



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 problemsstillinger. 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[1].

---

## 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.

### Koronavirus-pandemien i 2020

### 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>1</b>	<b>Introduksjon</b>	<b>1</b>
1.1	Rapportens oppbygging . . . . .	1
1.2	Oppgaven . . . . .	1
<b>2</b>	<b>Glatting</b>	<b>2</b>
<b>3</b>	<b>Inpainting</b>	<b>2</b>
3.1	Bakgrunn . . . . .	2
3.2	Teori . . . . .	2
3.3	Implementasjon . . . . .	2
<b>4</b>	<b>Kontrastforsterkning</b>	<b>3</b>
4.1	Bakgrunn . . . . .	3
4.2	Implementasjon . . . . .	4
<b>5</b>	<b>Demosaicing</b>	<b>5</b>
5.1	Bakgrunn . . . . .	5
5.2	Implementasjon . . . . .	6
<b>6</b>	<b>Sømløs kloning</b>	<b>7</b>
6.1	Bakgrunn . . . . .	7
6.2	Implementasjon . . . . .	7
<b>7</b>	<b>Konvertering av fargebilder til gråtone</b>	<b>8</b>
<b>8</b>	<b>Anonymisering av bilder med ansikter</b>	<b>9</b>
8.1	Bakgrunn . . . . .	9
8.1.1	Open CV . . . . .	9
8.1.2	Haar Cascade algoritmen . . . . .	9
8.2	Implementasjon . . . . .	10
<b>9</b>	<b>GUI</b>	<b>10</b>
9.1	Bakgrunn . . . . .	10
9.2	Implementasjon . . . . .	10
9.2.1	Glatting . . . . .	12
9.2.2	Inpainting . . . . .	12
9.2.3	Kontrastforsterkning . . . . .	12

---

9.2.4	Demosaicing . . . . .	12
9.2.5	Sømløs kloning . . . . .	12
9.2.6	Konvertering av fargetone til gråtone . . . . .	12
9.3	Anonymisering . . . . .	12
9.4	Brukermanual . . . . .	12
9.4.1	Glatting . . . . .	12
9.4.2	Inpainting . . . . .	12
9.4.3	Kontrastforsterkning . . . . .	12
9.4.4	Demosaicing . . . . .	12
9.4.5	Sømløs kloning . . . . .	12
9.4.6	Konvertering av fargetone til gråtone . . . . .	12
<b>Figurer</b>		<b>13</b>
<b>Bibliografi</b>		<b>14</b>

---

# 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). Utover 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.

## 2 Glatting

## 3 Inpainting

### 3.1 Bakgrunn

Inpainting er en utbredt teknikk innenfor bildebehandling [2]. Teknikken kan brukes til å fylle inn manglende informasjon eller fjerne uønsket støy eller elementer i bilder<sup>2</sup>. 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. Fjerning av visse elementer er i tillegg sett på som en løsning for forfalskning av bilder. 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 [4].

### 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 [5], hvor tilsynelatende ødelagte bilder kan gjenopprettes til meget høy kvalitet [6], illustrert ved Figur 1. I vår implementasjon er man i større grad avhengig av at masken befinner seg på et gunstig sted slik at ikke viktig informasjon tildekkes.

### 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
```

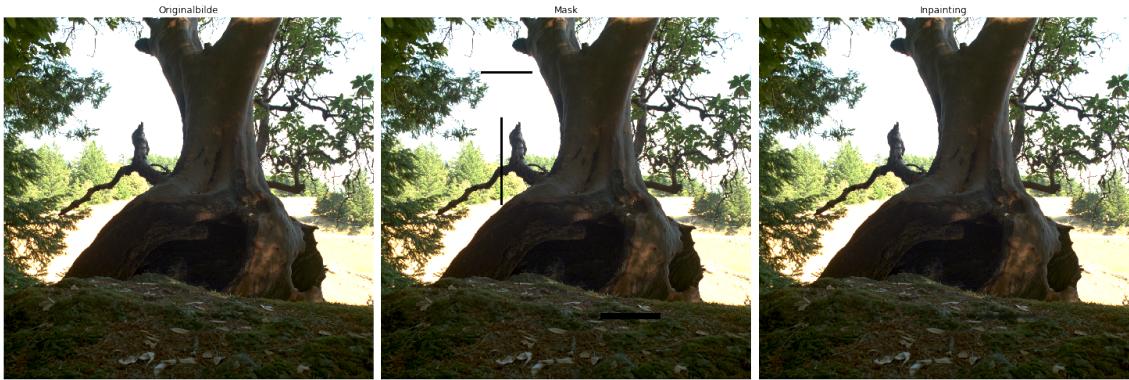
---

<sup>2</sup>Poisson Image Editing[3]



Figur 1: NVIDIAs inpaintingalgoritme

Kilde: NVIDIA [7]



Figur 2: Inpainting av tre

Deretter bruker man denne arrayen som en "view"<sup>3</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]
```

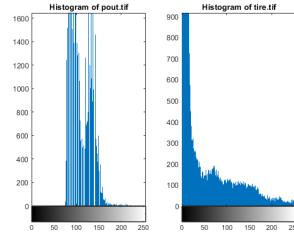
Tilslutt returneres originalbilde og det ferdig inpaintede bildet fra funksjonen som løste diffusjonslikningene. Figur 2 viser resultatet av vår inpaintingalgoritme, her med  $n=100$  iterasjoner. Her ble masken definert som 3 rette svarte klosser. Etter inpainting er fullført er de tilsynelatende borte.

## 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.[8] 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

<sup>3</sup><https://docs.scipy.org/doc/numpy/user/basics.indexing.html>



Figur 3: Graylevel Histogram

Kilde: MathWorks [11]

fargekontrasten som gjør at vi kan se objekter på et bilde. Et med bilde med høy motivkontrast[9] 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 mørkest 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.

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)[10]. Dette gjøres ved å strekke grånivåverdiene i bildet slik at histogrammet dekker alle grånivåene mellom 0 og 255, som vist i figur (3).

## 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

$$\frac{\partial u}{\partial t} = \nabla^2 u - k \nabla^2 u_0. \quad (5)$$

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])

```

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 kippe verdiene til et lovlig intervaller, som gjøres før  $u$  returneres av funksjonen.

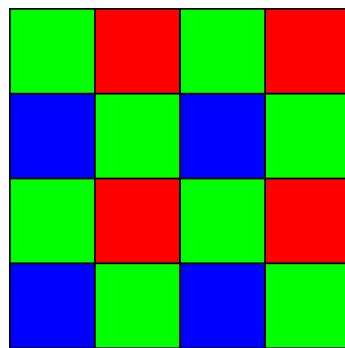
```

u[u < 0] = 0
u[u > 1] = 1

```



Figur 4: Kontrastforsterkning



Figur 5: Bayer filter

Kilde: Wikipedia [13]

Resultatet av vår implementasjon vises i figur 4. Til venstre vises originalbildet slik det blir lastet inn ved hjelp av `imageio.imread`<sup>4</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 [12], illustrert ved Figur 3. 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>5</sup>. Bayer-filteret er et mønster som alternerer mellom en rad av grønne og røde piksler, og en rad av blå og grønne piksler (5). 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 omtrent

<sup>4</sup><https://pypi.org/project/imageio/>

<sup>5</sup><https://en.wikipedia.org/w/index.php?title=Interpolation&oldid=949792579>



Figur 6: Demosaicing

6 millioner tapper<sup>6</sup> av typene S, L og M-tapper. S-tappene registrerer fiolett, altså blandinger mellom rød og blå [14]. Derimot registrerer både L- og M-tapper grønt, med mest følsomhet rundt bølgelengder tilsvarende 534-564nm [15]. Dermed trenger en mer informasjon fra grønne piksler for å produsere et bilde som menneskeøyne vil kunne oppfatte som naturtro.

## 5.2 Implementasjon

Fra kameraensoren kommer en gråtonemosaiikk, 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 gråtonemosaiikken 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 [12]. 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 6. Først presenteres originalbilde slik det blir lastet inn ved `imageio.imread`. Deretter vises gråtonemosaiikken, før det ferdig interpolerte bildet.

<sup>6</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: Sømløs kloning

## 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.

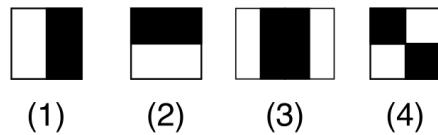
Problemet kan formuleres som et Poisson-problem dersom  $u_0$  er originalbildet det skal klones inn i, og  $u_1$  bildet utsnittet skal hentes fra. Problemstillingen blir da å finne  $u$  slik at  $u = u_0$  utenfor masken  $\Omega_i$  og  $\nabla^2 u = \nabla^2 u_1$  i  $\Omega_i$ . Dette kan løses 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 laplacen `laplace1` fra  $u_1$  før (2) løses med `h=laplace1`. Resultatet ved å velge et område på  $u_0$  som samsvarer noenlunde med fargene og objektene fra  $u_1$  vises i Figur 7.

---

## 7 Konvertering av fargebilder til gråtone



Figur 8: Haar Feature

Kilde: Wikimedia Commons [21]

## 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 forblir skarpt. Teknikken avhenger av at programmet klarer å gjenkjenne ansiktene[16] 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.

#### 8.1.1 Open CV

Open CV er et open source datasyn-[17] og maskinlæringsbibliotek som ble utviklet for å skape et felles infrastruktur for datasynsapplikasjoner og for å akselerere bruken av maskinoppfatning i kommersielle produkter[18]. 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[19], og da spesielt synet i dette tilfellet.

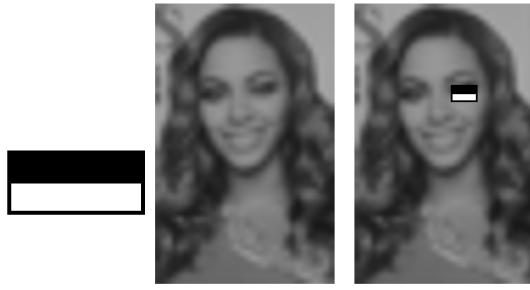
#### 8.1.2 Haar Cascade algoritmen

Haar Cascade er en maskinlæringsalgoritme [20] 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 8) 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 gjenkjenne linjer og kanter på objekter, og spesielt i ansikter. Som man ser i (figur 9) 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 (8), altså en mørk overdel med lys bunn.



Figur 9: Haar Feature demonstrert på et ansikt

Kilde: Wikimedia Commons [22]

## 8.2 Implementasjon

# 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 nавигerer ved hjelp av kommandoer, består et grafiske grensesnittet ofte av grafiske ikoner og lydindikatorer.<sup>[23]</sup> 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>7</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>8</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.

<sup>7</sup><https://pypi.org/project/PyQt5/>

<sup>8</sup><https://pypi.org/project/PyQt5Designer/>



Figur 10: Grafisk brukergrensesnitt - Glatting

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

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 10). Vi har valgt å bruke Matplotlib til å vise bilder. Dette gjør at vi kan bruke `plt.imshow()`<sup>9</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>10</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. 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>11</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

<sup>9</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>10</sup><https://doc.qt.io/archives/qt-4.8/qlabel.html#pixmap-prop>

<sup>11</sup><https://doc.qt.io/qt-5/qfiledialog.html>

---

funksjonen `FunctionGUI>ShowCode`. Her opprettes det et nytt `QMainWindow`, hvor koden vises i et rullbar tekstfelt.

### 9.2.1 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 eksplisitt løsning av diffusjonslikningen.

### 9.2.2 Inpainting

### 9.2.3 Kontrastforsterkning

### 9.2.4 Demosaicing

### 9.2.5 Sømløs kloning

### 9.2.6 Konvertering av fargetone til gråtone

## 9.3 Anonymisering

## 9.4 Brukermanual

### 9.4.1 Glatting

### 9.4.2 Inpainting

### 9.4.3 Kontrastforsterkning

### 9.4.4 Demosaicing

### 9.4.5 Sømløs kloning

### 9.4.6 Konvertering av fargetone til gråtone

Ta med til konklusjon: - gjort på nytt, droppet bruk av QtDesigner og heller skrevet alt i kode.  
Litt mer kontroll og lettere 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 cowboylosninger når man bygger på og bygger på, spesielt lagring av bilder ble ganske rotete og unødvendig komplisert til slutt -

---

## Figurer

1	NVIDIAs inpaintingalgoritme	3
2	Inpainting av tre	3
3	Graylevel Histogram	4
4	Kontrastforsterkning	5
5	Bayer filter	5
6	Demosaicing	6
7	Sømløs kloning	7
8	Haar Feature	9
9	Haar Feature demonstrert på et ansikt	10
10	Grafisk brukergrensesnitt - Glatting	11

---

## Bibliografi

- [1] Kjelsrud, Gulbrandsen, and Jegerud. Imt3881 2020 prosjekt. <https://git.gvk.idi.ntnu.no/casperfg/imt3881-2020-prosjekt/-/tree/master>, mars 2020.
- [2] Wikipedia bidragsytere. Inpainting — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=Inpainting&oldid=951383833>, 2020. [lesedato 27.04.2020].
- [3] 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].
- [4] 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).
- [5] 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].
- [6] Kevin J. Shih Ting-Chun Wang Andrew Tao Bryan Catanzaro Guilin Liu, Fitsum A. Reda. [1804.07723] image inpainting for irregular holes using partial convolutions. <https://arxiv.org/abs/1804.07723>. (lesedato: 04.27.2020).
- [7] 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).
- [8] Wikipedia bidragsytere. Kontrast — Wikipedia, den fria encyklopedin. <https://sv.wikipedia.org/wiki/Kontrast>, 2020. [lesedato 29.04.2020].
- [9] Wikimedia Commons. Kontrast — Wikipedia, die freie enzyklopädie. <https://de.wikipedia.org/wiki/Kontrast#Bilddarstellung>, 2020. [lesedato 30.04.2020].
- [10] University of Tartu. Chapter 9 image enhancement processing. [lesedato 29.04.2020].
- [11] MathWorks. Contrast enhancement techniques. [lesedato 30.04.2020].
- [12] 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).
- [13] 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.
- [14] Wikipedia bidragsytere. Lilla — Wikipedia, the free encyclopedia. <https://no.wikipedia.org/w/index.php?title=Lilla&oldid=18926294>, 2018. [lesedato 25.10.2018].
- [15] 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].
- [16] 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].
- [17] Jason Brownlee. A gentle introduction to computer vision. <https://machinelearningmastery.com/what-is-computer-vision/>, Mars 2019. [lesedato: 24.04.2020].
- [18] Open CV team. About open cv. <https://opencv.org/about/>, 2020. [lesedato: 24.04.2020].

- 
- [19] 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].
  - [20] Will Berger. Deep learning haar cascade explained. <http://www.willberger.org/cascade-haar-explained/>, 2018. [lesedato: 24.04.2020].
  - [21] Wikimedia Commons. File:vj featuretypes.svg — wikimedia commons, the free media repository, 2015. [lesedato 29.04.2020].
  - [22] Wikimedia Commons. File:ms. magazine cover - spring 2013 (cropped).jpg — wikimedia commons, the free media repository, 2020. lesedato 29.04.2020].
  - [23] 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].
  - [24] Svein Linge and Hans Petter Langtangen. *Programming for Computations - Python*, volume 15 of *Texts in Computational Science and Engineering*. SpringerOpen, 2016.