

Poisson Image Editing

Marius Lillevik

27. april 2018

Sammendrag

Bildebehandling er en stor del av hverdagen, alt fra reklame til personlige Intagrambilder, de fleste er redigert på en eller annen måte. Det finnes og mange bildebehandlingsprogrammer, fra simple gratisprogrammer til kraftige proffprogrammer til flere tusen kroner. Men billig eller dyrt så baserer alle disse programmene på de samme grunnprinsippene og en av disse er «Poisson Image Editing».

1 Innledning

Innen bildebehandling har man en teknikk kalt «Poisson Image Editing» som man kan bruke til å manipulere bilder. Det er mulig å forvrenge bildet og til og med lime inn fra et annet bilde relativt sømløst. Denne rapporten tar for seg 4 teknikker innen «Poisson Image Editing»; «glatting», «inpainting», «kontrastforsterkning» og «demosaicing» og implementerer dem i Python.

2 Metode

2.1 Matten

Med «Poisson Image Editing» fremstiller man bildet som en funksjon $u : \Omega \rightarrow C$ der $\Omega \subset R^2$ er området hvor bildet er definert, og C er fargerommet, med $C = [0, 1]$ for gråtonebilder og $C = [0, 1]^3$ for fargebilder. Bildet fremkommer da som en løsning av Poisson-ligningen (1) der randverdier på $\partial\Omega$ og funksjonen $h : \Omega \rightarrow R^{dim(C)}$ spesifiseres avhengig av hva som skal gjøres.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \equiv \nabla^2 u = h \quad (1)$$

Man kan løse Poisson-ligningen vha. gradientnedstigning ved å innføre en kunstig tidsparameter og la løsningen utvikle seg mot konvergens:

$$\frac{\partial u}{\partial t} = \nabla^2 u - h. \quad (2)$$

Her må vi også velge en initialverdi for bilde, $u(x, y, 0) = u_0(x, y, 0)$. Ut fra dette får vi da det eksplisitte skjemaet (3).

$$\frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} = \frac{1}{\Delta x^2}(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n) - h_{i,j} \quad (3)$$

Hvis vi da løser (3) for u^{n+1} får vi Laplace-ligningen (6).

$$u_{i,j}^{n+1} = \Delta t \frac{1}{\Delta x^2}(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n) - h_{i,j} + u_{i,j}^n \quad (4)$$

$$\alpha = \Delta t \frac{1}{\Delta x^2} \quad (5)$$

$$u_{i,j}^{n+1} = u_{i,j}^n + \alpha(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n) - h_{i,j} \quad (6)$$

2.2 Koden

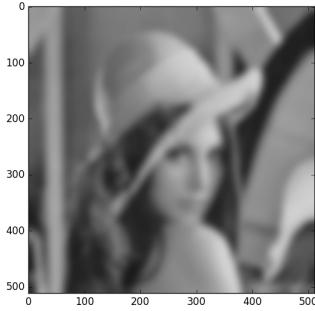
Git repo: https://bitbucket.org/marlil/imt3881_prosjekt.git
Laplace (6) implementert i python.

```
def explicitLaplace(img):    #Laplace transformation
    return (img[:-2, 1:-1] +
            img[2:, 1:-1] +
            img[1:-1, :-2] +
            img[1:-1, 2:] -
            4*img[1:-1, 1:-1])
```

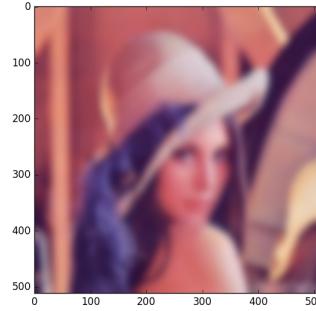
3 Resultater

3.1 Blur

Ved å ta utgangspunkt i originalbildet $u_0(x, y)$ kan man utføre «glatting» ved å iterere ligning (2) med $h = 0$ i hele Ω . Bildet vil da bli glattere med tiden t . Til randverdier er Neumann bukt, med $\partial u / \partial n = 0$ på $\partial\Omega$.



Figur 1: Greyscale blur



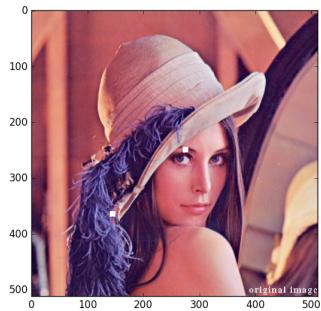
Figur 2: Colour blur

3.2 Inpainting

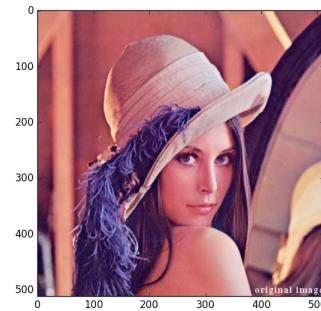
Om et bilde $u_0(x, y)$ mangler informasjon, eller vi ønsker å fjerne noe fra det, kan dette gjøres ved hjelp av «inpaint». Hvis $\Omega_i \subset \Omega$ er området som skal inpaintes, kan dette gjøres ved å sette $h = 0$ i Ω_i og løse ligning (2) i Ω_i med Dirichlet-betingelsen $u(x, y, t) = u_0(x, y)$ på $\partial\Omega_i$

I implementasjonen er det innført en maske i form av et boolsk array hvor alle pixler innenfor Ω_i er sanne og alle utenfor usanne. Denne masken kan da benyttes som et «view» av bildet og vi kan gjøre opperasjoner kun på pixlene innenfor eller utenfor dette.

Her har original bildet fått to små hull i seg, ved å kjøre inpaint på disse områdene får vi fylt igjen hullene. Fra en avstand ser man nesten ikke at bildet har vært skadet.

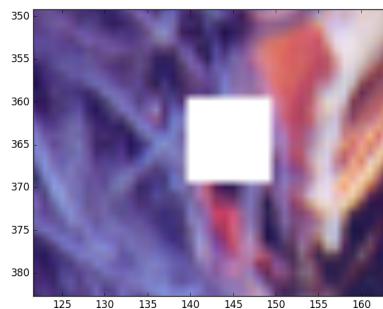


Figur 3: Broken image

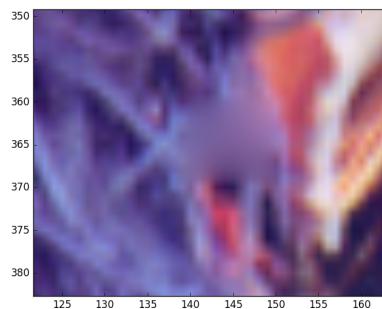


Figur 4: Fixed image

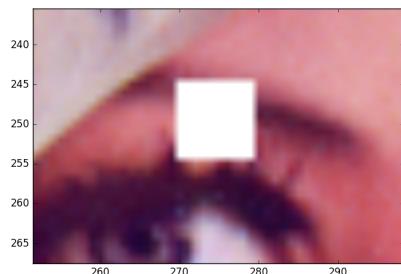
Om man zoomer inn på områdene kan man tydeligere se at noe er gjort.



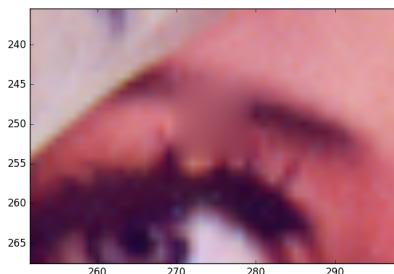
Figur 5: Broken area 1



Figur 6: Fixed area 1



Figur 7: Broken area 2



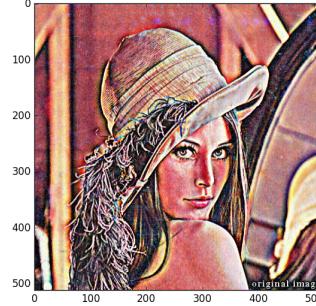
Figur 8: Fixed area 2

3.3 Kontrastforsterkning

Økt kontrast i et bilde u , betyr økt gradient ∇u . Vi kan da, for å finne en mer kontrastert utgave av originalbilde u_0 , finne et bilde med samme gradient som u_0 men forsterket med en konstant $k > 1$. Dette kan gjøres ved å sette $h = k\nabla^2 u_0$ inn i 2 og løse for u . Til randverdier brukes Neumann med $\partial u / \partial n = k\partial u_0 / \partial n$ på $\partial\Omega$. Iterering av (2) med $k > 1$ kan føre til løsninger med $u > 1$ eller $u < 1$ noe som er utenfor det tilgjengelige fargeområdet. Vi innfører derfor en føring at $u \in [0, 1]$ for å unngå dette.



Figur 9: Greyscale contrast



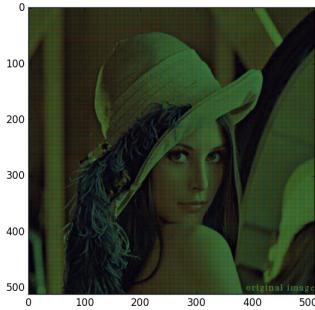
Figur 10: Colour contrast

3.4 Demosaicing

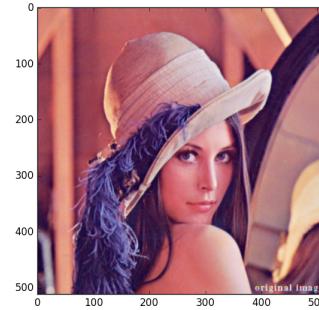
Bildesensoren i et digitalkamera er egentlig monokrom og kan bare måle hvor mye lys som treffer den, det brukes derfor en mosaikk av fargefiltere for å måle mengden av hver farge som treffer sensoren. Ut av sensoren får man da en gråtonemosaiikk. Det er mulig å simulere en slik gråtonemosaiikk i python ved å ta et fargebilde u representert ved en $M \times N \times 3$ numpy array og sette opp gråtonemosaiikken slik:

```
mosaic = np.zeros(im.shape[:2]) # Allocer plass
mosaic[ ::2, ::2] = im[ ::2, ::2, 0] # R-kanal
mosaic[1::2, ::2] = im[1::2, ::2, 1] # G-kanal
mosaic[ ::2, 1::2] = im[ ::2, 1::2, 1] # G-kanal
mosaic[1::2, 1::2] = im[1::2, 1::2, 2] # B-kanal
```

For å «demosaic» mosaikken må vi først flytte informasjonen fra mosaikken over i de rette fargekanalene og deretter kjøre «inpaint» på hele bildet for å fylle inn den manglende informasjonen. Her er mosaikken før og etter vi har kjørt «demosaicing» på den.

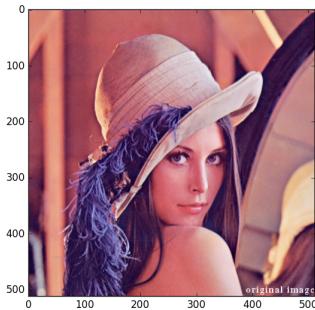


Figur 11: Mosaic

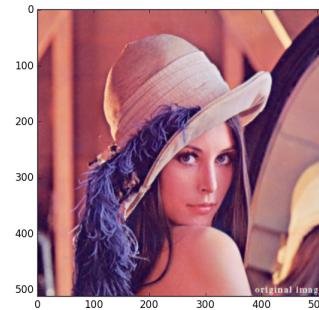


Figur 12: Demosaic

Om vi sammenligner med original bildet, ser vi at de er nesten identiske, men om man ser nøye etter kan man se at originalbildet er litt skarpere.



Figur 13: Original



Figur 14: Demosaic

4 Diskusjon

Når det kommer til koden er det mye som kunne vært forbedret. Flere klasser og funksjoner ville gjort programmet mer fleksibelt og gjenbrukbart. Det kunne også vært implementert flere sjekker og tiltak for generelt bedre og penere kode.

5 Konklusjon

Prosjektet i sin helhet er greit utført, men det er utvidelser og forbedringer som kunne vært gjort for å få et ekstra lag pollish, om man hadde tatt seg tid til det.