

# Poisson Image Editing

## Et innblikk i bildebehandling

Vitenskapelig programmering - IMT3881

Brage Arnkvaern  
kandidatnr : 10015

Sigurd Santiago Schoeler  
kandidatnr : 10034

14. mai 2020

### Sammendrag

Denne rapporten tar for seg implementasjon og fremgangsmåte av metoder fra oppgavesettet [Far20] utgitt av faglærer. Hvor oppgavene var å utforske Poisson likningen knyttet mot bildebehandling.

Mange metoder vil bli sett på, blant annet metoder for glatting, inpainting, sømløs kloning, kontrastforsterkning, anonymisering og konstruksjon av HDR(high dynamic range) bilder. Vi vil i tillegg diskutere resultatene og sammenligne mot andre eksisterende metoder.

Man finner koden på Gitlab<sup>1</sup> sammen med en kanban tavlen som har prøvd å dokumentere utviklingen underveis knytta mot commits<sup>2</sup>, kodens dokumentasjon er lagt ut på Github pages<sup>3</sup> og en interaktiv notebook mot koden finner man også på Gitlab<sup>4</sup>.

<sup>1</sup><https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt>

<sup>2</sup><https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/issues?scope=all&state=all>

<sup>3</sup><https://2xic.github.io/vitprog-prosjekt/>

<sup>4</sup><https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/examples.ipynb>

# Innhold

<b>1 Git</b>	<b>4</b>
<b>2 Innledning</b>	<b>4</b>
<b>3 Kode og kodelstandard</b>	<b>4</b>
3.1 Kode . . . . .	4
3.2 Testing . . . . .	4
3.3 Docstrings . . . . .	5
3.4 requirements.txt . . . . .	5
3.5 GUI . . . . .	5
<b>4 Poisson-ligningen</b>	<b>6</b>
4.1 Overordnet implementasjon . . . . .	6
<b>5 Glatting</b>	<b>8</b>
5.1 Intro . . . . .	8
5.2 Problemstilling . . . . .	8
5.3 Fremgangsmåte . . . . .	8
5.4 Implementering . . . . .	8
5.5 Diskusjon . . . . .	8
5.6 Alternativene . . . . .	9
5.6.1 (Gaussian) filter . . . . .	9
5.7 Resultat . . . . .	10
5.7.1 Gråtone - eksplisitt . . . . .	10
5.7.2 Farger - eksplisitt . . . . .	11
<b>6 Inpainting</b>	<b>12</b>
6.1 Intro . . . . .	12
6.2 Problemstilling . . . . .	12
6.3 Fremgangsmåte . . . . .	12
6.4 Implementering . . . . .	12
6.5 Diskusjon . . . . .	12
6.6 Resultat . . . . .	14
6.6.1 Gråtone - eksplisitt . . . . .	14
6.6.2 Farger - eksplisitt . . . . .	15
6.7 Alternativer . . . . .	16
6.7.1 Median filter . . . . .	16
6.7.2 Autoencoder . . . . .	16
<b>7 Kontrastforsterkning</b>	<b>17</b>
7.1 Intro . . . . .	17
7.2 Problemstilling . . . . .	17
7.3 Fremgangsmåte . . . . .	17
7.4 Resultat . . . . .	18
7.4.1 Gråtone - eksplisitt . . . . .	18
7.4.2 Farger - eksplisitt . . . . .	19
7.5 Diskusjon . . . . .	19

7.6	Alternativene . . . . .	19
7.6.1	Local adaptive histogram (LAH) . . . . .	19
7.7	Resultat . . . . .	21
<b>8</b>	<b>Demosaicing</b>	<b>22</b>
8.1	Intro . . . . .	22
8.2	Problemstilling . . . . .	22
8.3	Fremgangsmåte . . . . .	22
8.4	Implementering . . . . .	22
8.5	Diskusjon . . . . .	22
8.6	Resultat . . . . .	24
8.6.1	Eksplisitt . . . . .	24
<b>9</b>	<b>Somløs kloning</b>	<b>25</b>
9.1	Intro . . . . .	25
9.2	Problemstilling . . . . .	25
9.3	Fremgangsmåte . . . . .	25
9.4	Implementering . . . . .	25
9.5	Resultat . . . . .	26
9.5.1	Gråtone - Eksplisitt . . . . .	26
9.5.2	Farger - Eksplisitt . . . . .	27
9.5.3	Parametere . . . . .	28
9.5.4	Støy . . . . .	29
9.6	Diskusjon . . . . .	30
9.7	Alternativer . . . . .	30
9.7.1	Healing brush . . . . .	30
<b>10</b>	<b>Konvertering av fargebilder til gråtone</b>	<b>31</b>
10.1	Intro . . . . .	31
10.2	Problemstilling . . . . .	31
10.3	Fremgangsmåte . . . . .	31
10.4	Implementering . . . . .	31
10.5	Diskusjon . . . . .	32
10.6	Resultat . . . . .	34
10.6.1	Eksplisitt . . . . .	34
<b>11</b>	<b>Anonymisering</b>	<b>35</b>
11.1	Intro . . . . .	35
11.2	Problemstilling . . . . .	35
11.3	Fremgangsmåte . . . . .	35
11.4	Implementering . . . . .	35
11.5	Diskusjon . . . . .	35
11.6	Resultat . . . . .	36
11.6.1	Gråtona - eksplisitt . . . . .	36
11.6.2	Farger - eksplisitt . . . . .	37
11.6.3	Mange ansikt . . . . .	38

<b>12 Rekonstruksjon og visualisering av HDR-bilder</b>	<b>39</b>
12.1 Intro . . . . .	39
12.2 Problemstilling . . . . .	39
12.3 Fremgangsmåte . . . . .	39
12.4 Implementering . . . . .	40
12.5 Diskusjon . . . . .	41
12.6 Resultat . . . . .	42
12.6.1 Adjuster . . . . .	42
12.6.2 Ocean . . . . .	43
12.6.3 Ocean med ulikt antall bilder . . . . .	44
<b>13 Kantbevarende glatting</b>	<b>45</b>
13.1 Intro . . . . .	45
13.2 Problemstilling . . . . .	45
13.3 Fremgangsmåte . . . . .	45
13.4 Implementering . . . . .	45
13.5 Diskusjon . . . . .	46
13.6 Resultater . . . . .	47
13.6.1 Gråtone - eksplisitt . . . . .	47
13.6.2 Farger - eksplisitt . . . . .	48
13.7 Resultat . . . . .	49
13.7.1 Glatting og kantbevarende glatting - Side om side . . . . .	49
<b>14 Refleksjon</b>	<b>50</b>
<b>15 Konklusjon</b>	<b>50</b>
<b>A Appendix</b>	<b>52</b>
A.1 Kode for å simulere gråtone mosaikk . . . . .	52

# 1 Git

All koden knyttet til prosjektet finnes på <https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt>. Her ligger også koden som har laget (nesten) alle figurene i rapporten i mappen `src/rapport_snippets/*.py`.

## 2 Innledning

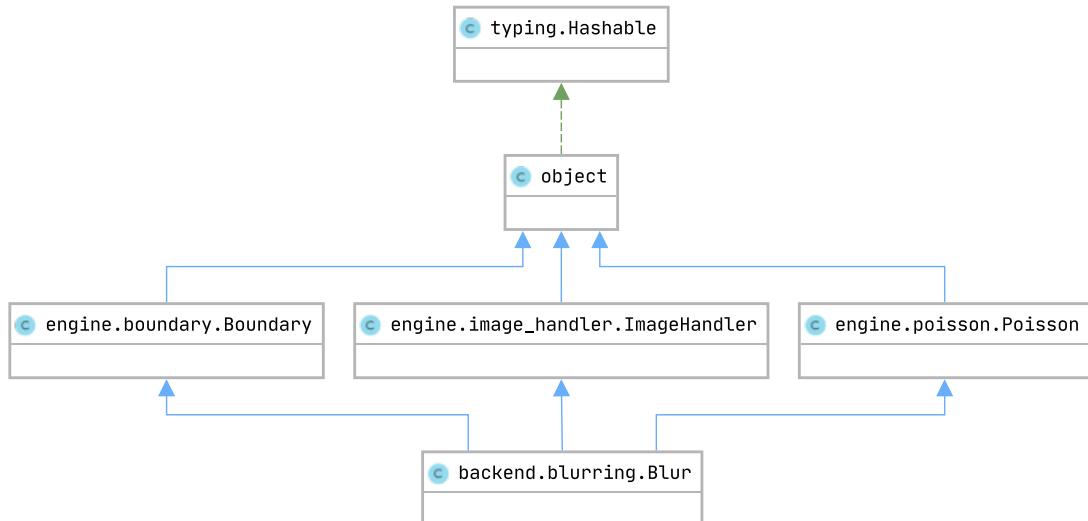
Bildebehandlingen er vi alle kjent med enten det er knytta til å rette opp småfeil i et bilde eller manipulere det med onde eller gode intensjoner. Vi vil i denne rapporten utforske bildebehandling ved å utnytte Poisson-ligningen. Dette er en meget fantastisk ligning som dukker opp overalt i fysikken i ulike former (blant annet i Newton sine lover for gravitasjon [Gow08, s. 483] og Schrödinger ligning [Gow08, s. 286]). Nå skal vi likevel fokusere på den langt mer interessante bruken av Poisson-ligningen nemlig til bildebehandling.

## 3 Kode og kodestandard

### 3.1 Kode

Koden er modulbasert. Dette gjør at alle utvidelsene ofte krever veldig lite kode siden det bygger på arbeid allerede gjort. Dette gjorde også at utvidelsen til et GUI var sømløst og at prosjektet ellers kan fungerer “headless” som et API<sup>5</sup>.

Prosjektet tar også stor inspirasjon fra hvordan bibliotekene scikit-learn<sup>6</sup> og keras<sup>7</sup> navngir sentrale funksjoner og ellers er bygd opp.



Figur 1: Klasse struktur for de fleste metodene (alle utenom HDR)

### 3.2 Testing

Siden Python ikke er “strongly typed” så har det vært et fokus på testing fra start til slutt av prosjektet. Vi benyttet travis-ci<sup>8</sup> for å sikre at alle commits har kode som

<sup>5</sup><https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/examples.ipynb>

<sup>6</sup><https://scikit-learn.org/stable/>

<sup>7</sup><https://keras.io/>

<sup>8</sup><https://travis-ci.com/2xic/vitprog-prosjekt/>

fungerer. Testene ser for det meste etter type feil og at ulike “code paths” fungerer.

En fremtidig forbedring er å utvide testene til å teste de numeriske beregningene også.

Det har i tillegg vært brukt “type hints”<sup>9</sup> som både hjelper for dokumentasjonen, men hjelper også å finne feil<sup>10</sup>. Vi prøvde likevel å benytte disse verktøyene på en slik måte at Python beholdt sin flyt som et skriptspråk.

### 3.3 Docstrings

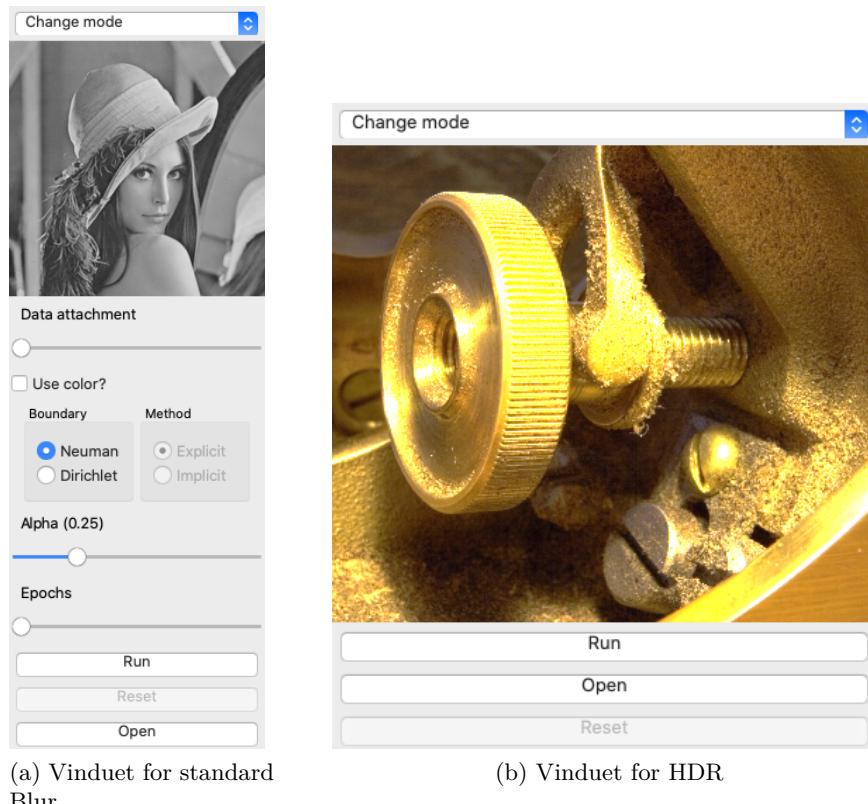
All koden har docstrings og prosjektet har også en medfølgende HTML dokumentasjon side<sup>11</sup> bygd opp av docstrings.

### 3.4 requirements.txt

For å både sikre at CI og sluttbruker er fornøyd er alle eksterne pakker listet i `requirements.txt`. Man kan enkelt installeres med pip, `pip install -r requirements.txt`.

### 3.5 GUI

Fra Figur. 2 kan man se deler av vårt GUI. Det er enkelt i sin form, men lar en styre de sentrale parameterne. Brukeren kan også åpne egne bildet ved å benytte “open” knappen og nullstille metodens endringer ved “reset”. GUI kan startes med `python main.py`.



Figur 2: Blur og HDR vindu

<sup>9</sup><https://docs.python.org/3/library/typing.html>

<sup>10</sup><http://mypy-lang.org/>

<sup>11</sup><https://2xic.github.io/vitprog-prosjekt/>

## 4 Poisson-ligningen

Vi vil først bruke litt tid til å betrakte Poisson-ligningen. Vi definere bildet som en funksjon  $u : \Omega \rightarrow C$ . Hvor  $C$  er fargerommet vi jobber med  $C = [0, 1]^n$ , hvor  $n$  er antall fargekanaler vi jobber med ( $n = 1$  for gråtone og  $n = 3$  for farge bilde).  $\Omega \subset \mathbb{R}^{\dim(C)}$  er området hvor bildet er definert og  $\partial\Omega$  er randen og spesifiseres avhengig av hvilket problem som løses.

Et bilde på formen  $u(x, y)$  vil gi følgende Poisson-ligningen

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \equiv \nabla^2 u = h,$$

Ligningen kan løses ved å benytte gradientnedstigning

$$\frac{\partial u}{\partial t} = \nabla^2 u - h. \quad (1)$$

Man kan deretter konstruere numeriske skjemaer for Likning. 1. Det eksplisitte skjema er konstruert slik

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

og løses for  $u^{n+1}$

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

Siden alle metodene bygger på dette numeriske skjemaene er det gunstig å implementere dette som en egen klasse med hjelpefunksjoner. På denne måten kan koden også enkelt utvides til støtte for andre skjemaer også (for eksempel implisitt skjema). Dette har vi gjort og vi vil videre diskuterte hvordan vi har implementert dette.

### 4.1 Overordnet implementasjon

Som man kan se fra Kodesnutt. 1 har vi gjort ting ganske abstrakt. Poisson klassen tar ikke hensyn til hva slags metode man jobber med men den vil bare iterere over funksjonene “operator” (som da ofte bare er Laplace) og “h” med den tilgitte alpha( $\frac{\Delta t}{\Delta x^2} = \alpha$ ) verdien. Den vil også automatisk løse for om det er et bilde med flere kanaler. Da vil operator klassen bli kalt med en indeks for hva slags kanal man jobber på. Man kan også påføre “boundary condition” direkte om det er ønskelig (noe det i flere av metodene er).

Koden er også skrevet slik at implementasjon av videre utvidelser med nye skjemaer eller andre “boundary conditions” er enkelt. Vi benytter også Python sin `@property` decorator<sup>12</sup> for å sikre at variabler for Poisson klassen blir satt riktig og at andre metoder sine variabler er sikret mot at brukeren taster feil. På denne måten blir man varslet om feil i sanntid om en variabel endres til feil type og ikke først når når man prøver å kjøre metoden.

---

<sup>12</sup><https://www.python.org/dev/peps/pep-0318/>

<sup>13</sup>Full kode med docstrings og ekstra metoder : <https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/engine/poisson.py>

```

1  class Poisson(object):
2      def get_laplace_explicit(self, data: Array[float, float] =
3          None, alpha:bool = True) -> Array:
4          if data is None:
5              data = self.data
6              laplace = data[0:-2, 1:-1] \
7                  + data[2:, 1:-1] \
8                  + data[1:-1, 0:-2] \
9                  + data[1:-1, 2:] \
10                 - 4 * data[1:-1, 1:-1]
11          if not alpha:
12              return laplace
13          return laplace * self.alpha
14
15      def solve(self, data, operator:callable, h=lambda i=None:
16          0, apply_boundary=True) -> Array:
17          result = data.copy()
18          if(len(data.shape) == 3):
19              for i in range(3):
20                  result[1:-1, 1:-1, i] += self.alpha * \
21                      (operator(i) - h(i=i))
22          else:
23              result[1:-1, 1:-1] += self.alpha * (operator() -
24                  h())
25
26          if apply_boundary:
27              return self.apply_boundary(result).clip(0, 1)
28          else:
29              return result.clip(0, 1)

```

Kodesnutt 1: Et utdrag fra vår Poisson klasse <sup>13</sup>

## 5 Glatting

### 5.1 Intro

Glatting er en av de enkleste formene for bildebehandling. Det kan både gi kule effekter til bilde, men glatting har også egenskapen at det kan anvendes til å løse flere andre problemer for eksempel Inpainting og Anonymisering.

### 5.2 Problemstilling

Vi ønsker å glatte bilde, utføre glatting (“blurring”) ved å benytte Likning. 1.

### 5.3 Fremgangsmåte

Dersom man lar  $h = \lambda(u - u_0)$  i Likning. 1 så vil bildet bli mer og mer uskarpt når man itererer over bildet (ved  $\lambda = 0$ ). Dersom  $\lambda \neq 0$  vil man kunne begrense hvor glatt bilde blir (“data-attachment”).

### 5.4 Implementering

Dette er en enkel metode å implementere om man har et fleksibelt rammeverk for å kalkulere Poisson likningen, koden under er fra vårt rammeverk.

```
class Blur(image_handler.ImageHandler, poisson.Poisson,
    ↵ boundary.Boundary):
    def iteration(self) -> Array:
        self.data = self.solve(self.data, self.operator, self.h)

    def operator(self, i=None) -> Array:
        if i is None:
            return self.get_laplace(self.data, alpha=True)
        else:
            return self.get_laplace(self.data[:, :, i],
                ↵ alpha=True)

    def h(self, i=None) -> Array:
        if i is None:
            return self.common_shape(self.lambda_size *
                ↵ (self.data - self.data_copy))
        else:
            return self.common_shape(self.lambda_size *
                ↵ (self.data[:, :, i] - self.data_copy[:, :, i]))
```

Kodesnutt 2: Vår implementasjon av glatting<sup>14</sup>

### 5.5 Diskusjon

Som man kan se fra Kodesnutt. 2 så er funksjonene enkle og har i hovedoppgave å sende tilbake riktig kanal (avhengig av hvorvidt bilde som brukes er fargebilde eller gråtone). ”h” gir støtte for data-attachment og gir dermed muligheten til å styre hvor glatt bilde blir (Figur. 5). Her vil i tillegg “solve” funksjonen fra Poisson være den som påfører valgt “boundary condition”.

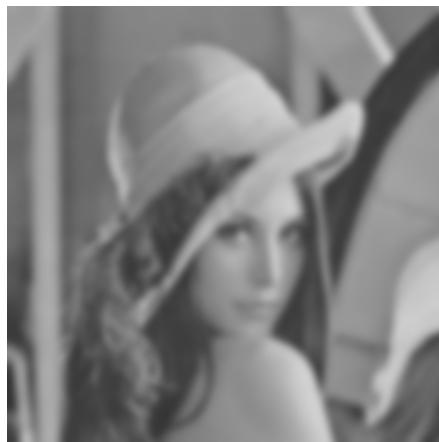
<sup>14</sup>Full kode med docstrings og ekstra metoder : <https://git.gvk.idi.ntnu.no/BrageA/intm3881-2020-prosjekt/-/blob/master/src/backend/blurring.py>

Man kan også se fra Diskusjon at det å styre parameterne ( $\alpha$  og antall iterasjoner) er noe som må balanseres for å få best resultat. Dersom man har for høy  $\alpha$  vil bildet fort bli bare støy, men øker man  $\alpha$  forsiktig kan man spare noen iterasjoner. Dette er et fenomen som vi vil se gjennom hele rapporten.

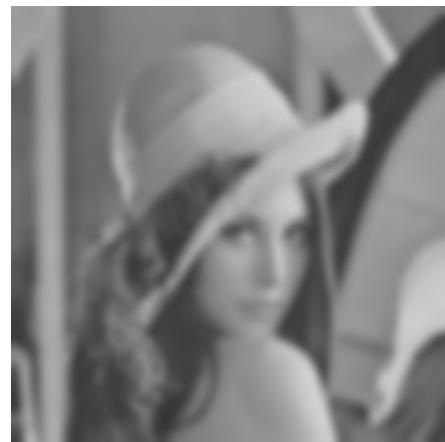
## 5.6 Alternativene

### 5.6.1 (Gaussian) filter

En av de enkleste metodene for å utføre glatting er å benytte filtre. Her vil man påføre et filter over hele bildet og som et resultat vil man få et uskarpt bilde. Det finnes mange filtre å velge blant, men av de mest populære er å benytte et gaussian filter<sup>15</sup>.



(a) Gaussian filter ( $\sigma = 2$ )



(b) Poisson ( $\alpha = 0.5$ , iterasjoner = 10)

Siden denne formen for metode både er rask og enkel å implementere har den sine fordeler over å bruke Poisson. Likevel en naive implementasjon i Python vil ofte være treg (Kodesnutt. 3). Så man bør enten skrive en modul i C/C++ for Python<sup>16</sup> eller benytte scipy<sup>17</sup> som allerede har denne funksjonen implementert<sup>18</sup>.

```
for row in range(image_row):
    for col in range(image_col):
        output[row, col] = np.sum(filter * image[row:row +
→ kernel_row, col:col + kernel_col])
```

Kodesnutt 3: Naive Python implementering av Gaussian filter

<sup>15</sup>[https://en.wikipedia.org/wiki/Gaussian\\_blur](https://en.wikipedia.org/wiki/Gaussian_blur)

<sup>16</sup><https://docs.python.org/3/extending/extending.html>

<sup>17</sup><https://scipy.org/>

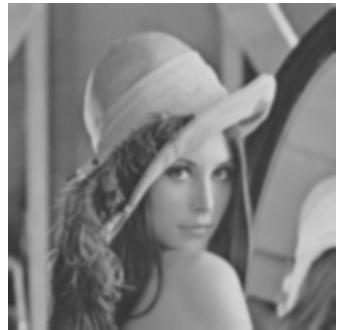
<sup>18</sup>[https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.gaussian\\_filter.html](https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.gaussian_filter.html)

## 5.7 Resultat

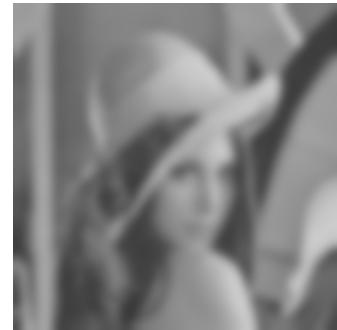
### 5.7.1 Gråtone - eksplisitt



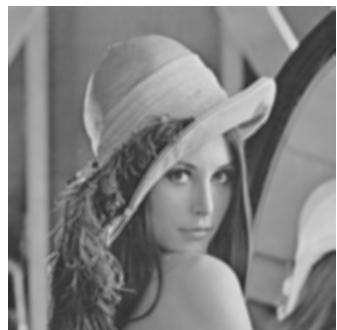
(a)  $\alpha = 0.25$   
iterasjoner = 1



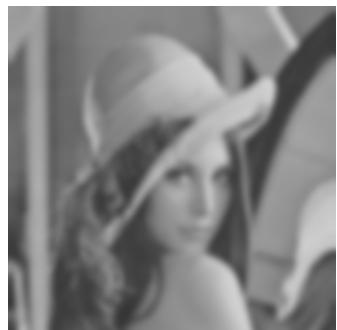
(b)  $\alpha = 0.25$   
iterasjoner = 10



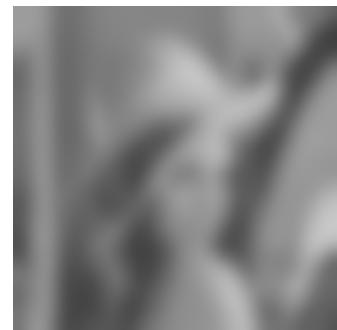
(c)  $\alpha = 0.25$   
iterasjoner = 100



(d)  $\alpha = 0.5$   
iterasjoner = 1



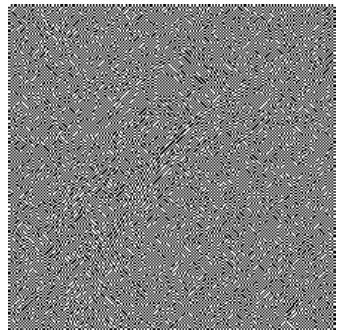
(e)  $\alpha = 0.5$   
iterasjoner = 10



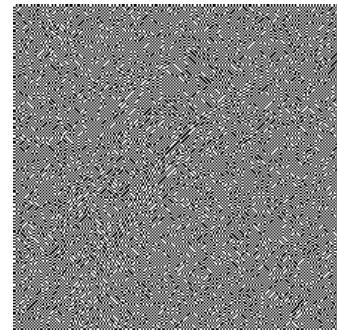
(f)  $\alpha = 0.5$   
iterasjoner = 100



(g)  $\alpha = 0.75$   
iterasjoner = 1



(h)  $\alpha = 0.75$   
iterasjoner = 10



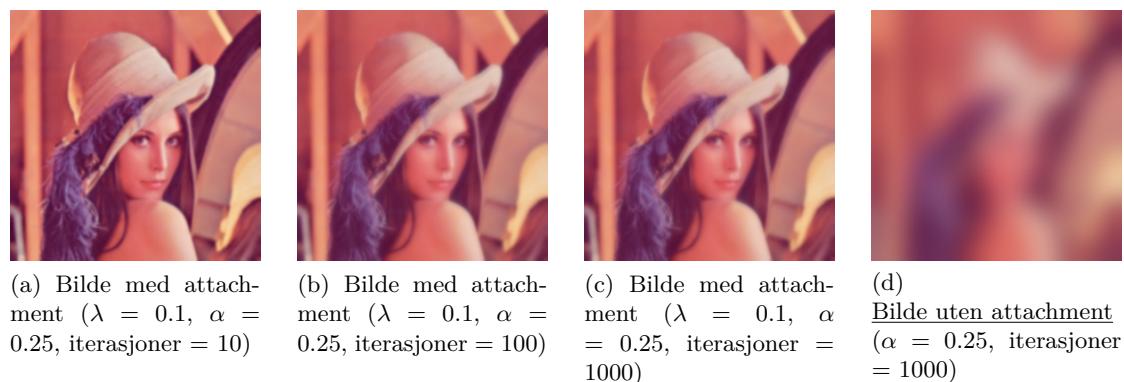
(i)  $\alpha = 0.75$   
iterasjoner = 100

Figur 3: Resultat for gråtonebilder med ulike verdier av  $\alpha$  og iterasjoner

### 5.7.2 Farger - eksplisitt



Figur 4: Resultat for fargebilder med ulike verdier av  $\alpha$  og iterasjoner



Figur 5: Data attachment begrenser glatting

## 6 Inpainting

### 6.1 Intro

Det finnes flere tilfeller hvor man enten ønsker å fjerne informasjon fra et bilde, eller hente frem igjen tapt informasjon. Man kan ha vært uheldig å være utsatt for salt og pepper noise<sup>19</sup> eller bare ha en defekt piksel fra kamera.

### 6.2 Problemstilling

Vi ønsker å fylle inn eller fjerne uønsket deler av et bilde.

### 6.3 Fremgangsmåte

Dersom vi benytter en maske for å definere arbeidsområdet kan vi utnytte glatte funksjonen fra forrige del (Glatting). Man vil i praksis overskrive manglende eller uønsket informasjon med innhold fra området rundt.

### 6.4 Implementering

Denne metoden er også nokså rett frem å implementere.

```
class Inpaint(image_handler.ImageHandler, poisson.Poisson,
→ boundary.Boundary):
    def iteration(self) -> None:
        response = self.solve(self.data, self.operator,
        → apply_boundary=False)
        if(len(self.data.shape) == 3):
            for i in range(self.data.shape[-1]):
                self.data[:, :, i] = (response[:, :, i] * (1 -
                → self.mask)) + (self.original_data_copy[:, :, i] *
                → self.mask))
        else:
            self.data = (response * (1 - self.mask)) +
            → (self.original_data_copy * (self.mask))
        self.data = self.dirichlet(self.data, self.mask)
        return self.data

    def operator(self, i=None):
        if i is None:
            return self.get_laplace(self.data)
        else:
            return self.get_laplace(self.data[:, :, i])
```

Kodesnutt 4: Vår implementasjon av inpainting<sup>20</sup>

### 6.5 Diskusjon

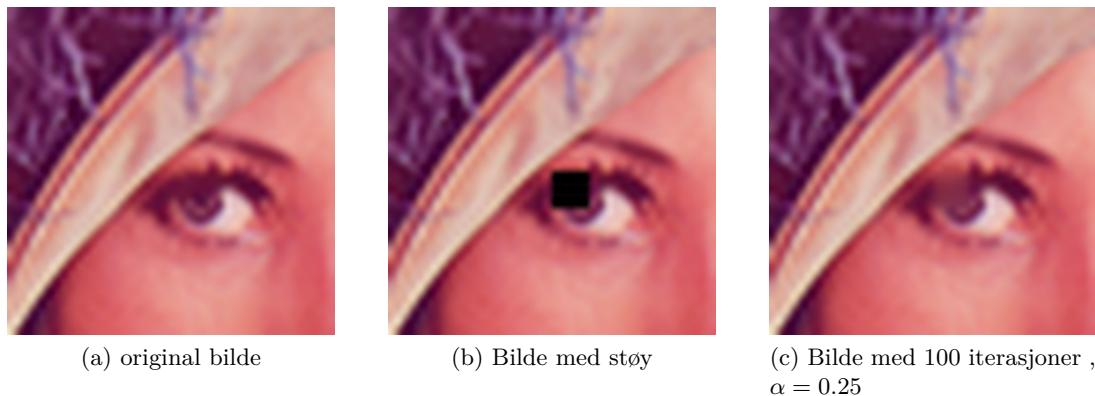
Funksjonene her er faktisk like med Blur (foruten at vi her alltid vil ha  $h = 0$ ) frem til man har løst Poisson likningen. Deretter vil man oppdatere bilde hvor bildet har mistet informasjon/har lyst å fjerne informasjon. Vi benytter her Dirichlet for

<sup>19</sup>[https://en.wikipedia.org/wiki/Salt-and-pepper\\_noise](https://en.wikipedia.org/wiki/Salt-and-pepper_noise)

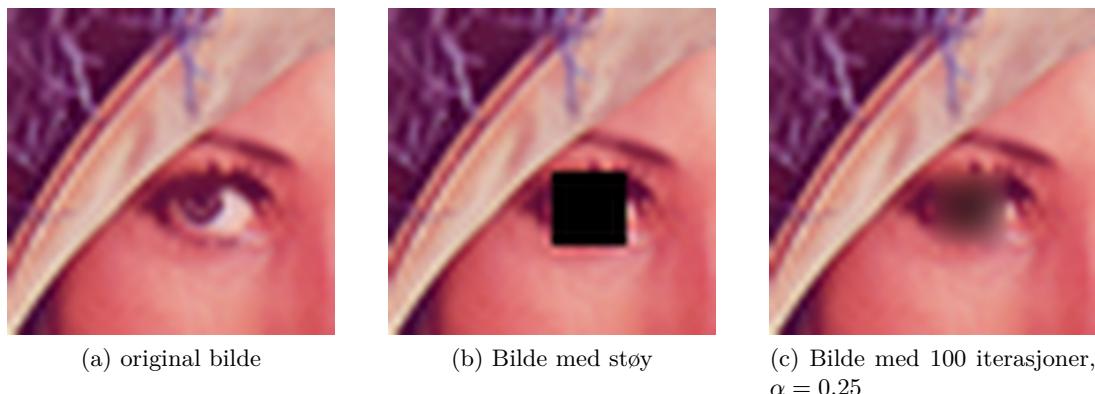
<sup>20</sup>Full kode med docstrings og ekstra metoder : <https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/backend/inpainting.py>

å nullstille bildet hvor informasjon ikke er tapt. Her har vi ikke tatt hensyn til om informasjonen som blir fjernet ligger på randen. Noe som er et forbedringspotensiale.

Dersom man studerer bildene nøyne så vil man se at bildet i praksis bare er smurt utover (Figur. 6, Figur. 7). Likevel som man kan se fra Figur. 9 så er dette noe man ikke nødvendigvis vil oppdage etter nok iterasjoner med gode parametere. Dersom man prøver å inpainte et forsvunnet øye blir ting selvsagt åpenbart Figur. 7. Informasjonen som skal inpaintes må være jevnt fordelt med ikke for store individuelle områder for best resultat.



Figur 6: Inpainting er kontrollert blurring



Figur 7: Inpainting er kontrollert blurring

Det man også ser her er som beskrevet i Diskusjon for Glatting at ulik  $\alpha$  kan spare en for mye tid. Man ser at ved å benytte  $\alpha = 0.5$  og 10 iterasjoner kan man få nesten samme resultat som ved  $\alpha = 0.25$  og 100 iterasjoner. Man ser også her at ved å benytte for høy  $\alpha$  vil ikke løsningen konvergere mot et bra resultat.

## 6.6 Resultat

### 6.6.1 Gråtone - eksplisitt



(a)  $u_0$



(b)  $\alpha = 0.25$   
iterasjoner = 1



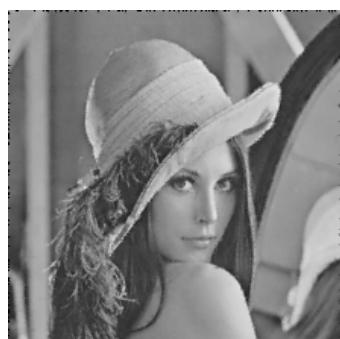
(c)  $\alpha = 0.25$   
iterasjoner = 10



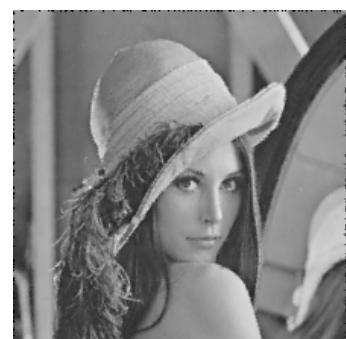
(d)  $\alpha = 0.25$   
iterasjoner = 100



(e)  $\alpha = 0.5$   
iterasjoner = 1



(f)  $\alpha = 0.5$   
iterasjoner = 10



(g)  $\alpha = 0.5$   
iterasjoner = 100



(h)  $\alpha = 0.75$   
iterasjoner = 1



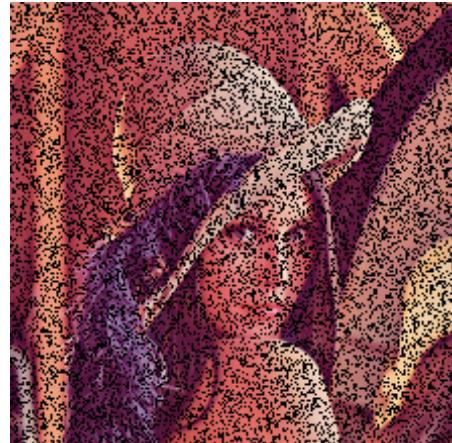
(i)  $\alpha = 0.75$   
iterasjoner = 10



(j)  $\alpha = 0.75$   
iterasjoner = 100

Figur 8: Resultat for gråtonebilder med ulike verdier av  $\alpha$  og iterasjoner

### 6.6.2 Farger - eksplisitt



(a)  $u_0$



(b)  $\alpha = 0.25$   
iterasjoner = 1



(c)  $\alpha = 0.25$   
iterasjoner = 10



(d)  $\alpha = 0.25$   
iterasjoner = 100



(e)  $\alpha = 0.5$   
iterasjoner = 1



(f)  $\alpha = 0.5$   
iterasjoner = 10



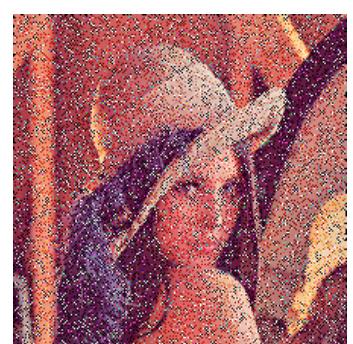
(g)  $\alpha = 0.5$   
iterasjoner = 100



(h)  $\alpha = 0.75$   
iterasjoner = 1



(i)  $\alpha = 0.75$   
iterasjoner = 10



(j)  $\alpha = 0.75$   
iterasjoner = 100

Figur 9: Resultat for fargebilder med ulike verdier av  $\alpha$  og iterasjoner

## 6.7 Alternativer

### 6.7.1 Median filter

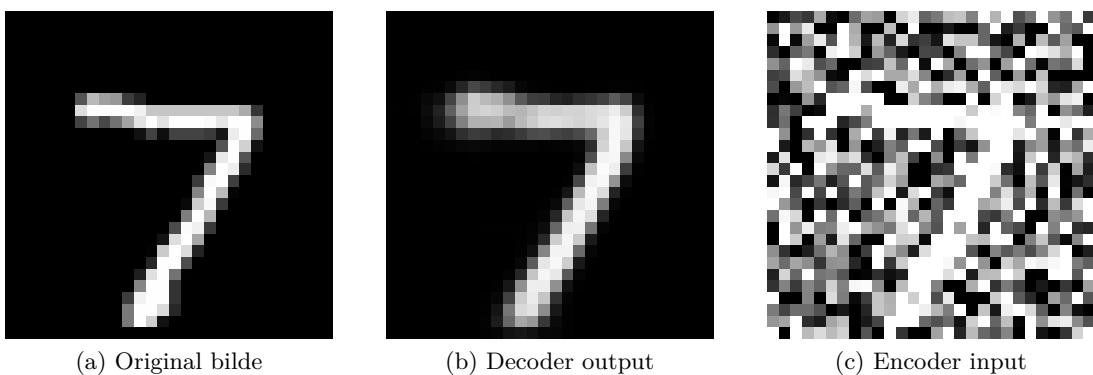
Det er også her muligheter for bruk av filter. Man vil i praksis bruke et median filter<sup>21</sup>. Man benytter da median verdien innenfor et piksel vindu (vi brukte  $3 \times 3$  størrelse) som ny piksel verdi, dette vinduet vil flytte seg over hele bildet og oppdatere en piksel om gangen. Vi benytter også her en maske, slik at vi kun oppdaterer hvor skaden har skjedd.



Man har også her problemet med at en naive filter implementasjon vil være treg. Metoden gir heller ikke et likte godt resultat som Poisson. Likevel man kan se at metoden klarer å gjenskape mye av informasjonen som forsvant.

### 6.7.2 Autoencoder

I en verden preget av maskinlæring og “kunstig intelligens” ville det være feil å ikke nevne bruk av en autoencoder<sup>22</sup>. En autoencoder består av en encoder( $x$ ) og en decoder( $y$ ), oppgavens den er å lære en representasjon  $f : x \mapsto y$ . Man kan dermed be den lære å gå fra et bilde med manglende informasjon til et med informasjonen fylt inn. Trenings-data kan enkelt lages ved å legg på Gaussian noise på hvert bilde. Her har vi gjort dette på MNIST<sup>23</sup>, Figur. 10.



Figur 10: Autoencoder rekonstruksjon av støy

<sup>21</sup>[https://en.wikipedia.org/wiki/Median\\_filter](https://en.wikipedia.org/wiki/Median_filter)

<sup>22</sup><https://en.wikipedia.org/wiki/Autoencoder>

<sup>23</sup><http://yann.lecun.com/exdb/mnist/>

## 7 Kontrastforsterkning

### 7.1 Intro

Det finnes flere tilfeller hvor man gjerne ønsker å få frem kontrasten i bildet, og gjøre bildet skarpere. Eldre bilder sliter med lav kontrast, noe som kan føre til at bildet ikke oppnår den samme sentimentale verdien det kunne hatt. Muligheten til å rette opp i dette kan gi mye glede for sluttbruker.

### 7.2 Problemstilling

Vi ønsker å gjøre kontrasten i bilde skapere.

### 7.3 Fremgangsmåte

Gradienten til et bilde er avhengig av kontrasten i et bilde. Så for å øke kontrasten i et bilde må vi å finne et bilde med samme gradient, men hvor gradienten til bildet er forsterket med en konstant.

Dette kan løses ved å benytte  $h = k\nabla^2 u_0$  inn i Poisson-ligningen.

```
class Contrast(image_handler.ImageHandler, poisson.Poisson,
    ↪ boundary.Boundary):
    def iteration(self) -> Array:
        self.data = self.solve(self.data, self.operator, self.h)

    def operator(self, i=None) -> Array:
        if i is None:
            return self.get_laplace(self.data, alpha=True)
        else:
            return self.get_laplace(self.data[:, :, i],
                ↪ alpha=True)

    def h(self, i=None) -> Array:
        if len(self.h_arr.shape) == 3:
            return self.h_arr[i, :, :]
        else:
            return self.h_arr
```

Kodesnutt 5: Vår implementasjon<sup>24</sup>

Her er

```
h_arr = self.k *
    ↪ (np.asarray([self.get_laplace(np.copy(self.data[:, :, i]))]
    ↪ for i in range(self.data.shape[-1])]))
```

og regnes ut i starter av klassen(ved `__init__`).

<sup>24</sup>Full kode med docstrings og ekstra metoder : <https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/backend/contrasting.py>

## 7.4 Resultat

### 7.4.1 Gråtone - eksplisitt



(a)  $\alpha = 0.25$   
iterasjoner = 1



(b)  $\alpha = 0.25$   
iterasjoner = 10



(c)  $\alpha = 0.25$   
iterasjoner = 100



(d)  $\alpha = 0.5$   
iterasjoner = 1



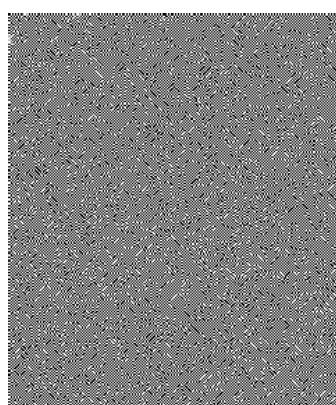
(e)  $\alpha = 0.5$   
iterasjoner = 10



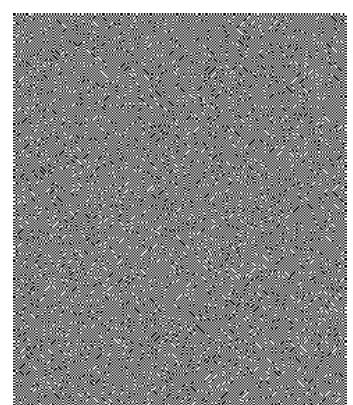
(f)  $\alpha = 0.5$   
iterasjoner = 100



(g)  $\alpha = 0.75$   
iterasjoner = 1



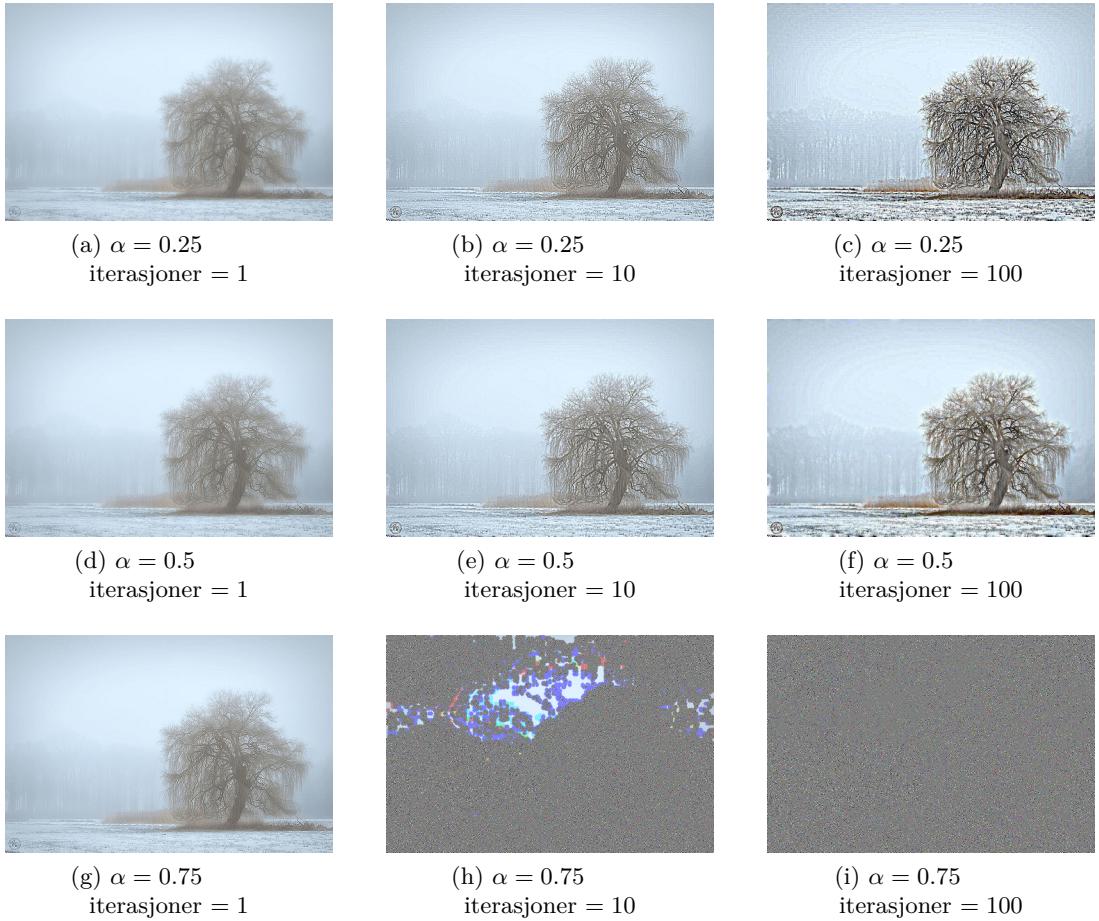
(h)  $\alpha = 0.75$   
iterasjoner = 10



(i)  $\alpha = 0.75$   
iterasjoner = 100

Figur 11: Resultat for gråtonebilder med ulike verdier av  $\alpha$  og iterasjoner.  $k = 10$  på alle bildene.

### 7.4.2 Farger - eksplisitt



Figur 12: Resultat for fargebilder med ulike verdier av  $\alpha$  og iterasjoner.  $k = 10$  på alle bildene.

## 7.5 Diskusjon

Som man kan se var også dette en ganske lett frem implementasjon basert på hva vi alt har vært gjennom. Man kan se at metoden funker bra, men at man bør her passe på bruken av parametere for å få den beste kvaliteten.

## 7.6 Alternativene

### 7.6.1 Local adaptive histogram (LAH)

Dette er en teknikk som ser på piksel histogrammet, det vil si fordelingen av intensiteten av hver piksel og benytter dem til å konstruere en ny fordeling [Sze11, s. 95]. Man setter så den nye intensiteten på samme måte som man hentet den ut.

Dersom man skal sammenligne metodene ser man nedenfor et histogram av hvordan bilde sin intensitet endret seg mellom en iterasjon av hver metode Figur. 13.

Fra Figur. 14 kan man se at ved å benytte Poisson likningen får man en langt mer konservativ kontrast-forsterking (noe som bare skulle mangle fordi den ikke er begrenset til å kun endre gradienten), mens “local adaptive histogram” prøver å fordele piksel intensiteten langt mer. Resultatet er at “local adaptive histogram” gjerne gir mere livlige bilder (igjen, fordi LAH kan endre på mer enn gradienten).

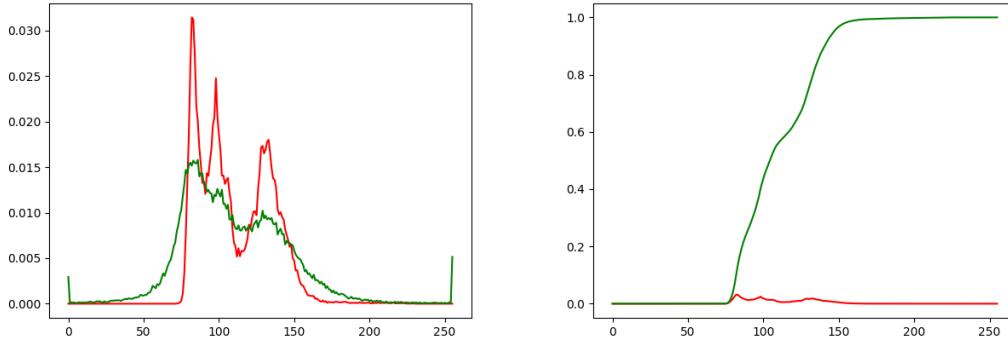
```

def intensity(img, channel) -> Array:
    intensity_scale = np.zeros(256, np.int32)
    for y in range(0, img.shape[0]):
        for x in range(0, img.shape[1]):
            intensity = img[y][x][channel]
            intensity_scale[intensity] += 1
    return intensity_scale / (img.shape[0] * img.shape[1] *
                               img.shape[2])

old_intensity = intensity(img)
new_intensity = np.cumsum(old_intensity)

```

Kodesnutt 6: Vår implementasjon

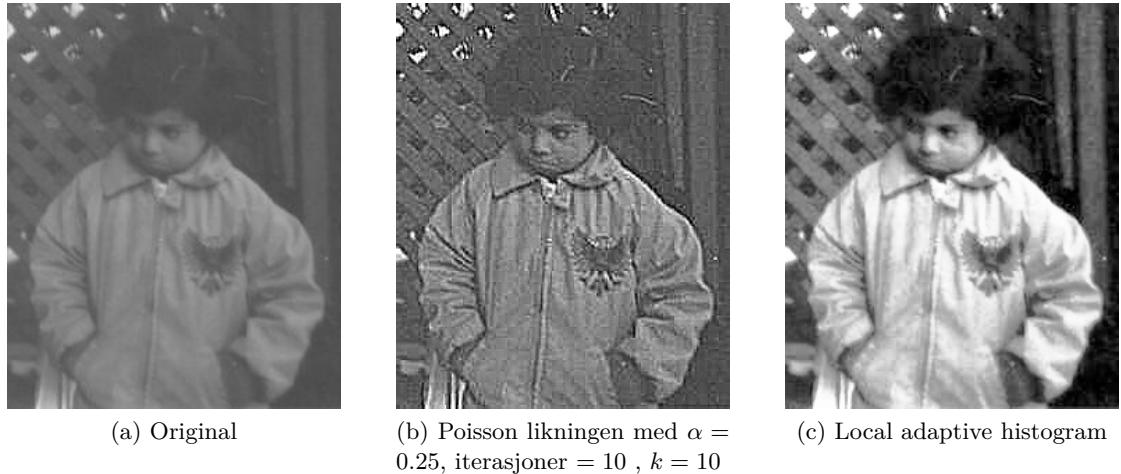


(a) Pixel fordelingen ved  $\alpha = 0.25$ , iterasjoner = 3    (b) Fordelingen etter local adaptive histogram med  $\alpha = 0.25$  og  $k = 10$

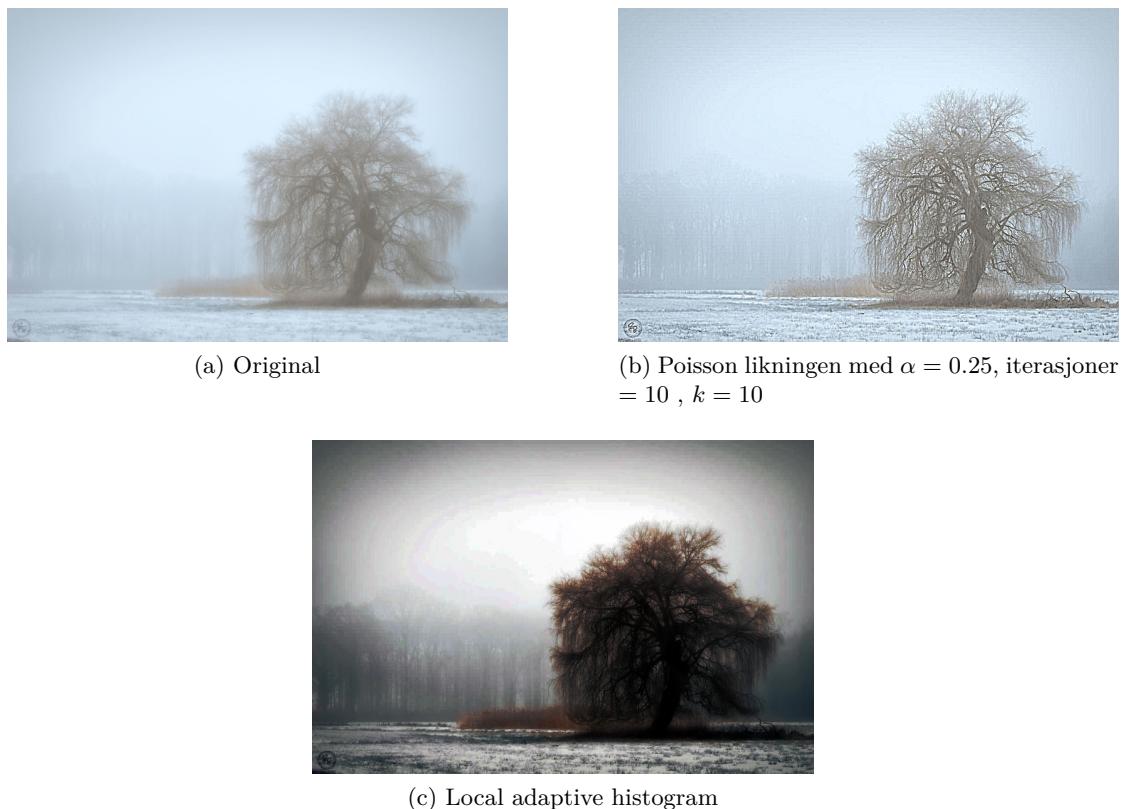
Figur 13: Fordelingen av piksel intensitet av Figur. 14. Fargene er **rød** for før og **grønn** for etter metoden er utført.

Likevel finnes det også tilfeller hvor Poisson gjør det bedre enn LAH som man kan se i Figur. 15. Selv om LAH her har produsert et langt mer livlig bilde så har det trolig gått på bekostning av at mye av bildets kontekst har blitt borte.

## 7.7 Resultat



Figur 14: Optimalt bilde for LAH



Figur 15: Et ikke optimalt bilde for LAH

## 8 Demosaicing

### 8.1 Intro

For at digitalkamera skal kunne lage fargebilder benyttes en mosaikk av fargefilter slik at bildesensoren kun behøver å måle en av fargekanalene. Resultatet er en gråtone mosaikk. Demosaicing-algoritme sin jobb er å rekonstruere et fargebilde fra en slik gråtone mosaikk.

### 8.2 Problemstilling

Vi ønsker å rekonstruerte et fargebilde ut fra et gråtone mosaikk.

### 8.3 Fremgangsmåte

Vi kan gjøre dette ved å først flytte mosaikken over i alle fargekanalene og bruker Poisson likningen til å inpainte den manglende informasjonen for hver kanal.

### 8.4 Implementering

Vi benyttet Kodesnutt. 15 fra [Far20, s. 3] for å lage et bilde med gråtonemosaiikk

```
class Demosaic(image_handler.ImageHandler, poisson.poisson,
    boundary.Boundary):
    def iteration(self) -> None:
        self.results = np.zeros(self.mosaic.shape + (3,))
        for i in range(3):
            self.results[:, :, i] =
                self.inpaint.fit(self.rgb_mosaic[:, :, i],
                self.results[:, :, i], self.mask[:, :, i])
        self.data = self.results
    return self.data
```

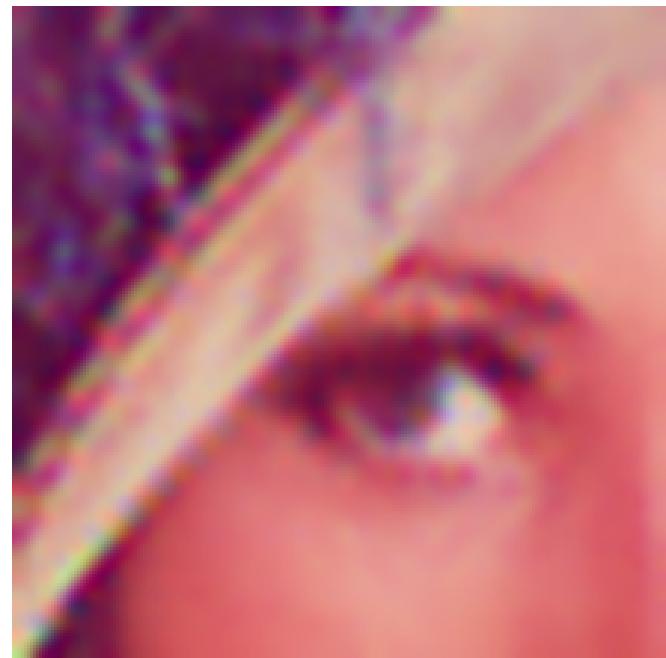
Kodesnutt 7: Vår implementasjon av Demosaic algoritme<sup>25</sup>

### 8.5 Diskusjon

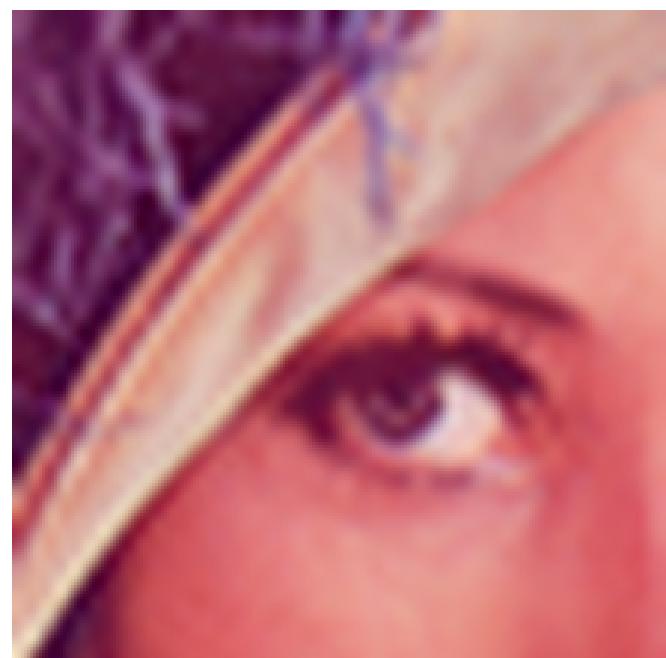
Siden dette er en metode som bygger på tidligere metode så gjelder det å ha abstrakt funksjon fra tidligere så man enkelt kan benytte den for inpainting. Vår “fit”<sup>26</sup> funksjon tar seg av å sette opp masken for så og sette  $u_0$  for så å utføre inpaint. Dersom man ser på Figur. 16 så ser man også her det samme som man så ved inpainting at bildet kan få en pikselert struktur om man studerer bilde med noe farge støy. Likevel er dette igjen noe man ikke ser med mindre man studerer bilde mer nøye.

<sup>25</sup>Full kode med docstrings og ekstra metoder : <https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/backend/demosaicing.py>

<sup>26</sup>inpaint sin fit funksjon på Git <https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/Brage/src/backend/inpainting.py#L146>



(a) Bilde med 100 iterasjoner,  $\alpha = 0.25$

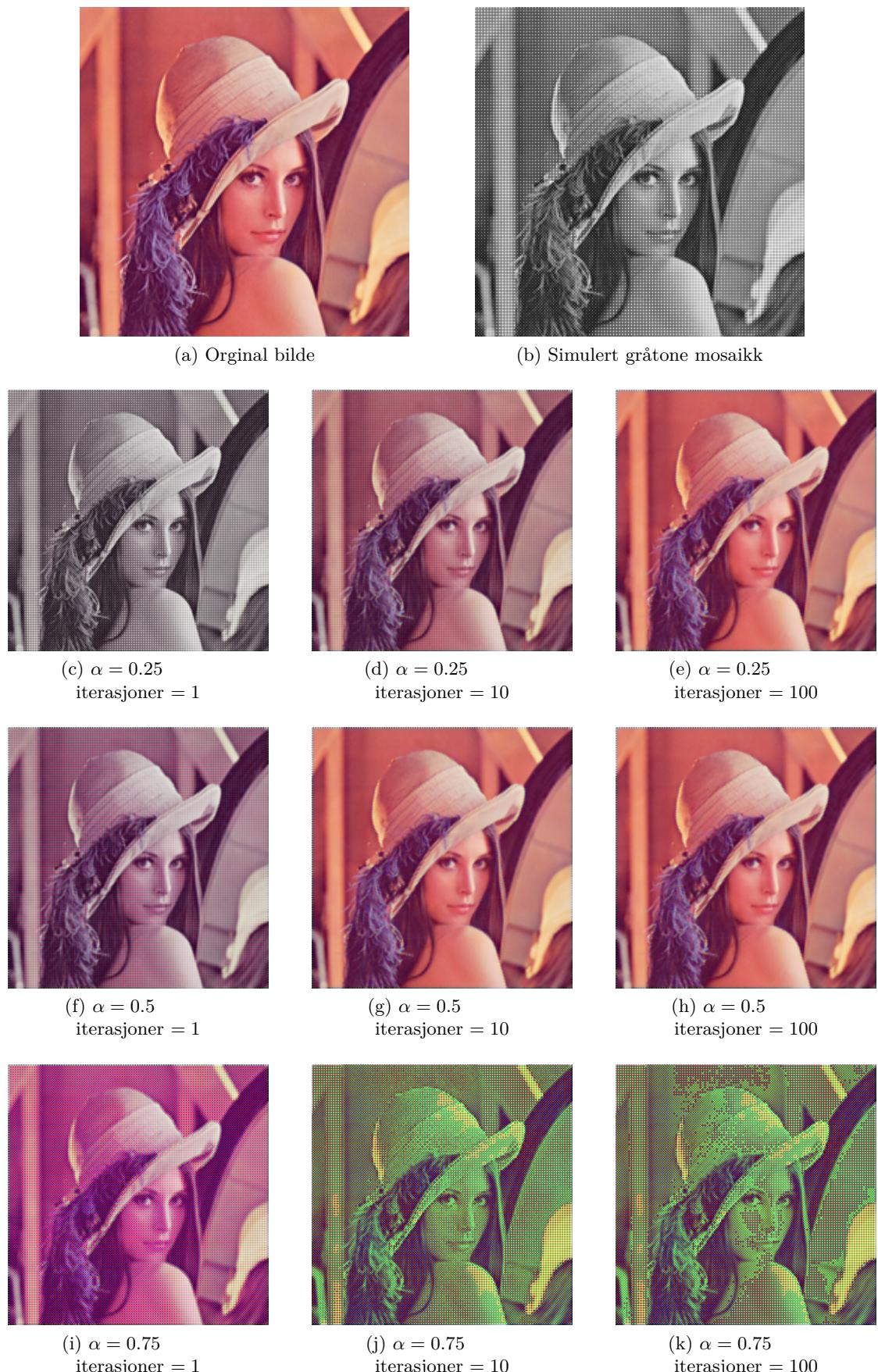


(b) Original bilde

Figur 16: Demosaicing algoritme er ikke perfekt

## 8.6 Resultat

### 8.6.1 Eksplisitt



Figur 17: Resultat med ulike verdier av  $\alpha$  og iterasjoner

## 9 Somløs kloning

### 9.1 Intro

Det finnes mange grunner for å ønske å flytte et bilde inn i et annet. Enten det er for humoristisk verdi eller for å skrive om historien. Dersom vi kloner et bilde rett inn i et annet, vil ikke effekten være realistisk fordi det vil være en hard kant mellom de 2 bildene, og det er lett å se at bildet er manipulert. Dette kan unngås ved å benytte somløs kloning.

### 9.2 Problemstilling

Vi ønsker å la et bilde( $u_{source}$ ) bli del av et annet( $u_{target}$ ) ved å benytte Poisson likningen.

### 9.3 Fremgangsmåte

Vi ønsker å finne  $u$  slik at den delen av bildet som skal bevares( $x \notin \Omega_i$ ) er  $u = u_0$  og hvor vi skal flytte over informasjon til original bilde skal ha  $\nabla^2 u_{target} = \nabla^2 u_{source}$ . Dette løses i praksis ved å sette  $h = \nabla^2 u_{source}$  og utnytte numpy sin view funksjon for å kun jobbe på det aktuelle området.

### 9.4 Implementering

Som man kan se blir implementasjonen nokså rett fram

```
class Matting(image_handler.ImageHandler, poisson.Poisson,
    ↪ boundary.Boundary):
    def iteration(self) -> Array:
        working_area = self.crop(self.data)
        solved_working_area = self.solve(working_area,
            ↪ self.operator, self.h, apply_boundary=False)
        self.apply(solved_working_area)
        return self.data

    def operator(self, i):
        working_area = self.crop(self.data)
        return self.get_laplace(working_area[:, :, i])

    def h(self, i):
        working_area_source = self.crop(self.source.data, False)
        return self.get_laplace((working_area_source)[:, :, i])
```

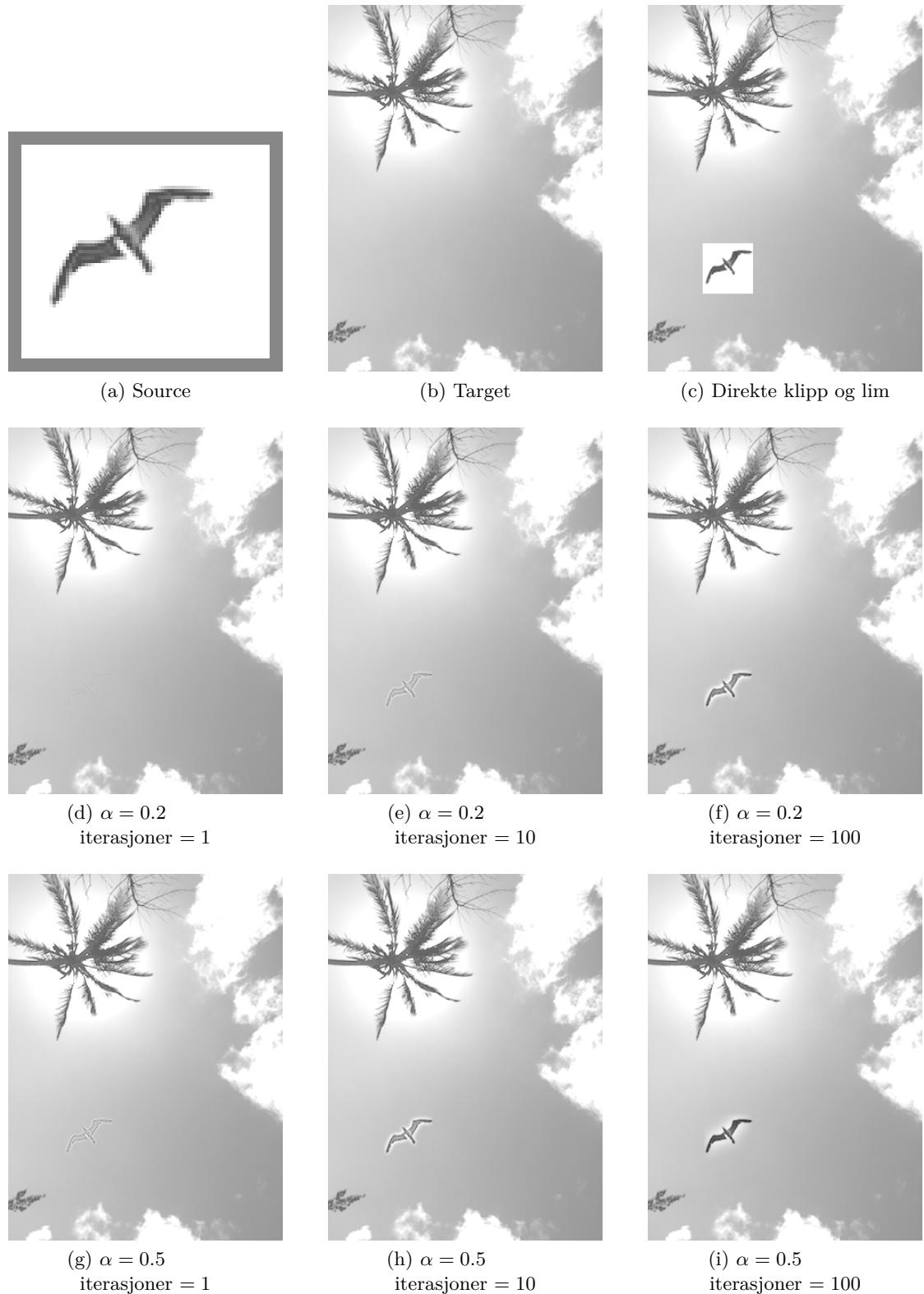
Kodesnutt 8: Vår implementasjon <sup>27</sup>

Her vil `crop()` funksjonen ta seg av å “aligne” bildene riktig ved å hente ut riktig del av bildet (dette er parametere som kan endres). `apply()` vil passe på at man legger resultatet på riktig sted. Begge metodene utnytter numpy sin view funksjon.

<sup>27</sup>Full kode med docstrings og ekstra metoder : <https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/backend/matting.py>

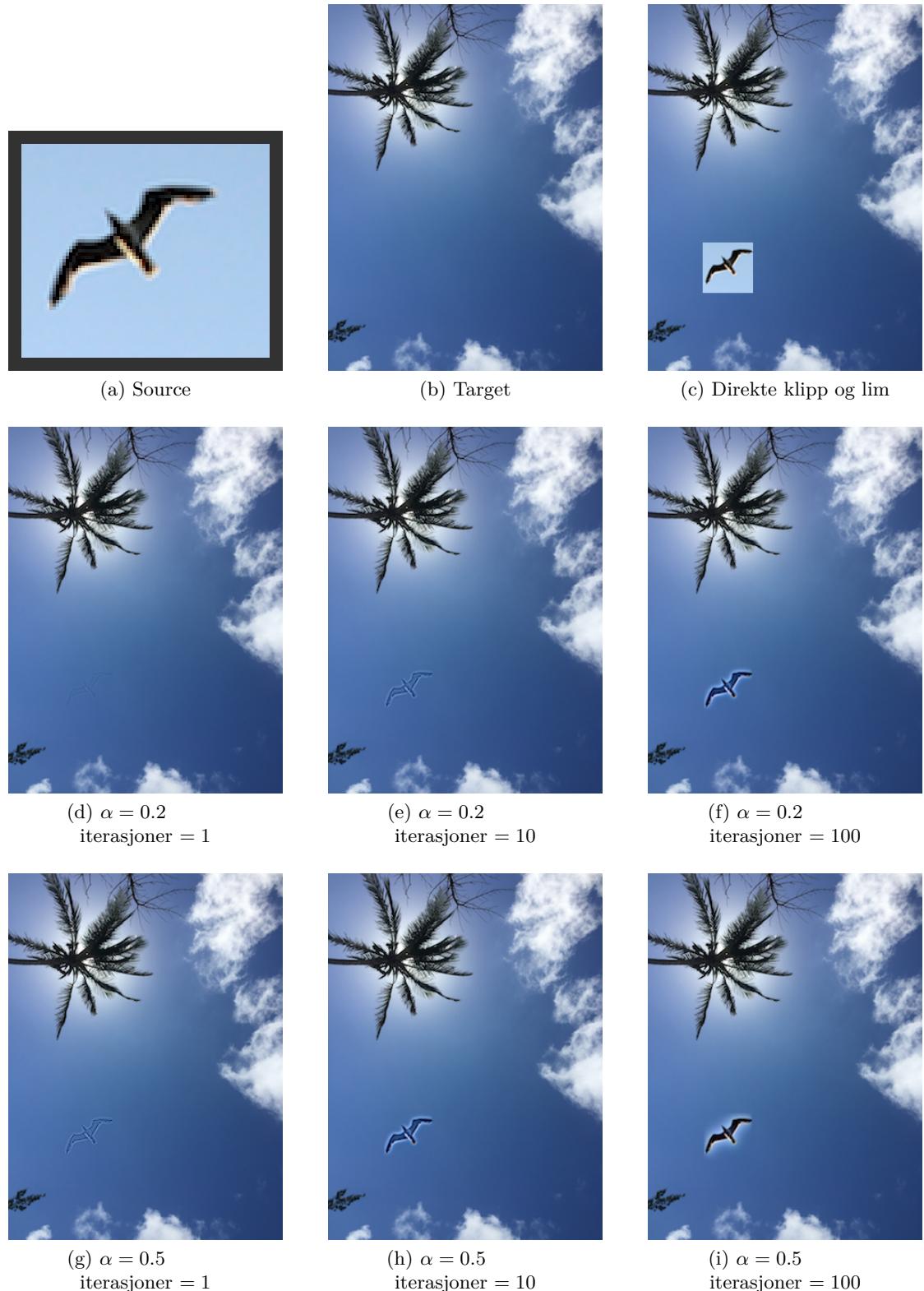
## 9.5 Resultat

### 9.5.1 Gråtone - Eksplisitt



Figur 18: Resultat for gråtonebilder med ulike verdier av  $\alpha$  og iterasjoner

### 9.5.2 Farger - Eksplisitt



Figur 19: Resultat for fargebilder med ulike verdier av  $\alpha$  og iterasjoner

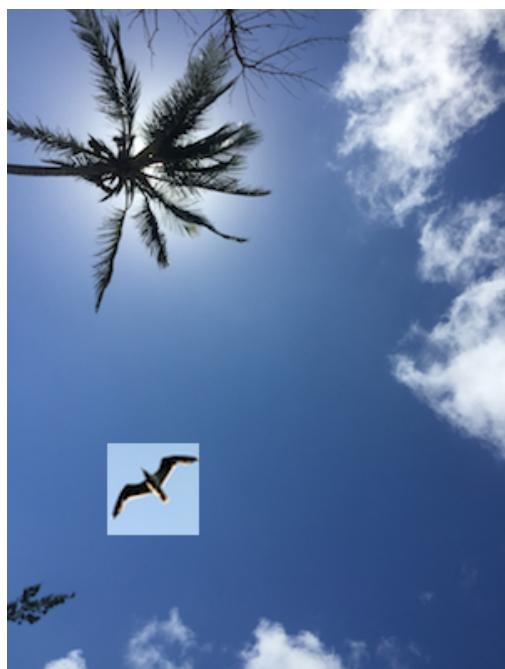
### 9.5.3 Parametere



(a) Target bilde lagt på source bilde



(b) Bilde med 100 iterasjoner,  $\alpha = 0.25$



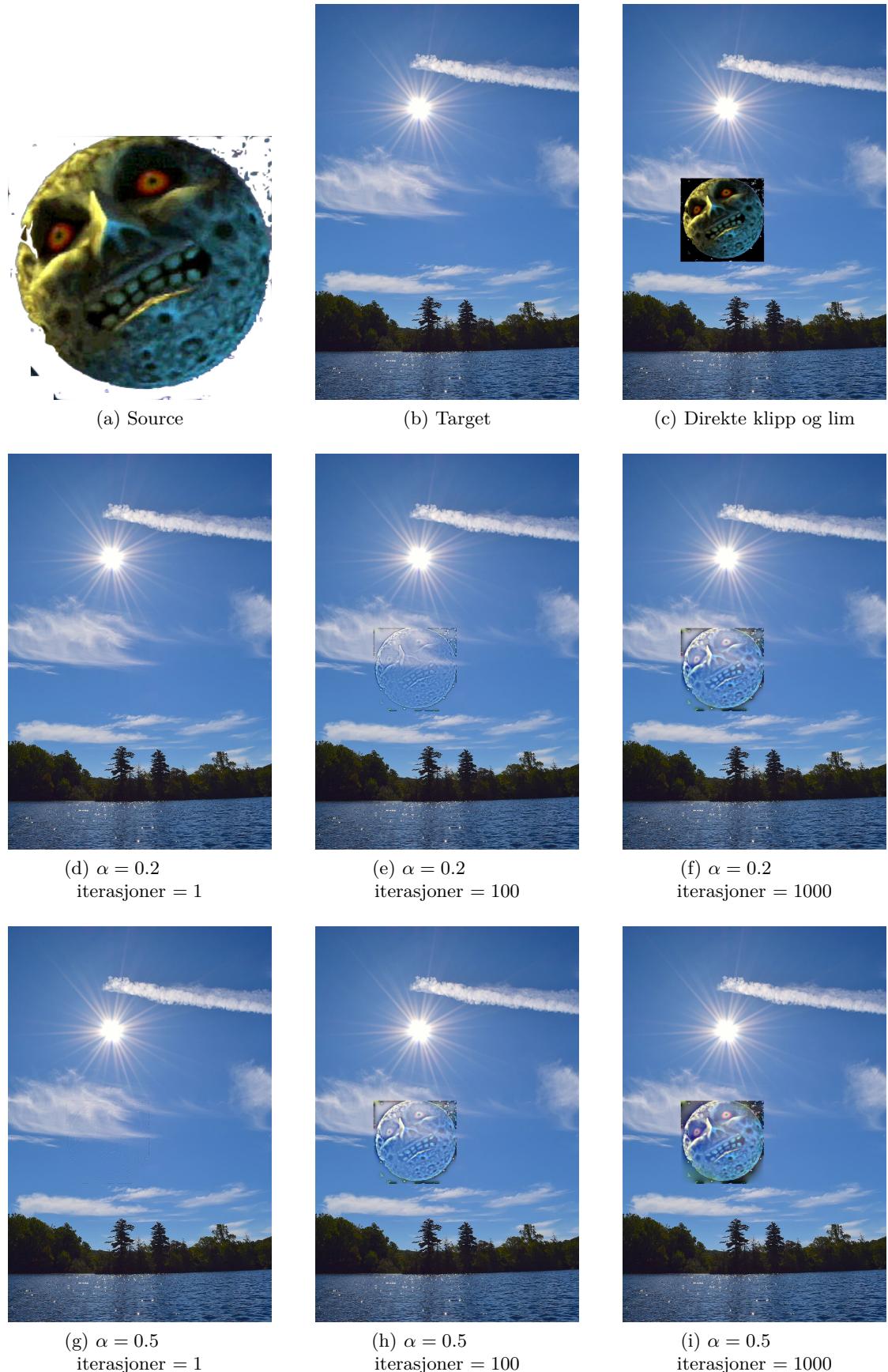
(c) Target bilde lagt på source bilde



(d) Bilde med 100 iterasjoner,  $\alpha = 0.25$

Figur 20: Parametere er viktig.

#### 9.5.4 Støy

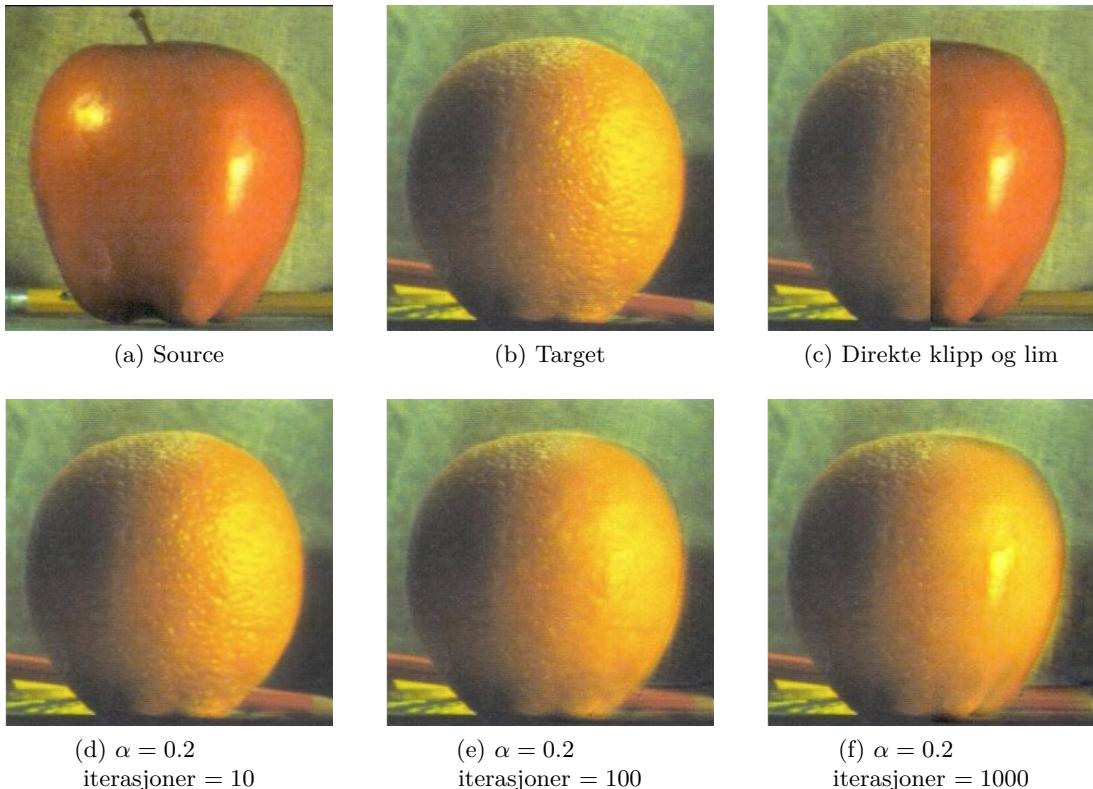


Figur 21: Resultat med ulike verdier av  $\alpha$  og iterasjoner

## 9.6 Diskusjon

Dette er en metode som er mer sensitiv til valg av parameterne enn tidligere diskuterte metoder (se Figur. 20). Siden man ikke bare skal forholde seg til ett bilde, men to, bør man passe på at parameterne er gode for å få best resultat. Det gjelder både å velge en god bounding box, men også en god bakgrunn (det finnes også bilder som bare fungerer dårlig sammen Figur. 21).

Det som også var meget interessant var når vi prøvde oss å utføre somløs kloning for å skape en *Orapple* (Figur. 22). Man kan se at kantene gradvis blir mer glatte og appelsinen som ellers har et noe mer “skrukkete” skall blir langt mer glatt.



Figur 22: Orapple

## 9.7 Alternativer

### 9.7.1 Healing brush

Adobe Photoshop lanserte i 2002 en funksjon kalt healing brush som er en mer sofistikert utgave av somløs kloning [Hig15, s. 6]. Man vil da velge å løse den Biharmoniske likningen<sup>28</sup>, som man kan se er det kvadratet av vår kjente Laplace.

$$\Delta^2 g(x, y) = \Delta^2 f(x + \delta x, y + \delta y) \quad (4)$$

$$\Delta^2 = \frac{\partial^4}{x^4} + \frac{2\partial^4}{\partial x^2 \partial y^2} + \frac{\partial^4}{y^4} = \nabla^2 u \quad (5)$$

Det ble dessverre ikke tid nok til å implementere denne og deretter sammenligne, men [Geo] går i dybden for hvordan metoden fungerer og anbefales for dem som er nysgjerrige.

---

<sup>28</sup>[https://en.wikipedia.org/wiki/Biharmonic\\_equation](https://en.wikipedia.org/wiki/Biharmonic_equation)

## 10 Konvertering av fargebilder til gråtone

### 10.1 Intro

Den enkleste metoden for å konvertere fargebilder til gråtone er ved å ta gjenomsnittet av fargekanalene. Dette fører til at vi mister deler av informasjonen, men heldigvis har Poisson likningen en løsning for oss.

### 10.2 Problemstilling

Vi ønsker å konstruere et gråtone-bilde ved hjelp av Poisson likningen.

### 10.3 Fremgangsmåte

Vi kan konstruerte en ny gradient med lengden  $\frac{\|\nabla u_0\|}{\sqrt{3}} = g$  med retningen gitt av  $\nabla(\sum_{c \in \{R,G,B\}} u_c) = \dot{h}$  (hvor  $u_c$  er fargekanalen  $c$  for bildet) og la  $h = \nabla \cdot g$ . Deretter itererer vi som vanlig setter og satt  $u_0$  lik det vektede gråtone bildet.

### 10.4 Implementering

Denne metoden krever litt mer penn og papir enn tidligere metoder. Når metoden først er implementert så er det lett å se hvor lett det var, men vi hadde noen utfordringer med å implementere denne metode.

Det å finne lengden er greit nok, man tar lengden av gradienten for bildet. Samme gjelder gradienten for summen av fargekanalene. Det var løsningen  $h = \nabla \cdot g$  hvor vi måtte tenke litt.

```
def h_func(self) -> Array:
    g_length =
        ← np.sum(self.get_gradient_norm(self.data_copy[:, :, i])) for i in range(3)) / np.sqrt(3)
    rgb_sum = np.sum(self.data_copy[:, :, i] for i in range(self.data_copy.shape[-1]))
    rgb_sx, rgb_sy = self.get_gradient(rgb_sum)

    length = np.sqrt(rgb_sx ** 2 + rgb_sy ** 2)
    length[length == 0] = np.finfo(float).eps
    rgb_sx /= length
    rgb_sy /= length

    h_prime_x, h_prime_y = rgb_sx * g_length, rgb_sy *
        ← g_length
    h_sx, _ = self.get_gradient(h_prime_x)
    _, h_sy = self.get_gradient(h_prime_y)

    return (h_sx + h_sy)
```

Kodesnutt 9: implementasjon av  $h$  funksjonen til konvertering av fargebilder til gråtone

Vi fant ut at vi ikke kunne utnytte gradienten direkte, men måtte benytte oss av dens enhetsvektor(unit vector). Dette er noe som selvsagt gir mening, siden  $g$  står for

<sup>28</sup>Full kode med docstrings og ekstra metoder : <https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/backend/mattting.py>

lengden og da skulle  $h$  stå for rettingen, men dette var ikke åpenbart ved å kun se på oppgaveteksten. For å få et optimalt resultat bør man også passe på at man ikke deler med null (kan løses med  $\text{length}[\text{length} == 0] = \text{np.finfo(float).eps}$ ).

Man kan se fra resten av koden at mye er som man har sett før. Det eneste å merke seg er at første gang funksjonen kjøres så vil man regne ut  $h$  slik at man ikke trenger å regne det mer enn en gang.

```
class Grayscale(image_handler.ImageHandler, poisson.Poisson,
→ boundary.Boundary):
    def iteration(self) -> None:
        if len(np.shape(self.data)) == 3:
            self.data = self.data.mean(axis=2)
            self.h_array = self.h_func()

        self.data = self.solve(self.data, self.opeartor, self.h,
→ apply_boundary=True)

    def opeartor(self):
        return self.get_laplace(self.data, alpha=False)

    def h(self, data=None):
        return self.common_shape(self.h_array)
```

Kodesnutt 10: Vår implementasjon for konvertering av fargebilder til gråtone

## 10.5 Diskusjon

Dersom man sammenligner denne metoden mot den tradisjonelle vektende metoden så ser man at denne metoden tilsynelatende gjør bildet mer skarpt ved å tydeliggjøre kantene (Figur. 23) .

---

<sup>28</sup>Full kode med docstrings og ekstra metoder : <https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/backend/grayscale.py>



(a) Vektet bilde

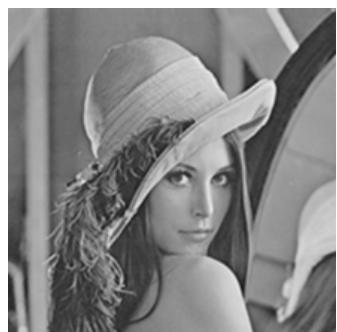


(b) Poisson ( $\alpha = 0.25$ , iterasjoner=10)

Figur 23: Sammenligning av metode

## 10.6 Resultat

### 10.6.1 Eksplisitt



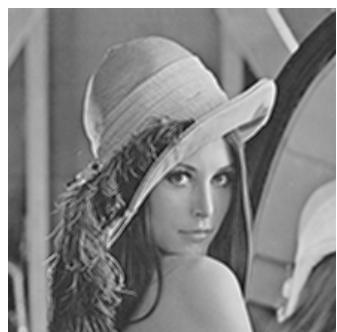
(a)  $\alpha = 0.25$   
iterasjoner = 1



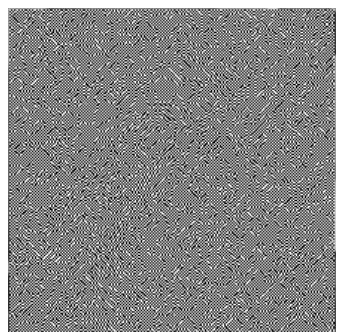
(b)  $\alpha = 0.25$   
iterasjoner = 10



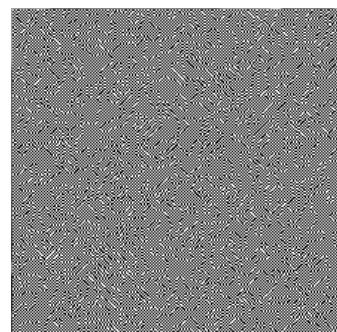
(c)  $\alpha = 0.25$   
iterasjoner = 100



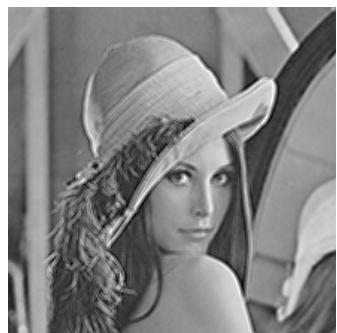
(d)  $\alpha = 0.5$   
iterasjoner = 1



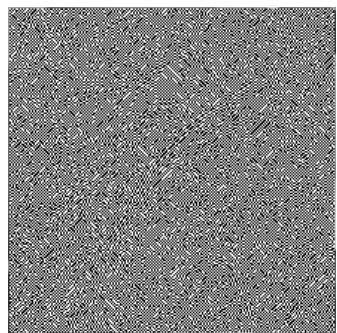
(e)  $\alpha = 0.5$   
iterasjoner = 10



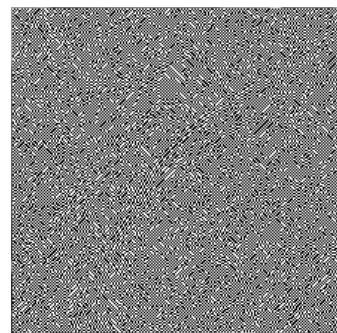
(f)  $\alpha = 0.5$   
iterasjoner = 100



(g)  $\alpha = 0.75$   
iterasjoner = 1



(h)  $\alpha = 0.75$   
iterasjoner = 10



(i)  $\alpha = 0.75$   
iterasjoner = 100

Figur 24: Resultat med ulike verdier av  $\alpha$  og iterasjoner

## 11 Anonymisering

### 11.1 Intro

Det er ikke uvanlig at et behov for å anonymisere flere på et bilde vil være nødvendig før publisering. En metode for dette er å gjøre ansikt uskarpt, men beholde resten av bildet skarpt.

### 11.2 Problemstilling

Vi ønsker å gjøre ansiktene på et bilde uskarpt.

### 11.3 Fremgangsmåte

Vi benytter Glatting funksjonen fra tidligere og glatter ut delen av bildet. For å finne masken for ansiktene benytter vi OpenCV<sup>29</sup>.

### 11.4 Implementering

```
class Anonymous(image_handler.ImageHandler, poisson.Poisson,
    ↪ boundary.Boundary):
    def iteration(self) -> Array:
        blur = blurring.Blur(None)
        for mask in self.mask:
            blur.set_data(self.data.copy() [mask[0]:mask[1],
                ↪ mask[2]:mask[3]])
            blur.set_boundary("dirichlet")
            self.data[mask[0]:mask[1], mask[2]:mask[3]] =
                ↪ blur.itearasjoner()
        return self.data
```

Kodesnutt 11: Vår implementasjon av anonymisering<sup>30</sup>

Som man kan se er koden ganske rett frem når man har et allerede oppsatt rammeverk.

### 11.5 Diskusjon

Som man kan se fra Resultat funker metoden godt i praksis. Både for en person og når det er flere mennesker i et bildet. Man kan se at blur boksen noen ganger blir veldig tydelig, en mulig fremtidig forbedring er å gradvis øke styrken på blur når man nærmer seg midten av bildet og utnyttet hodets symmetri.

Man kan også se fra Figur. 25 at OpenCV noen ganger kan være litt aggressiv og markere ting som ikke er ansikt som “ansikt”. En metode for å forbedre dette er å benytte flere modeller (Ensemble model) og la dem sammen komme til enighet<sup>31</sup>.

---

<sup>29</sup><https://opencv.org/>

<sup>30</sup>Full kode med docstrings og ekstra metoder : <https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/backend/anonymizing.py>

<sup>31</sup>[https://en.wikipedia.org/wiki/Ensemble\\_learning](https://en.wikipedia.org/wiki/Ensemble_learning)

## 11.6 Resultat

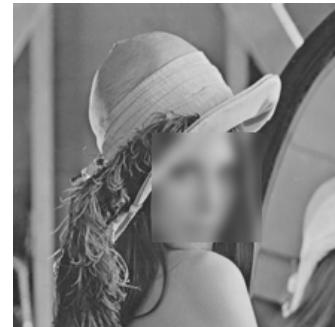
### 11.6.1 Gråtona - eksplisitt



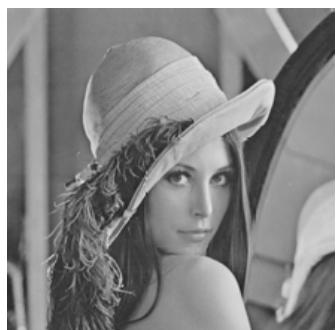
(a)  $\alpha = 0.25$   
iterasjoner = 1



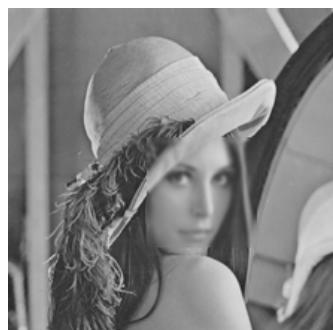
(b)  $\alpha = 0.25$   
iterasjoner = 10



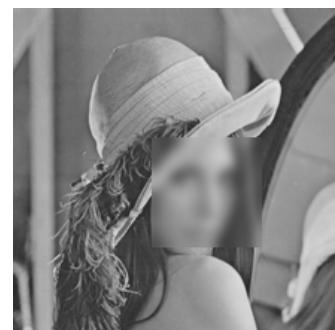
(c)  $\alpha = 0.25$   
iterasjoner = 100



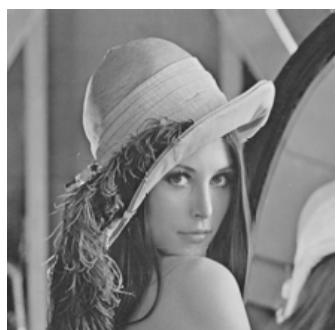
(d)  $\alpha = 0.5$   
iterasjoner = 1



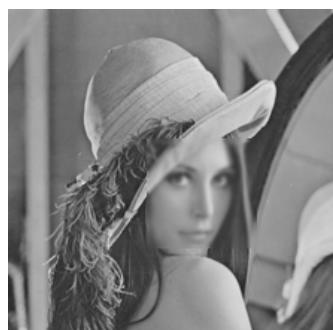
(e)  $\alpha = 0.5$   
iterasjoner = 10



(f)  $\alpha = 0.5$   
iterasjoner = 100



(g)  $\alpha = 0.75$   
iterasjoner = 1

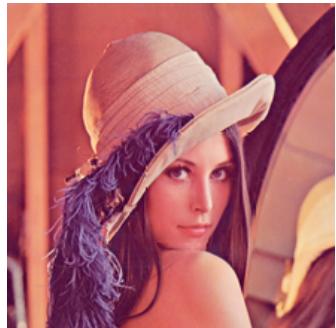


(h)  $\alpha = 0.75$   
iterasjoner = 10

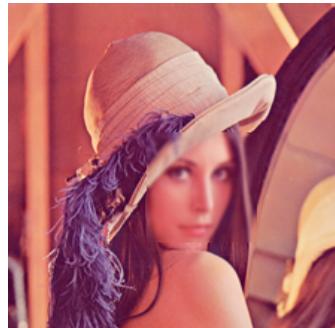


(i)  $\alpha = 0.75$   
iterasjoner = 100

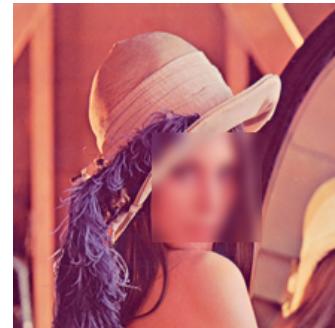
### 11.6.2 Farger - eksplisitt



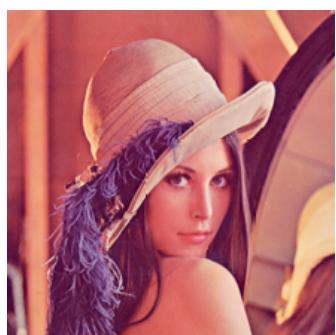
(a)  $\alpha = 0.25$   
iterasjoner = 1



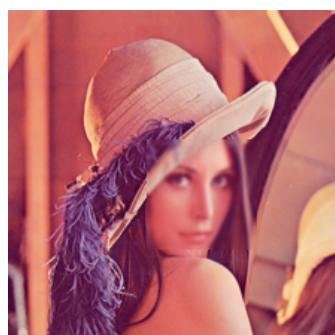
(b)  $\alpha = 0.25$   
iterasjoner = 10



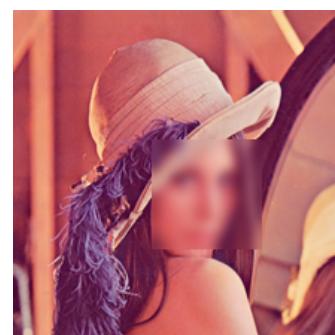
(c)  $\alpha = 0.25$   
iterasjoner = 100



(d)  $\alpha = 0.5$   
iterasjoner = 1



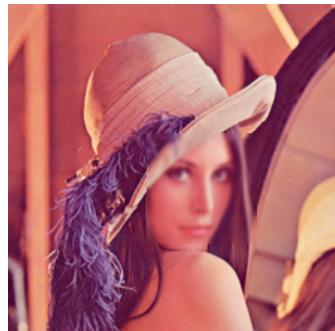
(e)  $\alpha = 0.5$   
iterasjoner = 10



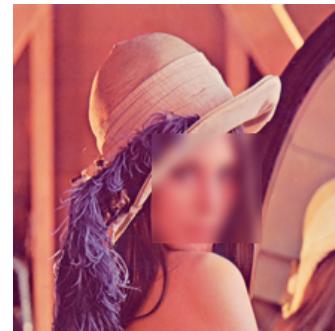
(f)  $\alpha = 0.5$   
iterasjoner = 100



(g)  $\alpha = 0.75$   
iterasjoner = 1



(h)  $\alpha = 0.75$   
iterasjoner = 10



(i)  $\alpha = 0.75$   
iterasjoner = 100

### 11.6.3 Mange ansikt



(a) Ansikt



(b) Anonyme

Figur 25: Metoden funker også for mange forskjellige ansikt (Som man kan se er OpenCV litt agresiv noen steder)

## 12 Rekonstruksjon og visualisering av HDR-bilder

### 12.1 Intro

Bilder som er tatt i dagslys kan fort bli over- og undereksponert, man ender da opp med bilder som har områder som er helt svarte og andre helt hvite. En teknikk for å unngå dette er å kombinere bildene til et HDR (High Dynamic Range) bilde.

### 12.2 Problemstilling

Vi ønsker å gjøre flere bilder med ulik eksponeringstid til et bilde som ikke er over- eller undereksponert.

### 12.3 Fremgangsmåte

Fremgangsmåten er beskrevet i utdypet form i [DM97]. I korte trekk går metoden ut på følgende :

1. Man henter inn en liste med bilder med ulik eksponeringstid  $\{u_0, u_1, \dots, u_n\}$ .
2. For hver  $u \in \{u_0, u_1, \dots, u_n\}$  "sampler" man piksel intensiteten ved tilfeldige piksler av  $u(x, y)$ . Disse lagrer man til et eget array ( $\mathcal{Z}$ ).
3. Basert på dette arrayet ( $\mathcal{Z}$ ) vil man sette opp et likningssystem  $Ax = B$  (gsolve i [DM97])
4. Løsningen på likningsette benyttes videre for å regne utstråling verdien for hver piksel for det endelige bilde.

$$\ln E_i = \frac{\sum_{j=1}^P w(Z_{ij})(g(Z_{ij}) - \ln \Delta t_j)}{\sum_{j=1}^P w(Z_{ij})}$$

Figur 26: Likning 6 fra [DM97]

5. Det vil tilslutt være gunstig å normalisere resultatet over slik at man kan vise det frem. Her er  $\mathcal{U}$ , resultatet av likningen over og  $\bar{u}$  resultatet av normaliseringen. Vi utfører denne operasjonen over alle farge kanalene  $c$ .

$$\bar{u}_c = \frac{\mathcal{U}_c - \max(\mathcal{U}_c))}{\max(\mathcal{U}_c) + |\min \mathcal{U}_c|}$$

## 12.4 Implementering

Klassen for å håndtere HDR klasser bygger på mange funksjoner som ikke resten av klassene benytter. Siden det er mye kode vil vi nå se på de to sentrale funksjonene for å sette opp likningsystemet(Kodesnutt. 12) og for å løse dette(Kodesnutt. 13). Resten av koden er å finne [https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/engine/hdr\\_image\\_handler.py](https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/engine/hdr_image_handler.py)

```
def get_Ab(self, Z, n=256) -> tuple:
    k = 0
    A = np.zeros((Z.shape[0] * Z.shape[1] + n + 1, n +
                  Z.shape[0]))
    b = np.zeros((A.shape[0], 1))

    for i in range(Z.shape[0]):
        for j in range(Z.shape[1]):
            Z_ij = int(round(Z[i, j]))
            w_ij = self.weight_function(Z_ij + 1)

            A[k, Z_ij] = w_ij
            A[k, n + i] = -w_ij

            w_ij = self.weight_function(int(Z[i, j]) + 1)
            b[k, 0] = w_ij * self.B[j]
            k += 1

    A[k, 128] = 1
    k += 1

    for i in range(0, n - 3):
        A[k, i] = self.lambda_constant * self.weight_function(i +
                                                                2)
        A[k, i + 1] = -2 * self.lambda_constant *
                      self.weight_function(i + 2)
        A[k, i + 2] = self.lambda_constant *
                      self.weight_function(i + 2)
        k += 1
    return A, b, n
```

Kodesnutt 12: Kjerne funksjoner for HDR<sup>32</sup>

Under utvikling ble funksjonene `scipy.io.loadmat` og `scipy.io.savemat` ble mye brukt for å gjøre “root cause analysis” raskt og enkelt. Da kunne vi laste inn Matlab arrays og lagre numpy arrays til Matlab så overgangen mellom verktøyene ble sømløs.

Vi ble også nødt til å benytte en alternativ implementasjon av “leastsq” siden “`np.linalg.leastsq`” ikke så ut til å fungere slik vi ønsket.

<sup>32</sup>Full kode med docstrings og ekstra metoder : [https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/engine/hdr\\_image\\_handler.py](https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/engine/hdr_image_handler.py)

```

def gsolve(self, Z) -> tuple:
    A, b, n = self.get_Ab(Z)
    #   https://github.com/numpy/numpy/issues/9563
    def leastsq(X, Y):
        """ Solves the problem  $Y = XB$  """
        inv = np.linalg.pinv(np.dot(X.T, X))
        cross = np.dot(inv, X.T)
        beta = np.dot(cross, Y)
        return beta
    x = leastsq(A, b)
    g = x[1:n]
    lE = x[n + 1:x.shape[0]]
    return g, lE

```

Kodesnutt 13: Kjerne funksjoner for HDR<sup>33</sup>

Koden ble også sjekk mot et annen implementasjon<sup>34</sup> under utviklingen da vi også hadde andre problemer med Matlab og trengte noe å teste mot.

## 12.5 Diskusjon

Som man kan se fra Resultat så fungerer metoden meget bra i praksis. Man ser også at man ikke trenger så mange bilder for å få et godt resultat. Dersom man ser på Figur. 28 så ser man at med et godt valgt av bilder kan få et bra resultat (alt avhengig av hva man anser som godt resultat, i denne sammenheng tenker vi realistisk bilde).

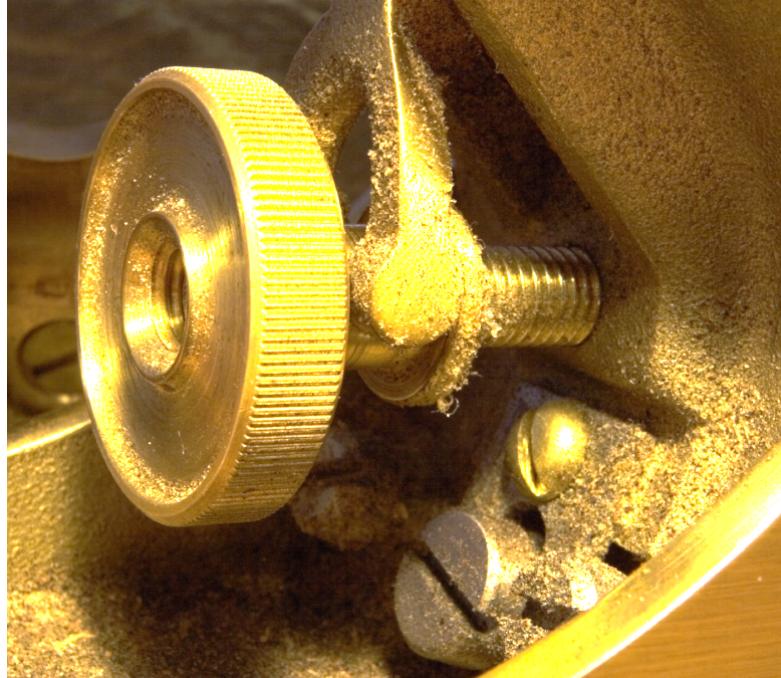
---

<sup>33</sup>Full kode med docstrings og ekstra metoder : [https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/engine/hdr\\_image\\_handler.py](https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/engine/hdr_image_handler.py)

<sup>34</sup><https://github.com/warmspringwinds/hdr-photography/>

## 12.6 Resultat

### 12.6.1 Adjuster



(a) Adjuster\_00064.png



(b) Adjuster\_00128.png



(c) Adjuster\_00256.png



(d) Adjuster\_00512.png



(e) output

### 12.6.2 Ocean



(a) Ocean\_00064.png



(b) Ocean\_00128.png



(c) Ocean\_00256.png



(d) Ocean\_00512.png



(e) Resultat

Figur 27: Konstruksjonen av et HDR bilde fra 4 bilder med ulik eksponeringstid (se filnavn)

### 12.6.3 Ocean med ulikt antall bilder



(a) Resultatet fra alle bildene (eksponeringstid  $\{2^x \mid 0 \leq x \leq 14\}$ )



(b) Resultatet fra 4 bilder (eksponeringstid  $\{64, 128, 256, 512\}$ )

Figur 28: Antall bilder og dems eksponeringstid påvirker resultatet

## 13 Kantbevarende glatting

### 13.1 Intro

Det finnes tilfeller når man ønsker å glatte et bilde uten å gjøre kantene i bildet uskarpe. Dette kan vi gjøre ved å innføre en posisjonsavhengig diffusjonsparameter.

### 13.2 Problemstilling

Vi ønsker kun at glatting skal påvirke deler av bilde, vi vil bevare kanten og gjøre resten av bilde glatt. Styrken skal kunne styres med en parameter  $k$ .

### 13.3 Fremgangsmåte

Fra oppgaveteksten ble vi gitt følgende likninger som vi kan bruke til å konstruere et nytt skjema.

$$D(x, y) = \frac{1}{1 + \kappa \|\nabla u_0(x, y)\|^2} \quad (6)$$

$$\frac{\partial u}{\partial t} = \nabla \cdot (D \nabla u) \quad (7)$$

Konstruksjon av nytt numerisk skjema.

Utnytter definisjonen av en gradient

$$\frac{\partial u}{\partial t} = \nabla \cdot (D \nabla u) = \frac{\partial}{\partial x}(D \frac{\partial u}{\partial x}) + \frac{\partial}{\partial y}(D \frac{\partial u}{\partial y}) \quad (8)$$

Benytter produkt regelen

$$\frac{\partial D \partial u}{\partial x^2} + D \frac{\partial^2 u}{\partial x^2} + \frac{\partial D \partial u}{\partial y^2} + D \frac{\partial^2 u}{\partial y^2} \quad (9)$$

Samler opp

$$\frac{\partial u}{\partial t} = D \nabla^2 u + \frac{\partial D \partial u}{\partial x^2} + \frac{\partial D \partial u}{\partial y^2} \quad (10)$$

$$= D \nabla^2 u + (\nabla D \cdot \nabla u) \quad (11)$$

Man ender da opp med følgende numeriske skjema for eksplisitt.

$$u_{i,j}^{n+1} = \frac{\Delta t}{\Delta x^2} (D_{i,j}^n (u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n) + u_{i,j}^n) \quad (12)$$

$$+ (D_{i+1,j}^n - D_{i-1,j}^n)(u_{i+1,j}^n - u_{i-1,j}^n) \quad (13)$$

$$+ (D_{i,j+1}^n - D_{i,j-1}^n)(u_{i,j+1}^n - u_{i,j-1}^n)) \quad (14)$$

### 13.4 Implementering

Denne implementasjonen (Kodesnutt. 14) er også ganske lett frem så fort man har fått satt opp skjema. Vi benytter i tillegg `numpy.gradient` for å få gradienten til  $D$  og  $u$ . Som hjelpefunksjonen `get_gradient` står for.

```

class NonEdgeBlur(image_handler.ImageHandler, poisson.Poisson,
                   boundary.Boundary):
    def D(self) -> Array:
        fraction = 1 / \
            (1 + self.k *
             (self.get_gradient_norm(self.data_copy))
             ** 2)
        return fraction

    def set_D_gradient(self):
        self.D_arr = self.D()
        assert np.all(self.D_arr <= 1), "D function error"

        self.d_xy = np.asarray(self.get_gradient(self.D_arr))

    def operator(self, i=None):
        data_xy = np.asarray(self.get_gradient(self.data))
        combined = np.sum(self.d_xy * data_xy, axis=0)
        if i is None:
            return ((self.common_shape(self.D_arr) *
                    self.get_laplace(self.data, alpha=False)
                    + self.common_shape(combined)))
        else:
            return ((self.common_shape(self.D_arr)[:, :, i] *
                    self.get_laplace(self.data[:, :, i], alpha=False)
                    + self.common_shape(combined[:, :, i])))

```

Kodesnutt 14: Vår implementasjon av kantbevarende glatting<sup>35</sup>

### 13.5 Diskusjon

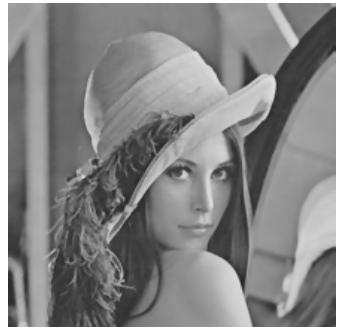
Man ser også her at man bør passe på bruken av parametere, dersom man itererer for lenge med en høy  $k$  verdi får man ikke optimale resultater (Resultater). Likevel benytter man fornuftige parametere kan man se at kantbevarende glatting gir flotte resultater (Glatting og kantbevarende glatting - Side om side).

---

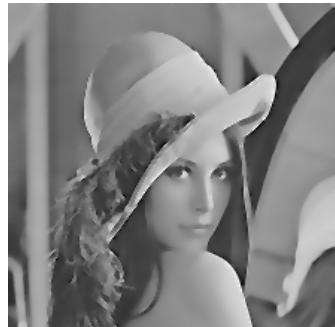
<sup>35</sup>Full kode med docstrings og ekstra metoder : [https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/backend/non\\_edge\\_blurring.py](https://git.gvk.idi.ntnu.no/BrageA/imt3881-2020-prosjekt/-/blob/master/src/backend/non_edge_blurring.py)

## 13.6 Resultater

### 13.6.1 Gråtone - eksplisitt



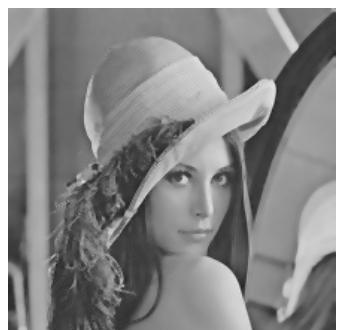
(a)  $\alpha = 0.25$   
iterasjoner = 1



(b)  $\alpha = 0.25$   
iterasjoner = 10



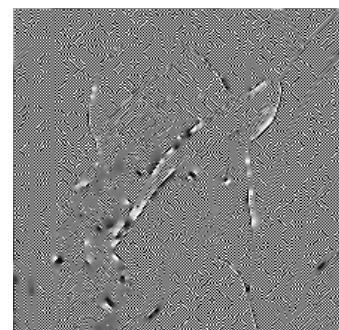
(c)  $\alpha = 0.25$   
iterasjoner = 100



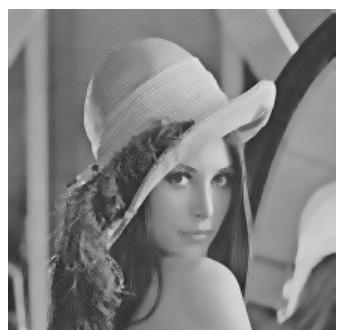
(d)  $\alpha = 0.5$   
iterasjoner = 1



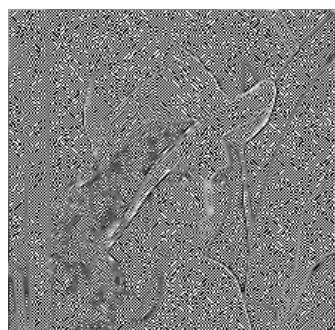
(e)  $\alpha = 0.5$   
iterasjoner = 10



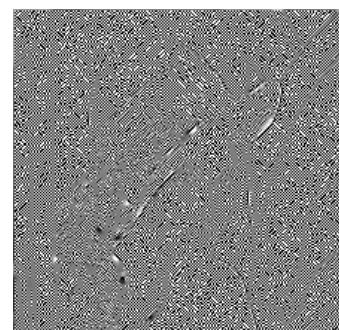
(f)  $\alpha = 0.5$   
iterasjoner = 100



(g)  $\alpha = 0.75$   
iterasjoner = 1



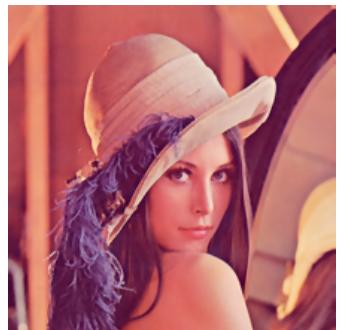
(h)  $\alpha = 0.75$   
iterasjoner = 10



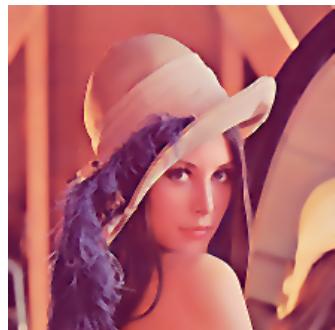
(i)  $\alpha = 0.75$   
iterasjoner = 100

Figur 29: Resultat med ulike verdier av  $\alpha$  og iterasjoner.  $K = 500$  på alle bildene.

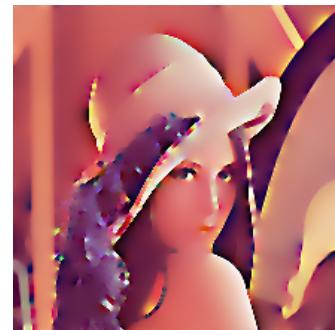
### 13.6.2 Farger - eksplisitt



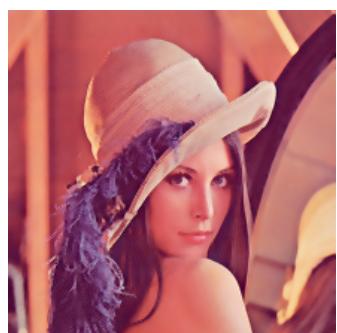
(a)  $\alpha = 0.25$   
iterasjoner = 1



(b)  $\alpha = 0.25$   
iterasjoner = 10



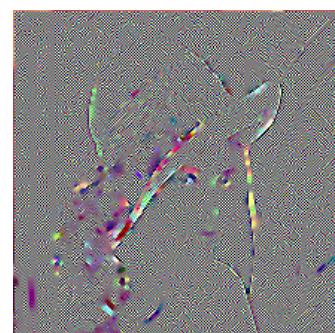
(c)  $\alpha = 0.25$   
iterasjoner = 100



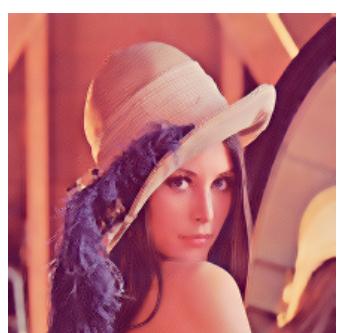
(d)  $\alpha = 0.5$   
iterasjoner = 1



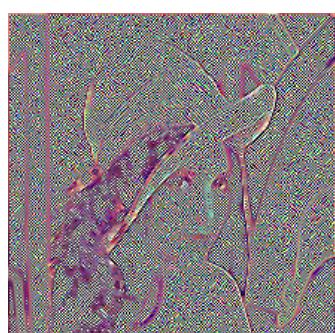
(e)  $\alpha = 0.5$   
iterasjoner = 10



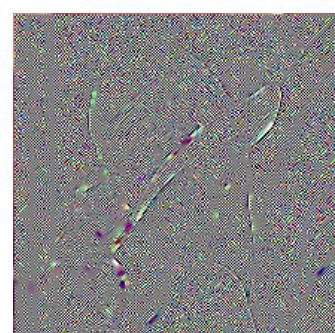
(f)  $\alpha = 0.5$   
iterasjoner = 100



(g)  $\alpha = 0.75$   
iterasjoner = 1



(h)  $\alpha = 0.75$   
iterasjoner = 10

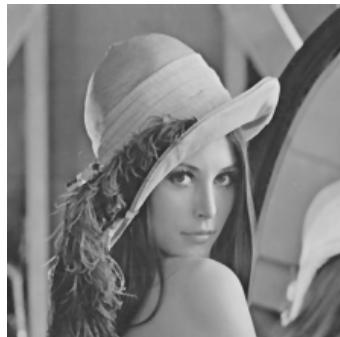


(i)  $\alpha = 0.75$   
iterasjoner = 100

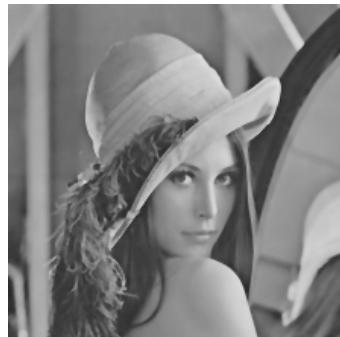
Figur 30: Resultat med ulike verdier av  $\alpha$  og iterasjoner.  $K = 500$  på alle bildene.

## 13.7 Resultat

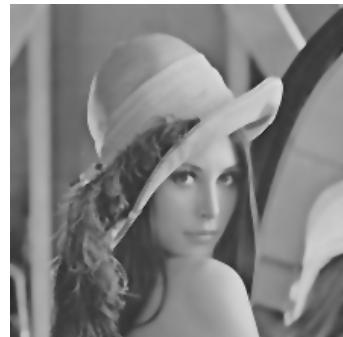
### 13.7.1 Glatting og kantbevarende glatting - Side om side



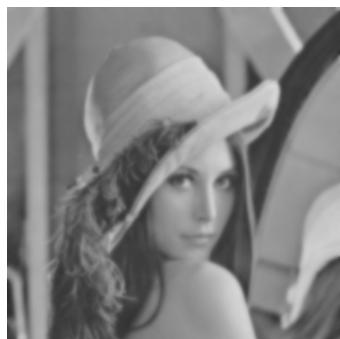
(a) Kantbevarende blur( $K = 100, \alpha = 0.2$ , iterasjoner=5)



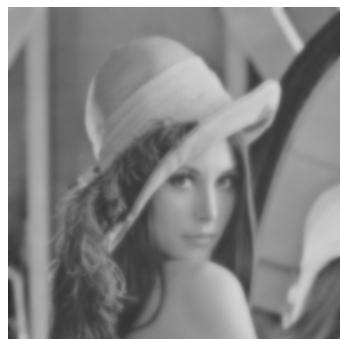
(b) Kantbevarende blur( $K = 100, \alpha = 0.2$ , iterasjoner=10)



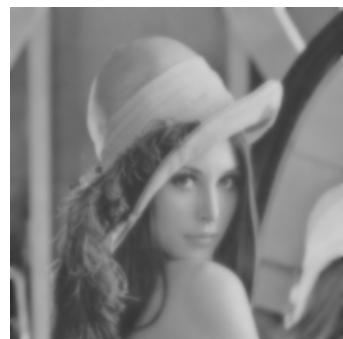
(c) Kantbevarende blur( $K = 100, \alpha = 0.2$ , iterasjoner=25)



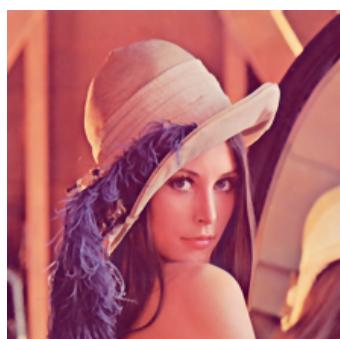
(d) Vanlig blur( $\alpha = 0.2$ , iterasjoner=5)



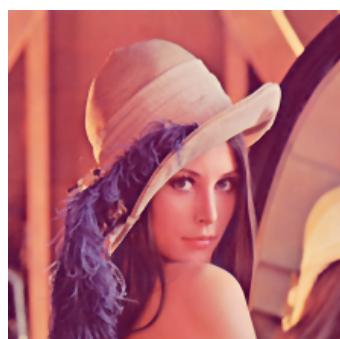
(e) Vanlig blur( $\alpha = 0.2$ , iterasjoner=10)



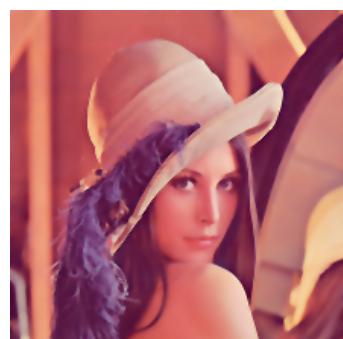
(f) Vanlig blur( $\alpha = 0.2$ , iterasjoner=25)



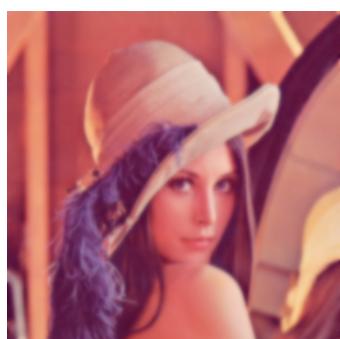
(g) Kantbevarende blur( $K = 100, \alpha = 0.2$ , iterasjoner=5)



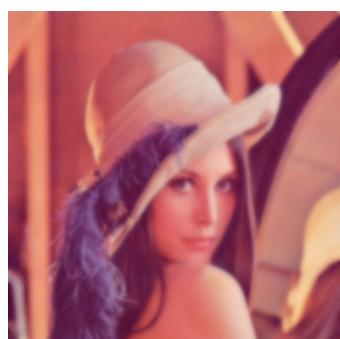
(h) Kantbevarende blur( $K = 100, \alpha = 0.2$ , iterasjoner=10)



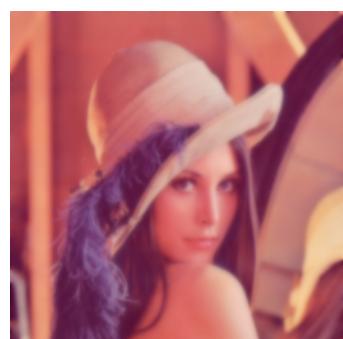
(i) Kantbevarende blur( $K = 100, \alpha = 0.2$ , iterasjoner=25)



(j) Vanlig blur( $\alpha = 0.2$ , iterasjoner=5)



(k) Vanlig blur( $\alpha = 0.2$ , iterasjoner=10)



(l) Vanlig blur( $\alpha = 0.2$ , iterasjoner=25)

Figur 31: Blur mot Blur

## 14 Refleksjon

Vi er fornøyd med resultatet av prosjektet. Vi implementerte 9 av 11 foreslalte metoder fullstendig. Alle metodene har et ganske generelt grensesnitt som gjør dem lette å anvende. Vi implementerte i tillegg utvidelser for fargebilder der det virker riktig og lagde et enkel GUI med QT<sup>36</sup>. Vi har i denne rapporten sammenlignet flere av metodene mot andre metoder som eksisterer for å gi et bedre inntrykk av hvordan metoden funger i praksis og hvor “gode” de er.

Som med mye annet finnes det alltid områder man kan forbedret. Vi skulle gjerne ha fått implementert de to nye metodene som ble lagt inn i ettertid (samt å testet metodene mot flere andre eksisterende metoder) og fått med det implisitte skjema, men mangel på tid gjorde dette vanskelig. Selv om vi fikk kodet opp deler av koden for de ekstra metodene (og lekt litt med implisitt skjema), var det for lite tid til å lage like gode implementasjoner som vi ellers hadde. Vi tenker det er bedre å implementere mange gode implementasjoner og gå i dybden av disse og handlet deretter.

## 15 Konklusjon

Vi har nå sett at Poisson likningens kan ta mange former og kan til flere forskjellige operasjoner for bildebehandling. Man ser at anvendelsene er mange og dette “bare” ved å sette inn for ulike verdier av  $h$ .

---

<sup>36</sup><https://www.qt.io/qt-for-python>

## Referanser

- [DM97] P. E. Debevec og J. Malik. “Recovering High Dynamic Range Radiance Maps from Photographs”. I: *Proceedings of SIGGRAPH 97*. Computer Graphics Proceedings. 1997, s. 369–378.
- [Gow08] Timothy Gowers. *the princeton companion to mathematics*. 3. utg. Princeton University Press, sep. 2008. ISBN: 0691118809.
- [Sze11] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer; 2011 edition, okt. 2011. ISBN: 1848829345.
- [Hig15] Nicholas J. Higham. *the princeton companion to applied mathematics*. 1. utg. Princeton University Press, sep. 2015. ISBN: 3257227892.
- [Far20] Ivar Farup. “Poisson Image Editing”. I: *NTNU* (2020). URL: <https://git.gvk.idi.ntnu.no/course/imt3881/imt3881-2020-prosjekt>.
- [Geo] Todor Georgiev. “Photoshop Healing Brush:a Tool for Seamless Cloning”. I: (). URL: <https://pdfs.semanticscholar.org/f982/29e953f8073ee7533eca55ab192197a5b6.pdf>.

## A Appendix

### A.1 Kode for å simulere gråtone mosaikk

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

Kodesnutt 15: Simulering av gråtone mosaikk