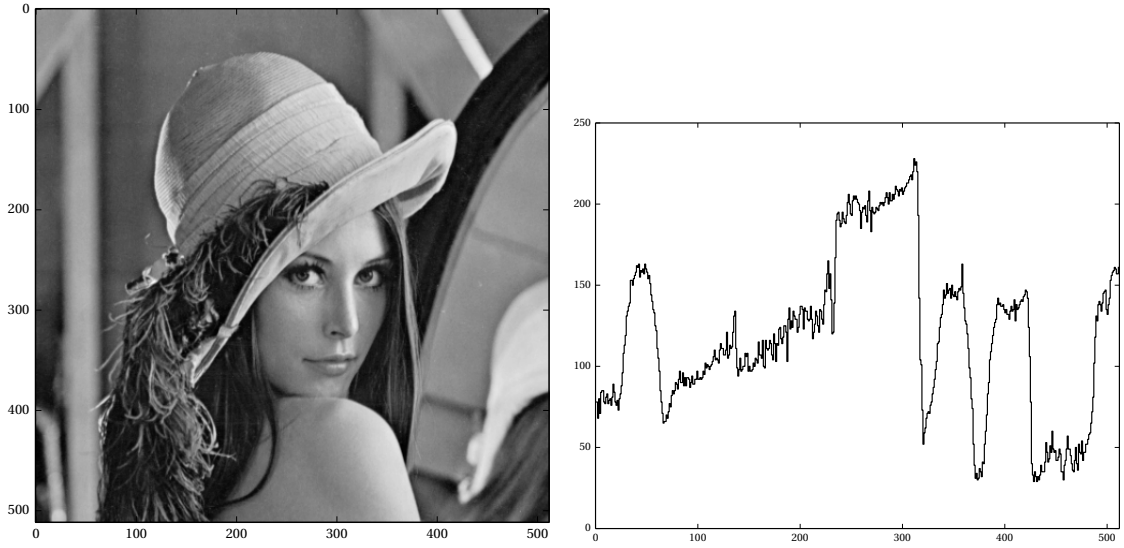
Oppgave 1 Test ut `imagestuff` ved å laste inn et av eksemplene og vise bildet:

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

Eksempelet over viser bildet av Lena, som til venstre i figur 3. Plott også R^{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 R^{100} vokser i verdi omtrent midt i definisjonsmrådet – dette er det hvite i hatten til Lena.

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

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



Figur 3: **Venstre:** Eksempelbilde «lena» slik det vises ved hjelp av `imagestuff.py`. **Høyre:** Radfunksjonen R^{100} for dette bildet.

3 Todimensjonal DFT

I Kreyszigs 11.9 lærer vi at hvis en funksjon f samples i de ekvidistante punktene $x_0, x_1, \ldots, x_{N-1}$ over en lengde 2π, så er

$$f(x_k) = \sum_{n=0}^{N-1} c_n e^{inx_k},$$

hvor

$$c_n = \frac{1}{N} \sum_{k=0}^{N-1} f(x_k) e^{-inx_k}.$$

Vektoren $(c_0, \ldots, c_{N-1})$ kalles kalles den *diskrete Fourier-transformasjon (DFT)* til dataene $f(x_0), \ldots, f(x_{N-1})$. I Kreyszig er f reell-valuert, men DFT fungerer like fint for kompleksvaluerte funksjoner.

I vårt tilfelle passer det bedre å se på funksjonsverdiene som samplet i punktene $0,1,\ldots,N-1$. Da kan vi tenke på f som en vektor av lengde N, og DFT tar formen

$$c_n = \frac{1}{N} \sum_{k=0}^{N-1} f_k e^{-2\pi i \frac{nk}{N}}.$$

for $0 \le n < N$. Inversen er gitt av

$$f_k = \sum_{n=0}^{N-1} c_n e^{2\pi i \frac{nk}{N}}$$

for $0 \le k < N$.

La nå B være en $m \times n$ matrise (et bilde, om du vil). Dersom vi skriver

$$(c_0^{(i)}, c_1^{(i)}, \ldots, c_{n-1}^{(i)})$$

for den diskrete Fourier-transformasjonen til radfunksjon nummer i for matrisen B, kan vi danne oss en ny (kompleks, siden $c_j^{(i)}$'ene generelt er komplekse) $m \times n$-matrise

$$A = \begin{pmatrix} c_0^{(0)} & c_1^{(0)} & \cdots & c_{n-1}^{(0)} \\ c_0^{(1)} & c_1^{(1)} & \cdots & c_{n-1}^{(1)} \\ \vdots & \vdots & & \vdots \\ c_0^{(m-1)} & c_1^{(m-1)} & \cdots & c_{n-1}^{(m-1)} \end{pmatrix}.$$

Hver *kolonne* i A kan nå gjennomgå en DFT. Da ender vi opp med nok en (kompleks) $m \times n$-matrise

$$C = \begin{pmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,n-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,n-1} \\ \vdots & \vdots & & \vdots \\ c_{m-1,0} & c_{m-1,1} & \cdots & c_{m-1,n-1} \end{pmatrix}.$$

Denne kalles den *todimensjonalle diskrete Fourier-transformasjonen* til B. Vi kommer til å skrive $C = \mathrm{DFT}(B)$.

Oppgave 2 Vis at

$$c_{p,q} = \frac{1}{mn} \sum_{j=0}^{m-1} \sum_{k=0}^{n-1} B_{j,k} e^{-2\pi i \left(\frac{pj}{m} + \frac{qk}{n} \right)},$$

hvor $B_{j,k}$ er elementet på rad j og kolonne k i den opprinnelige matrisen B.

3.1 DFT i Python

NumPy har innebygde funksjoner for å gjøre DFT i vilkårlige dimensjoner. Hvis NumPy er importert med `import numpy as np` er `np.fft.fft2(B)` den todimensjonale DFT'en til en matrise `B`. Funksjonsnavnet FFT kommer av at NumPy bruker Fast Fourier Transform-algoritmer for å beregne DFT.

Det viser seg at dersom B er reell, har $\mathrm{DFT}(B)$ visse symmetrier; omtrent halvparten av elementene er komplekskonjugerte av andre elementer.[4] Siden vi skal kaste bort en andel av matrisen $\mathrm{DFT}(B)$, er det viktig at vi ikke gjør det på en måte som ødelegger denne essensielle symmetrien. NumPy har en funksjon `np.fft.fftshift` som stokker om på en matrise slik at vi alltid kan velge ut en *undermatrise sentrert om sentrum* uten å bryte med disse symmetriene. En konsekvens av dette, er at de laveste frekvensene havner i sentrum, med økende frekvenser jo lengre vi beveger oss fra sentrum.

Skal vi invers-transformere, må vi først reversere omstokkingen. Følgende funksjoner kan derfor brukes for å gjøre 2D DFT og invers-DFT med riktig omstokking:

```
import numpy as np

def dft(b):
    return np.fft.fftshift(np.fft.fft2(b))

def invdft(c):
    return np.fft.ifft2(np.fft.ifftshift(c))
```

Oppsummert: Hvis `B` er en bildematrise, er `dft(B)` en matrise hvor elementene nærmest midten representerer de laveste frekvensene i `B`. Frekvensene øker jo lengre vi kommer fra sentrum.

Oppgave 3 I Python/NumPy kan vi hente ut en undermatrise ved hjelp av såkalt *slicing*. Hvis B er en $m \times n$-matrise, kan en $k \times l$-matrise bestående av rad i til og med rad $k-1$ og kolonne j til og med kolonne $l-1$ fra B lages med `B[i:k, j:l]`. For eksempel:

```
B = np.array([[1,2,3],
              [4,5,6],
              [7,8,9]])
print "B:"
print B
print "Undermatrise:"
print B[1:3, 1:3]
```

gir output

```
 B:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Undermatrise:
[[5 6]
 [8 9]]
```

[4] Vis gjerne dette. Det er rotete, men ikke vanskelig.

Skriv en funksjon `submatrix` som tar inn en matrise X og to tall i og j, og returnerer undermatrisen til X som består av i rader og j kolonner fra X, *sentrert midt i* X. Hvordan du velger å runde av hvis i eller j er odde er uviktig, men matrisen du returnerer må være sentrert i midten av X.

Dersom

$$X = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 5 & 5 & 1 & 1 \\ 1 & 1 & 5 & 5 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

skal `submatrix(X, 2, 2)` returnere

$$\begin{pmatrix} 5 & 5 \\ 5 & 5 \end{pmatrix}.$$

4 En enkel bildekomprimeringsalgoritme

Oppgave 4 (krevende) Skriv en Python-funksjon `compress`, som tar inn et bilde (en $m \times n$-matrise) B og et reelt tall $0 < r \leq 1$. Funksjonen skal beregne DFT(B) (se avsnitt 3.1 for hjelp) og så returnere $\lfloor rm \rfloor \times \lfloor rn \rfloor$-matrisen sentrert i midten av DFT(B). (Husk at $\lfloor rm \rfloor$ betyr «rm rundet ned til nærmeste heltall». Du kan selvfølgelig runde opp istedet.)

Oppgave 5 Anta[5] at et kompleks element i DFT(B) opptar dobbelt så mye plass (reell del pluss imaginær del) som et heltallselement i B. Vis at hvis B opptar S byte, så opptar matrisen returnert av `compress(B, r)` omtrent $2r^2S$ byte.

Oppgave 5 viser for eksempel at `compress(B, 0.25)` kan komprimere et bilde B til å oppta bare

$$2 \cdot 0{,}25^2 \approx 12{,}5\%$$

av opprinnelig lagringsplass. Komprimering er dog lite verd uten dekomprimering...

Vi ser av definisjonene i avsnitt 3 at `dft` og `invdft` fra avsnitt 3.1 tar inn og gir ut matriser av samme størrelser. Når vi nå skal gå fra et komprimert bilde (en $\lfloor rm \rfloor \times \lfloor rn \rfloor$-matrise slik returnert av `compress`), må vi vite noe om størrelsen til bildet vi skal gjenskape, og fylle inn 0 for de ukjente frekvenskomponentene.

Oppgave 6 (krevende) Skriv nå en funksjon `decompress` som tar inn et komprimert bilde C (en matrise som returnert av `compress`), samt det opprinnelige bildets størrelse (m og n). Funksjonen `decompress` skal opprette en $m \times n$-matrise som har matrisen C i sentrum, og 0 overalt ellers[6]. Denne matrisen gis så til `invdft`, som returnerer en matrise med samme dimensjoner som det opprinnelige bildet vårt. Funksjonen `decompress` må returnere *realdelen*[7] av denne matrisen, da komprimeringen vår har ført til at en (bitteliten) imaginærdel har oppstått. Denne vil forpurre fremvisning av bildet.

Oppgave 7 (krevende) Prøv ut komprimeringsalgoritmen vår på eksempelbildene (eller andre bilder) ved først å bruke `compress`, og deretter `decompress`. Husk at bilder kan leses inn og vises ved hjelp av funksjonene fra `imagestuff.py` som ble lenket til tidligere.

5 Demonstrasjon

Siden noen av nøkkeloppgavene er ganske krevende, følger her en demonstrasjon av oppgave 7, slik at man kan få demonstrert effekten av komprimeringsalgoritmen vår uten å ha fått til alle oppgavene.

Figur 4 og **??** viser effekten av `compress` etterfulgt av `decompress` for forskjellige r.

[5] Dette er ikke umiddelbart sant, men kan ordnes ved hjelp av *kvantisering*, som vi ikke vil gå inn på her.

[6] NumPys funksjon `zeros` kan være til hjelp: `http://docs.scipy.org/doc/numpy/reference/generated/numpy.zeros.html`

[7] `X.real` er realdelen til en matrise X i NumPy.

Figur 4: **Øverst:** Original. **Deretter:** $r = 1/2$ (50% plass) og $r = 1/4$ (12,5% plass).

Figur 5: **Øverst:** Original. **Deretter:** $r = 1/2$ (50% plass) og $r = 1/4$ (12,5% plass).

Kommentarer

Avsnitt 5 viser at komprimeringsalgoritmen vår fungerer, men at den ikke akkurat er storartet. (Likevel, hva kan man forvente seg av bildekomprimering utviklet iløpet av en øving i et grunnleggende matematikk-kurs?) Hva skiller algoritmen vår fra for eksempel JPEG, som vi vet fungerer strålende i praksis?

- JPEG sparer ytterligere plass ved bare å bry seg om Fourier-cosinus-koeffisienter. Den benytter seg av en såkalt diskret Fourier-cosinus-transformasjon (analogt med Fourier-cosinus-integralet i det kontinuerlige tilfellet).
- Mens vi Fourier-transformerte hele bildet, deler JPEG det først opp i 8×8-*blokker*, og transformerer hver av disse for seg. Dette er svært hensiktsmessig.
- Viktigst av alt: Vi tar med de laveste frekvensene, opp til en viss grense, og deretter sier vi stopp. JPEG tar med store *deler av* de lave frekvensene, og mindre *deler av* de høyere. Denne glidende overgangen har svært stor effekt i praksis, og gir mye bedre resultat enn vår brå stans.
- Farger er en (overraskende?) liten detalj. JPEG behandler hver fargekomponent (rød, grønn, blå) for seg selv, akkurat som vi kunne gjort om vi tok oss tid til det.