

Poisson Bilderedigeringsapplikasjon

Aslak Tøn, Håkon Muggerud, Isac Mikal Kallevig

April 2020

Contents

1	Introduksjon	3
1.1	Bakgrunn	3
1.2	Vårt arbeid	3
1.3	Oppsummering	4
2	Systemkrav	4
2.1	Generell diffusjonsligning	4
2.2	Glatting	5
2.3	Inpainting	5
2.4	Demosaicing	5
2.5	Anonymisering av ansikter	5
2.6	Konvertering til gråskakkabilder	5
2.7	Kontrastforsterkning	5
2.8	Sømløs kloning	6
2.9	Kantbevarende glatting	6
2.10	HDR bilder	6
2.11	Grafisk brukergrensesnitt	6
2.12	Bruk av GPU	7
3	Teknisk design og implementasjon	7
3.1	Generell eksplisitt løsning	7
3.2	Glatting	12
3.2.1	Implisitt, Crank-Nicolson og Direkte Glatting	13
3.2.2	Implisitt skjema	14
3.2.3	Crank-Nicolson	15
3.2.4	Direkte Løsning	15
3.3	Inpainting	16
3.4	Demosaicing	16
3.5	Anonymisering av ansikter	18
3.5.1	Annsiktsgjenkjenning	18
3.5.2	Øyegjenkjenning	20
3.5.3	Precision og recall	20
3.5.4	Implisitt metode	20

3.5.5	Avsluttende tanker	22
3.6	Konvertering til gråskalabilder	22
3.7	Kontrastforsterkning	23
3.8	Sømløs kloning	24
3.9	Kantbevarende glatting	24
3.10	HDR bilder	25
3.10.1	Optimalisering	26
3.11	Grafisk brukergrensesnitt	27
3.11.1	Brukergrensesnittet i korte trekk	27
3.11.2	Objekt Orientert Design	28
3.11.3	Sekvensdiagram	29
3.11.4	Argumentasjon for Arv	29
3.11.5	Argumentasjon for bruken av pipelines	30
4	Utviklingsprosess	30
4.1	Rollebeskrivelser	30
4.1.1	Prosjektleder	30
4.1.2	Numerisk utledning og matematikk-ansvarlig	30
4.1.3	Grensesnitt og test-Ansvarlig	30
4.2	Smidig Systemutviklingsmodell	31
4.3	Styrende Faktorer	31
5	Kvalitetssikring	31
5.1	Møter	31
5.2	Kommunikasjonskanal	31
5.3	Testing	32
5.3.1	Testkompetanse	32
5.3.2	Planlegging	32
5.3.3	Akseptansetesting	32
5.4	Avsluttende tanker	33
5.5	Arbeidsfordeling	33
5.6	Kodestandard	33
5.7	Robust kode	33
6	Resultater og diskusjon	34
6.1	Generell diffusjon	34
6.2	Glatting	34
6.3	Inpaint	34
6.4	Demosaicing	34
6.5	Anonymisering av ansikter	35
6.6	Konvertering til gråskalabilder	35
6.7	Kontrastforsterkning	37
6.8	Sømløs Kloning	38
6.9	Kantbevarende glatting	39
6.10	HDR bilder	42
6.11	Grafisk brukergrensesnitt	43

6.11.1	Resultat og innføring i bruken av applikasjonen	43
6.11.2	Feil og Problemer	45
6.11.3	Optimaliseringer og Forbedringer	46
7	Testing	46
8	Diskusjon	47
8.1	Måloppnåelse	47
8.2	Forbedringspotensiale	47
8.3	Videre arbeid	48
9	Konklusjon	48
10	Bibliografi	48
11	Apendix A: Bildekilder	49

1 Introduksjon

1.1 Bakgrunn

Bilderedigering er et bredt felt, og utvikling av applikasjoner for dette formålet kan benytte seg av en rekke metoder for å gjennomføre effekter som blurring, invertering, kloning, mm. Vi har fra oppdragsgiver fått oppdrag å utvikle en applikasjon som benytter seg av "Poisson Image Editing [1]. "Poisson Image Editing" er en metode innenfor bildebehandling som løser Poisson-ligningen:

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

til å løse en rekke bilderedigeringsproblemer. En av metodene for å løse en slik ligning er å iterere seg frem til løsningen numerisk ved å ta:

$$\frac{\partial u}{\partial t} = \nabla^2 u - h$$

Ved å benytte forskjellige verdier for h avhengig av hvilke mål man har for et bilde og hvilken region man er jobber i et bilde kan man oppnå forskjellige effekter av en slik "Poisson Image Editing". Vår applikasjon er utviklet for å utnytte en utvalgt undergruppe av disse metodene for å oppnå forskjellige effekter av bildebehandling.

1.2 Vårt arbeid

Vi vil i løpet av arbeidsperioden satt fra 11. Mars ved undertegning av arbeidskontrakt fram til 15. Mai jobbe med å inkludere en rekke moduler foreslått av

oppdragsgiver i sammenheng med hans prosjektbeskrivelse¹. Vi ble enige gjennom møter at vi ønsket å gjennomføre alle punkter oppdragsgiver la fram av forslag. I tillegg ønsket vi tidlig å inkludere en brukbar GUI til applikasjonen. Gjennom senere brainstorming kom også ideer om bruk av maskinlæringsalgoritmer for konvertering av gråskalabilder til fargebilder.

Dette har i senere tid blitt forkastet da vi valgte å bruke tiden til å gjøre allerede eksisterende kodebase mer robust. Vi konkluderte etter et par diskusjoner at muligheten for god implementasjon av gråskala til fargebilder ville vært en for stor arbeidsmengde, og at resten av applikasjonen ville ikke oppnå en tilfredsstillende kvalitet hvis vi brukte mye tid på utvikling av denne metoden.

Modulene som dermed ble inkludert i sluttproduktet var: Glattning, inpainting, kontrastforsterkning, demosaicing, sømløs kloning, fargetoner til gråtone vha. bildets gradient, anisksitsanonymisering, rekonstruksjon og visualisering av HDR bilder og kantbevarende glattning. Disse modulene skal fungere for fargebilder og svart-hvitt bilder. Fargebilder vil generelt anta et RGBA format, men skal også fungere for RGB formater slik som JPEG.

Alle moduler for bilderedigering vil være tilgjengelige innenfor GUI som er utviklet sammen med bildeprosesseringsmodulene. GUI vil kunne ta input bilder og masker tilhørende bilder og etter bruker har bekreftet ønsket transformasjon vil vise ferdig prosessert bilde til bruker.

Prosjektet har benyttet seg av GitLab som sentralt repository for utvikling. Kildekoden kan finnes på <https://git.gvk.idi.ntnu.no/aslakto/imt3881-2020-prosjekt>.

1.3 Oppsummering

Vi har altså i dette prosjektet på to måneder utviklet et bilderedigeringsprogram med tilhørende GUI. Programmet benytter seg av ”Poisson Image Editing” [1] for å gjennomføre forskjellige effekter av et bilde.

2 Systemkrav

Ved ferdigstilling leverer applikasjonen et visuelt brukergrensesnitt (GUI) for valg og bruk av forskjellige moduler som utnytter Poisson-bilderedigering. Modulene som tilbys med applikasjonen er:

2.1 Generell diffusjonsligning

Programmet tilbyr en generell diffusjonsligning som andre metoder kan kalle på for å kjøre en diffusjon på ønskelig måte.

¹<https://git.gvk.idi.ntnu.no/course/imt3881/imt3881-2020-prosjekt/-/blob/master/oppgave/prosjekt.pdf>

2.2 Glatting

En funksjonalitet som fjerner høyintense regioner av et bilde vha. en diffusjon fra piksler rundt en region inn i regionen.

Denne funksjonaliteten kan gjøres ved å løse Poisson-ligning iterativt med eksplisitt, implisitt og Crank-Nicolson skjemama, i tillegg til direkte løsning (Se teknisk design 3.2).

2.3 Inpainting

Inpainting skal sørge for en jevn overgang fra en region i et bilde til en annen region i et bilde. Dette kan være nyttig i applikasjonen som fjerning av små skader i bilder eller fjerning av tekst oppå bildet.

2.4 Demosaicing

Demosaicing vil ta de tre kanalene med monokromatisk lysinformasjon og benytte inpainting til å fylle ut informasjon i piksler den ikke har informasjon til.

Metoden er viktig for bruk av kameraer, da det kun er en lyssensor i et kamera. Kamera benytter for eksempel et Bayer filter² for å trekke ut informasjon om mengden av rødt, grønt og blått lys i hver piksel. Denne skapte mosaikken trengs å ”avmosaikkes” for å se ut som et vanlig fargebilde.

2.5 Anonymisering av ansikter

Personer ønsker ikke alltid at deres ansikt er lett gjenkjennelig i et bilde. En funksjonalitet applikasjonen skal tilby mot dette er en anonymiseringsfunksjon hvor ansikter blir glattet ut og ugenkjennelig. Anonymisering av ansikter burde ha som mål å anonymisere alle ansikter i et bilde. Vi ønsker heller at noen regioner som ikke burde bli anonymisert blir anonymisert grunnet en false-positive.

2.6 Konvertering til gråskakalabilder

En enkel operasjon for å konvertere fargebilder til gråskala er å ta snittet av rød, grønn og blå og legge det inn i et nytt bilde med en fargekanal. Dette beholder ikke fullstendig detaljer, og spesielt overgangen mellom mørke og lyse regioner i et bilde. En annen måte å skape et gråskalabilde er å regne gradienten til et bilde og bruke dette som informasjon for å skape et gråskala bilde som reflekterer bedre kontrastene i et bilde.

2.7 Kontrastforsterkning

Kontrastforsterkning vil bruke de samme konseptene som Konvertering til gråskala. Her regnes gradienten ut til alle fargekanalene i ett bilde. Ved å øke lengden

²<https://patents.google.com/patent/US3971065>

til denne gradienten vil kontrasten i fargekanalen øke, og kontrasten i bilde vil dermed også øke.

2.8 Sømløs kloning

Med denne funksjonen vil du kunne lime inn en region av et bilde inn i et annet slik at det ser naturlig ut uten synlige overganger. Dette oppnås ved å beholde gradienten til bildet som limes inn, og diffundere med originalbildet som Dirichlet randbetingelse. Bildene og maskene må være samme størrelse, men klone-området kan finne sted på ulike steder i de to bildene.

2.9 Kantbevarende glatting

Applikasjonen tilbyr å glatte ut et bilde ved å bevare noe av kantene. Kantbevaringen skal kunne reguleres ved å vekte kantbevaringen med en parameter k . I vår implementasjon skal det tilbys ekplisitt løsning for kantbevarende glatting.

2.10 HDR bilder

Dersom man har tatt flere bilder av en scene med ulike eksponeringstider kan denne funksjonen rekonstruere og visualisere et HDR bilde. Dette gjøres ved å estimere kamera-responskurven og lysheten i bildet som et minste kvadraters metode problem. Det er også mulig å kontrollere glattheten til kurven for å unngå støy i blidet.

2.11 Grafisk brukergrensesnitt

Applikasjon tilbyr et lettvekts brukergrensesnitt med en enkel pipeline per bilde som skal gjøre det mulig å gjøre flere transformasjoner interaktivt. Brukergrensesnittet kan:

- Laste inn og eksportere et bilde
- Laste inn en mappe med en serie med bilder som gjøres om til et HDR bilde.
- Legge til 0 til mange transformasjoner på et bilde
- Slette bilder og transformasjoner gjort i applikasjonen
- Bytte om på rekkefølgen transformasjoner blir gjort
- Bake resultatet av flere transformasjoner inn i et nytt bilde
- Tilby hjelpemoduler som f.eks. støy, mosaicing, binærmodulen for å kunne interaktivt hovedmodulene.

Brukergrensesnittet tilbyr ikke:

- Tegneverktøy, for direkte tegning bilder. Er det behov for dette, f.eks. å tegne masker må dette gjøres i et eksternt program som f.eks krita eller ms paint på windows.
- Lagring og importering av prosjekter.
- Trådbasert optimalisering, med noen få unntak.

2.12 Bruk av GPU

Det tilbys ikke GPU akselerasjon. Det ble vurdert å benytte seg av dette, da GPU-er raskt kan utføre matriseregning, men ble senere forkastet, da nødvendig ekstra ressursbruk ville gått utover rammene i prosjektet.

3 Teknisk design og implementasjon

3.1 Generell eksplisitt løsning

Det eksplisitte skjemaet for den generelle diffusjonsligningen kan utledes som i ligning 1.

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

Metoden alene vil kun benytte en Laplace og alpha for å oppdatere pikselverdier i bildet. En rask metode for å gjennomføre dette er ved bruk av en kodeblokk som denne:

```
laplace = (
    u[:-2, 1:-1] +
    u[2:, 1:-1] +
    u[1:-1, :-2] +
    u[1:-1, 2:] -
    4 * u[1:-1, 1:-1]
)
u[1:-1, 1:-1] += laplace * alpha
```

Denne løsningen inkluderer ikke randen i metoden, men i de fleste bilder vil randen være tilnærmet lik pikselverdi innenfor og vi kan dermed bruke Neumann-betingelser etter en Laplace-transformasjon.

```
u[0, :] = u[1, :]
u[:, 0] = u[:, 1]
u[-1, :] = u[-2, :]
u[:, -1] = u[:, -2]
```

Ved å gjennomføre disse to operasjonene sekvensielt flere ganger vil regioner i et bilde med høy intensitet glattes ut. Effekten er mest åpenbar hvis algoritmen kjøres på et 2dimensjonalt objekt. Merk at den mest markante forskjellen i figur 1 skjer mellom iterasjon 0 og 1. Her vil de punktene med største variasjon

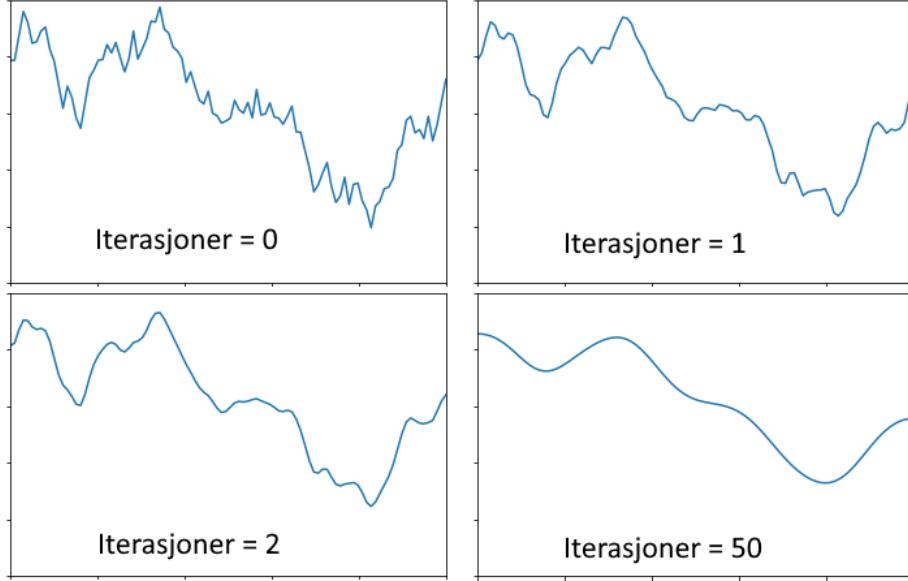


Figure 1: Forskjellige iterasjoner av diffusjonsligning brukt på et 1d objekt

i intensitet til sine naboer raskt bli glattet ut. Vi kan også se denne forskjellen matematisk vha. fraksjonale dimensjoner [2].

Lett modifisering av koden funnet på github³ for utregning av fraktale dimensjoner gir oss disse Minkowski-Bouligand dimensjonene [3] (D):

- Itr 0: $D = 1.31036$
- Itr 1: $D = 1.25888$
- Itr 2: $D = 1.24480$
- Itr 50: $D = 1.20610$

Vi ser at forandringen i den fraktale dimensjonen er 0.05148 mellom iterasjon 0 og iterasjon 1. Mellom iterasjon 1 og iterasjon 50 ligger differansen på 0.05278. Det vil si at ”jevnheten” til dette endimensjonale objektet går omrent like mye ned den første iterasjonen som de neste 49 iterasjonene.

Denne metoden har noen problemer når den utføres på bilder. Den vil kun kjøre en diffusjon så lenge u er et $m \cdot n$ bilde. Dette betyr at flere fargekanaler ikke støttes. Operasjonen vil også kjøres på hele bilde, som ikke alltid er nødvendig. En løsning til dette problemet er bruk av boolske masker i samme lengde og bredde som bilde.

³<https://gist.github.com/rougier/e5eafc276a4e54f516ed5559df4242c0>

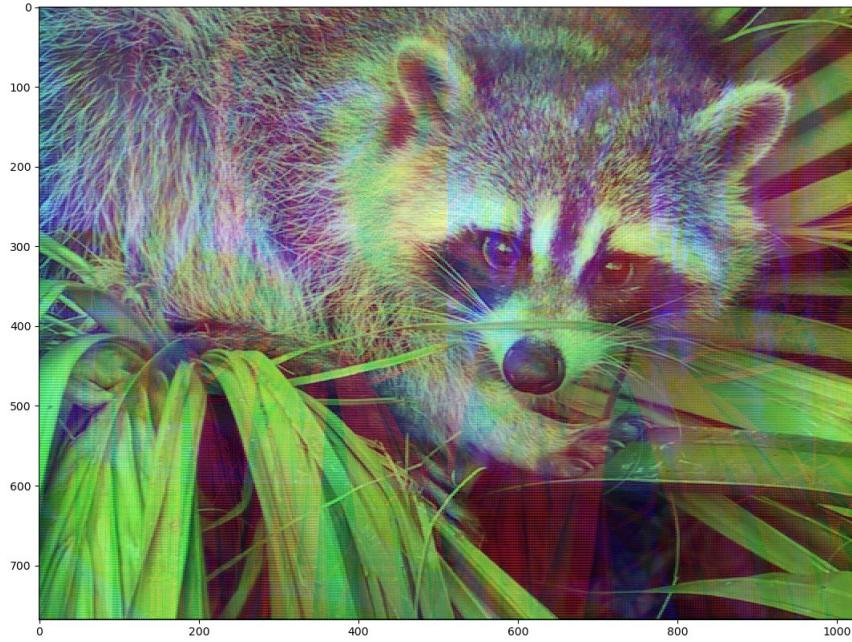


Figure 2: Wrapping av bilde etter et forsøk på demosaicing. Orginalbilde 35

```
t_mask = np.roll(mask, -1, axis=0)
b_mask = np.roll(mask, 1, axis=0)
l_mask = np.roll(mask, -1, axis=1)
r_mask = np.roll(mask, 1, axis=1)
laplace = (
    u[t_mask] +
    u[b_mask] +
    u[l_mask] +
    u[r_mask] -
    4 * u[mask]
)
u[mask] += laplace * alpha
```

Denne metoden vil generere vektorer av bildet som så summeres riktig sammen og vil legges tilbake i indeksene der *mask* har en true verdi. Det er relevant ved bruk av denne metoden å passe på at randen til mask ikke er true, da dette vil forårsake wrapping. Vi kan se denne wrappingen i 2. Her har grønn fargekanal blitt riktig diffundert, mens rød og blå kanal har problemer med wrapping. Vi kan se effekten av wrapping hvis vi kun viser fram den blå kanalen 3 Denne effekten hindres ved å sette hele randen til boolmasken til false før man kjører noen av de overnevnte kodelinjene. Igjen benyttes en Neumann-betingelse på randen for å sette pikselverdiene der riktig.



Figure 3: Wrapping er mer åpenbar når vi kun ser på blå kanal. Orginalbilde 35



Figure 4: 2 regioner i et bilde diffunderes. Orginalbilde 35

Et annet problem som dukker opp fortsatt er at programmet fortsatt vil prosessere en diffusjon over hele bildet. Dette vil være en stor faktor for prosessering av bilder hvis man kun ønsket å glatte en region på $10 \cdot 10$ men har et bilde på $10000 \cdot 10000$. Løsningen til dette problemet er å benytte masken igjen til å generere et view akkurat rundt punktet som skal diffunderes og kjøre prosessering der. En enkel implementasjon av denne metoden er å finne fire ytre kanter i den boolske matrisen som omringer alle sanne verdier i matrisen. Dette kan gjennomføres med en slik kodesnutt:

```
mask = mask.astype(bool)
maskCords = np.argwhere(mask)

top = np.amin(maskCords[:, 0])
bottom = np.amax(maskCords[:, 0]) + 1
left = np.amin(maskCords[:, 1])
right = np.amax(maskCords[:, 1]) + 1

view = u[top:bottom, left:right]
new_view = u1[top:bottom, left:right]
mask = mask[top:bottom, left:right]
```

Her vil $u1$ være en kopi av u men vil være bildet som diffusjon gjennomføres. Dette er en implementasjon av det generelle eksplisitte skjemaet i python, og brukes i vår kode i filen "Diffusion.py".

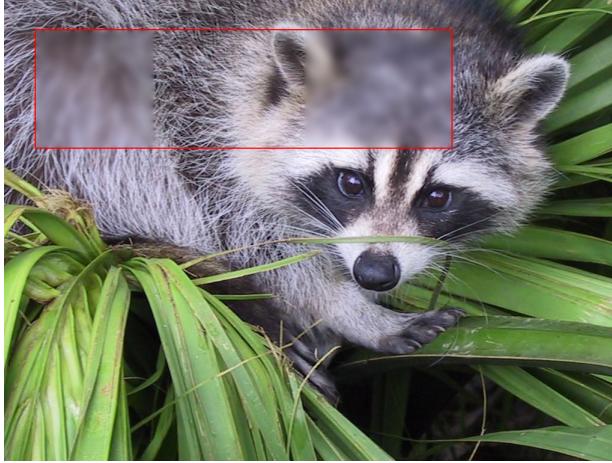


Figure 5: View som programmet produserer. Orginalbilde 35

3.2 Glatting

Generell løsning med Neumann-randbetingelser: $\partial u / \partial x = 0$. Kan også bruke:

$$h = \lambda(u - u_0) \quad \Rightarrow \quad h_{i,j} = \lambda(u_{i,j}^n - u_{i,j}^0)$$

for å konvergere mot en løsning. Dette bruker vi også for å løse Poisson-ligningen for glatting direkte.

$$\Rightarrow u_{i,j}^{n+1} = u_{i,j}^n + \frac{\Delta t}{\Delta x^2} (u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n) - \Delta t \lambda (u_{i,j}^n - u_{i,j}^0) \quad (2)$$

Med $\lambda > 1$ må det klippes. Desto nermere λ blir lik 0, desto glattere blir bildet. Når λ går mot 0 vil bildet være den gjennomsnittlige fargen i bildet.

Glatting, sammen med inpainting kan sees på å være de enkleste metodene å implementere i en bildetransformasjon.

Metoden vil essensielt gå ut på en sum av pikslene rundt seg selv, dette kan sees i likning 2. Vi ser i skjemaet at $i+1, i-1, j+1$ og $j-1$ benyttes for å bestemme verdi for u_{ij}^{n+1} . Dette vil iterativt fjerne skarpe regioner, som diskutert i 3.1, i et bilde som til slutt skaper et utvasket bilde tilnærmet lik andre glattingsalgoritmer slik som Gaussian blur.

3.2.1 Implisitt, Crank-Nicolson og Direkte Glatting

s	n		n											px0
n	s	n		n										px1
	n	s	n		n									px2
	n	s	n			n								px3
		n	s				n							px4
n			s	n			n							px5
	n		n	s	n			n						px6
		n		n	s	n			n					px7
			n		n	s	n			n				px8
			n			n	s				n			px9
				n			s	n						px10
					n		n	s	n					px11
						n		n	s	n				px12
							n		n	s	n			px13
								n		n	s			px14

Figure 6: En matrise som kan brukes i implisitt, Crank-Nicolson eller direkte løsnings skjema for et 5×3 bilde.

2n	n													I M A G e
	n													
	n													
		2n												
			n											
				n										
					n									
						2n								
							n							
								n						
									2n					

Figure 7: Matrisen viser hvordan man legger til Neumann rand betingelser ved å si at piksler på randen legger til n eller $2n$, ”nabo”-konstanten på seg selv. Vi kan lett se hvor hjørnene er i matrisen, da disse cellene vil ha $2n$ i Neumann-randbetingelsesmatrisen

Implisitt, Crank-Nicolson og direkte glatting skjemaene har mye tilfelles. I alle tre tilfellene få et utrykk som

$$s \cdot u_{i,j} + n \cdot u_{i+1,j} + n \cdot u_{i-1,j} + n \cdot u_{i,j+1} + n \cdot u_{i,j-1} = b \quad (3)$$

b er den kjente i dette utrykket, og s og n er konstanter. Dette utrykket kan skrives på matrise form som $Ax = b$. På venstre side får vi en matrise på en form som i figur 6. Måten vi kom frem til dette mønsteret var bare gå ut i fra ligningen 3

Reglene for oppsettet av denne matrisene er som følger:

- Bildet gjøres om til en vektor. Dette gjøres ved å legge å bare sette radene (eller kolonnene) inntil hverandre i rekkefølge.
- På diagonalene nær hoved-diagonalen vil hver x-te ”nabo”-konstant være lik 0, der x refererer til bredden i bildet.
- de to ytterste diagonalene vil være x celler ned og tilsvarende x celler opp (eller til sida) for den øverste.
- en slik matrise vil være $(x \cdot y) \cdot (x \cdot y)$ celler i størrelse

Matrisen (6) inneholder ikke leddene som går utenfor kanten. Hvis vi vil bruke Neumann randbetingelsen, kan vi legge til ”nabo”-konstanten i diagonalen der pikslene ikke har 4 naboyer, se (7). Der en piksel har kun 2 naboyer, vil den legge til nabo-konstanten i seg selv 2 ganger. Siden mye av utledningene av skjemaene blir de samme, viser vi kun utledningen av implisitt skjema under:

3.2.2 Implisitt skjema

$$\frac{u_{n+1} - u_n}{\Delta t} = \frac{1}{\Delta x^2} \nabla^2 u - h$$

hvis $\Delta x = 1$ og $\Delta t = \alpha$ får vi

$$u_{n+1} - u_n = \alpha \nabla^2 u - \alpha h$$

Laplace skal være uttrykt ved u_{n+1} , så vi flytter den over på venstre side, og u_n på høyre side:

$$u_{n+1} - \alpha \nabla^2 u = u_n - \alpha h$$

Vi setter inn Laplace uttrykt ved u_{n+1}

$$u_{i,j}^{n+1} + \alpha(4u_{i,j}^{n+1} - u_{i+1,j}^{n+1} - u_{i-1,j}^{n+1} - u_{i,j+1}^{n+1} - u_{i,j-1}^{n+1}) = u_n - \alpha h$$

Dette setter vi på matriseform. Vi ser at:

$$s = 1 + 4\alpha$$

$$n = -\alpha$$

Høyre siden er $b = u_{i,j}^n - \alpha \cdot h$. Vi konstruer en matrise $A = M + N$ der M er hovedmatrisa (6) og N er Neumann-randbetingelsesmatrisa (7). Dermed kan utrykket løses ved $x = A^{-1}b$.

Implisitt løsning vil også være mye mer stabil, og α kan være svært høy, så i glatting modulen ganger vi α med antall iterasjoner og setter iterasjoner til 1 hvis implisitt metode brukes.

Et slike ligningssystem vil fort bli veldig stort, så i praksis bruker vi sparse modulen i `scipy` for å lage og løse systemer av store glisne matriser. Videre bruker vi en iterative løser "bicgstab" (BICGConjugate Gradient Stabilized), med et initialt gjett på at løsningen blir bildet selv, for å finne en tilnærmet løsning, da dette går mange ganger raskere. En siste ting vi har gjort for å løse ligninger på formen $A(s, n)x = b$, er at for bilder med flere kanaler, kjøres det en egen tråd for hver kanal. Dette gjør det omrent 3x raskere for fargebilder dersom en en datamaskin har 4 kjerner.

3.2.3 Crank-Nicolson

Utledningen av Crank-Nicolson ligner veldig på utledningen av implisitt skjema, men har med en eksplisitt del. Siden Crank-Nicolson bruker et gjennomsnitt av de to, får vi i praksis en $\frac{\alpha}{2}$, med høyre side av ligningen som ren eksplisitt metode:

$$u_{n+1} - \frac{\alpha}{2} \nabla_i^2 u = u_n + \frac{\alpha}{2} \nabla_e^2 u - \alpha h$$

der ∇_i^2 og ∇_e^2 er Laplace operatorene for implisitt og eksplisitt. På grunn av den eksplisitt delen, er ikke Crank-Nicolsons metode stabil for $\alpha > 0.5$, og vi må iterere mer som ved Eksplisitt metode. På en annen side er ikke dette et veldig stort problem i praksis av to grunner. Det ene er at vi allerede bruker en iterativ løser som gjetter på at resultatet vil være bildet selv. Siden en lavere alpha vil forårsake et mindre transformert bilde, ligger allerede den faktiske løsningen mye nærmere orginalbildet enn det gjør for implisitt. Dette gjør at utregningen ved iterativ løsning går mye forttere. Den andre grunnen er at vi fortsatt kan doble α og halvere iterasjoner når vi går over fra eksplisitt metode. Dette gir oss rom for å eksperimentere med hvilke α som er raskest. Et alternativ til Crank-Nicolson kunne vært å bruke Heuns metode, og løse det implisitt, men dette ville gitt oss en glissen matrise med 13 diagonaler siden vi får en ∇^4 i skjemaet vårt, og ville tatt signifikant mer tid å løse enn det allerede implisitt gjør.

3.2.4 Direkte Løsning

For å løse Poisson-ligningen for glatting direkte, løser vi

$$u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} = \Delta x^2 h$$

For at det skal være løselig må resultatet konvergere. Dette gjør vi ved å sette $h = \lambda u - \lambda u_0$. Da får vi følgende ligning:

$$u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} = \Delta x^2 \lambda u - \Delta x^2 \lambda u_0$$

$$u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} - \Delta x^2 \lambda u_{i,j} = -\Delta x^2 \lambda u_{0,i,j}$$

$$(4 + \Delta x^2 \lambda)u_{i,j} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} = \Delta x^2 \lambda u_{0,i,j}$$

Vi kjenner igjen dette mønsteret i fra 3. Vi setter

$$s = 4 + \Delta x^2 \lambda$$

$$n = -1$$

$$b = \Delta x^2 \lambda u_0$$

og konstruerer matrisen A med 6 og randbetingelsen 7, og løser systemet med

$$x = A^{-1}b$$

Løse systemet direkte vil ta omtrent like lang tid som den implisitte. Et triks vi gjør for å gå fra eksplisitt metode til direkte løsning er å si at $\lambda = \frac{1}{1+\alpha \cdot i}$, der i er antall iterasjoner. Desto lavere λ er, desto glattere vil blidet bli.

3.3 Inpainting

Inpainting vil i stor grad benytte seg av de samme strategiene som glatting for å gjennomføre en transformasjon på et bilde. Den største forskjellen er et hyppig bruk av Dirichlet-betingelser, hvor verdier mellom de to glattete regionene settes tilbake til orginalverdi etter en diffusjon. Inpainting ved denne metoden har ikke en særlig god effekt over store regioner, da den er best på naturlige overganger, og ikke innfylling av tapt informasjon. Eksempel på bruk av inpaint kan sees i bilde 8 som inpainted til bilde 9.

Kode benyttet i ”diffuse.py” benytter seg av masker for å angi hvilke regioner hvor verdien skal oppdateres. Det er derfor unødig i vår kode å spesifisere at pikselverdier utenfor regionen som skal inpainted settes tilbake til orginalverdi etter hver iterasjon.

3.4 Demosaicing

Demosaicing benytter seg av inpainting for å male ut piksler rundt den pikselen den har informasjon fra. Modulen for demosaic benytter seg av tre bilder, et rødt, et grønt og ett blått bilde. For at inpainting skal skje i de riktige pikslene trengs det også tre masker for de respektive fargene. Disse kan dannes relativt rask slik:

```
redMask = ~red.astype(bool)
greenMask = ~green.astype(bool)
blueMask = ~blue.astype(bool)
```

Ved å benytte binær not operatoren på hver fargekanal vil python ta inn alle pikslene hvor en fargekanal har en verdi og markere dem som false, alle andre verdier vil være true i den boolske masken og vil bli inpainted. Metoden håndterer regioner med mye informasjon relativt bra, med liten forskjell mellom



Figure 8: En trist vaskebjørn med mange ”hull”. Orginalbilde 34



Figure 9: En mye mer fornøyd, inpainted, vaskebjørn. Orginalbilde 34



Figure 10: Demosaicing av vaskebjørn. Orginalbilde 34

orginalbildet og det rekonstruerte bilde. Skarpe kanter derimot vil være strevsomt da det er kun en eller to piksler å trekke informasjon fra. Det vil dermed være en del rene røde, grønne eller blå piksler på kanter etter at demosaic har kjørt ferdig. Denne effekten kan sees i bilde 10, merk den hvite (tomme) regionen mellom ørene, værhårene og en blå stilk ved vaskebjørnens høyre forlabb som ikke ser riktige ut.

Denne metoden for demosaicing antar et Bayer filter eller lignende for et godt resultat av demosaicing⁴.

3.5 Anonymisering av ansikter

3.5.1 Annsiktsgjenkjenning

Anonymisering fungerer på de samme prinsippene som glatting, så lenge en boolsk maske ligger over alle ansikter i et bilde. Problemet er å generere denne masken automatisk. OpenCV⁵ (Open Source Computer Vision Library) tilbyr en rekke metoder og ferdigstilte nettsideler for gjenkjenning av ansikter. Vi har benyttet oss av to slike moduler for å gjenkjenne ansikter i et bilde. Den ene er en ren ansiktsgjenkjenning og fungerer bra hvis et ansikt har et godt antall piksler (rundt $200 \cdot 200$ og oppover vil den være relativt pålitelig for alle forovervendte ansikter). Denne typen anonymisering er essensielt en blackbox ansiktsgjenkjenning og kan ikke diskuteres mye mer. Effekten av ansiktsgjenkjenning kan sees i bilde 11

⁴<https://patents.google.com/patent/US3971065>

⁵<https://opencv.org>

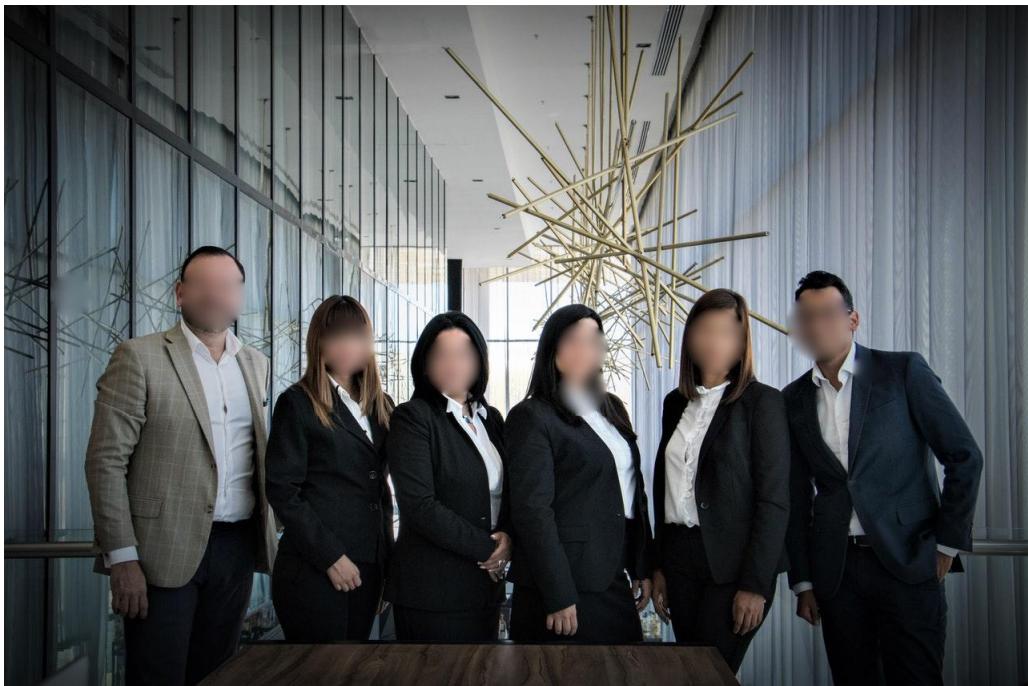


Figure 11: Ansikter anonymisert vha. ansiktsgjenkjenning. Orginalbilde 37

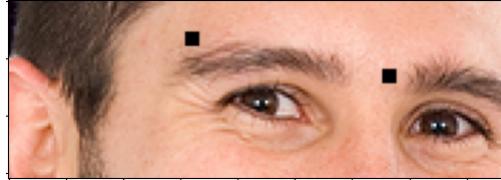


Figure 12: Regionen hvor det letes etter 2 øyne. Orginalbilde 36

3.5.2 Øyegjenkjenning

Det var desverre en del bilder som ikke ble fanget opp av denne ansiktsgjenkjenningen, og gitt at målet med anonymisering var å få alle ansikter anonymisert var et høyt antall falske negative ikke gunstig for formålet. Vi valgte derfor heller å benytte flere metoder for å øke antallet gjenkjente ansikt, med risiko om at regioner som ikke var ansikter også ble anonymisert. Metoden vi benyttet her var også innenfor OpenCV's biblioteker. Vi benyttet oss av funksjonaliteten av øyegjenkjenning som virket til å være mer effektiv når bilder var små og ansiktet tok opp en markant andel av bildet. Metoden gikk ut på å skape et view rundt hvert mulig øye og se om OpenCV's øyedeteksjon hadde funnet et annet øye i denne regionen. Hadde den funnet flere eller færre øyne i regionen var sjansen stor for at det var en falsk positiv, og regionen ble ikke blurred. Vi kan se hvordan et slikt view så ut i bilde 12. I dette bilde er de svarte firkantene 2 punkter hvor den har funnet venstre hjørne av et øye. I dette tilfelle er det 2 treff og den vil anonymisere regionen. Dette gjøres ved å skape en større region midt mellom punktene og sette masken til true i regionen. Bilde av større region kan sees i bilde 13.

3.5.3 Precision og recall

Denne prosessen reduserte antallet falske negative, men økte også antalle falske positive. Vi fikk dermed en høyere recall ($\frac{t_p}{t_p + f_n}$) men også en lavere precision ($\frac{t_p}{t_p + f_p}$). Vi kan se et eksempel på en falsk positiv i figur 14. I dette tilfelle er regionen ganske nær riktig plassering. Dette er ikke alltid tilfelle, og helt tilfeldige regioner i bilde kan markeres for anonymisering.

3.5.4 Implisitt metode

Anonymisering er en modul som ville tjent godt på bruk av implisitt metode⁶. Noen tidligere bilder som ble prosessert kan vi se effekten av en iterasjon med $\alpha = 200$ i bilde 15 og $\alpha = 500$ i bilde 16. I dette bilde brukte implisitt ca. 2 minutter per iterasjon og ville derfor ikke vært mer effektiv en eksplisitt metode. Senere implementasjon av implisitt metode ble aldri stabil nok til å inkluderes lett tilgjengelig gjennom GUI.

⁶Brukt poisson.py, 8. mai. Hash: 6616f540d40a017b380419eab4aaf05a668673e4



Figure 13: Region hvor maske skal settes til true. Orginalbilde 36



Figure 14: En falsk positiv med 2 regioner den tror er øyne. Orginalbilde 36



Figure 15: implisitt diffusjon med alpha = 200. Orginalbilde 36



Figure 16: implisitt diffusjon med alpha = 500. Orginalbilde 36

3.5.5 Avsluttende tanker

Ved kombinert bruk av ansiktsgjenkjenning og øyegjenkjenning fikk vi en akseptabel ansiktsgjenkjenner. Den slipper fortsatt igjennom noen ansikter, spesielt hvis ansikter ikke viser to øyne eller ikke ser mot kamera. Et viktig notat å ta framover er at den slutter helt å fungere hvis ansikter blir rotert ca. ± 30 grader. Det gjelder derfor å sikre seg at bilder er rotert med hake pekene nedover og øyne i en omtrent horisontal linje på hverandre. Dette er en utvidelse som kunne legges til anonymisering hvis det ikke nærmet seg deadline for ferdigstilt applikasjon.

3.6 Konvertering til gråskalabilder

En vanlig måte å konvertere et bilde til gråtone, er å bruke et vektet gjennomsnitt, eller se på en RGB piksel som en vektor der lysstyrken er $\frac{\|P\|}{\sqrt{3}}$ der P er en piksel. Gjør man dette kan informasjon og detaljer i bildet gå tapt, siden flere farger ”mappes” til den samme gråfargen.

Vi bruker en farge til gråtone teknikk som forsøker å ha lite lokal variasjon. Dette gjør vi ved å gjøre følgende:

Vi vil konstruere en gradient g som ligner en gråtone gradient. For å lage en

slik gradient, setter vi lengden

$$l = \sqrt{2} \frac{\|\nabla u_0\|}{\sqrt{6}}$$

. Vi deler på $\sqrt{6}$ for å få lengden l til å ligge mellom 0 og 1. Så ganger vi den med $\sqrt{2}$ for å få gradientens lengde til å ligge i samme ”rom” som en gråtonegradient. Dette gir oss lengden

$$l = \frac{\|\nabla u_0\|}{\sqrt{3}}$$

$$\|\nabla u_0\| = \sqrt{\|\nabla u_{0R}\|^2 + \|\nabla u_{0G}\|^2 + \|\nabla u_{0B}\|^2}$$

Retningen til den nye gradienten kan vi si er gradienten til summen av fargekanalene:

$$\vec{k} = \nabla(u_{0R} + u_{0G} + u_{0B}) + \epsilon$$

Vi gjør dette om til enhetsvektorer med lengde 1 og setter

$$\vec{v} = \frac{\vec{k}}{\|\vec{k}\|}$$

Siden denne gradienten i mange tilfeller ville vært lik 0, har vi introdusert en vektor ϵ med lav lengde slik at k ikke vil ha en lengde på 0. Retningen blir da:

$$\vec{v} = \frac{\vec{k} + \epsilon}{\|\vec{k}\|}$$

Vi får da $g = \vec{v}l$. Vi ser at der \vec{v} går i ϵ retning, vil lengden l være lik 0, og det vil derfor ikke ha en stor innvirkning på resultatet.

Deretter bruker vi Poisson-ligningen og løser $h = \nabla g$ I vår implementasjon har vi satt initialverdien til å være gjennomsnittet av fargekanalene. I tillegg har vi introdusert en ekstra input variabel k slik at $h = k\nabla g$. Dette gir oss mer kontroll over hvor ”sterke detaljer” man vil få frem.

3.7 Kontrastforsterkning

Kontrastforsterkning vil benytte seg av den generelle eksplisitte løsningen og med en h-funksjon:

$$h = k\nabla^2 u_0$$

Denne implementeres ved å innføre en ny Laplace-matrise for det opprinnelige bildet.

```
laplace_0 = (new_img[2:, 1:-1] +
             new_img[:-2, 1:-1] +
             new_img[1:-1, 2:] +
             new_img[1:-1, :-2] -
             4 * new_img[1:-1, 1:-1])
```

Den vil være den samme gjennom alle iterasjonene av diffusjonen og ganges med en konstant k som bestemmer størrelsen på gradienten. Det vil gi mer kontroll over hvor stor kontrastforsterkningen vil være.

Når gradienten blir større kan man risikere at noen verdier faller utenfor definisjonsområdet av pikselverdier. Dette kan løses ved å kippe bildet etter diffusjonen.

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

3.8 Sømløs kloning

Hvis vi kaller originalbildet det skal klones inn i for u_0 , og bildet som det skal klones inn fra for u_1 så vil $u = u_0$ for $x \notin \Omega_i \Rightarrow h = \nabla^2 u_1$ i Ω_i med Dirichlet-betingelsen $u = u_0$ på $\partial\Omega_i$.

Dersom klone-området er på ulike områder kan man sende med to masker til funksjonen. Det implementeres med en liste med to masker. Maskene vil være like dersom kun en maske sendes med.

```
if mask2 is None:
    mask = [mask1, mask1]
else:
    mask = [mask1, mask2]
```

h -matrisen lages ved hjelp av den andre masken på følgene måte

```
h = (clone_source[np.roll(mask[1], -1, axis=0)] +
     clone_source[np.roll(mask[1], 1, axis=0)] +
     clone_source[np.roll(mask[1], -1, axis=1)] +
     clone_source[np.roll(mask[1], 1, axis=1)] -
     4 * clone_source[mask[1]])
```

Laplace-matrisen lages på samme måte med den første masken og vil oppdateres ved hver iterasjon. Diffusjonen vil skje på hele bildet og vi må derfor revertere endringene på området som ikke er en del av masken. Det gjøres på følgende måte.

```
new_img[~mask1] = img[~mask1]
```

Etter diffusjon må man kippe verdier utenfor definisjonsområdet på samme måte som i 3.8.

3.9 Kantbevarende glatting

Kantbevarende glatting gjøres ved å lage et vektet bilde $D = \frac{1}{1+k||\nabla u_0||^2}$, Se 17. Når størrelsen på gradienten blir stor, vil D gå mot 0. Dette kan vi bruke til å sette opp ligningen:

$$\begin{aligned}\frac{\partial u}{\partial t} &= \nabla \cdot (D \nabla u) \\ \frac{\partial u}{\partial t} &= D \nabla^2 u\end{aligned}$$

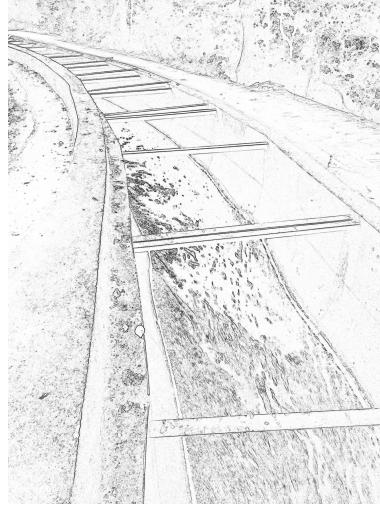


Figure 17: Vektet bilde D

Og vi får følgende eksplisitt skjema:

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

$$u_{i,j}^{n+1} = u_{i,j}^n + \alpha D_{ij}(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n)$$

I praksis blir det som å bruke en maske, men i stedet for at en piksel enten glattes eller ikke, blir den vektet glattet.

3.10 HDR bilder

HDR har vi implementert vha. utledningene og matlab-koden tilDebevec og Malik [4]. For å finne responskurven til bildet må vi hente prøver av pixler fra bildene. Dette er gjort ved å ta utgangspunkt i et av bildene med middels eksponering for å finne områdende i bildet hvor det er mørkt/lyst. Deretter går man igjennom alle bildene og tar tilfeldige prøver av pixler. Dette er implementert på følgende måte

```
for i in range(num_intensities):
    rows, cols = np.where(mid_img == i)
    if len(rows) != 0:
        idx = random.randrange(len(rows))
        for j in range(num_images):
            intensity_values[i, j] = images[j][rows[idx], cols[idx]]
```

Figur 3.10 viser hvordan Z-matrisen kan se ut.

For å estimere responskurven til bildet tar man utgangspunktet i likning 4

$$\mathcal{O} = \sum_{i=1}^N \sum_{j=1}^P \{w(Z_{ij})[g(Z_{ij}) - \ln E - \ln \Delta t_j]\}^2 - \lambda \sum_{z=Z_{min}+1}^{Z_{max}-1} [w(z)g''(z)]^2 \quad (4)$$



Figure 18: Eksempel på prøver av pikselverdier i 15 bilder

Der Z er matrisen diskutert tidligere med i som piksel lokasjon og j som bilde-index. g er responskurven, E er lysintensitet og Δt er eksponeringstiden. w er en vektfunksjon som gjør at pixler i midten av skalaen får mer vekt enn de på kantene. Den er implementert på følgende måte.

```
def w(piksel_value):
    if piksel_value <= (0.5 * (Z_MIN + Z_MAX)):
        return piksel_value - Z_MIN
    return Z_MAX - piksel_value
```

For å løse likningen brukes algoritmen fra [4] ved å oversette matlabkoden til Python.

Når g er funnet kan vi iterere igjennom alle pixlene i bildet og rekonstruere et HDR-bilde vha likning 5

$$\ln E_i = \frac{\sum_{j=1}^P w(Z_{ij})(g(Z_{ij}) - \ln \Delta t_j)}{\sum_{j=1}^P w(Z_{ij})} \quad (5)$$

Disse algoritmene baserer seg på gråskalabilder. Dersom man ønsker å rekonstruere et fargebilde må hver fargekanal rekonstrueres selvstendig. Deretter kan de settes sammen til et fargebilde.

3.10.1 Optimalisering

For å konstruere et HDR bilde må man gå gjennom hver piksel, og for hver piksel gå gjennom hvert bilde og bruke funksjonen g som ble returnert for å bestemme den nye piksel verdien. Siden g ikke er en funksjon, men en array kan man ikke direkte gjøre:

```
# x : numpy.ndarray
# g : numpy.ndarray
newX = g[x]
```

Bruken av forløkker i python fører til at prosessen går tregt, og kunne ta flere minutter å få lastet inn.

Vi løste dette med å bruke `numpy.take()`. Den tar inn to arrayer, en array man henter elementer fra, og en annen array som indekserer de elementene man vil hente. Dette bruket vi til å få ”mappet” en array med en annen.

3.11 Grafisk brukergrensesnitt

3.11.1 Brukergrensesnittet i korte trekk

I brukergrensesnittet kan brukeren velge lett til et til mange bilder ved å klikke på + tegnet opp til venstre, eller lage et hdr bilde ut av flere bilder ved å klikke på HDR knappen øverst til venstre.

Når man legger til et bilde, må man klikke på ”åpne” ikonet for å hente inn et bilde. Til høyre kan brukeren velge å legge en til mange transformasjoner. Man kan slette en transformasjon, eller bytte om på rekkefølgen på de med pilene. Transformasjonene på bildet vil skje i rekkefølgen fra øverst til nederst. Sluttdokumentet vil vises i det store bildet i midten.

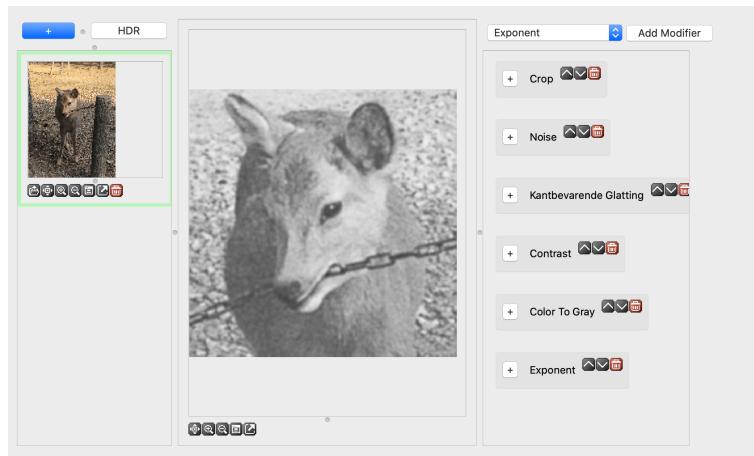


Figure 19: Bilde viser hvordan grensesnittet ser ut. Her har en bruker eksperimentert med blant annet kantbevarende glatting og kontrastforsterkning, og har forsøkt å få bildet til å se ut som et bilde fra 1800 tallet. Orginalbilde 38

3.11.2 Objekt Orientert Design

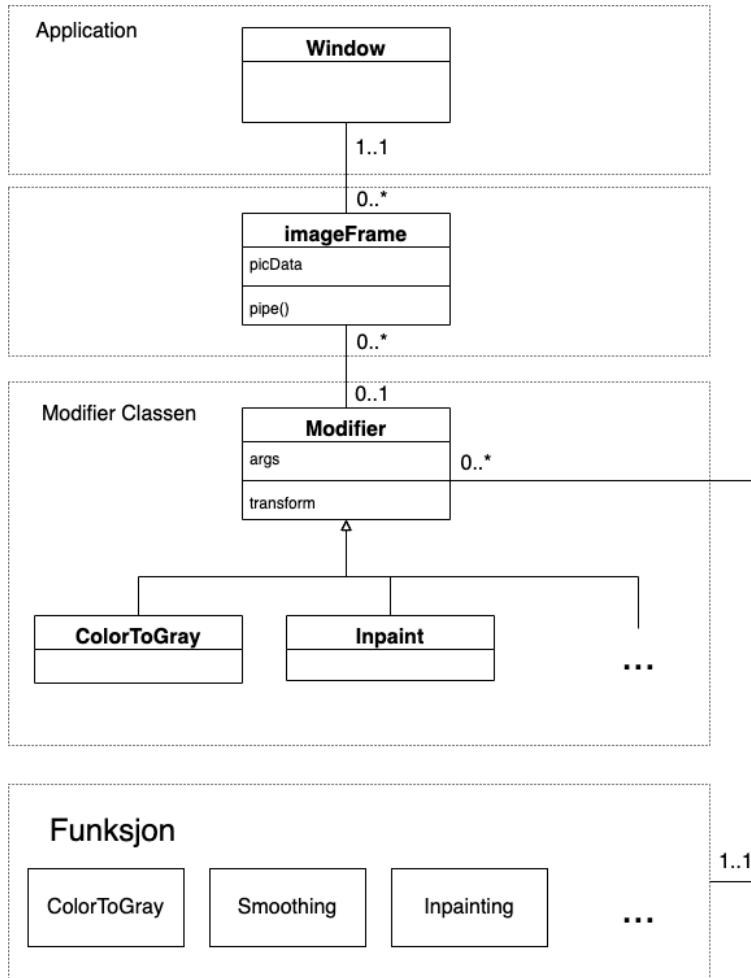


Figure 20: Figuren viser i korte trekk hvordan brukergrensesnittet er bygd opp. Applikasjonen har imageFrames som har en modifier. Denne brukes for å transformere imageFrame sin data

Noen av de viktigste objektene og klassene å forklare er

- Vindu
- imageFrame
- Modifier

Se figuren 20 Vinduet har flere imageFrame-er. Disse imageFrame-ene inneholder enten 1 eller ingen Modifier objekter. imageFrame brukes for å holde bilde-data,

vise bilde-data og sende bilde data videre til andre bilder. Se programflyten 21. Alle modifierer er underklasser av Modifier klassen. Dette gjør at Modifier klassen kan ta kontroll over formatering av bilde data. For eksempel: er input bildet et svart hvitt bilde? Skal kun rød-kanalen brukes? Er det rgb, eller rgba, eller er det en maske, og hva slags format er det på output? og så videre. En modifier underklasse inneholder funksjon som kalles for å transformere data. Denne funksjonen kalles når Modifier klassens transform() kalles. Denne funksjonen tar også hånd om clipping, og sørger for at output ikke går over 1 eller under 0.

3.11.3 Sekvensdiagram

For å forklare flyten i programmet, tar vi for oss et tilfelle der bruker har lagt til et bilde og en modifier på bildet, og endrer på en parameter i modifieren, se 21.

I diagrammet er imageFrame1 på toppen av pipelinen, altså bildet brukeren selv har valgt fra sin datamaskin, og imageFrame2 er bildet som hører til en ModifierWidget brukeren akkurat har endret på. Bildene er koblet sammen gjennom "pipes" som i praksis er lister over imageFrames en imageFrame skal sende data til. Når brukeren endrer en verdi, vil en on-change funksjon kjøres. Denne funksjonen vil oppdatere verdiene til dets bildes modifier. Deretter sier den til dets bildes "kilde", image1 at den må sende data på nytt gjennom pipe(). Når bilde-dataen er sendt kjøres det først gjennom modifieren sin transform() før det blir lagret i imageFrame2. Dette vil fortsette rekursivt om det ved tilfellet med flere modifier widgets, til det når et bilde som ikke har noen "pipes", som regel hoveddisplayet.

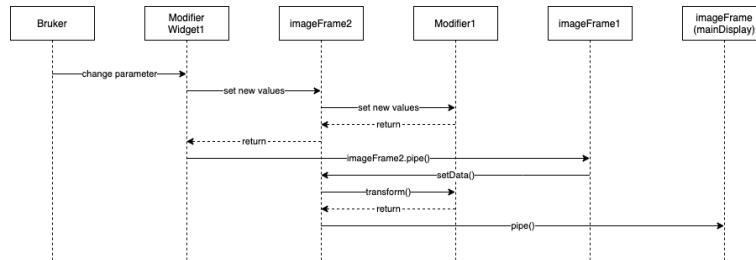


Figure 21: Diagrammet viser hvordan flyten av programmet går når brukeren endrer på en verdi.

3.11.4 Argumentasjon for Arv

Det hadde også vært mulig å lage én klasse "Modifier" som ikke har noen underklasser, men som inneholder en funksjon og et navn. Grunnen til at vi har delt det opp slik vi har gjort, er for å få mere fleksibilitet. For eksempel, er det mulig å overskrive transform funksjonen til å gjøre flere ting.

En annen grunn var at det gjorde det lett for oss å instansiere en modifier, noe

som vi følte det det gjorde det mer leselig, og lettere å unngå at en og samme modul blir referert til når det skulle ha vært flere.

3.11.5 Argumentasjon for bruken av pipelines

En av fordelene ved å bruke pipelines er at man kan lage mer komplekse systemer av transformasjoner, som kan deles opp i små deler som brukeren har kontroll over. For eksempel: Du har et stort bilde du legger en crop transformasjon på. Deretter pipes det til noise for å simule et bilde med mye støy, før det pipes til kantbevarende glatting for å visualisere fjering av støy igjen. Deretter kan man eventuelt se om bildet eventuelt blir klarere igjen hvis man legger på kontrastforsterkning. Herfra kan man prøve å endre på parameterene og se hvordan resultatet blir.

4 Utviklingsprosess

Siden vi er en gruppe på tre, har vi valgt å gå for en enkel ”prosjektorganisasjon”. Vi har:

- Prosjektleder
- Numerisk utledning og Matematikk-ansvarlig
- Grensesnitt og Test-ansvarlig

Alle rollene tar del i utviklingen, kvalitetssikring av kode og testingen, men får ansvar for forskjellige deler av prosjektet.

4.1 Rollebeskrivelser

4.1.1 Prosjektleder

Prosjektleder vil få ansvaret for planlegging, estimering og gjennomføring av prosjektet. Aslak Tøn tok på seg ansvar som prosjektleder, da han har tidligere erfaring med gruppeledelse og organisering.

4.1.2 Numerisk utledning og matematikk-ansvarlig

Håkon Trøan Muggerud har erfaring i matematikk og er derfor tilordnet ansvaret for utledningen av de numeriske skjemaene og kvalitetssikring av utledede skjemaer gjort av andre i gruppen.

4.1.3 Grensesnitt og test-Ansvarlig

Isac Kallevig har litt erfaring innenfor utvikling av applikasjoner i python, og tok derfor de delene av oppgaven som var minst relatert med arbeid gjennomført tidligere i semmesteret i sammenheng med obligatoriske arbeidskrav. Han har

som hovedansvar å utvikle brukergrensesnittet, og er også tilordnet ansvaret for integreringen og testingen av modulene opp i mot brukergrensesnittet, siden de henger mye sammen.

4.2 Smidig Systemutviklingsmodell

Siden vi er få utviklere på dette prosjektet, har vi valgt å gå for en ”forenklet” hybrid av scrum og extreme programming, med arbeidflyten til scrum og noen av verdiene fra extreme programming, som for eksempel allemanns eie.

4.3 Styrende Faktorer

Vi valgte å gå for denne utviklingsmodellen da vi har et behov for fleksibilitet i utviklingen. Det er også vist at en mer smidig utviklingsmetode fungerer bra i mindre grupper slik som vår. Ved riktig bruk av modellen vil også problemer som en kort tidsfrist, mange moduler og et behov for testing være håndtert bedre en mer planbasserte utviklingsmodeller.

5 Kvalitetssikring

Prosjektgruppen skrev under på en arbeidskontrakt i starten av arbeidsperioden. Denne arbeidskontrakten vil stå til grunnlag for punktene diskutert under denne seksjonen.

5.1 Møter

Arbeidskontrakten beskrev at ukentlige møter skulle holdes kl. 12 onsdag. Disse møtene ble holdt ukentlig, og var generelt nyttig for alle på gruppen. Det nevnes ikke i kontrakten at disse møtene kan flyttes ved muntlig avtale, som ble gjort ved noen tilfeller. Dette burde nevnes i framtidige kontrakter, slik at klarhet rundt hvordan møtene organiseres.

Det ble heller ikke alltid skrevet referater fra møtene, og referatene var ikke alltid utfyllende nok. Skriving av referater ble bedre en stund etter overgang til ny kommunikasjonskanal (se 5.2). Det var et par eksempler hvor vi ønsket å peke tilbake til tidligere møter, men hadde ikke noe stikkord fra møtet som støttet under påstandene vi ønsket å gi. Dette problemet kunne rettes ved en klarere referat mal, og ansvarsfordeling av hvem som burde skrive referat.

5.2 Kommunikasjonskanal

Det ble skrevet i kontrakten at Messenger⁷ skulle brukes som en felles kommunikasjonskanal i gruppen. Det ble muntlig bestemt innad i gruppen å bytte over til Discord⁸ 18. Mars (siste melding sent i Messenger kl 11.58 Onsdag

⁷<https://www.messenger.com>

⁸<https://discord.com>

18. Mars). Dette er grunnet et mer fleksibelt oppsett på Discord, som tillot seksjoner med bilde-eksempler, referater, generell diskusjon og andre nyttige grupperinger av samtaler. Det ble ikke skrevet en oppdatert arbeidsavtale når dette ble innført, som strider med kontraktens siste avsnitt under ”Arbeid som forventes”. Dette resulterte ikke i store problemer da hele gruppen var stort sett enige i alle beslutninger, men kunne vært et problem hvis store problemer oppstod under utviklingsfasen.

5.3 Testing

Kontrakten nevner at metoder for test driven development (TDD) skal benyttes under utviklingen av prosjektet. Dette ble overholdt i svært liten grad i selve prosjektet. Et par problemer oppstod som hindret denne metodikken. Dette var en mangel på testkompetanse og planlegging før utvikling.

5.3.1 Testkompetanse

Det var liten kompetanse innen feltet fra noen av gruppemedlemmene før starten av utviklingsfasen. Det var derfor vanskelig å generere gode tester før applikasjonen var utviklet. Vi unngikk derfor utvikling av tester før moduler i håp om at det ble lettere etter en modul var utviklet. Dette viste seg å ikke være tilfellet, og et nytt forsøk på å implementere TDD burde forsøkes ved neste prosjekt. Dette kan gjennomføres ved et større fokus på TDD og en sterk enighet innad i gruppen om at vi ønsker å benytte oss av TDD. Gruppen må være enige om at TDD er et nyttig utviklingsprinsipp for å gjennomføre det i neste prosjekt.

5.3.2 Planlegging

Ved start av prosjekt var det mye usikkerhet om hvordan applikasjonen skulle utvikles, det ble derfor en enighet om at vi tidlig skulle begynne å utvikle prototyper for å se hvordan applikasjonsflyten ville bli framover. Det ble derimot ikke avtalt at disse prototypene skulle forkastes eller at et møte skulle diskutere en generell arkitektur på programvaren etter litt klarhet var kommet fra arbeid på prosjektet. Arbeid framover ble derfor ofte bassert på en ustødig base. Det har dermed i senere tid kommet en del behov for refactoring av kode og en ustabil interface som jobber opp mot de forskjellige modulene. Vi burde spesielt ha sett litt mer på en generell diffusjonsmodell da denne står til grunn for mye av det andre arbeidet som blir gjort i applikasjonen.

5.3.3 Akseptansetesting

Et område som ikke er implementert for programmet er akseptansetesting. Vi ønsker i framtiden å inkludere en tabell for hver modifier i UI som ser ut som tabell 1. Her vil en tester bruke 10 minutter per modifiers som finnes i GUI, forandre på alle parametere som er tilgjengelig i GUI. Resultatet av testen markeres med en ”OK” eller ”NOT OK” i siste boksen i tabellen.

Modifier	Parm1	Param2	...	Param N	Test OK?
----------	-------	--------	-----	---------	----------

Table 1: Mulig testtabell

5.4 Avsluttende tanker

Prosjektet startet for gruppen relativt raskt etter oppgavebeskrivelsen var utgitt, men vi burde som gruppe tatt et par møter ekstra hvor vi diskuterte oss igjennom en generell struktur før hovedarbeidet med utvikling startet.

5.5 Arbeidsfordeling

Arbeidskontrakten nevnte at timelister skulle til en viss grad føres opp, slik at arbeid ble jevnt fordelt mellom gruppemedlemmene. Dette punktet ble ikke fulgt opp av noen av gruppemedlemmene. Det var fortsatt en god fordeling av arbeidsmengde mellom gruppemedlemmene. Ingen i gruppen ga utsynk for en særdeles over- eller underbelastning i arbeidsmengde under arbeidsperioden. Et par møter nevnte en ubalanse mellom utviklere, da spesielt arbeid på grafisk brukergrensesnitt tok opp mye tid for ansvarlig utvikler. Dette punktet ble aldri tatt opp i særdeles stor grad. Det var enighet innad i gruppen at det ikke forventes mer enn 10 uketimer med arbeid direkte på prosjektet, og det kom fram at alle i gruppen overholdt denne muntlige avtalen.

5.6 Kodestandard

Ingen helt spesifiserte kodestandard ble gitt under utviklingen av applikasjonen. Det var likevel forventet at alle skulle overholde kodestandarder gitt i python linteren Flake8⁹. Denne linteren, benyttet gjennom VS Code ga en relativt god kodestandard. Dette gjorde leselighet i koden mye bedre en uten bruk av noen linter. Denne delen av kvalitetsikringen har fungert meget bra. Et problem som oppstod med bruk av linter var at kodestandard var ikke krevd for å pushe ny kode til master branch. Dette burde overholdes ved neste prosjekt for å hindre store arbeidsmengder i slutten av prosjektet for å overholde kodestandard.

5.7 Robust kode

Kodebasen og bruk av koden gjennom GUI er fortsatt relativt ustabil med hyppige krasjer av programmet. Dette kunne til en viss grad bli bedret av en del try-catch blokker i koden. Dette ville ikke hindret at problemet oppstod, men ville i hvertfall gitt feilmeldinger til bruker istedenfor å avslutte programmet før bruker får mulighet til å lagre foreløpig arbeid. Dette vil være en relevant utvidelse av prosjektet hvis det hadde vært mer tid avsatt til utvikling.

⁹<https://pypi.org/project/flake8>

6 Resultater og diskusjon

6.1 Generell diffusjon

Den generelle diffusjonsløsningen som er benyttet i "diffusion.py" fungerer i stor grad men har et par problemer. Diffusjonsmetoden benyttet i "diffuse.py" er ikke benyttet av kontrastforsterkning og sømløs kloning. Dette er grunnet en markant forandring i hvordan disse kodeblokkene utfører diffusjon i forhold til de andre modulene.

Kontrastforsterkning benytter en Laplace₀ som er bassert på orginalbildet. Denne modulen er unik i å benytte dette, og selv om det hadde vært mulig å introdusere det inn i den generelle funksjonen pre_diffuse ville dette skapt en arkitektur som hadde vært vanskelig å vedlikeholde framover. På lignende måte var det heller ikke mulig å benytte sømløs kloning.

I denne modulen benyttes $\alpha \cdot \text{laplace} - \alpha \cdot h$. Dette er forskjellig fra de resterende modulene som generelt benytter $\alpha \cdot \text{laplace} - h$. Dette ville også vært mulig med if-tester langt ned i funksjonsstacken, men vanskelig å vedlikeholde. Vi valgte også her å benytte en egen diffusjonsmetode for denne modulen.

6.2 Glatting

Glatting oppnådde mye av de målene vi hadde satt fram ved starten av prosjektet. Modulen har mulighet for glatting av spesifikke regioner, og vil også kun kjøre glatting over et view av bilde for potensielt raskere prosessering av bildet. Modulen har både en eksplisitt og en implisitt løsning, den eksplisitte ble utviklet først.

6.3 Inpaint

Inapint oppnådde også mange av målene som ble satt ut i starten av prosjektpérioden. Den kjører relativt raskt igjennom en diffusjon og får en pen overgang fra utenfor inpaintet område og inn som kan sees i bildene 8 og 9. Inpaint som glatting fungerer til en viss grad med implisitt metode, men har vist noen problemer, spesielt i sammenheng med demosaic som benytter seg av inpainting. Her skulle vi gjerne vist til noen bildeeksempler, men da kodebasen har blitt forandret siden eksempelbildene ble generert, og bildene ikke ble lagret er dette ikke mulig.

6.4 Demosaicing

Som diskutert i 3.4 vil demosaic produsere en god rekonstruksjon av et bilde i regioner der mye informasjon er tilgjengelig, men vil streve i regioner den ikke kan trekke data fra 4 nabopiksler. Denne effekten ville vært mindre åpenbar ved bruk av flere iterasjoner, men ville aldri forsvinne helt. Ved å prioritere hastighet høyere får vi en relativt god demosaic algoritme med 10 iterasjoner gjennom diffusjons metodene.

6.5 Anonymisering av ansikter

Anonymisering av ansikter har oppnådd en 100% recall ($\frac{t_p}{t_p + f_n}$) og en akseptabel presisjon. Vi er her konsekvent litt vase i språket da vi ikke har klart å bygge opp et testrammeverk for hvor god anonymiseringsalgoritmen er, og siden mange av bildene brukt som test for anonymiseringen er av ukjent opphavssrett har vi valgt å ikke inkludere dem i rapporten. Dette er definitivt et område som burde bli mer utforsket i sammenheng med anonymisering. Tilgjengelighet på testdata og et stort bibliotek med bilder av ansikter, både i grupper og portrettfoto hindret utvikling å testing av denne modulen.

Andre utvidelser som burde blitt implementert er mulighet for rotasjon av bilder som diskutert i 3.5. Anonymisering er ofte en metode som er brukt på store databaser av fotografier, og å sikre seg at disse alle er rotert riktig på forhånd ville redusert effekten til en anonymiseringsalgoritme.

6.6 Konvertering til gråskalabilder

Vi mener at vi har oppnådd et bra resultat med implementasjonen av konverteringer til gråtone bilde.



Figure 22: Bildet viser resultatet av konverteringen til gråtone med Poisson-implementasjonen vår.



Figure 23: Bildet viser resultatet av konverteringen til gråtone med gjennomsnittet av fargekanalene.

Det øverste bildet som bruker Poisson-teknikken får frem flere detaljer, mens gråtonebildet som benytter seg av et gjennomsnitt ser litt uskarpt ut i forhold. Forskjellen mellom de to kommer tydelig frem hvis vi tar differansen mellom de 25.

Figure 24: Vaskebjørt midt på svarte natten /s



Figure 25: Bildet viser absoluttverdien til differansen mellom de to bildene. Det er blitt tatt en kvadratrot av bildet for å gjøre forskjellene mer tydelige

6.7 Kontrastforsterkning

Kontrastforsterkning har oppnådd et veldig bra resultat som vist på figur 6.7. Eksempelet er etter kun én iterasjon med diffusjon, så den er også ganske effektiv. Ettersom man kan styre brattheten til gradienten og antall iterasjoner får man ganske mye kontroll over hvor stor forsterkningen blir.

Som nevnt i 6.1 så benytter ikke denne modulen seg av den generelle diffusjonsmodulen. Dersom dette hadde blitt implementert ville det lagt til funksjonalitet som for eksempel forsterkning bare på et område av bildet.



Figure 26: Eksempel på kontrastforsterkning (imgur)

6.8 Sømløs Kloning

Sømløs kloning har oppnådd et brukbart resultat som vist i figur 6.8. Et problem som oppsto er at den er veldig ineffektiv. Det kreves fryktelig mange iterasjoner før man får et greit resultat. Kloningen i figur 6.8 er oppnådd etter 10.000 iterasjoner. Dersom vi kunne brukt implisitt løsning kunne alpha vært mye større og dermed hadde vi ikke trengt like mange iterasjoner. Som nevnt i 6.1 hadde vi ikke ressurser til å integrere generell diffusjon i denne modulen, og dermed fikk vi ikke testet implisitt løsning for sømløs kloning.

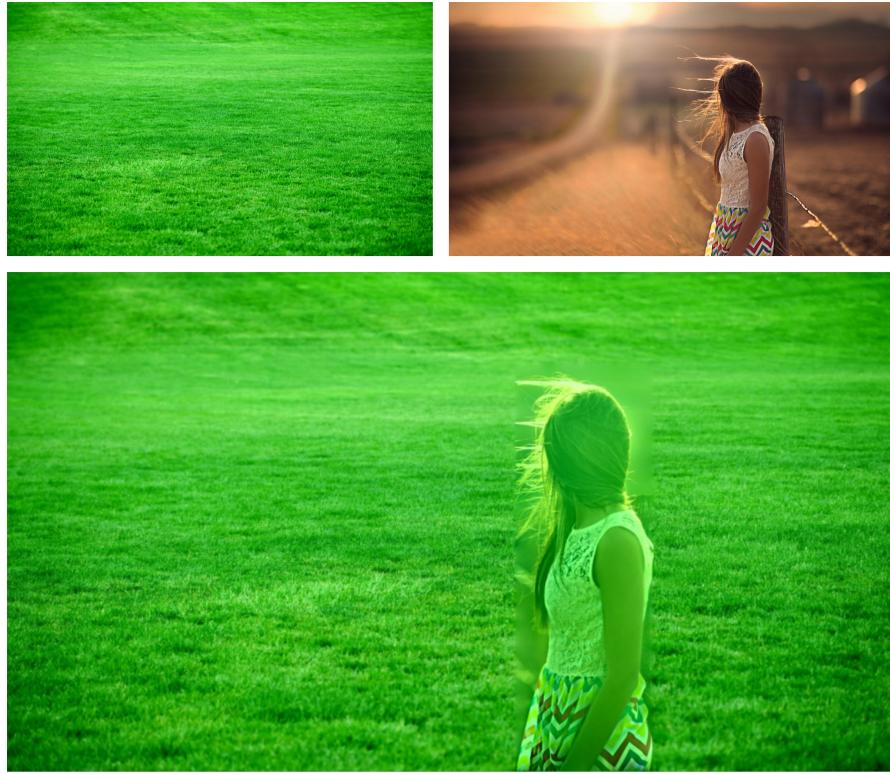


Figure 27: Eksempel på et klonet bilde. Damen i *høyre* bilde er klonet inn i *venstre* bilde

6.9 Kantbevarende glatting

Den kantbevarende glattings-modulen funger godt for mindre bilder, men det skal en del iterasjoner og en høy k (kantbevaringsvariablen) til for å se en signifikant forskjell.



Figure 28: Bildet viser et kantbevart stort bilde.



Figure 29: Kantbevart Glattet Rådyr

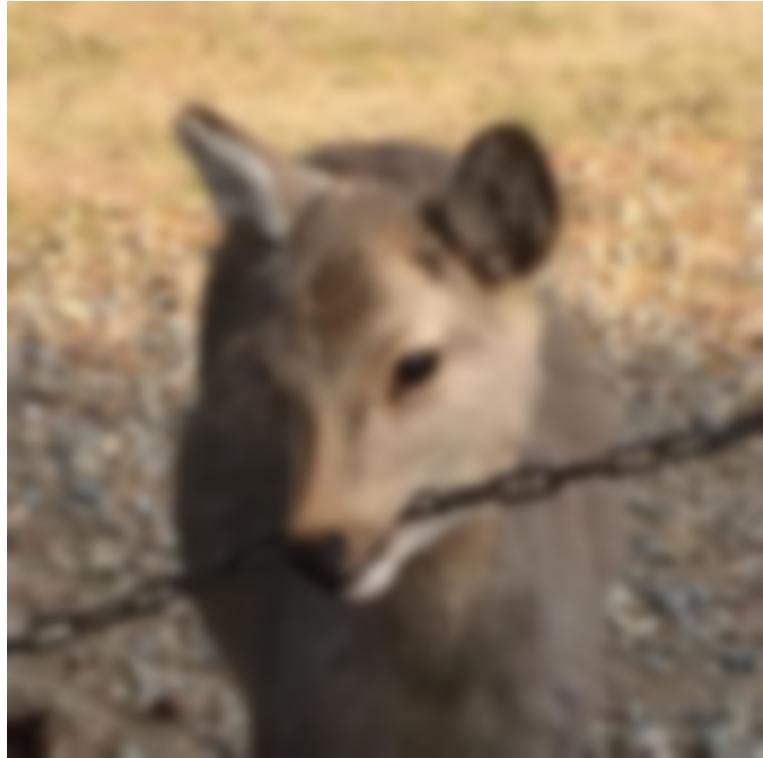


Figure 30: Glattet rådyr uten kantbevaring

Siden vi må kjøre en del iterasjoner for å få til en signifikant forskjell hadde det for mindre bilder vært nyttig å få til direkte løsning for kantbevarende glatting også.

6.10 HDR bilder

Rekonstruksjon ga ikke alltid tilfredstillende resultater. Etter introduksjon av `numpy.take()` ble generelt alle bilder lysere. En tidligere versjon som bruker mange minutter på å gjennomføre en HDR konstruksjon kan finnes med hash `516db3baccf9550e5ecd8b422c0078cbc858a732`.

Et kjent problem i HDR modulen er at den vil krasje ved innlasting av monokromatiske bilder. Dette må unngås ved å ikke benytte monokromatiske bilder for rekonstruksjon av HDR bilder.



Figure 31: Rekonstruksjon av HDR-bilde. Basert på 15 bilder med ulik eksponering

6.11 Grafisk brukergrensesnitt

6.11.1 Resultat og innføring i bruken av applikasjonen

Bildet (32) viser sluttresultatet av brukergrensesnittet.



Figure 32: Bildet viser hvordan sluttresultatet av brukergrensesnittet ser ut

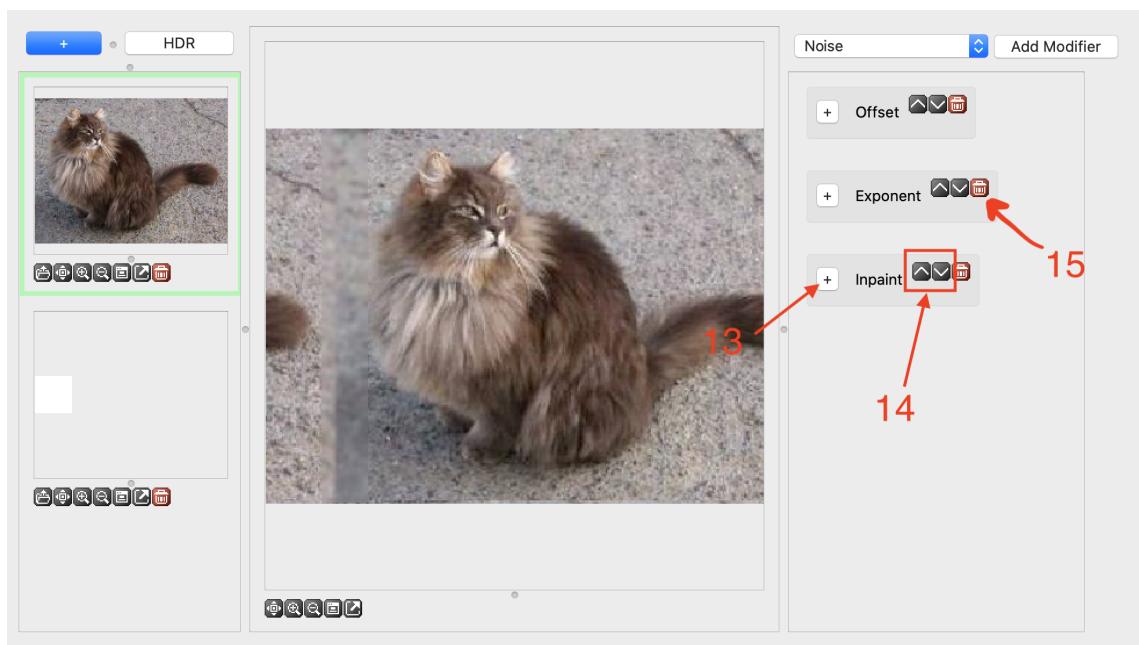


Figure 33: Bildet viser blant annet hvordan pipelinen ser ut

En kort innføring:

1. Leser inn et bilde fra fil.
2. Tilpasser bildet sin ramme
3. Sletter bildet
4. Bildet kan dobbeltklikkes for å laste inn et bilde
5. Legger til en ny ressurs (tomt bilde)
6. Velger en mappe for å lage et HDR bilde.
7. Resultatet av en pipeline bakes inn i et nytt bilde
8. Eksporterer resultatet som .png
9. Velger en modifier fra liste. Poisson modifierene ligger høyt opp og hjelpe-moduler ligger lenger nede.
10. Legger til valgt modifier
11. Input widget. Her kan det skrives inn et tall, eller man kan klikke å dra for å endre input-en.
12. Akkurat denne input widgeten brukes for å velge en løsningsmetode. 0 er eksplisitt, 1 er Crank-Nicolson, 2 er implisitt og 3 er direkte løsning.
13. Utvidider grensesnittet til en modifier eller kollapser den. (Vis eller skjul parametere)
14. Piler som brukes til å endre rekkefølgen på pipeline widgetene
15. Sletter en modifier

I bildet (33) ser vi et eksempel på en rekke modifiers som transformerer bildet. Først flyttes katten med en offset. Deretter brukes en eksponent $x < 1$ slik at bildet blir lysere. Deretter kjøres inpaint for dekke over litt av kanten som kom med når bildet ble flyttet.

6.11.2 Feil og Problemer

Applikasjonen kan kjøre stabil hvis man vet hva man gjør, eller ikke prøver å krasje det med hensikt. Det ligger fortsatt mange skjulte bugs i den, men noen av de kjente problemene er:

- I Noise modifier, hvis man velger en seed under 0
- På windows vil man kanskje få feilmelding når man lukker applikasjonen
- Små feil i moduler som gjør at applikasjonen krasjer ved for eksempel dele på 0 feil.
- Applikasjonen fryser ved store bilder eller mange iterasjoner på grunn av kalkulasjonstiden det tar å få resultatet.

6.11.3 Optimaliseringer og Forbedringer

En ting som burde vært optimalisert, men som dessverre ikke er det, er at kalkulasjoner skulle kjørt på en egen tråd, for å unngå at programmet fryser hver gang man gjør endringer. I utgangspunktet var det meningen å kunne se live endringer man gjør, men med store bilder ender det fort opp med at applikasjonen konstant fryser hver gang man endrer på input verdier.

I tillegg skulle det vært introdusert en "mute" på slik at man ikke trenger å slette en modifier for å se forskjellen uten den hver gang.

7 Testing

Testing av applikasjonen er nok den delen av prosjektet som ble mest nedprioritert av prosjektgruppen. Dette var ikke grunnet en tanke om lite behov for testing, det var heller forårsaket av en usikkerhet på hvordan testing skulle gjennomføres på en god måte. Testing har derfor gjennom utviklingsperioden gått ut på å prøve modulen som ble laget eller modifisert, og se om resultatet ble omtrent som forventet.

Testrammeverk ble til dels implementert mot slutten av utviklingsperioden, og benyttet seg av python unittest og coverage for testing og testdokumentasjon. Testrammeverket tester stort sett veldig brede faktorer som om et bilde har difusert, eller om noe er klonet inn i et annet bilde vha. Testing av kloningskoden er et godt eksempel på dette.

```
self.img1 = np.zeros((11, 11, 3)).astype(float)
self.img2 = np.zeros((11, 11, 3)).astype(float)
self.img2[2:4, 2:4] = 1
self.mask1 = np.copy(self.img2).astype(bool)
self.itr = 5
self.alpha = 0.24

self.img = cloning.cloning(
    self.img1, self.img2, self.itr, self.mask1, None, self.alpha
)

self.sum = np.sum(self.img)
self.assertNotAlmostEqual(self.sum, 0)
```

Her testes det kun om summen i bilde 1 er forskjellig fra 0 etter en kloning av noen piksler. Dette betyr ikke at kloning var god eller riktig, bare at en forandring av pikselverdier i bilde en har forekommet.

Det skal forsatt sies at noe testing er bedre enn ingen testing, da vi her har en sjanse for å oppdage om noe er helt feil etter forandring i kode. Dette kan hindre skulte bugs som blir introdusert senere i utvikling av programvaren.

8 Diskusjon

8.1 Målloppnåelse

Målene med utviklingen av applikasjonen gitt i systemkrav 2 har nådd et akseptabel nivå innenfor alle modulene som var spesifistert. Disse modulene var: Glatting, inpainting, demosaicing, kontrastforsterkning, konvertering til gråskala, kantbevarende glatting, rekonstruering og visualisering av HDR bilder, anonymisering av ansikter og en GUI for å representer disse modulene. Manglene i disse modulene har blitt tatt opp i resultatene for hver enkelt modul, og kan finnes under seksjon 6.

8.2 Forbedringspotensiale

Som diskutert under målloppnåelse har alle moduler oppnådd en viss grad av funksjonalitet, men alle moduler kunne dratt nytte av en gjennomgang og forbedring av produsert kode. Vi mener at to til tre uker til med arbeidstid ville bedre kvaliteten på store deler av applikasjonen markant. Da prosjektet skal ferdigstilles til fredag 15. Mai er det ikke realistisk å få en utsettelse av arbeidstid på prosjektet. Vi må dermed heller se hvilke steg som kunne blitt tatt for å hindre at denne situasjonen oppstod.

Da applikasjonen ikke var ferdigstilt ved leveringsfrist, kan vi konkludere at scope for prosjektet var for stort for utviklingsgruppen. Denne situasjonen kunne blitt unngått hvis utviklerene hadde gode kunnskaper om hvor lang tid forskjellige moduler i applikasjonen ville ta å utvikle. Planning poker¹⁰ har blitt benyttet under tidsestimering, men har vist seg å være en underestimering av faktisk krevd arbeid. Dette problemet er ikke lett løselig uten å skaffe seg mer kunnskap innen et felt.

Det finnes også andre måter å løse dette problemet på. Vi kunne vært strengere på oppdeling og tidfesting av ferdigstilte moduler. Tidfesting av moduler hadde gitt gruppen et tidlig varsel om at utviklingen lå bak planlagt skjema. Vi ville dermed tidligere kunne tatt en vurdering om det ble behov for å redusere scope eller øke arbeidsmengden.

Et annet område som burde fått mer oppmerksomhet var testing. Vi ville med tidligere utvikling av automatiserte tester kunne fange opp problemer som oppstod i kode etter utvikling av enkelte modeller. Dette ville spart oss noen timer med senere bugfixing samt tid brukt til å manuelt kjøre igjennom forskjellige funksjoner for å bekrefte visuelt at modulene fungerte som de burde. Vi burde ved neste prosjekt legge mer vekt på automatiserte enhetstester, om disse er så enkle som å kun kjøre igjennom alle funksjonene, eller om de er mer sofistikerte er lavere i prioriteten. En funksjon som krasjer i automatisert testing er fortsatt nyttig informasjon for utvikling av en applikasjon.

¹⁰<https://www.planningpoker.com/>

8.3 Videre arbeid

Framtidig arbeid innen applikasjonen burde prioriteres for mer robuste tester og ferdigstilling av eksisterende moduler. Applikasjonen tilbyr allerede mye funksjonalitet, men mangler stabilitet. Det er derfor vår vurdering at det mest relevante framtidige arbeidet med denne applikasjonen burde gå innenfor forbedring av eksisterende kodebase, og ikke på utvikling av ny funksjonalitet.

9 Konklusjon

Vi konkluderer med at alle moduler har blitt lagt til med akseptabel funksjonalitet i utviklet applikasjon.

Vi erkjenner også at applikasjonen ville trukket nytte av tidligere opprettelse av et enkelt testrammeverk.

Avsluttende vil vi anbefale at framtidig arbeid på applikasjonen fokuser på utvidelse av testrammeverket og ferdigstilling av eksisterende moduler før utvikling av nye moduler utvikles.

10 Bibliografi

References

- [1] Patrick Pérez, Michel Gangnet, and Andrew Blake. Poisson image editing. *ACM Transactions on Graphics*, 22(3):313–318, 2003.
- [2] Benoit B Mandelbrot. How long is the coast of britain? statistical self-similarity and fractional dimension. *Science*, (156):636–638, 1967.
- [3] A. R. IMRE and J. BOGAERT. The minkowski-bouligand dimension and the interior-to-edge ratio of habitats. *Fractals*, 14(01):49–53, 2006.
- [4] P. E. Debevec and J. Malik. Recovering high dynamic range radiance maps from photographs. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, pages 369–378, 1997.

11 Apendix A: Bildekilder



Figure 34: <https://cdn.mos.cms.futurecdn.net/xCknw5UTKR8rM97eZE7reF-320-80.jpg>



Figure 35: https://scipy-lectures.org/_images/face.png



Figure 36: https://s3.amazonaws.com/mobisitdeletedimages/5575b10c52903-1433776396-CDG_Welcome_3.jpg



Figure 37: <https://www.pexels.com/photo/group-of-people-in-dress-suits-776615>



Figure 38: Fotograf: Isac Kallevig