



IMT3881 - VITENSKAPELIG PROGRAMMERING

## Poisson Image Editing - Prosjektoppgave

---

Anders Kjelsrud  
Casper F. Gulbrandsen  
Kristian Jegerud

### Sammendrag

Emnet IMT3881 Vitenskapelig Programmering kombinerer realfaglige teknikker og høynivå programmering, og gir studentene et innblikk i hvordan dette kan benyttes til å løse hverdagslige problemsstiller[1]. Et godt eksempel på dette er bildebehandling, hvor numeriske løsninger av differensielllikninger og ikke-lineære algebraiske likninger sammen med biblioteker fra Python kan benyttes til å behandle bilder.

Oppgaven vår har vært å implementere ulike anvendelser for dette, både for fargebilder og gråtonebilder. Vi har også skrevet et lite program med grafisk brukergrensesnitt (GUI) som viser eksempler på anvendelsene vi har implementert. Ved å lese denne rapporten får du innsikt i hvordan vi har jobbet, hvilke teknikker som er brukt og hvordan disse er benyttet. All kildekoden ligger tilgjengelig på GitLab[2].

---

## Forord

Prosjektgruppen vår består av tre studenter fra dataingeniørstudiet ved NTNU Gjøvik. I løpet av semesteret har vi blitt introdusert for matematiske teknikker for numeriske løsninger av bestemte integral, ordinære og partielle differentialslikninger, og systemer av disse, samt ikkelineære algebraiske likninger. Vi har også fått lære om høynivå programmering for lineær algebra, optimalisering, bildebehandling og maskinlæring. Oppgaven som ble gitt hadde som hensikt at vi skulle benytte oss av alt vi har lært i løpet av dette semesteret. Tross mange utfordringer har vi fått et enormt læringsutbytte i form av teoretisk kunnskap og praktiske ferdigheter.

De to første årene av dataingeniørstudiet består utelukkende av obligatoriske fag. Derfor har gruppens medlemmer stort sett de samme forkunnskapene og forutsetningene før prosjektets start. En fordel med at vi er en gruppe på 3 at vi møter utfordringer med ulike tilnærminger, og nettopp dette har vært essensielt for å løse problemstillingene som er blitt gitt på en best mulig måte.

Vitenskapelig Programmering er det første emnet vi deltar i hvor Python benyttes som programmeringsspråk, og vi på gruppa hadde derfor liten erfaring med Python før dette semesteret. Før prosjektets start har vi benyttet det i alle arbeidskrav som er blitt gitt i emnet. Dette har gitt oss en viss kjennskap til hvordan Python fungerer og hvilke fordeler og ulemper dette medfører. Vår største utfordring med Python har vært hvordan vi skal strukturere et såpass stort prosjekt med mange ulike filtyper. Ingen av gruppens medlemmer har tidligere erfaring med å utvikle stort annet enn applikasjoner som kjører i kommandolinje. Dette har ført til en bratt læringskurve og mye nyttig læring.

Rapporten er skrevet med LaTeX, et språk vi har brukt på et mindre prosjekt tidligere på studiet. Når vi har skrevet rapporten har det ikke gitt oss noen nevneverdig utfordringer. Det har derimot gitt oss utfordringer når rapporten måtte holdes lagret sammen med kildekoden i GitLab. Vi slet særlig når vi skulle finne en god editor for å jobbe på de ulike rapportfilene samtidig som det var en løsning som vi synes var effektiv. Vi endte opp med å skrive rapporten i Overleaf<sup>1</sup> og pushe endringer til repositoriet med jevne mellomrom, slik at det skal være mulig å følge utviklingen i ettermiddag.

## Git Repository

Progresjonen på prosjektet ligger i git-repositoriet på GitLab:

<https://git.gvk.idi.ntnu.no/casperfg/imt3881-2020-prosjekt>

---

<sup>1</sup><https://www.overleaf.com>

---

# Innhold

<b>Figurer</b>	<b>iii</b>
<b>1 Introduksjon</b>	<b>1</b>
<b>2 Glatting</b>	<b>2</b>
<b>3 Inpainting</b>	<b>4</b>
<b>4 Kontrastforsterkning</b>	<b>5</b>
<b>5 Demosaicing</b>	<b>7</b>
<b>6 Sømløs kloning</b>	<b>9</b>
<b>7 Konvertering av fargebilder til gråtone</b>	<b>10</b>
<b>8 Anonymisering av bilder med ansikter</b>	<b>11</b>
<b>9 GUI</b>	<b>13</b>
<b>10 Konklusjon</b>	<b>18</b>
<b>Bibliografi</b>	<b>20</b>
<b>Appendiks</b>	<b>22</b>
<b>A Arbeidsflyt og prosjektorganisering</b>	<b>22</b>

---

## Figurer

1	Glatting	3
2	Implisitt glatting med $\alpha = 2, n = 3$	3
3	NVIDIAs inpaintingalgoritme	5
4	Inpainting av tre	6
5	Graylevel Histogram	6
6	Kontrastforsterkning	7
7	Bayer filter	8
8	Demosaicing	9
9	Sømløs kloning	10
10	Konvertering til gråtone	11
11	Haar Feature	12
12	Haar Feature demonstrert på et ansikt	12
13	Eksempel på anonymisering	13
14	Grafisk brukergrensesnitt - Glatting	15
15	Gitlab issue board	22
16	Arbeidsflyten gjennom ukedagene	22

---

# 1 Introduksjon

## 1.1 Rapportens oppbygging

Vi har valgt å følge rapportmalen til Jon Arnt Kårstad fra institutt for marin teknikk ved NTNU. Denne malen ligger klar til bruk som mal i den nettbaserte LaTeX-editoren Overleaf. Når vi i rapporten skal vise til andre kapitler og seksjoner vil det bli brukt kapittelets og seksjonens nummer. For listformer som f.eks. punktlister vil vi bruke terminologien punkt. Vedlegg blir lagt bakerst i rapporten og vil bli referert til med vedleggets bokstav og navn. Rapporten har følgende oppbygning:

- Introduksjon - beskriver raskt omstendighetene rundt prosjektet. Gir også en kort beskrivelse av gruppen og rapporten.
- Oppgaven - beskriver oppgaven som er gitt, hva som er forventet og hvordan det kan løses.
- Løste oppgaver - viser hvordan vi har løst oppgavene, hvordan vi har tenkt og hvilke forutsetninger som er gjort
- Avslutning - refleksjoner rundt prosjektet blir gjort. Det blir også presentert diskusjoner og evaluering som gjort underveis.
- Konklusjon - inneholder avsluttende tanker, hva som fungerte og hva som kunne vært gjort annerledes.
- Bibliografi - inneholder alle kilder vi har benyttet i løpet av prosjektets periode.
- Vedlegg - inneholder kode som vi ønsket å forklare nærmere.

## 1.2 Oppgaven

En rekke problemer i bildebehandling kan løses med en teknikk som kalles «Poisson Image Editing». Metoden går i korthet ut på at man representerer bildet man ønsker å komme frem til som en funksjon  $u : \Omega \rightarrow C$ , der  $\Omega \subset \mathbb{R}^2$  er det rektangulære området hvor bildet er definert, og  $C$  er fargerommet, vanligvis  $C = [0, 1]$  for gråtonebilder og  $C = [0, 1]^3$  for fargebilder. Bildet  $u(x, y)$  fremkommer som en løsning av Poisson-ligningen

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

der randverdier på  $\partial\Omega$  og funksjonen  $h : \Omega \rightarrow \mathbb{R}^{\dim(C)}$  spesifiseres avhengig av hvilket problem som skal løses. Randverdiene er av Dirichlet- eller Neumann-typen.

En måte å løse Poisson-ligningen på er å iterere seg frem til løsningen vha. såkalt gradientnedstigning («gradient descent»). I praksis gjøres dette ved å innføre en kunstig tidsparameter og la løsningen utvikle seg mot konvergens:

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

Når man velger denne fremgangsmåten, må man også velge en initialverdi for bildet,  $u(x, y, 0) = u_0(x, y)$ .

To diskrete numeriske skjemaer for (2) kan finnes ved henholdsvis eksplisitt og implisitt tidsintegrasjon og sentrerte differanser for de partielle deriverte:

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

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

---

Fra dette skulle vi implementere en rekke anvendelser på bilder som for hver anvendelse blir forklart nøyere i denne rapporten. Oppgaveteksten inneholdt noen eksempler og generelle forklaringer på hvordan man kan implementere anvendelsene, men vi fikk også frihet til å inkludere egne anvendelser.

Det ble definert en minimumsløsning som skulle inneholde implementasjon av det eksplisitte skjemaet (3) og anvendelsene glatting (avsnitt 2), inpainting (avsnitt 3), kontrastforsterkning for gråtonebilder (avsnitt 4) og sømløs kloning (avsnitt 6). Ut over dette har vi også implementert demosaicing (avsnitt 5), en mer avansert konvertering av fargebilder til gråtone (avsnitt 7) og anonymisering av ansikter i bilder (avsnitt 8). I tillegg har vi implementert det implisitte numeriske skjemaet og laget et grafisk brukergrensesnitt (avsnitt 9). Disse blir diskutert i detalj senere i rapporten. Alle bilder hvor det er avbildet mennesker er hentet fra <https://unsplash.com/>.

## 2 Glatting

### 2.1 Bakgrunn

Glatting, også kalt Gaussian Blur[3], er en veldig mye brukt teknikk innenfor bildebehandling. Man glatter et bilde for å fjerne støy og redusere detaljene i bildet og kan sammenlignes med effekten man får av å se på et bilde gjennom en skjerm som er nesten helt gjennomsiktig. Metoden går essensielt ut på å erstatte verdien i hver piksel med en vektet sum som følger normalfordelingen av pikslene rundt[4].

### 2.2 Implementasjon

#### Eksplisitt

Det eksplisitte skjemaet gitt ved likning (3) kan løses ved at man tar man utgangspunkt i originalbilde  $u_0$  og itererer dette med likning (2) i hele  $\Omega$ . Bildet `im` leses derfor inn ved bruk av `imageio.imread`<sup>2</sup>, før det legges litt støy på bildet.

```
im = im + .05 * np.random.randn(* np.shape(im))
```

Bildet `im` sendes til `eksplisittGlatting()` sammen med originalbilde `orig_im` og en konstant `k`. Her blir det i for-løkka som itereres 20 ganger funnet en laplace for bildet `image`

```
laplace = (image[0:-2, 1:-1] +
           image[2:, 1:-1] +
           image[1:-1, 0:-2] +
           image[1:-1, 2:] -
           4 * image[1:-1, 1:-1])
```

For å ha mer kontroll over hvor mye bildet glattes benyttes “data attachement”. Dette vil si at  $h$  settes lik  $\lambda(u - u_0)$ .

```
h = k * delta_t * (image[1:-1, 1:-1] - orig_im[1:-1, 1:-1])
```

Da kan det ved å justere parameteren  $\lambda$  styres hvor glatt bildet maksimalt kan bli selv om det iteres i det uendelige. Som randverdier brukes Neumann med  $\frac{\partial u}{\partial \nu} = 0$  på  $\partial\Omega$ . Når en itererer likning (2) vil noen verdier kunne gå ut av lovlig intervall, det vil si over 1 eller under 0. Derfor er siste ledd i før-løkka å kippe verdiene i `image` til et lovlig intervall.

```
image[image < 0] = 0
image[image > 1] = 1
```

Til slutt returnes det glattede bildet som vist i Figur 1

---

<sup>2</sup><https://pypi.org/project/imageio/>



Figur 1: Glatting



Figur 2: Implisitt glatting med  $\alpha = 2, n = 3$

## Implisitt

For å implementere det implisitte skjemaet gitt ved (4) brukes “sparse matrices”<sup>3</sup>. Her er de aller fleste elementene i matrisa 0, bortsett fra noen utvalgte diagonaler. I dette tilfellet får vi en tridiagonal matrise, pluss to ekstra diagonaler. Dersom  $u$  er bildet som leses inn, befinner de siste to diagonalene seg henholdsvis  $u.\text{shape}[1]$  til høyre og venstre for senterdagonalen. Ved å omskrive (4) og sette  $\frac{\Delta t}{\Delta x^2} = \alpha$  får vi med  $h=0$ :

$$-\alpha u_{i+1,j}^{n+1} - \alpha u_{i-1,j}^{n+1} + (1 + 4\alpha)u_{i,j}^{n+1} - \alpha u_{i,j+1}^{n+1} - \alpha u_{i,j-1}^{n+1} = u_{i,j}^n. \quad (5)$$

For å skrive dette på matriseform må en altså bruke glisne matriser. Her blir hovedjobben å lage nevnte diagonaler.

```
size=u.shape[0]*u.shape[1]
nupper = np.concatenate((np.zeros(u.shape[1]),
                        -alpha * np.ones(size - u.shape[1])))
upper = np.concatenate(([0, 0],
```

<sup>3</sup><https://docs.scipy.org/doc/scipy/reference/sparse.html>

---

```

        -alpha * np.ones(size - 2)))
center = np.concatenate(([1],
                      (1 + 4 * alpha) * np.ones(size - 2),
                      [1]))
lower = np.concatenate((-alpha * np.ones(size - 2),
                       [0, 0]))
nlower = np.concatenate((-alpha * np.ones(size - u.shape[1]),
                         np.zeros(u.shape[1])))
diags = np.array([nupper, upper, center, lower, nlower])
A = spdiags(diags, [u.shape[1], 1, 0, -1, -u.shape[1]], size, size).
tocsc()

```

`nupper` og `nlower` posisjoneres riktig utfra bildets dimensjoner, mens de tre midterste diagonalene plasseres i midten. Deretter lages en array `diags` av alle diagonalene. Til slutt settes de sammen til en glissen matrise ved hjelp av `spdiags()`. I tillegg gjøres de kolonnesentrert med `tocsc()`. Denne glissne koeffisientmatrisen `A` brukes videre i `spsolve`<sup>4</sup> som dersom `u` og `im` er fargebilder, løser det lineære likningssystemet for hver fargekanal.

```

for i in range (u.shape[2]):
    im[:, :, i]=spsolve(A,im[:, :, i].flatten()).reshape(u[:, :, i].shape)

```

`flatten()` brukes her for å lage en  $[1, m \times n]$  - vektor, der størrelsen på bildet `u` er  $m \times n$ . Ved å bruke et implisitt skjema kontra et eksplisitt, vil en være mindre utsatt for numerisk ustabilitet. Man kan dermed benytte seg av færre iterasjoner og større tidsskritt. I vår implementasjon derimot, opplevde vi at kjøretiden ble langt lengre med implisitt skjema, og de fleste anvendelser benytter derfor det eksplisitte. Figur 2 viser glatting med implisitt skjema. Her er det brukt 3 iterasjoner,  $\alpha=2$  og  $h=0$

## 3 Inpainting

### 3.1 Bakgrunn

Inpainting er en utbredt teknikk innenfor bildebehandling. Teknikken kan brukes til å fylle inn manglende informasjon eller fjerne uønsket støy eller elementer i bilder [5]. Selvom kunsten å reparere gamle og forfalne bilder har eksistert i mange år, har det nylig opplevd en økning i popularitet grunnet den raske utviklingen av bildebehandlingsteknikker. Bilder er for tiden en av de mest vanlige formene for informasjon. Derfor er teknikker som kan endre bilder uten spor blitt et sikkerhetsproblem. Et eksempel på dette er den svært utbredte delingen av personlige bilder i sosiale medier. Disse kan inneholde objekter som lett kan fjernes for å drastisk endre semantikken i hele bildet. Litteraturen som beskriver fjerning av objekter deles gjerne inn i to kategorier: inpainting og kloning. Kloning blir diskutert i detalj i (6). Tidligere ble inpainting i større grad brukt på gamle bilder for å fjerne riper o.l. I nyere tid brukes det mer for å fjerne gjenstander som har blitt lagt til i bilder. I tillegg brukes inpainting til å fjerne forstyrrelser som linjer, flekker og tekst [6].

### 3.2 Teori

Metoden går ut på å fylle informasjon i et ønsket området ut fra informasjon i det omkringliggende området. Dette gjøres ved å sette  $h = 0$  i  $\Omega_i$  og løse likning (3) i  $\Omega_i$ , dersom  $\Omega_i \subset \Omega$  er det området hvor inpaintingen skal utføres. Implementasjonen i dette prosjektet krever at en fornuftig maske velges for at resultatet skal oppleves tilfredsstillende. NVIDIA har implementert en algoritme som bruker deep learning [7], hvor tilsynelatende ødelagte bilder kan gjenopprettes til meget høy kvalitet [8], illustrert ved Figur 3. I vår implementasjon er man i større grad avhengig av at masken befinner seg på et gunstig sted slik at ikke viktig informasjon tildekkes.

---

<sup>4</sup><https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.sparse.linalg.spsolve.html>



Figur 3: NVIDIAs inpaintingalgoritme

Kilde: NVIDIA [9]

### 3.3 Implementasjon

For implementasjon i Python konstrueres en maske i form av en bool-array som er sann for alle piksler innenfor målområdet  $\Omega_i$ , og usann utenfor.

```
mask = np.zeros(im.shape)      #lag maske
mask[135:140, 250:380] = 1
mask[250:470, 300:305] = 1
mask[740:755, 550:700] = 1
mask = mask.astype(bool)      #lag bool-array
```

Deretter bruker man denne arrayen som en ”view”<sup>5</sup> på array-representasjonen av bildet for å løse diffusjonslikningen (3) innenfor området  $\Omega_i$ , før man sender med array-representasjonen og masken til funksjonen som løser diffusjonslikningen eksplisitt med dirichlet randbetingelser. For hver iterering blir området innenfor målområdet satt til å være likt tilsvarende område i originalbildet.

```
im[~mask] = im0[~mask]
```

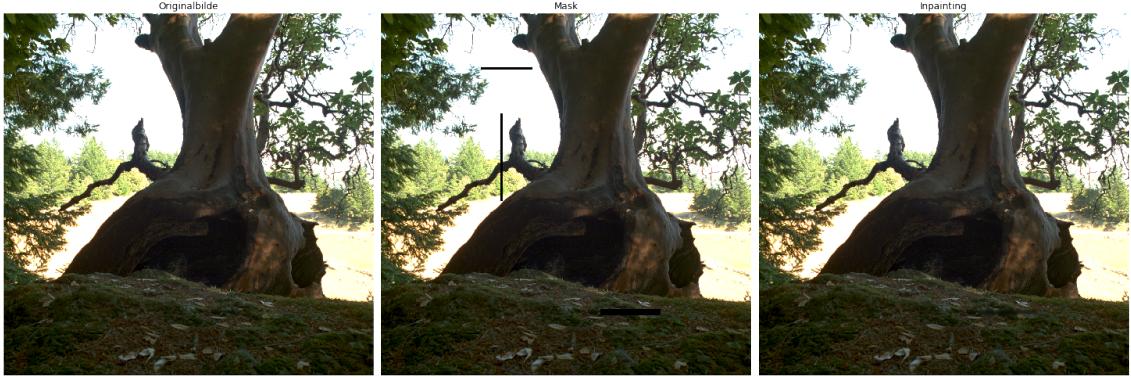
Tilslutt returneres originalbildet og det ferdig inpaintede bildet fra funksjonen som løste diffusjonslikningene. Figur 4 viser resultatet av vår inpaintingalgoritme, her med  $n=100$  iterasjoner. Her ble masken definert som 3 rette svarte klosser. Etter inpainting er fullført er de tilsvarende borte.

## 4 Kontrastforsterkning

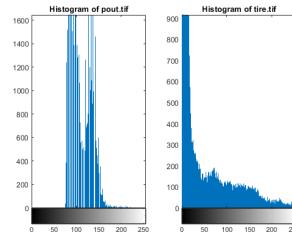
### 4.1 Bakgrunn

I et bilde er det kontrasten som gjør at vi kan skille et objekt fra bakgrunnen og eventuelle andre objekter.[10] Et godt eksempel på dette er svart tekst på hvit bakgrunn, hvor kontrasten mellom bokstavene og bakgrunnen er veldig stor. Dette gjør teksten lett å se og lese. I et bilde er det fargekontrasten som gjør at vi kan se objekter på et bilde. Et med bilde med høy motivkontrast[11] viser de lyse delene av bildet enda lysere og de mørke delene av bildet enda mørkere. Dette gjør at bildet ser mindre flatt ut for seeren, men gjør i realiteten at det blir mindre detaljer i bildet. Spesielt i de mørkeste og lyseste områdene blir bildene udetaljerte og støyete. Når man skal behandle et bilde vil det være veldig lett å øke kontrastene, mens muligheten for å redusere kontrasten vil være veldig begrenset.

<sup>5</sup><https://docs.scipy.org/doc/numpy/user/basics.indexing.html>



Figur 4: Inpainting av tre



Figur 5: Graylevel Histogram

Kilde: MathWorks [13]

De fleste metodene for å gjøre en kontrastforsterkning benytter seg av et gråtonehistogram med verdier fra 0 til 255. Et gråtonehistogram beskriver ikke hvor de ulike verdiene finnes på bildet, men beskriver heller fordelingen av grånivåene i bildet. Et slikt diagram opprettes ved å telle antall ganger hver en grånivåverdi i bildet oppstår. Deretter deler man med det totale antall piksler i bildet slik at man får en prosentvis andel av hver grånivåverdi. En måte å omdanne et lavkontrastbilde til et høykontrastbilde er ved å gjøre en dynamic range adjustmen(DRA)[12]. Dette gjøres ved å strekke grånivåverdiene i bildet slik at histogrammet dekker alle grånivåene mellom 0 og 255, som vist i figur (5).

## 4.2 Implementasjon

For å finne en mer kontrastert versjon av originalbildet  $u_0$  finner vi et bilde som har samme gradient som originalbildet, men forsterket med en konstant  $k$  som er større enn 1. Dette gjør vi ved å ta utgangspunkt i likning (2), og setter  $h = k\nabla^2 u_0$ . Vi har nå denne likningen

$$\frac{\partial u}{\partial t} = \nabla^2 u - k\nabla^2 u_0. \quad (6)$$

som løses for  $u$ . Først finner vi et eksplisitt skjema for hver av tre fargekanalene i  $u_0$ , før vi for hver av de tre fargekanalene itererer oss frem til en løsning for  $u$ .

```

u[1:-1, 1:-1, 0] += (0.25 * (u[:-2, 1:-1, 0] + u[2:, 1:-1, 0] +
u[1:-1, :-2, 0] + u[1:-1, 2:, 0] -
4 * u[1:-1, 1:-1, 0]) - k * u0[:, :, 0])

u[1:-1, 1:-1, 1] += (0.25 * (u[:-2, 1:-1, 1] + u[2:, 1:-1, 1] +
u[1:-1, :-2, 1] + u[1:-1, 2:, 1] -
4 * u[1:-1, 1:-1, 1]) - k * u0[:, :, 1])

u[1:-1, 1:-1, 2] += (0.25 * (u[:-2, 1:-1, 2] + u[2:, 1:-1, 2] +
u[1:-1, :-2, 2] + u[1:-1, 2:, 2] -
4 * u[1:-1, 1:-1, 2]) - k * u0[:, :, 2])

```



Figur 6: Kontrastforsterkning

$$u[1:-1, :-2, 2] + u[1:-1, 2:, 2] - \\ 4 * u[1:-1, 1:-1, 2]) - k * u0[:, :, 2])$$

For randbetingelser har vi valgt Neumann,  $\partial u / \partial n = \partial u_0 / \partial n$ . Når man multipliserer  $u_0$  med  $k$  vil flere pikselverdier  $u > 1$  eller  $u < 0$ , som er utenfor fargeområdet. Derfor må vi klippe verdiene til et lovlig intervaller, som gjøres før  $u$  returneres av funksjonen.

$$\begin{aligned} u[u < 0] &= 0 \\ u[u > 1] &= 1 \end{aligned}$$

Resultatet av vår implementasjon vises i figur 6. Til venstre vises originalbildet slik det blir lastet inn ved hjelp av `imageio.imread`<sup>6</sup>. Til høyre vises bildet som er kontrastforsterket med  $k = 1.9$ .

## 5 Demosaicing

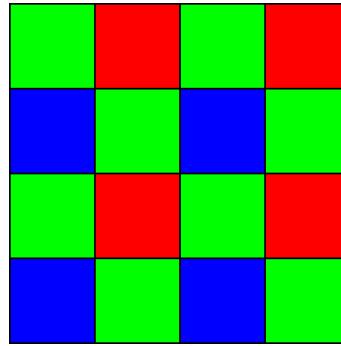
### 5.1 Bakgrunn

En både økonomisk og praktisk måte å fange opp primærfarger på er å plassere et fargefilter over fargesensorene i et kamera. Det finnes ulike filtre, men det mest populære er Bayer-filteret [14], illustrert ved Figur 5. Ved å se på sensoren som en matrise og legge på ulike fargefiltre på de individuelle elementene kan en samle informasjon ved å la de respektive pikslene fange lysstyrken i sitt område. Denne informasjonen er tilstrekkelig til å gjøre veldig nøyaktige gjett om den ekte fargen i dette området. Nettopp denne metoden som ser på omkringliggende piksler for å estimere verdier, kalles interpolasjon<sup>7</sup>. Bayer-filteret er et mønster som alternerer mellom en rad av grønne og røde piksler, og en rad av blå og grønne piksler (7). Fra dette ser man enkelt at det finnes dobbelt så mange grønne piksler, som det gjør røde og blå. Dette er fordi fargereseptorene som sitter på netthinnen i menneskeøyet er mer følsomme for grønt lys. På netthinnen finnes omrent 6 millioner tapper<sup>8</sup> av typene S, L og M-tapper. S-tappene registrerer fiolett, altså blandinger mellom rød og blå [16]. Derimot registrerer både L- og M-tapper grønt, med mest følsomhet rundt bølgelengder tilsvarende 534-564nm [17]. Dermed trenger en mer informasjon fra grønne piksler for å produsere et bilde som menneskeøyne vil kunne oppfatte som naturtro.

<sup>6</sup><https://pypi.org/project/imageio/>

<sup>7</sup><https://en.wikipedia.org/w/index.php?title=Interpolation&oldid=949792579>

<sup>8</sup>[https://en.wikipedia.org/w/index.php?title=Cone\\_cell&oldid=949785199](https://en.wikipedia.org/w/index.php?title=Cone_cell&oldid=949785199)



Figur 7: Bayer filter

Kilde: Wikipedia [15]

## 5.2 Implementasjon

Fra kamerasensoren kommer en gråtonemosaikk, som i praksis altså er mengden lys i hver av de tre fargekanalene, R, G og B. For implementasjon i Python kan man simulere denne mosaikken ved å betrakte et bilde `im` representert ved en  $M \times N \times 3$  numpy array. Da kan gråtonemosaikken lages slik:

```
mosaic = np.zeros(im.shape[:2])          #Allocates space
mosaic[::2, ::2] = im[::2, ::2, 0]      #R
mosaic[1::2, ::2] = im[1::2, ::2, 1]    #G
mosaic[::2, 1::2] = im[::2, 1::2, 1]    #G
mosaic[1::2, 1::2] = im[1::2, 1::2, 2]  #B
```

Videre kan en flytte informasjonen fra mosaikken inn i det ønskede fargebildet. Dette gjøres enkelt ved og for hver dimensjon i det nye bildet, fylle inn med informasjon fra hver av dimensjonene R, G og B fra mosaikken. Også her må man ta høyde for at grønne piksler forekommer dobbelt så ofte som de andre [14]. Videre kan en bruke inpainting beskrevet i (3) for å fylle inn den manglende informasjonen. Dette gjøres ved å definere en maske  $\Omega_i$  for hver av fargekanalene. Deretter gjøres et funksjonskall til inpainting-algoritmen for hver dimensjon, der `im_ed` er det resulterende fargebildet:

```
eks.Inpainting_mosaic(im_ed[:, :, 0], mask[:, :, 0])
eks.Inpainting_mosaic(im_ed[:, :, 1], mask[:, :, 1])
eks.Inpainting_mosaic(im_ed[:, :, 2], mask[:, :, 2])
```

Resultatet av vår implementasjon vises i figur 8. Først presenteres originalbildet slik det blir lastet inn ved `imageio.imread`. Deretter vises gråtonemosaikken, før det ferdig interpolerte bildet.



Figur 8: Demosaicing

## 6 Sømløs kloning

### 6.1 Bakgrunn

En populær anvendelse av Poisson-problemet er sømløs kloning. Det går ut på å flytte en partisjon av et bilde over i et annet med den hensikt å minimalisere synligheten av overgangen mellom dem. De fleste profesjonelle bilderedigeringsverktøy har en form for sømløs kloning implementert. Her gjøres operasjonene grafisk ved hjelp av noen museklikk, men i bakgrunnen ligger altså en kode ikke altfor ulik den som er implementert her.

Problemet kan formuleres som et Poisson-problem dersom  $u_0$  er originalbildet det skal klones inn i, og  $u_1$  bildet utsnittet skal hentes fra. Problemstillingen blir da å finne  $u$  slik at  $u = u_0$  utenfor masken  $\Omega_i$  og  $\nabla^2 u = \nabla^2 u_1$  i  $\Omega_i$ . Dette kan løses ved Poisson-likningen (2) ved å sette  $h = \nabla^2 u_1$ , med Dirichlet randbetingelser:  $u = u_0$  på  $\partial\Omega_i$ .

### 6.2 Implementasjon

Først må en velge to bilder hvor deler av det ene skal flyttes inn i det andre. Deretter må en lage en partisjon av bildet man vil lime inn deler av. Dette har vi implementert ved å definere utsnittet som et rektangel. Deretter hentes `laplace1` fra  $u_1$  før (2) løses med `h=laplace1`:

```
for i in range(1000):
    laplace2 = (im_ed[0:-2, 1:-1] +
                im_ed[2:, 1:-1] +
                im_ed[1:-1, 0:-2] +
                im_ed[1:-1, 2:] -
                4 * im_ed[1:-1, 1:-1])
    im_ed[1:-1, 1:-1] += alpha*(laplace2-laplace1) # Los diffusjonslikningen
```

Resultatet ved å velge et område på  $u_0$  som samsvarer noenlunde med fargene og objektene fra  $u_1$  vises i Figur 9. Her fører de sømløse overgangene mellom de to bildene til at det er vanskelig å se hvor skillet går med det blotte øyet.



Figur 9: Sømløs kloning

## 7 Konvertering av fargebilder til gråtone

### 7.1 Bakgrunn

Et gråskalabilde har en verdi i hver piksel som angir mengden lys som den opplever. Gråskalabilder består utelukkende av ulike nyanser av grått, hvor lavest intensitet nærmer seg svart og større verdier går mot hvitt [18]. Disse skiller seg altså fra rene svart-hvitt bilder som i motsetning bare inneholder nettopp disse to fargene. En vanlig måte å konvertere fargebilder til gråtone er å ta et veiet gjennomsnitt av fargekanalene. Ulempen med denne enkle metoden er at informasjon som hovedsakelig har lik lyshet, men skiller ved at kanter, teksturer og andre detaljer har ulik fargetone eller metning, lett kan forsvinne i konverteringen [19].

En bedre teknikk er å konstruere et gråtonebilde med så lik lokal variasjon som det originale fargebildet som mulig.

### 7.2 Implementasjon

Dette gjøres i Python ved å først konstruere en ny gradient  $\mathbf{g}$  med lengde  $||\nabla u_0||/\sqrt{3}$  og retning  $\nabla(u_{0R} + u_{0G} + u_{0B})$ , for deretter å løse (2) med  $\mathbf{h}=\nabla \cdot \mathbf{g}$ .

Som initialverdi brukes det veide gjennomsnittet fra den enkle metoden:

```
u0 = np.sum(picture.astype(float),2)/(3*255)
```

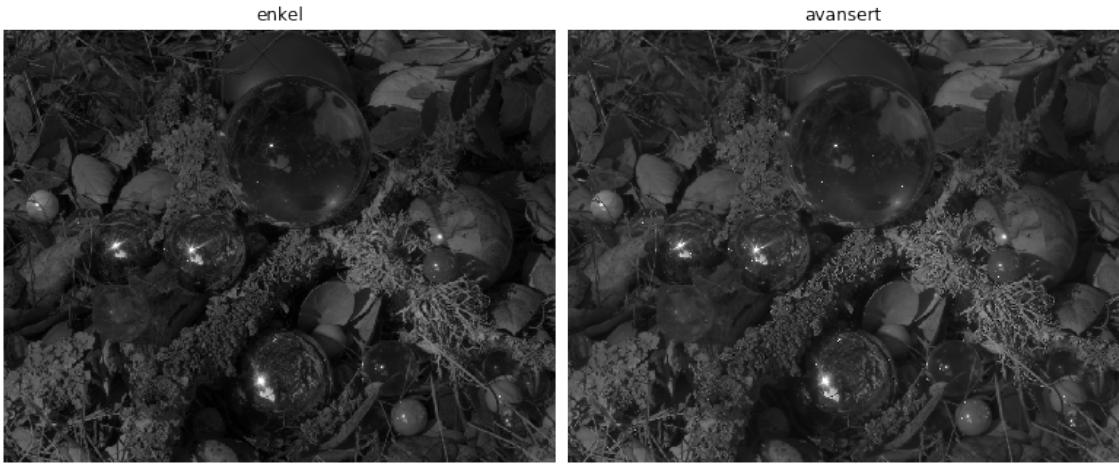
Videre lages den nye gradienten og dens retning slik

```
gradient = abs(finnLaplace(u0))/np.sqrt(3)
retning = finnLaplace(orig_im[:, :, 0] + orig_im[:, :, 1] + orig_im[:, :, 2])
```

Deretter gjenstår bare å løse (2) med modifisert  $\mathbf{h}$ :

```
u0[1:-1, 1:-1] += alpha * laplace - retning*gradient
```

Figur 10 viser de to ulike teknikkene.



Figur 10: Konvertering til gråtone

## 8 Anonymisering av bilder med ansikter

### 8.1 Bakgrunn

Anonymisering av bilder er enkelte ganger nødvendig før det blir offentlig vist frem for å skjerme den avbildedes personvern. En teknikk som ofte blir brukt er å gjøre ansiktene uskarpe og resten av bildet blir skarpt. Teknikken avhenger av at programmet klarer å gjenkjenne ansiktene[20] for å definere en maske rundt, slik at man kan isolere anvendingen av bildet kun på ansiktet, og i likhet som tidligere er det flere teknikker som benyttes. I denne oppgaven brukte vi biblioteket OpenCV (Open Source Computer Vision Library) og maskinlæringsalgoritmen Haar Cascade.

#### Open CV

Open CV er et open source datasyn-[21] og maskinlæringsbibliotek som ble utviklet for å skape et felles infrastruktur for datasynsapplikasjoner og for å akselerere bruken av maskinoppfatning i kommersielle produkter[22]. Kort fortalt betyr dette datamaskinenes evne til å tolke data på en måte som ligner på hvordan mennesker bruker sansene til å oppfatte verden rundt seg[23], og da spesielt synet i dette tilfellet.

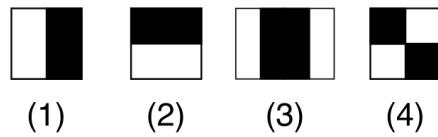
#### Haar Cascade algoritmen

Haar Cascade er en maskinlæringsalgoritme [24] som brukes til å identifisere objekter i et bilde eller i en video. Den er mest kjent til å identifisere ansikter, men kan trenes til å identifisere nesten hva det måtte være av objekter.

Algoritmen trenger å trenes med flere positive bilder av ansikter og negative bilder uten ansikter, deretter må man trekke ut kjente egenskaper fra disse bildene. Første steget er å samle inn Haar-features.

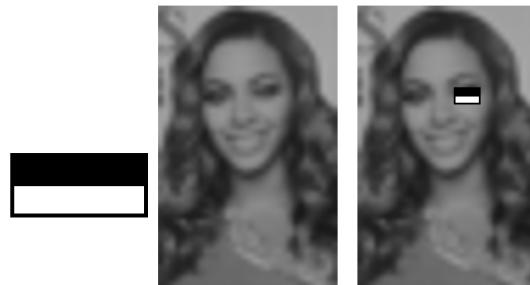
De to første elementene (figur 11) kalles for edge-features som benyttes til å detektere kanter i objekter. Den tredje er en line-feature, mens den fjerde heter four rectangle-feature som oftest benyttes til å gjengjennie en skrå linje.

Haar-Features er veldig god til å gjengjennie linjer og kanter på objekter, og spesielt i ansikter. Som man ser i (figur 12) klarer en edge-feature å gjengjennie essensielle deler ved et ansikt, som f.eks et øye. Grunnen til dette er at i bildet blir det en kontrast mellom øyet og kinnet, som kan minne om objekt 2 fra figur (11), altså en mørk overdel med lys bunn. Algoritmen går så gjennom hele



Figur 11: Haar Feature

Kilde: Wikimedia Commons [26]



Figur 12: Haar Feature demonstrert på et ansikt

Kilde: Wikimedia Commons [27]

bildet helt til det oppdager en samling av slike features” og ved nok positive treff markerer den det område over ansiktet.

Algoritmen trenger to parametere som heter scale factor og minimum neighbor

## 8.2 Implementasjon

Det mest krevende var nettopp å få programmet til å gjenkjenne ansikter. Viktigste var å importere xml-filen<sup>9</sup> som inneholdt en ferdig trennt algoritme for gjenkjenning av ansikter som er vendt direkte mot kamera. Valg av parameterne måtte tas basert på bildet.

```
def detectFace(file, scaleFactor = 1.2, minNeighbors = 5):
    image = cv2.imread(file)                      # Leser inn bilde
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # Konverterer til RGB

    # Importerer haarscade biblioteket
    face_cascade = cv2.CascadeClassifier('Resources/
haarcascade_frontalface_default.xml')
    faces = face_cascade.detectMultiScale(image, scaleFactor,
                                           minNeighbors, minSize = (30,30))
    # Antall ansikter
    faces_rects = face_cascade.detectMultiScale(image, scaleFactor,
minNeighbors)
    for (x,y,w,h) in faces_rects:      # For hvert oppdagede ansikt tegner
    rektangel rundt
        cv2.rectangle(image, (x, y), (x+w, y+h), (0, 255, 0), 2)
```

<sup>9</sup>haarcascade\_frontalface\_default.xml[25]



Figur 13: Eksempel på anonymisering

Kilde: Wikimedia Commons [27]

## 9 GUI

### 9.1 Bakgrunn

GUI, graphical user interface, er et grensesnitt som lar brukerne samhandle med et dataprogram grafisk, vanligvis ved hjelp av tastatur og datamus. I motsetning til typiske shellapplikasjoner hvor man navigerer ved hjelp av kommandoer, består et grafiske grensesnittet ofte av grafiske ikoner og lydindikatorer.[28] Grunnet den bratte læringskurven på en kommandobasert datamaskin med shellapplikasjoner ble grafisk brukergrensesnitt med mulighet for å bruke mus til å klikke på knapper, tekstfelt og informasjonsfelt raskt populært. Hos de to mest populære operativsystemene på markedet i dag, Windows og MacOS, er det GUI-et står i sentrum, mens det i UNIX er det mer en blanding av et kommandobasert system og grafisk brukergrensesnitt.

Vi ønsket at det skulle være lettere å benytte seg våre implementasjoner nevnt fra kapittel 2 - 8, og bestemte oss for å lage vårt eget program med et grafisk brukergrensesnitt. Her skal det være mulighet for å utføre alle de implementerte operasjonene interaktiv på ulike utvalgte bilder. Vi endte opp med å bruke det foreslalte rammeverket **PyQt5**<sup>10</sup>, som gjør det enkelt å bruke implementasjonene skrevet vi allerede hadde skrevet med Python i GUI-applikasjonen. Ved å skrive programmet i PyQt5 kan man kjøre programmet på både Windows, MacOS og Linux, noe som gjør at vi forminsker arbeidsmengden betraktelig.

### 9.2 Implementasjon

Programmet kjøres fra `Main.py` hvor main ligger. Vi har valgt en vindusbasert applikasjon hvor man fra hjemskjermen kan velge hvilken implementasjon man har lyst til å benytte seg av. Vi har funnet det hensiktsmessig å dele applikasjonen opp i ulike moduler, hvor det er en modul for hjemskjermen og en modul for hver av implementasjonene. Hver modul har sin egen .py-fil i mappa `src/GUI` hvor all kode er lagret. Dette har vi gjort for god organisering ved å skille de ulike modulene i applikasjonen fra hverandre. Det er også mer oversiktlig når vi importerer de ulike bibliotekene vi har bruk for i den aktuelle modulen.

På grunn av at vi har valgt en vindusbasert applikasjon har vi derfor laget et design for hver modul. Designet til hver modul lagres som en .ui-fil som importeres når en modul initialiseres. Designet er gjort i **Qt5 Designer**<sup>11</sup>, som gjør det enkelt å lage et oversiktlig UI med tekstfelter, knapper og bilder. Alle .ui-filene er lagret i mappa `src/GUI/UI`. Hver gang Main.py lastes tilpasses applikasjonen slik at den oppfører seg likt på skjermer med ulik pikseltetthet.

```
QtWidgets.QApplication.setAttribute(QtCore.Qt.AA_EnableHighDpiScaling, True)  
QtWidgets.QApplication.setAttribute(QtCore.Qt.AA_UseHighDpiPixmaps, True)
```

<sup>10</sup><https://pypi.org/project/PyQt5/>

<sup>11</sup><https://pypi.org/project/PyQt5Designer/>

---

Når en modul initialiseres må vinduets dimensjoner justeres. Dette må gjøres fordi designet er laget med Qt5 Designer, og oppfører seg derfor annerledes på skjermer med ulike dimensjoner. Disse justeringene gjøres ved hjelp av funksjonen `adjustScreen()`. Her får man dimensjonen ved å hente skjermens høyde og bredde, og bruker denne til å justere vinduet. Hvert vindu er forskjellig fra hverandre, og derfor har hver modul sin egen `adjustScreen()` funksjon.

```
def adjustScreen(self):
    screenWidth = app.primaryScreen().size().width()
    screenHeight = app.primaryScreen().size().height()
    if screenWidth/screenHeight == 1.5:
        width = int(screenWidth / 1.8)
        height = int(screenHeight / 2)
    else:
        width = int(screenWidth / 2.22222)
        height = int(screenHeight / 1.95298)
    self.setGeometry(500, 80, width, height)
```

Hjemskjermen består av et antall knapper med tilhørende tekst som forklarer hva den aktuelle implementasjonen gjør. Når man klikker på en knapp opprettes en dialog til denne modulen. Hvis det lykkes med å opprette en dialog vil et nytt vindu åpnes hvor den aktuelle modulens design og funksjonaliteter vises.

Hovedelementet i hver modul er bildet som man ser i (figur 14) Vi har valgt å bruke MatPlotLib til å vise bilder. Dette gjør at vi kan bruke `plt.imshow()`<sup>12</sup> som vi har brukt i hver enkelt implementasjon i Jupyter Notebook-filene. I vår applikasjon er dette en bedre metode å vise bilder på sammenliknet med alternativet hvor vi da ville brukt QLabel og `setPixmap`<sup>13</sup>. Selve bildet er et imagewidget av typen FigureCanvas, og vi har derfor opprettet en egen klasse `imagewidget` i filen `src/imagewidget`. Ved oppstart av hver modul initialiseres imagewidget hvor `self.img` opprettes og initialiseres. Hver gang et nytt bilde skal vises kalles `showImage(self, image, colour=True)`. Her blir det lagt til subplot, aksene fjernes og bildet justeres slik at det fyller mest mulig av bilde-rammen. Avhengig av om colour er True eller False, vises enten et fargebilde eller et gråtonebilde ved bruk av `imshow()`. Det er i hver modul lagret `self.path` som hele tiden er oppdatert med filstien til bilde som er valgt. Denne benyttes når brukeren vil se annet bilde eller ta bort effekten gjort på bildet og vise originalbildet. Sett bort fra `Anonymising.py` har hver modul har også en lokal variabel `self.image` hvor det bildet som vises i applikasjonen ligger lagret som en numpy array. Denne brukes når brukeren vil lagret bildet, hvor den sendes til `FunctionGUI/saveImage`. Her åpnes en `QFileDialog`<sup>14</sup>, som returnerer ønsket navn på filen og filsti til ønsket mappe.

I hver modul importeres klassen `showCode(QMainWindow)`, som viser koden i et nytt vindu for implementasjonen. Koden som vises ligger lagret i en .txt-fil i `src/codes`, som vises ved å bruke funksjonen `FunctionGUI>ShowCode`. Her opprettes det et nytt `QMainWindow`, hvor koden vises i et nullbar tekstfelt.

## Glatting

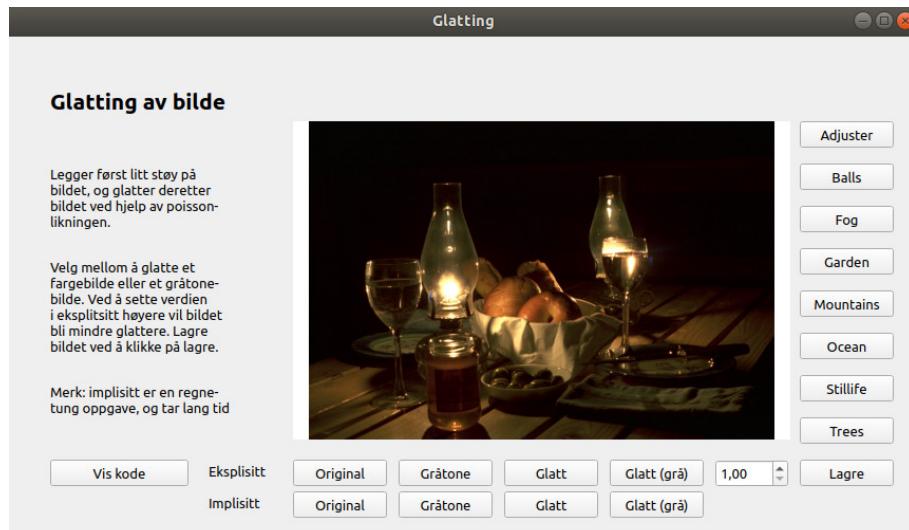
Et bilde kan glattes ved å bruke en eksplisitt løsning av diffusjonslikningen. Dette gjøres av funksjonen `blurImage(self, colour)`. Denne henter først et fargebilde hvis colour er True eller gråtonebilde hvis colour er False. Videre lages en kopi av originalbilde som det legges litt støy på, før dette glattes ved å bruke funksjonen `eksplisittGlatting()` fra `Eksplisitt.py`. Tilslutt vises bildet ved å bruke modulens `showImage`-funksjon.

Bildet kan også glattes ved å bruke en implisitt løsning av diffusjonslikningen. Dette gjøres at funksjonen `blurImageImplsitt(self, colour=True)`. Hvis colour er True leses det inn et fargebilde, og hvis colour er False leses det inn et gråtonebilde ved bruk av `grayscale()` fra `Grayscale.py`. Disse bildene lagres som en numpyarray `u` som videre omformes til å inneholde verdier mellom 0 og 1, hvor verdier over 1 og under 0 klippes til lovlige verdier. Deretter glattes bildet `u` ved å bruke

<sup>12</sup>[https://matplotlib.org/3.2.1/api/\\_as\\_gen/matplotlib.pyplot.imshow.html](https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.imshow.html)

<sup>13</sup><https://doc.qt.io/archives/qt-4.8/qlabel.html#pixmap-prop>

<sup>14</sup><https://doc.qt.io/qt-5/qfiledialog.html>



Figur 14: Grafisk brukergrensesnitt - Glatting

funksjonen `implisitt()` fra `implisitt.py`. Tilslutt vises bildet ved å bruke `showImage`-funksjonen i modulen.

## Inpainting

I Inpainting sitt GUI er det mulig å få vist bildet med manglende informasjon før det fylles inn. Dette gjøres ved å bruke funksjonen `showMask(self)`, som sender med `self.path` og tallet 2 til funksjonen `Inpaint()` fra `Inpainting.py`. Denne funksjonen returnerer bildet med masken, som vises ved å bruke `showImage()` i modulen. Informasjonen i bildet fyller inn i funksjonen `inpaint(self)`, som sender med `self.path` og tallet 3 til `Inpaint()` fra `Inpainting.py`. Her returneres et bilde hvor informasjonen er fylt inn, som vises av `showImage()`.

## Kontrastforsterkning

Når knappen `contrastColour` triggas starter funksjonen `contrastImage(self)`. Denne funksjonen kaller på `contrastEnhance()` fra `contrastEnhancement.py` og sender med `self.path` og den aktuelle verdien på `QDoubleSpinBox`<sup>15</sup> som brukeren har satt en verdi på. `contrastEnhancement()` returnerer et kontrastforsterket bilde som vises i GUI av `showContrastImage(self, im, colour=True)`. Hvis brukeren trigger knappen `contrastOrigGray` startes funksjonen `contrastGrayImage(self)`. Her sendes `self.path` og verdien på `QDoubleSpinBox` med til `contrastEnhanceBW()` i `contrastEnhancement.py`, og returnerer et kontrastforsterket gråtonebilde. Dette vises av `showContrastImage()`.

## Demosaicing

Det er mulig å bruke to forskjellige metoder å gjøre demosaicing på et bilde i denne modulen. Hvis man vil bruke metoden som innebærer at man manuelt fyller inn informasjonen fra gråtonemosaiikken inn i hver fargekanal, opprette en maske for fargekanalene og inpainte den manglende informasjonen med bruk av diffusjonslikningen, kan gjøres dette med `demosaicImage()`. Denne bruker `mosaicToRGB()` til å utføre demosaicingen, som vises på skjermen ved å bruke `showImage()`. Hvis man har lyst til å se gråtonemosaiikken før informasjonen inpaintes gjøres dette av `mosaic()`, hvor gråtonemosaiikken returneres av `getMosaic()` og vises av `showImage()`. Ønsker man heller bruke biblioteket Colour-demosaicing<sup>16</sup> til å gjøre demosaicing av bildet gjøres

<sup>15</sup><https://doc.qt.io/qtforpython/PySide2/QtWidgets/QDoubleSpinBox.html>

<sup>16</sup><https://pypi.org/project/colour-demosaicing/>

---

dette ved å bruke `demosaicImagePackage`. Her brukes `Colour` først til å lage en gråtonemosaikk, før denne gråtonemosaikken demosaices ved å bruke `Colour-demosaicicing`. Hvis man har lyst til å se gråtonemosaikken som lages med dette biblioteket gjøres dette med `mosaicPackage()`. Gråtonemosaikken og bildet som det er gjort demosaicing på vises begge med `showImage()`.

## Sømløs kloning

For å sømløst klone en av de tre forhåndsvagte sammensetningene av bilder bruker `seamlessImage(self, number, img1, img2)`. `Number` er verdien på hvilken av kombinasjonene som klones. Deretter sjekkes verdi på den aktuelle av de tre lokale boolvariablene. Disse boolvariable fungerer som status på om noen av de tre forhåndsvagte kombinasjonene av bilder allerede er regnet ut. Disse settes til `False` ved oppstart av modulen, og oppdateres til `True` hvis den aktuelle kombinasjonen av bilder er blitt klonet. Dette er gjort fordi det å sømløst klone et bilde er en regnet oppgave og tar lang tid, og ved å lagre bildet kan brukeren raskt kan sammenlikne bildet før og etter cloning. Hvis den aktuelle boolverdien er `False` klones bildet av `seamless(img1, img2)` og lagres i `self.seamlessImageOne`, før den aktuelle `self.imgXReady` settes til `True`. Er boolverdien ikke `False` er dette allerede gjort og `showImage()` kan vise det ferdig klonede bildet.

## Konvertering av fargetone til gråtone

Når den enkle metoden for å konvertere et bilde til gråtone skal benyttes bruker `convertGrayEasy()` til å kalle på funksjonen `grayscale`. Her returneres gråtonebildet som sendes til `showImage()` som viser bildet på skjermen. Velges den mer sofistikerte metoden for å konverte et gråtonebilde bruker `convertGrayAdvanced()`. Her returneres det et gråtonebilde fra `rgb2gray()` som vises på skjermen av `showImage()`.

## Anonymisering

Når GUIet til Anonymisering lastes, blir hvert bilde nummerert og hvert bilde blir lastet inn i en variabel. Dette er gjort med hensyn til funksjonalitet for å kunne lagre bildet, og en felles bildevariabel slik som i de andre modulene gjorde det vanskelig grunnet bruk av cv2 til å oppdaget ansikt og anonymisere disse. Hver gang et nytt bilde velges hentes det aktuelle bildes filsti ved bruk av `getPath(self, nr)` og bildet leses inn og vises av `showImage()`. Variabelen `self.number` som sier hvilket bilde som vises blir også oppdatert, og `self.updateCount(self, count)` oppdaterer variablen `self.faceCount` som sier hvor mange ansikt som er blitt oppdaget. `detectFaces(file, scaleFactor, minNeighbours)` bruker til å finne ansikt på et bilde, hvor `self.getPath(self.number)` bruker til å finne filsti som sendes med som `file`. Fra `detectFaces()` returneres antallet ansikter på bildet og et bilde hvor ansiktene er markerte. Bildet vises ved å bruke `showFaces()`. For å anonymisere ansiktene bruker `anonymiseFaces()` som bruker `blurFace(file, scaleFactor, minNeighbours)` til å returnere antall ansikter som er oppdaget og et bilde med de anonymiserte ansiktene. Deretter oppdateres `self.number` til antall ansikter som er oppdaget, før bildet vises av `showImage()`.

## 9.3 Brukermanual

Applikasjonen startes ved å kjøre `Main.py` fra `/src` med en Pythonkompilator som støtter Python3 eller nyere, eller ved å kjøre `Main.py` med Python3 fra terminalen. Følgende biblioteker er nødvendig å ha installert for å kjøre applikasjonen:

- PyQt5
- MatPlotLib
- Numpy
- Imageio
- Colour

- 
- Colour-Demosaicing
  - PIL
  - cv2

Når applikasjonen starter er det første som vises et vindu som viser oversikt over alle implementasjonene og hva de gjør. For å benytte seg av en implementasjon trykker man på en knappene til venstre for teksten, og et nytt vindu med den aktuelle implementasjonen vil åpnes. Nederst på startvinduet finner man knappen **Om oss**, som åpner et nytt vindu når den triggges. Det nye vinduet viser en tekst med informasjon om gruppen og emnet applikasjonen er laget i, i tillegg til en morsom animasjon nederst i vinduet.

## Glatting

Når et nytt vindu med Glatting av bilde åpnes vil et standardbilde være lastet inn. Til venstre er det informasjonstekst om hva implementasjonen gjør, en liten instruksjon på hvordan det brukes og merknad om glatting av bilde med implisitt løsning. Under informasjonsteksten er det en ”Vis Kodeknapp som åpner et nytt vindu hvor koden som står bak implementasjonen vises i et scrollbart tekstfelt. Under bildet er det to rader med 4 knapper hvor den øverste raden med knapper glatter bildet med en eksplisitt løsning, og den nederste raden glatter bildet implisitt løsning. Velger man eksplisitt løsning har man også mulighet til å stille inn en verdi i spinboxen til høyre mellom 1 og 5, hvor bildet glattes mest når verdien er 1 og minst når verdien er 5. Til høyre for bildet er det åtte knapper, som gjør at brukeren kan velge mellom åtte forhåndsvalgte bilder og dermed endre hvilket bildet som skal glattes. Nede til høyre under disse knappene er det en Lagrekapp som gir brukeren mulighet til å lagre bildet i .png-format eller .jpg-format lokalt på PC-en.

## Inpainting

Ved åpning av nytt vindu med Inpainting er det lastet inn et standardbilde. Til venstre er det mulig å velge mellom tre valgte eksempelbilder. Ved å klikke på ”maskekappen under den valgte eksempelkappen vises bildet med masken over. For å inpainte den manglende informasjonen i bildet klikker man på ”Inpaintknappen, og det inpaintede bildet vil etterhvert vises i bildefeltet. Til venstre for bildet er det en informasjonstekst som sier hva implementasjonen gjør og kort om hvordan man benytter seg av implementasjonen. Nederst til venstre er det to knapper hvor ”Vis Kode” viser koden bak implementasjonen, mens Lagregir brukeren mulighet til å lagre bildet i .png-format eller .jpg-format lokalt på PC-en.

## Kontrastforsterkning

Det er lastet inn et standardbilde i bilderammen når et nytt vindu med kontrastforsterkning åpnes. Dette bildet kan endres ved å klikke på en av de 8 øverste knappene til høyre. Under disse knappene er det en Lagrekapp, som gir brukeren mulighet til å lagre det kontrastforsterkede bildet lokalt på PC-en i .png-format eller .jpg-format. Under bilderammet er det fire knapper og en spinbox. Her har brukeren mulighet til å se originalbildet, en gråtoneversjon av bildet, kontrastforsterke bildet i farger eller kontrastforsterke bildet i gråtoner. Verdien i spinboxen går fra 1 til 3, og ved å sette verdien høyere vil kontrasten økes mer. Til venstre for bildet er det en informasjonstekst som forteller hva implementasjonen gjør og kort om hvordan man bruker applikasjonen. Nederst til venstre er det en ”Vis Kodeknapp som åpner et nytt vindu hvor implementasjonens kode vises.

## Demosaicing

Når et nytt vindu med demosaicing åpnes vil et standardbilde være lastet inn. Dette bildet kan endres ved å klikke én av åtte øverste knappene til høyre for bildet, hvor hver knapp representerer hvert sitt bilde. Til venstre for bildet er det en informasjonstekst som forteller kort hva implementasjonen gjør og hvordan man kan bruke den. Under bildet kan man velge to måter å gjøre

---

demosaicing av bildet på. De tre knappene på den øverste raden bruker våre egenlagde funksjoner, mens de tre knappene på den nederste raden bruker biblioteker fra Colour som nevnt i seksjon 9.2. Ved å klikke på **Gråtonemosaiikk** vises gråtonemosaiikken av bildet med den valgte metoden. For å demosaice bilde klikker man på **Demosaicced** og bildet vises i bilderammen. Nederst til venstre er det en ”Vis Kodeknapp som åpner et nytt vindu hvor koden som brukes for å demosaice et bilde vises i et scrollbart tekstfelt. Nederst til høyre finner man knappen **Lagre** som gir brukeren mulighet til et .png-bilde eller et .jpg-bilde lokalt på PC-en.

### Sømløs kloning

Ved åpning av et nytt sømløs kloningvindu er det lastet inn et standardbilde. Til høyre er det knapper som representerer de tre kombinasjonene av bilder som kan sys sammen. Ved å klikke på **bilde 1** eller **bilde 2** vises originalversjonen av det bildet, og man syr sammen disse to bildene ved å klikke på **Sy sammen**. Til venstre for bildet er det en informasjonstekst som forteller kort om hva implementasjonen gjør og hvordan man bruker den. Nede til venstre er det to knapper. **Vis Kode** åpner et nytt vindu med et scrollbart tekstfelt som viser brukeren koden bak implementasjonen, mens **Lagre** gir brukeren mulighet til å lagre bildet lokalt på datamaskinen.

### Konvertering av fargetone til gråtone

Når brukeren åpner et nytt vindu med Konvertering til gråtone er det alltid lastet inn et standardbilde. Til venstre for dette bildet er det en informasjonstekst som forteller kort om hva implementasjonen gjør og forklarer hvordan man bruker den. Under bildet er det tre knapper som henholdsvis viser originalversjonen av bildet, viser et gråtonebilde hvor den enkle versjonen av konvertering er brukt og viser et gråtonebilde hvor den avanserte løsningen er brukt. Til høyre for bilde er mulig å velge mellom åtte forskjellige bilder, mens det nede til høyre er en lagrekapp som gir brukene mulighet til å lagre gråtonebildet. Nederst til venstre finner man knappen **Vis Kode** som åpner et nytt vindu hvor koden bak implementasjonen vises for brukeren.

### Anonymisering

Når et nytt vindu med anonymisering av ansikter åpnes er det lastet inn et standardbilde i bilderammen. Dette bildet kan endres ved å klikke på én av de åtte knappene til høyre for bildet. For å vise originalversjonen av bildet hvor ingen ansikter er funnet klikker man på **Original**. Ønsker man å finne ansiktene på bildet uten å anonymisere dem klikker man på **Finn ansikter**. For å anonymisere ansiktene klikker man på **Anonymiser**, og et bilde med anonymiserte ansikter vises i bilderammen. Når brukeren enten har klikket på **Finn ansikter** eller **Anonymiser** oppdateres teksten nede til høyre for bildet med hvor mange ansikter som er funnet på bildet. De to spinboxene til høyre under bildet gir mulighet for å finpusse ansiktsgjenkjenningen. Spinboxen til venstre endrer scaleFactor for bildet, det vil hvor mye algoritmen kompenserer for ansikter som er nærmere kamera enn andre. Spinboxen til høyre endrer minNeighbours for bildet, det vil si hvor mange naboorer en rektangel bør ha for å bli kalt ansett som et ansikt. Under denne teksten er det en lagrekapp som gir brukeren mulighet til å lagre et bilde med anonymiserte eller markerte ansikter i .png-format eller .jpg-format. Nederst til venstre finner man knappen **Vis Kode** som viser koden bak implementasjonen i et nytt vindu.

## 10 Konklusjon

- Brancher og mergerequest

Fra GUI: - gjort på nytt, droppet bruk av QtDesigner og heller skrevet alt i kode. Litt mer kontroll og lettare å justere visning av vinduer. Kunne da også enkelt implementert støtte for å kunne justere vinduer ved å dra i kantene - droppet bruk av .ui-filer som hadde gjort det mye enklere å

---

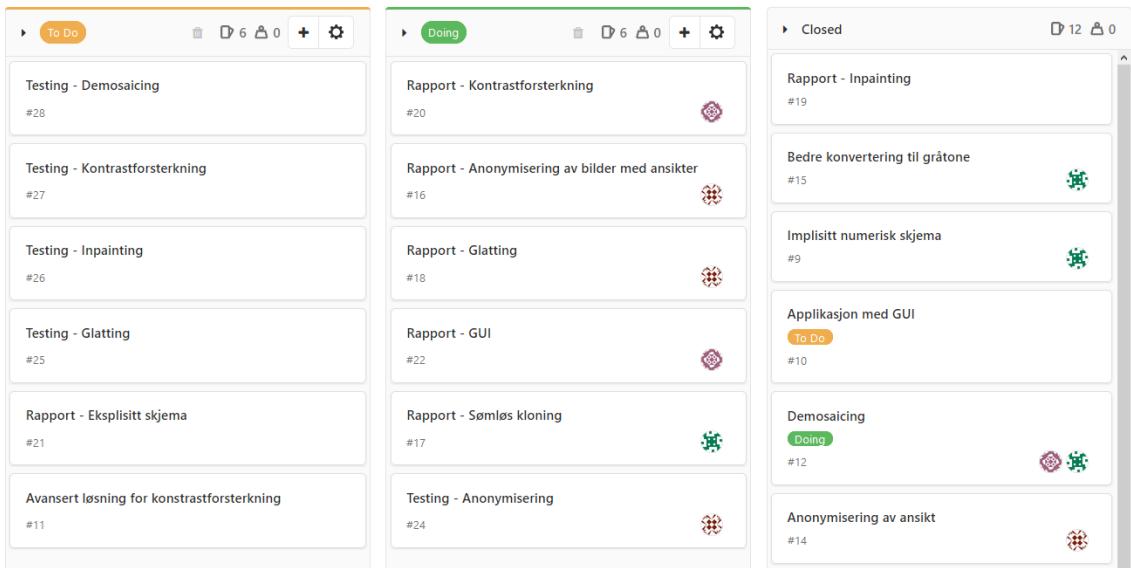
organisere prosjektet på en bedre måte - En del cowboyløsninger når man bygger på og bygger på, spesielt lagring av bilder ble ganske rotete og unødvendig komplisert til slutt - Mange steder som man kanskje burde laget GUI annerledes - Lagring av bilder har hatt verdier mellom 0 og 1 har lavere kvalitet

---

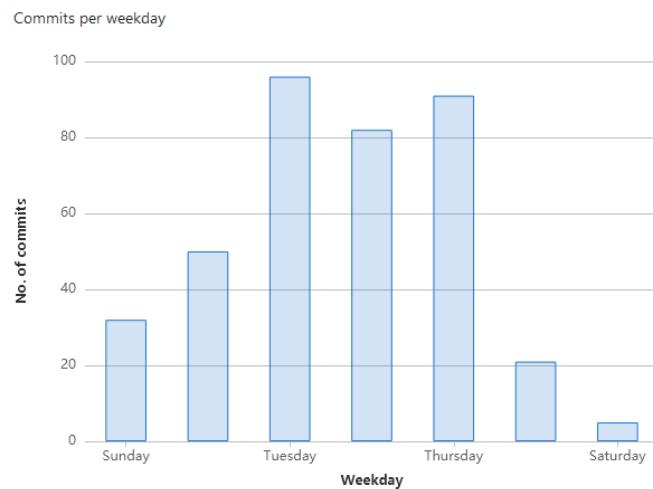
## Bibliografi

- [1] Svein Linge and Hans Petter Langtangen. *Programming for Computations - Python*, volume 15 of *Texts in Computational Science and Engineering*. SpringerOpen, 2016.
- [2] Kjelsrud, Gulbrandsen, and Jegerud. Imt3881 2020 prosjekt. <https://git.gvk.idi.ntnu.no/casperfg/imt3881-2020-prosjekt/-/tree/master>, mars 2020.
- [3] Wikipedia bidragsytere. Gaussian blur — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/wiki/Gaussian\\_blur](https://en.wikipedia.org/wiki/Gaussian_blur), 2020. [lesedato 10.05.2020].
- [4] Wikipedia bidragsytere. Gaussisk oskärpa — Wikipedia, the free encyclopedia. [https://sv.wikipedia.org/wiki/Gaussisk\\_oscarkarpa](https://sv.wikipedia.org/wiki/Gaussisk_oscarkarpa), 2020. [lesedato 10.05.2020].
- [5] Andrew Blake Patrick Perez, Michel Gangnet. papers\_0156\_final.dvi. [http://www.cs.virginia.edu/~connelly/class/2014/comp\\_photo/proj2/poisson.pdf](http://www.cs.virginia.edu/~connelly/class/2014/comp_photo/proj2/poisson.pdf). [lesedato: 04.27.2020].
- [6] Somaya Al-Maadeeda Younes Akbaria Omar Elharroussa, Noor Almaadeeda. 1909.06399.pdf. <https://arxiv.org/ftp/arxiv/papers/1909/1909.06399.pdf>. (lesedato: 04.27.2020).
- [7] Wikipedia bidragsytere. Deep learning — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/w/index.php?title=Deep\\_learning&oldid=953563597](https://en.wikipedia.org/w/index.php?title=Deep_learning&oldid=953563597), 2020. [lesedato 29.04.2020].
- [8] Kevin J. Shih Ting-Chun Wang Andrew Tao Bryan Catanzaro Guilin Liu, Fitzsum A. Reda. [1804.07723] image inpainting for irregular holes using partial convolutions. <https://arxiv.org/abs/1804.07723>. (lesedato: 04.27.2020).
- [9] Nvidia deep learning based image inpainting demo is impressive. <https://www.geeks3d.com/20180425/nvidia-deep-learning-based-image-inpainting-demo-is-impressive/>. (lesedato: 04.27.2020).
- [10] Wikipedia bidragsytere. Kontrast — Wikipedia, den fria encyklopedin. <https://sv.wikipedia.org/wiki/Kontrast>, 2020. [lesedato 29.04.2020].
- [11] Wikimedia Commons. Kontrast — Wikipedia, die freie enzyklopädie. <https://de.wikipedia.org/wiki/Kontrast#Bilddarstellung>, 2020. [lesedato 30.04.2020].
- [12] University of Tartu. Chapter 9 image enhancement processing. [lesedato 29.04.2020].
- [13] MathWorks. Contrast enhancement techniques. [lesedato 30.04.2020].
- [14] TRACY V. WILSON & GERALD GUREVICH KARIM NICE. Demosaicing algorithms: Color filtering — howstuffworks. <https://electronics.howstuffworks.com/cameras-photography/digital/digital-camera5.htm>. lesedato: 04.28.2020).
- [15] Wikipedia bidragsytere. Color filter array — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/w/index.php?title=Color\\_filter\\_array&oldid=948604579](https://en.wikipedia.org/w/index.php?title=Color_filter_array&oldid=948604579), 2020. lesedato: 28.04.2020.
- [16] Wikipedia bidragsytere. Lilla — Wikipedia, the free encyclopedia. <https://no.wikipedia.org/w/index.php?title=Lilla&oldid=18926294>, 2018. [lesedato 25.10.2018].
- [17] Wikipedia bidragsytere. Cone cell — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/w/index.php?title=Cone\\_cell&oldid=949785199](https://en.wikipedia.org/w/index.php?title=Cone_cell&oldid=949785199), 2020. [lesedato: 29.04.2020].
- [18] Wikipedia contributors. Grayscale — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=Grayscale&oldid=951234469>, 2020. [Online; accessed 12-May-2020].

- 
- [19] Farup I. Poisson image editing. <https://git.gvk.idi.ntnu.no/course/imt3881/imt3881-2020-prosjekt>.
  - [20] Wikipedia bidragsytere. Face detection — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/w/index.php?title=Face\\_detection&oldid=947281932](https://en.wikipedia.org/w/index.php?title=Face_detection&oldid=947281932), 2020. [lesedato: 22.04.2020].
  - [21] Jason Brownlee. A gentle introduction to computer vision. <https://machinelearningmastery.com/what-is-computer-vision/>, Mars 2019. [lesedato: 24.04.2020].
  - [22] Open CV team. About open cv. <https://opencv.org/about/>, 2020. [lesedato: 24.04.2020].
  - [23] Wikipedia bidragsytere. Machine perception — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/w/index.php?title=Machine\\_perception&oldid=949933213](https://en.wikipedia.org/w/index.php?title=Machine_perception&oldid=949933213), 2020. [lesedato: 24.04.2020].
  - [24] Will Berger. Deep learning haar cascade explained. <http://www.willberger.org/cascade-haar-explained/>, 2018. [lesedato: 24.04.2020].
  - [25] Vadim Pisarevsky. Github: haarcascade\_frontalface\_default.xml. [https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade\\_frontalface\\_default.xml](https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_default.xml), desember 2013. [Nedlastingsdato: 06.04.2020].
  - [26] Wikimedia Commons. File:vj featuretypes.svg — wikimedia commons, the free media repository, 2015. [lesedato 29.04.2020].
  - [27] Wikimedia Commons. File:ms. magazine cover - spring 2013 (cropped).jpg — wikimedia commons, the free media repository, 2020. lesedato 29.04.2020].
  - [28] Wikimedia Commons. Graphical user interfaces — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/wiki/Graphical\\_user\\_interface](https://en.wikipedia.org/wiki/Graphical_user_interface), 2020. [lesedato 30.04.2020].



Figur 15: Gitlab issue board



Figur 16: Arbeidsflyten gjennom ukedagene

## Appendiks

### A Arbeidsflyt og prosjektorganisering