

IMT3881 - Vitenskapelig Programmering  
HDR-avbildning: rekonstruksjon og rendring

Mats Eikeland Mollestad  
**Kandidatnummer:** 10006  
**Studentnummer:** 488086

Stian Andersen Negård  
**Kandidatnummer:** 10044  
**Studentnummer:** 484306

27. april 2019

## Innledning

Denne rapporten inneholder dokumentasjon og informasjon rundt vår arbeidsprosess i prosjektarbeid med rekonstruksjon og rendring av HDR-bilder. De fleste oppgaver er hentet fra emneansvarlig sin oppgavetekst[1]. I tillegg har det blitt implementert noe ekstra funksjonalitet. Arbeidsflyt, problemstillinger, resultater og kvalitetssikring blir gjennomgått underveis i rapporten. Mot slutten av rapporten nevner vi læringsutbytte, hva vi kunne gjort annerledes og en oppsummering av arbeidet som er gjennomført.

Alle bildene vi har benyttet for behandling og testing ligger i mappen '*eksempelbilder*'. Samtlige av disse fulgte med oppgaven, bortsett fra innholdet i mappen '*eksempelbilder/Balls Unaligned*'. Disse har vi selv lagd og er brukt for å teste opplinjeringsfunksjonaliteten.

## Git repository

Se vår progresjon og arbeidsflyt i git repositoriet på BitBucket:

<https://bitbucket.org/MatsEikelandMollestad/imt3881-2019-prosjekt/src/master/>

# Innhold

<b>1 Arbeidsflyt</b>	<b>1</b>
1.1 Samarbeid . . . . .	1
1.2 Overordnet kvalitetssikring . . . . .	1
<b>2 Funksjonalitetsliste</b>	<b>5</b>
<b>3 Resultater</b>	<b>6</b>
3.1 Rekonstruksjon . . . . .	6
3.2 Global HDR rendering . . . . .	13
3.3 Lokal HDR rendering . . . . .	17
3.4 GUI . . . . .	23
<b>4 Oppsummering</b>	<b>27</b>
4.1 Læringsutbytte . . . . .	27

# 1 Arbeidsflyt

I hele prosessen har det vært viktig for oss å forstå hva vi skal gjøre i stedet for å ta en barbarisk tilnærming til programvareutviklingen. Det å forstå teorien er ikke alltid like lett, men vi har alltid prøvd å gjøre et forsøk. I tillegg vil man naturligvis lære og skjønne mer ettersom man prøver seg frem.

## 1.1 Samarbeid

### 1.1.1 Innkommende hjelp

Ved usikkerhet eller behov for utveksling av kunnskap har vi tatt initiativ til faglig diskusjon med våre medelever. I all hovedsak har vi kun diskutert med Bjørn Christian Weinbach i forbindelse med HDR rekonstruksjon.

### 1.1.2 Utgående hjelp

Gjentatte ganger opplever vi at medelever oppsøker oss for faglige diskusjoner, spørsmål eller oppklaringer. Ved alle anledninger tar vi oss tid til å diskutere og sammenligne metoder. Samtidig har vi hatt fokus på å ikke gi løsningen direkte, men heller gi hint eller få studentene til å forstå problemstillingen bedre.

Dette føler vi har hjulpet oss til en dypere forståelse av teorien og praktisk bruk av fagstoffet. I tillegg har det vært motiverende for oss å oppleve at andre medstudenter har kommet til oss for veiledning og for å få svar på spørsmål.

### 1.1.3 Internt

I de tilfeller vi er usikre eller har oppdaget noe spennende, utveksler vi informasjonen internt i gruppen. Vår opplevelse er at man aldri vet hva slags kunnskap den andre har behov for, selv om man jobber med en helt annen oppgave. Dette motiverer også til videre arbeid med sin arbeidsoppgave som en effekt av en positiv, faglig diskusjon.

## 1.2 Overordnet kvalitetssikring

Fra starten av prosjektet har det vært viktig for oss å ha god struktur og følge gode vaner. Det bidrar til å skape et ryddig utviklingsmiljø hvor vi alltid har oversikt. Samtidig kan vi se hvor ting går feil underveis. Dette har vi gjort ved hjelp av kjente verktøy fra tidligere emner og fra egne erfaringer.

### 1.2.1 Pipeline

Vårt viktigste verktøy for kvalitetssikring av koden er bruk av pipeline. Det blir sjekket for kodestandard og dekningsgraden til testene. Begge disse kravene må være oppfylt for at pipelinens skal bli suksessfull. Allerede på første dag i prosjektet, 14. Mars, ble pipelinens satt opp. Vi har valgt å benytte oss av Bitbuckets

integrerte pipeline tjeneste.<sup>1</sup> Ved kjøring av pipeline vises kodestandarden på en skala fra 0-10 og dekningsgraden til testene vises i prosent. Ved siste commit ligger vår kodestandard på *9.95/10*.

#### **1.2.1.1 Konfigurasjon**

'*bitbucket-pipelines.yml*' inneholder basiskonfigurasjon som forteller om hvilke script som skal kjøres i pipelinen.

Filen '*pipelinelint.bash*' inneholder hvilke kommandoer som skal kjøres kvalitetssikring. Her blir også terskelen for kravene satt.

Kodestandarden ligger i '*lint-config.rc*'. Utgangspunktet til filen er hentet fra skaperne av Pylint og ligger i repoet til pylint<sup>2</sup>. Her er det satt anbefalte grenser som følger good practises for programmering i Python. I løpet av utviklingsperioden har vi modifisert enkelte parametre, eksempelvis veldig strenge advarsler som fanges opp.

#### **1.2.2 requirements.txt**

Filen er benyttet for å vise hvilke pakker og avhengigheter som kreves for å kjøre programvaren vi har utviklet. I tillegg er filen essensiell for at pipelinen skal vite hva som kreves for å kjøre programvaren.

#### **1.2.3 Docstrings**

Emneansvarlig har både informert om og sterkt anbefalt denne dokumenteringsmetoden. Det er en god vane vi har vektlagt i utviklingsprosessen. Det er også lagt inn krav om docstrings i pipelinen.

#### **1.2.4 Testing**

Etter hvert som vi har utviklet ny funksjonalitet har det vært en prioritet å skrive tester. Etter siste commit endte vi med en dekningsgrad på *85%*. Dette vises i Coverage rapporten i figur 1.

#### **1.2.5 git branches**

Fra første stund i utviklingen har vi benyttet oss av ulike branches. Vi har forsøkt å følge navngivningsstandarder fra GitFlow<sup>3</sup> med bruk av '*feature/<funksjonalitet>*'. Dette har gjort det enkelt å separere kode og begrense omfanget til feil i koden. Arbeidet med branches har gått knirkefritt og vi har unngått merge conflicts. Oversikten over våre branches vises i figur 2.

---

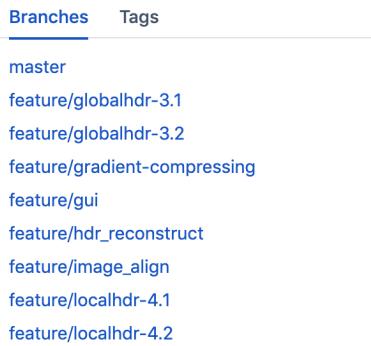
<sup>1</sup><https://bitbucket.org/product/features/pipelines>

<sup>2</sup><https://github.com/PyCQA/pylint/blob/master/pylintrc>

<sup>3</sup><https://nvie.com/posts/a-successful-git-branching-model/>

Name	Stmts	Miss	Cover
align_image.py	58	7	88%
align_image_test.py	40	0	100%
app_test.py	54	51	6%
filter_config.py	22	0	100%
globalHDR.py	48	25	48%
globalHDR_test.py	28	0	100%
gradient_compression.py	73	2	97%
gradient_compression_test.py	50	0	100%
hdr.py	58	3	95%
hdr_test.py	102	0	100%
image_set.py	65	13	80%
localHDR.py	52	12	77%
localHDR_test.py	101	0	100%
TOTAL	751	113	85%

Figur 1: Coverage report



Figur 2: Oversikt over git branches

### 1.2.6 Hjelpefunksjoner

Underveis i utviklingen har vi valgt å lage noen midlertidige hjelpefunksjoner for analyse og beskrive arbeidsmaterialet vårt. Vi har valgt å behandle de midlertidige hjelpefunksjonene som fullverdige funksjoner med blant annet docstrings. Ettersom de ikke gir noen nytteverdi til sluttproduktet har disse blitt fjernet siste commit. Et eksempel på en hjelpefunksjon er vist i kodelisting 1.

Listing 1: Sammenligning av to bilder

---

```
def compare(im1, im2):
    """
    Help function that takes two input images and prints
    the difference between their pixel values.

    :param im1: Input image 1.
    :type im1: Numpy array.

    :param im2: Input image 2.
    :type im2: Numpy array.
    """

    print(im2-im1)
```

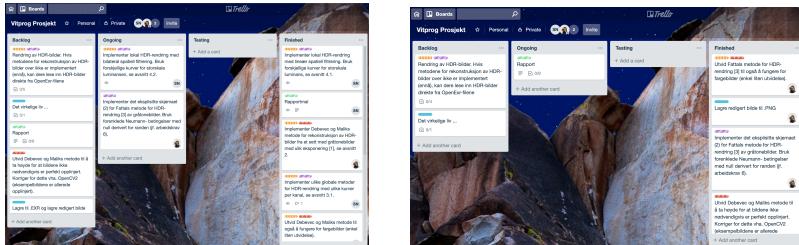
---

### 1.2.7 git submodule

Det ble opprettet en git submodule<sup>4</sup> for rapporten. Poenget er at rapporten skal leve et eget liv. Dessverre har submodulen blitt lite brukt.

### 1.2.8 Trello

For prosjektorganisering og ansvarsfordeling har vi valgt å bruke Trello. Dette har skapt god oversikt over hvilke oppgaver som arbeides med og hvilken status de har. I figur 3 er en oversikt over utviklen i Trello brettet vårt over tid. Merk at scrollle-baren til høyre under ferdigstilte oppgaver er mindre, noe som tilsvarer en større mengde kort. Det er også enkelte kort som har blitt splittet opp over tid.



(a) Status i prosjektets tredje uke

(b) Status i prosjektets siste uke

Figur 3: Utvikling i Trello over tid

<sup>4</sup><https://git-scm.com/docs/git-submodule>

### 1.2.9 PyCharm

Alt arbeid med prosjektet har foregått i PyCharm<sup>5</sup>. Vi anser verktøyene i denne IDE-en som svært nyttig, spesielt Scientific Mode<sup>6</sup>. Denne modusen simplifiserer visualisering av bilder og viser et samlebånd av plotter og bilder etter hvert som man kjører kjører koden.

### 1.2.10 Kodestandard

Kodestandarden vår kommer fra retningslinjene til Pylint. I tillegg har vi snakket om og fulgt interne retningslinjer når det gjelder navngivning<sup>7</sup>. Eksempelvis skal alle variabler være skrevet i '*snake case*' og klassenavn skal være skrevet i '*pascal case*'. Det har vært et fokus fra første stund å lage generelle funksjoner og moduler som skal kunne brukes flere steder i koden.

## 2 Funksjonalitetsliste

Nedenfor er en komplett funksjonsliste som forteller om hvilken funksjonalitet som er implementert fra emneansvarliges oppgavetekst.

- Rekonstruksjon

- Debevec og Maliks metode for HDR-rekonstruksjon av gråtonebilder
- Utvidelse av Debevec og Maliks metode for fargebilder
- Vektorisering av Debevec og Maliks metode
- Opplinjering av bilder

- Rendring

- Global HDR-rendring med en global funksjon
- Global HDR-rendring på luminanskanalen
- Lineær spatiell filtrering med Gaussisk blur
- Ikke-lineær spatiell filtrering ved hjelp av Bilateral filtrering
- Komprimering i gradientdomenet

- GUI

- Grafisk brukergrensesnitt med alle implementerte funksjoner
- Lagring av rekonstruert bilde til .PNG-format\*
- Lagring av innlastet .EXR-bilde til .PNG-format\*
- Lagring av innlastet .EXR-bilde til .EXR-format\*

\* Denne funksjonaliteten er implementert utover oppgaveteksten.

---

<sup>5</sup><https://www.jetbrains.com/pycharm/>

<sup>6</sup><https://www.jetbrains.com/help/pycharm/matplotlib-support.html>

<sup>7</sup><https://medium.com/@pddivine/string-case-styles-camel-snake-and-kebab-case-981407998841>

## 3 Resultater

### 3.1 Rekonstruksjon

#### 3.1.1 Introduksjon

I denne deloppgaven skulle vi rekonstruere et HDR-bilde basert på et bildesett som inneholdt samme motiv med forskjellige blenderåpninger.

#### 3.1.2 Hjelpeklasse

For å gjøre det enklere å håndtere de forskjellige bildene, har vi introdusert en egen klasse som heter `ImageSet`. Denne inneholder alle de forskjellige bildene, blenderåpningene og den originale bildedimensjonen. I denne klassen finnes det forskjellige funksjoner som gjør det enklere å bruke de implementerte funksjonene. Det finnes også funksjoner som `gray_images()` og `channels()` for å ytterligere forenkle koden.

#### 3.1.3 Debevec og Maliks metode for HDR-rekonstruksjon av gråtonebilder

##### 3.1.3.1 Problemstilling

I denne deloppgaven skulle vi rekonstruere et gråtone bilde fra et sett med bilder.

##### 3.1.3.2 Framgangsmåte

Får å oppnå målet vårt måtte vi benytte metoden presentert i Debevec og Maliks artikkkel[2]. Her bruker de grunnprinsippet med at pixelverdien  $Z_{ij} = f(E_i \Delta t)$ . Målet her er å finne  $E_i$  og Debvec og Maliks har derfor forandret på utrykket for oppnå en minstekvadraters problemstilling, som vist i (1).

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

Her er det også introdusert en vektionsfunksjon som vil vektlegge verdier nær midten. Dette er viktig ettersom det finnes fysiske grenser som bildesensorer. Disse klarer ikke å representer hele lysspekteret, og vi har derfor ikke en god representasjon av en verdi som ligger nær pixelverdi  $Z_{min}$  eller  $Z_{max}$ .

Dessverre har vi ikke fått et nytt bilde, men heller en oppslagstabell med alle  $E_i$  verdiene, basert på  $Z_i$ . Herfra må vi regne oss fram til den *reelle* pixelverdien på plassen  $i$ . Debevec og Malik har derfor gitt en funksjon for dette (2).

$$\ln E_i = \frac{\sum_{i=1}^P w(Z_{ij})(g(Z_{ij}) - \ln \Delta t_j)}{\sum_{j=1}^P w(Z_{ij})} \quad (2)$$

### 3.1.3.3 Kodekvalitet

For å forsikre oss at koden fungerer som vi ønsker har vi laget fire tester.

- *test\_weighting\_function*: Tester at vektefunksjonen for generering av HDR-oppslagstabellen fungerer som forventet.
- *test\_weighting\_function\_vector*: Tester at en vektorisert versjon av vektefunksjonen fungerer som forventet.
- *test\_hdr\_reconstruction*: Tester at HDR-oppslagstabellen og *ImageSet.hdr\_image(...)* er innenfor en godkjent margin.
- *test\_reconstruct\_image\_from\_graph*: Tester at det rekonstuerte bildet er innen for en godkjent margin basert på en kjent HDR-oppslagstabell.

### 3.1.3.4 Sluttresultat

Koden for å finne løsningen til (1) finnes ved *hdr.hdr\_channel(image, shutter, smoothness, weighting)*, eller en forenklet versjon *ImageSet.hdr\_curve(smoothness)*.

For å renge ut resultatet til (2) må man kjøre *hdr.reconstruct\_image(channels, weighting, hdr\_graph, shutter)*.

Vi har valgt å unngå en egen rekonstruksjonsfunksjon på *ImageSet*. I stedet benyttes en funksjon som regner ut oppslagstabellen og det nye bildet. For å gjøre dette må man kalle *ImageSet.hdr\_image(smoothness)*.

Resultater fra rekonstrueringen ses i figur 4.

## 3.1.4 Utvidelse avDebevec og Malik s metode for fargebilder

### 3.1.4.1 Problemstilling

I 3.1.3.1 viste vi hvordan vi fikk rekonstruert et bilde for en gråkanal. Allikevel er det ofte mer relevant å benytte fargebilder.

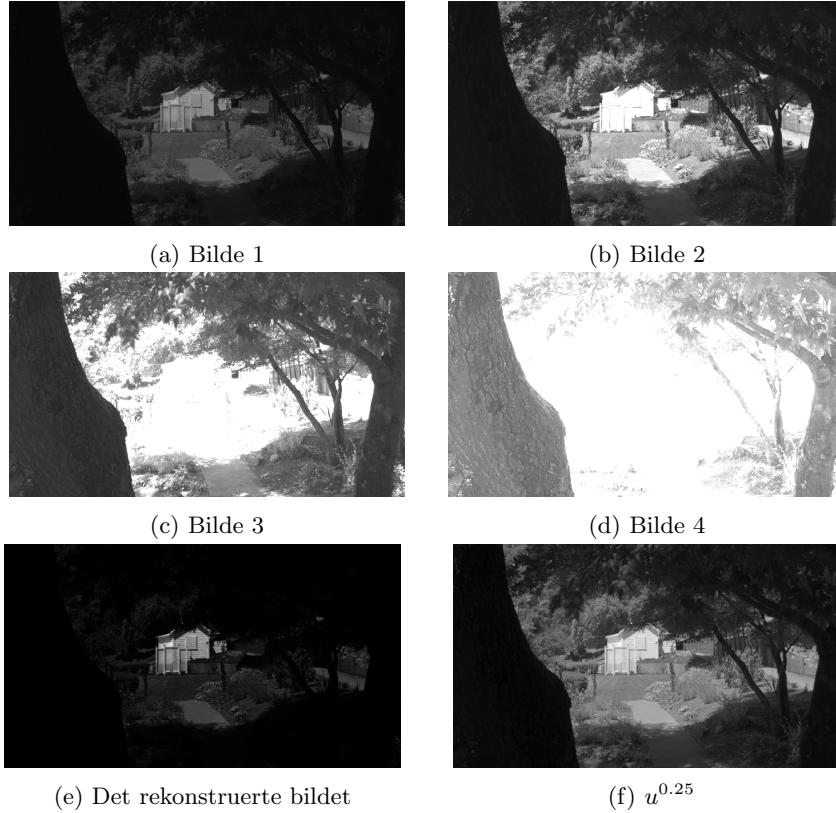
### 3.1.4.2 Framgangsmåte

I artikkelen til Debevec og Malik nevnes også hvordan man behandler fargebilder. Dette gjøres ved å behandle fargekanalene hver for seg selv, for så å sy det sammen til et nytt fargebilde. For å få til dette, måtte koden restruktureres litt og legge til en funksjon *ImageSet.channels()* som separerer de forskjellige kanalene. Etter dette, gjør vi det samme som i 3.1.3.1 men for hver kanal.

### 3.1.4.3 Kodekvalitet

For å forsikre oss at det fungerer for fargebilder også, har vi laget to tester som er veldig like de i 3.1.3.1.

- *test\_hdr\_reconstruction\_color*: Tester at det rekonstuerte bildet er innen en godkjent margin.
- *test\_reconstruct\_image\_from\_graph\_color*: Tester at det rekonstuerte bildet er innen for en godkjent margin, basert på en kjent HDR-oppslagstabell.



Figur 4: HDR-rekonstruksjon av monokromebilder

#### 3.1.4.4 Sluttresultat

For å lage HDR-bildet må du kjøre samme funksjon som i 3.1.3.4, altså *ImageSet.hdr\_image(smoothness)*, men denne vil se på dimensjonene til bilde og derfra vite om det er monokromt eller fargebilde.

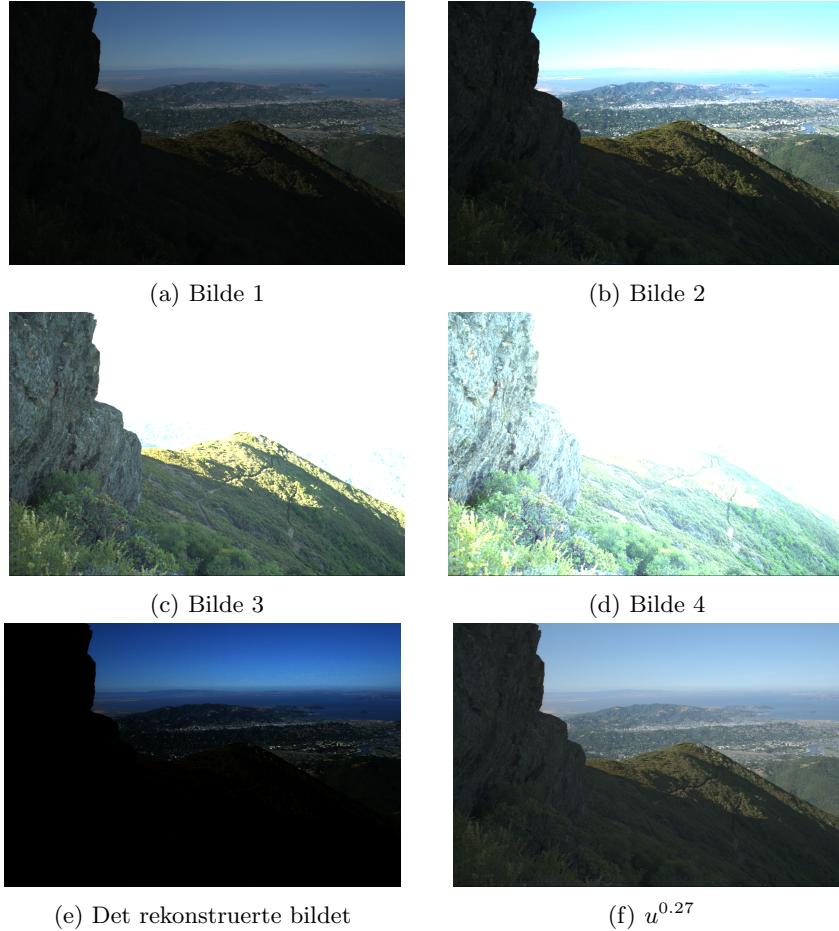
Etter å ha kjørt en test med et bildesett på seks 1255 x 876 fargebilder, brukte koden vår 4.486s på å regne ut HDR-grafen og rekonstruere bilde. Vi prøvde også med et bildesett på seks 11255 x 876 fargebilder og det brukte tilnærmet 20s. Dette skal i teorien ikke øke tiden på å regne ut HDR-grafen, noe vi kommer til å snakke om i 3.1.5, men det vil øke arbeidet med rekonstrueringen. Det kan forklare en tilnærmet femdobling i tidsbruk.

Resultater fra rekonstrueringen ses i figur 5.

#### 3.1.5 Vektorisering av Debevec og Maliks metode

##### 3.1.5.1 Problemstilling

Ettersom 3.1.3.1 og 3.1.4 baserer seg på minste kvadraters metode, kan det fort bli krevende å finne en løsning når man bruker mange datapunkter som et sett



Figur 5: HDR-rekonstruksjon av fargebilde

høyoppløsningsbilder.

### 3.1.5.2 Framgangsmåte

For å løse dette problemet fant vi hovedsaklig to løsninger. Den første er å vektorisere koden for å få den til å kjøre mer effektivt. Den andre er å minske størrelsen på matrisen, og dermed minske arbeidet som må utføres.

- Vektorisering

Optimalt sett ønsker vi å vektorisere så godt at vi kan bruke alle pixlene i alle bildene. Dette fordi vi vil beholde presisjonen på resultatet, noe som vi mener er viktig.

- Datautsnitt

Dessverre er det så mye data å analysere at vi ikke får gjort alt med vektorisering. Derfor må vi kombinere de to alternativene våre. Løsningen blir å minske datamengden så går fort nok, men fortsatt beholde nok data til å oppnå et godt resultat.

For å klare dette har vi valgt ut opptil 1000 pixler fra samme utsnitt i hvert av bildene. Dette senker datamengden som må proseseres i 3.1.3.1, men inneholder fortsatt nok informasjon til at det blir et presist resultat.

### 3.1.5.3 Kvalitetssikring

Her har vi laget en test, *test\_find\_reference\_points\_for*, som tester at antall punkter brukt i to forskjellige bilder er like mange som vi forventer.

### 3.1.5.4 Sluttresultat

På grunn av vektoriseringen endte vi opp med å endre koden som regner ut formel (2). Dette endte opp med at vi endret fra kodelisting 2 til 3.

Vi prøvde også å vektorisere koden som regner ut formel (1), men vi klarte ikke å finne et system som gjorde dette på riktig måte. Den har derfor forblitt uvektorisert.

Når det kommer til velge et mindre datautsnitt har vi lagd en funksjon som returnerer pixelindeksene som skal brukes under formel (1). Dette skjer i *hdr.find\_reference\_points\_for* med koden i kodelisting 4.

Listing 2: Før Vektorisering

---

```
for x in range(0, shape[-2]):  
    for y in range(0, shape[-1]):  
        weighted_sum = 0  
        g_value_sum = 0  
        for j in range(0, shape[0]):  
            weighted_sum += weighting(channels[j][x][y] + 1)  
            z_value = int(channels[j][x][y])  
            g_value_sum += weighted_sum * \  
                (hdr_graph[z_value] - shutter[j])  
        hdr_image[x][y] = g_value_sum / weighted_sum
```

---

Listing 3: Etter Vektorisering

---

```
w_value = weighting(channels.copy() + 1)  
denum_w = w_value.sum(0)  
denum_w[denum_w == 0] = 1  
return np.exp((w_value * (hdr_graph[channels.astype(int)] \  
- shutter[:, None, None])).sum(0) / denum_w)
```

---

Listing 4: Datautsnitt

---

```
channels = images.channels()
shape = np.shape(channels)
n_pixels = shape[-2] * shape[-1]
spacing = max(int(n_pixels / 1000), 1)
return np.arange(0, n_pixels, spacing)
```

---

### 3.1.6 Opplinjering av bilder

#### 3.1.6.1 Problemstilling

Ettersom HDR algoritmen bruker flere bilder som er tatt med forskjellig blenderåpning, og dermed også på forskjellig tidspunkt, kan man risikere at bildene ikke er perfekt opplinjert. For at HDR algoritmen skal fungere, krever den at pixlene er opplinjert riktig.

#### 3.1.6.2 Fremgangsmåte

For å klare å opplinjere de forskjellige bildene har vi basert oss på artikkelen<sup>8</sup>, som igjen baserer seg på konseptet om homografi. På grunn av dette finner vi kjennetegn i hvert bilde, for så å se om det er mulig å opprette en matrise som kan transformere bildene slik at disse kjennetegnene er opplinjert likt.

I tillegg kan vi ikke opplinjere bildene tilfeldig med hverandre, men vi må ha et bilde som fungerer som *sannheten*, og dermed opplinjere bildene til dette.

Deretter kan det bli problematisk å sammenligne *sannhetsbildet* med alle de andre, ettersom de har forskjellig blanderåpninger og dermed forskjellige kjennetegn. For å løse dette så velger vi å sammenligne de bildene som er nærmest i blanderåpning, men da med et allerede opplinjert bilde for at det skal ha pixlene på samme posisjon som *sannhetsbildet*.

Til slutt er det ikke sikkert alle pixlene inneholder en verdi etter opplinjeringen. Siden vi trenger pixelverdier på alle bildene når vi regner ut formel (2), har vi valgt å nedskalere bildet slik at det finnes en pixelverdi for alle pixlene.

#### 3.1.6.3 Kvalitetssikring

For å kvalitetssikre koden har vi skrevet tre tester.

- *test\_image\_cropping*: Tester om et bilde blir nedskalert til riktig størrelse.
- *test\_alignment\_function*: Tester om den opplinjerer to bilder riktig.
- *test\_alignment*: Tester om *ImageSet.aligned\_image\_set()* inneholder riktig metadata.

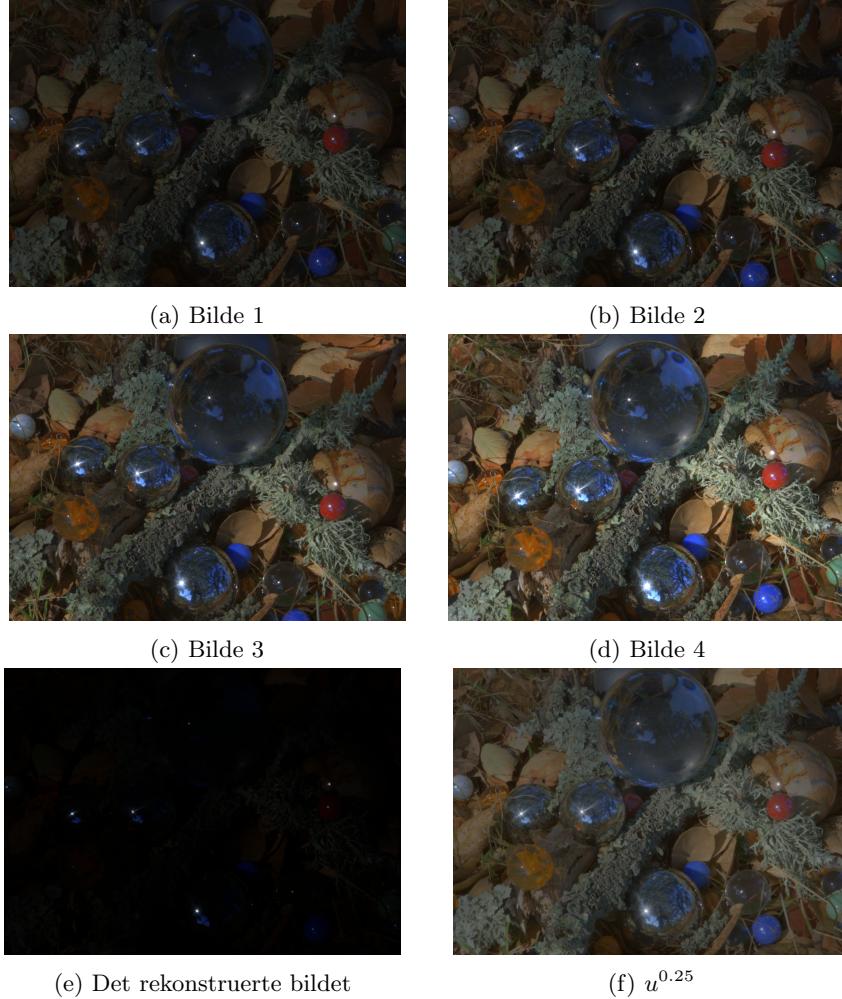
---

<sup>8</sup><https://www.learnopencv.com/image-alignment-feature-based-using-opencv-c-python>

### 3.1.6.4 Sluttresultat

For å opplinjere et sett med bilder kan man kalle `align_image.align_images(images)` eller `ImageSet_aligned_image_set()`. Det er også mulig å kalle `align_image.align_image_pair(first_im, second_im)` for å opplinjere to bilder mot hverandre.

Resultatene kan ses i figur 6.



Figur 6: HDR-rekonstruksjon av ulinjerte bilder

### 3.1.7 Forbedringspotensiale

I seksjon 3.1.2 har vi en hjelpeklasse som gjør behandling av bildesett lettere. I denne klassen gjør vi antagelsen at de innleste bildene har formatet `/sti/til/bilde_blendertid`, hvor `blendertid` er et tall. Dette vil ikke gjelde for alle bilder som

finnes, men det gjelder for de vi har i vårt prosjekt. For å gjøre det mer generelt kunne det vært en god idé å stole på metadataen til bildet som burde inneholde denne informasjonen.

I seksjon 3.1.5 brukte vi et lite utvalg av hovedbildet for å øke hastigheten. Den nåværende implementasjonen velger maksimalt 1000 pixler per bilde. For å øke hastigheten burde denne størrelsen enten senkes eller basere seg på hvor mange bilder som finnes. Blir dette utsnittet for lavt kan man risikere et upresist resultat.

I tillegg finnes det nok flere måter å vektorisere koden på som gjør at koden gårtt enda raskere.

Når det kommer til seksjon 3.1.6 har den en svakhet hvis bildet har for lite detaljer. Vi prøvde å løse dette med å bruke standardavviket til bildet og deretter bestemme om det inneholdt nok detaljer til å bli opplinjert. Det ville vært mulig å slette enkeltbilder så lenge andre bilder har høy nok verdi men det prøvde vi ikke.

I tillegg er det en svakhet i koden som nedskalerer bildet. Denne koden antar at bilde har en alpha-verdi på 0 der det ikke finnes pixelverdier, men det ser ut til at dette ikke stemmer for alle bilder. Det vil derfor være noe som kunne forbedres. Eksempelet på dette er i 6, hvor 6f skal være av en lavere dimmensjon.

## 3.2 Global HDR rendring

### 3.2.1 Introduksjon

Deloppgaven går ut på å utføre HDR-rendring med ulike funksjoner. Felles for all global rendring er at de skal kunne utføres globalt eller på luminanskanalen alene, samt i det logaritmiske domenet og i lineære verdier.

La  $[R, G, B]$  være kanalene i et bilde. For enkelhetens skyld er det gjort noen forenklinger i definisjonen av luminans og kromatisitet. Luminans er definert som  $L = R + G + B$ ; kromasitet er definert som  $[\frac{R}{L}, \frac{G}{L}, \frac{B}{L}]$ .

### 3.2.2 Hjelpeklasse

For å forenkle funksjonskall, øke gjenbrukbarheten og lesbarheten til koden, har vi introdusert klasser som inneholder editeringkonfigurasjon. Disse ligger i filen '*kode/filter\_config.py*'. Det finnes klasser med innstillingar for uklarhet, effekter og tilhørende skalering. Gjennom hele koden vil disse klassene bli brukt i stedet for å sende med enkeltparameter. Klassene som benyttes i denne delen er *BlurImageConfig* og *EffectConfig*.

### 3.2.3 En global funksjon

#### 3.2.3.1 Problemstilling

Første del av den globale rendringen går ut på å rendre et bilde med en global funksjon. Dette kan gjøres på de lineære verdiene, typisk fra et innlest .exr-bilde; det kan også gjøres i det logaritmiske domenet i et konstruert HDR-bilde fra en serie med .png-bilder beskrevet i 3.1.3.4.

### 3.2.3.2 Fremgangsmåte

Ved implementasjon er det viktig å bruke funksjoner som har en kurve. Hensikten er å fremheve mørke eller lyse områder i bildet. Grunnen til at dette fungerer bra, er at slike funksjoner behandler verdiene ulikt.

Dersom man implementerer lineære funksjoner vil bildet bare parallelldorskysves. Dette er ikke en ønsket effekt ettersom forholdet mellom pikslene vil være den samme. I tillegg ønsker MatPlotLibs pyplot verdier mellom 0 og 1. For å tilfredsstille dette kravet og beholde forholdet mellom pixlene etter rendering har vi skalert verdiene på følgende måte. I ligningen(3) er  $I =$  bildet som vises og  $P =$  pixelverdiene på posisjonen  $i, j$ .

$$I_{i,j} = \frac{P_{i,j} - P_{min}}{P_{max} - P_{min}} \quad (3)$$

Vi har valgt å implementere en håndfull ulike funksjoner som har en effektiv kurve. Enkelte funksjoner er kraftigere enn andre og noen funksjoner har parameter som kan justeres for å endre effekten.

#### Liste over implementerte funksjoner

- Eksponentialfunksjonen;  $e^u$
- Logaritmen;  $\ln u$
- Gamma;  $u^\gamma, \gamma \in [0, 1]$
- Eksponentfunksjonen;  $u^x, x \in [0, 4]$
- Kvadratrot;  $\sqrt{u}$

### 3.2.3.3 Kvalitetssikring

På grunn av at denne funksjonaliteten er veldig enkel, har vi kun valgt å implementere en test. `test_edit_globally` tester at global rendering med kvadratrot-funksjonen gir forventede verdier.

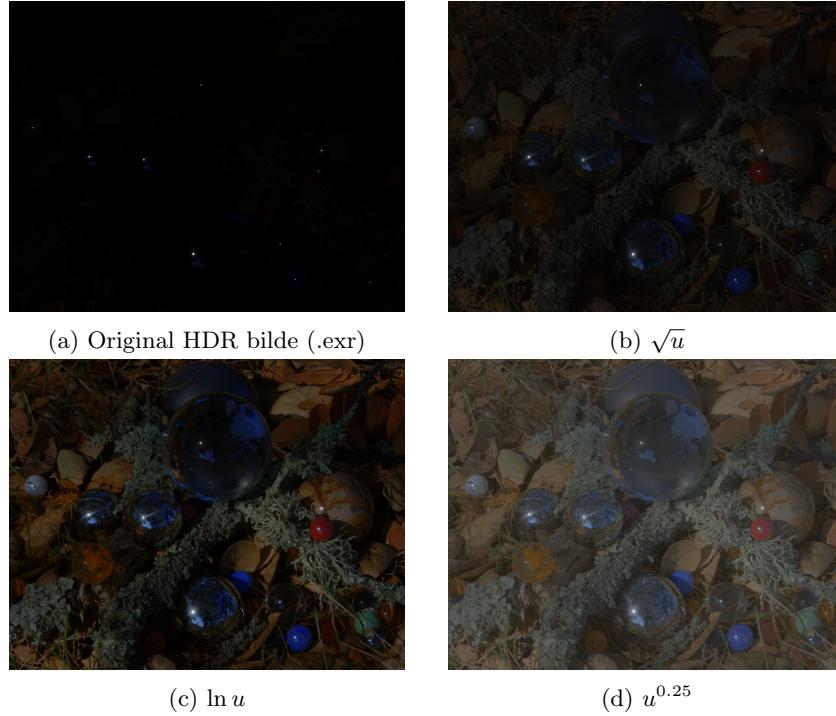
### 3.2.3.4 Sluttresultat

Funksjonen kalles ved `globalHDR.edit_globally(image, effect)`. Merk at effect er en instans av klassen `EffectConfig`, nevnt i 3.2.2

I figur 7 vises bilder rendret globalt med ulike funksjoner nevnt i 3.2.3.2.

### 3.2.4 Forbedringspotensiale

Ved å skalere om bildet som vist i formel (3) før fremvisning antar vi at bildets laveste verdi er 0 og den høyeste verdien er 1. Det er en påstand som ikke er sann i alle bilder. Dermed kan enkelte farger representeres forskjellig fra sitt egentlige utseendet.



Figur 7: Tre globale renderinger

### 3.2.5 Luminans og kromatisitet hver for seg

#### 3.2.5.1 Problemstilling

For å forhindre at fargene blir vasket ut, som er et problem nevnt i oppgave-teksten, ved å bruke en felles funksjon for alle pixlene, kan man dele bildet opp i luminans og kromasititet. Ved å rendre luminanskanalen alene før man legger den tilbake, vil man kunne oppnå et bedre resultat. Allikevel kan det resulterende bildet ende med høy fargemetning.

#### 3.2.5.2 Fremgangsmåte

For å oppnå ønsket resultat må man dele opp bildet i luminans og kromatisitet, rendre luminanskanalen med en funksjon og multiplisere den nye luminansen med kromatisiteten. Når luminans-kanalen skiller ut fra bildet tar programvaren vår hensyn til om bildet er monokromt eller har fargekanaler. Rendringen av luminanskanalen vil gjøres på samme måte som i forrige deloppgave med en global funksjon, nevnt i 3.2.3

Forholdet mellom luminans og kromatisitet kan justeres for å minske den høye fargemetningen. Dette kan sluttbrukeren gjøre i GUI-en.

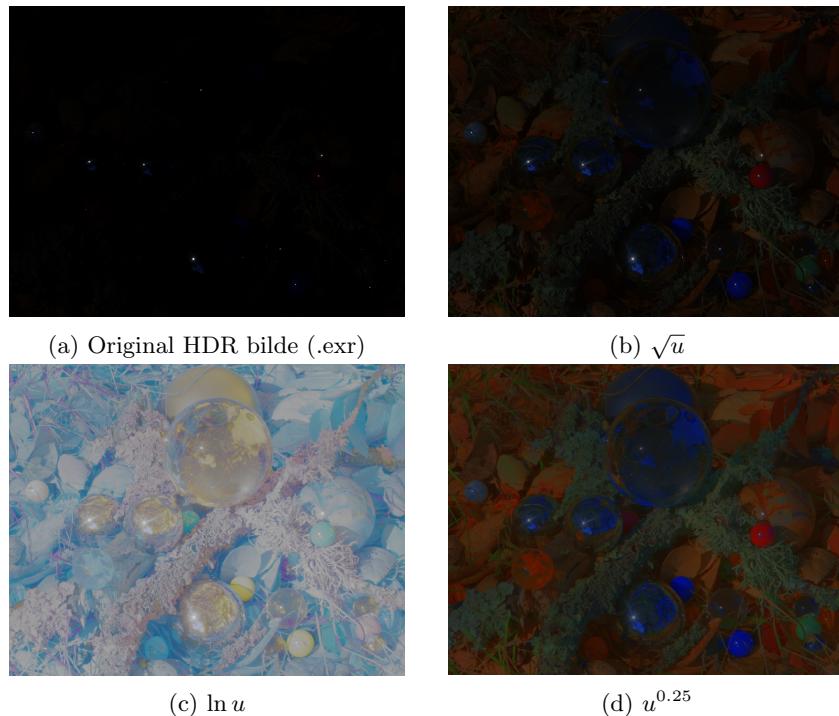
### 3.2.5.3 Kvalitetssikring

Ved en enkel funksjon som dette har vi kun valgt å implementere en test. `test_edit_luminance` tester at global rendring av luminanskanalen med vektning mellom luminans og kromatisitet gir forventede verdier.

### 3.2.5.4 Sluttresultat

Funksjonen kalles ved `globalHDR.split_image(image, effect)`. Merk at effect er en instans av klassen `EffectConfig`, nevnt i 3.2.2.

I figur 8 vises bilder rendret i luminanskanalen med ulike funksjoner fra 3.2.3.2.



Figur 8: Tre globale renderinger på luminanskanalen

Ved en nærmere sammenligning mellom global rendring på alle kanaler og global rendring på luminanskanalen alene ser man en tydelig forskjell. Som forventet har rendringen på luminanskanalen klart høyere fargemetning. Dette illustreres i figur 9.



(a) Global rendering på alle kanaler



(b) Global rendering på luminanskanalen

Figur 9: Sammenligning for global rendring med  $\sqrt{u}$

### 3.3 Lokal HDR rendring

#### 3.3.1 Introduksjon

Deloppgaven går ut på å utføre HDR-rendring med ulike funksjoner. I dette tilfellet skal vi ta hensyn til hvor de ulike pikslene er og hvordan de er i forhold til hverandre. Grunnen til at vi gjør dette, er for å forbedre bildet både i mørke og lyse partier samtidig. Også her skal rendringen kunne utføres globalt eller på luminanskanalen alene, samt i det logaritmiske domenet og i lineære verdier.

#### 3.3.2 Hjelpeklasse

På samme måte som i 3.2.2 benytter vi oss av en klasse som inneholder editeringskonfigurasjon. Klassene som benyttes i denne delen er *FilterImageConfig*, *BlurImageConfig*, *EffectConfig* og *GradientFilterConfig*.

#### 3.3.3 Lineær spatiell filtrering

##### 3.3.3.1 Problemstilling

Ved å skille på små og store lokale endringer i bildet, kan man hente ut detaljene. Dette kan gjøres ved å øke uklarheten i bildet, for så å ta differansen mellom originalbildet og det uskarpe bildet. Før detaljene legges tilbake utføres en rendring på det uskarpe bildet. Rendringen kan utføres på samme måte som i 3.2.3. Metoden vil gi en bedre behandling av bildets visuelle egenskaper.

##### 3.3.3.2 Fremgangsmåte

En metode som behandler pixlene på nevnt måte er et Gaussisk filter. Se formel 4. Teorien[3] er at den tar et område fra bildet og multipliserer med et vektet gjennomsnitt. Utfallsverdien settes for det aktuelle området. Sigmaen i ligningen står for standardavviket til den Gaussiske kjernen[4] som avgjør størrelsen på

filteret. Størrelsen på filteret stiger med verdien til sigma.

$$GB[I]_p = \sum_{q \in S} G_\sigma(||p - q||)I_q \quad (4)$$

### 3.3.3.3 Kvalitetssikring

For å sikre at koden fungerer som ønsket har vi laget noen tester.

- *test\_has\_alpha*: Tester om den finner en eventuell fjerde (alpha) kanal.
- *test\_extract\_alpha*: Tester om man suksessfullt fjerner en eventuell alpha kanal.
- *test\_blur\_image\_linear*: Tester om lineær uskarping (blur) gir forventede verdier.
- *test\_find\_details*: Tester om man finner detaljene til et gitt bilde.
- *test\_edit\_blurred\_image*: Tester om rendring av et uskarpt bilde returnerer forventede verdier.
- *test\_reconstruct\_image*: Tester om rekonstruksjon av et bilde fungerer som forventet.
- *test\_append\_channel*: Tester om en fjerde kanal blir skjøtet på et numpy array på riktig måte.
- *test\_filter\_linear*: Tester den lineære filtreringen med ulike parametere. Denne funksjonen er den ytterste i koden som foretar kall til andre mindre funksjoner.

### 3.3.3.4 Sluttresultat

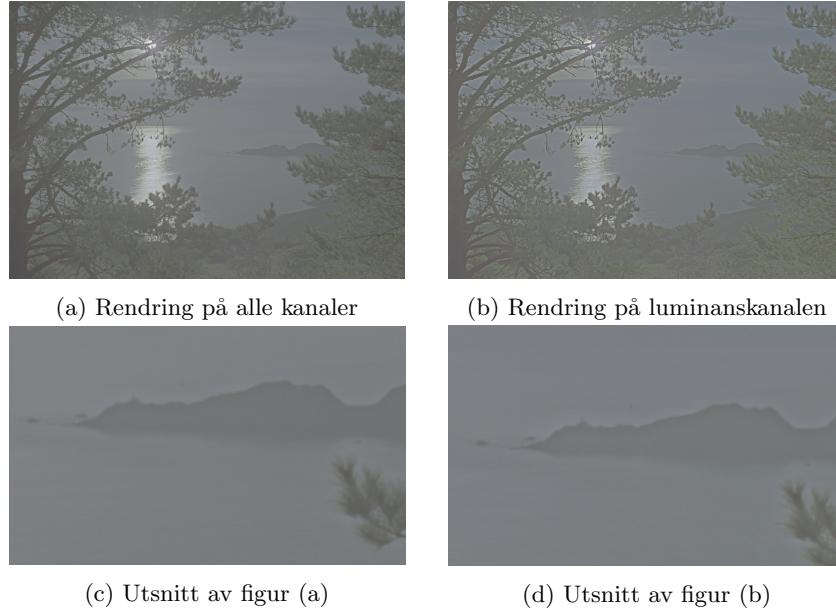
Funksjonen kalles ved `localHDR.filter_image(image, filters)`. Merk at *filters* er en instans av klassen *FilterImageConfig*, på samme måte som i 3.2.2.

I figur 10 vises bilder hvor den uskarpe delen av bildet er rendret globalt med ulike funksjoner nevnt i 3.2.3.2. Bildene i figuren har først et gammafilter med verdi 0.1 og så et gaussisk filter med sigma verdi 5. I det gaussiske filteret er det lagt et kvadratrotsfilter; i (a) på alle kanaler og i (b) på luminanskanalen. Ved å inspirere utsnittene i (c) og (d) ser man forskjellen på hvor mye haloer man ser.

## 3.3.4 Ikke-lineær spatiell filtrering

### 3.3.4.1 Problemstilling

Til tross for at metoden i 3.3.3 fungerte bedre enn de globale funksjonene, kan det oppstå noen problemer. Spesielt «halo»-effekter ved store lokale kontrastforskjeller. Et kantbevarende filter kan implementeres for å minske disse uønskede effektene.



Figur 10: Gaussisk filter med ulik render konfigurasjon

### 3.3.4.2 Fremgangsmåte

Ved utvikling av 3.3.4 forsøkte vi ulike metoder. Vår første idé gikk ut på å prøve et Sobel filter[5], men dette skapte utrolig store haloer i det resulterende bildet. Videre forsøkte vi å optimalisere Sobel filteret ved å hente og bruke kjernen i filteret, uten den ujevne børstekanten som karakteriserer filteret. Dette resulterte i mye arbeid uten store fremskritt. Dermed gikk jakten videre.

Etter mer undersøkelse satt vi igjen med tre alternativer. Anisotropisk diffusjon[6], Canny edge detection[7] og Bilateral filtering[3]. Alle lovte å leve et bedre resultat enn Sobel. Teknikken som virkelig trigget interessen vår og vi virkelig fikk sansen for, var Canny edge detection. Fremgangsmåten skulle løse problemet på en meget hendig måte. Spesielt delen med dobbel terskling av bildet ble vi fascinert av. Ettersom vi trodde disse filtrene måtte lages fra bunnen av, begynte vi med å implementere Cannys metode. For å implementere Canny edge detection fra bunnen av trengs følgende steg.

1. Konvertere til et monokromt bilde
2. Utføre av Gaussisk blur
3. Finne intensitetsgradientene i X- og Y-retning
4. Finne maksimumspunktet i en kant (Non maximum suppression)[8]
5. Dobbelt terskling for sterke og svake kanter

6. Sjekke hvilke svake kanter som er tilkoblet sterke kanter

7. Opprydding av unødig støy

Dette gikk fint en stund, inntil vi skulle implementere *non maximum suppression*. Her satt vi fast en stund før det ble informert at vi kunne bruke et ferdiglagd filter. Dette gjorde jobben ekstremt mye lettere, ettersom vi kun trengte å kjøre et ferdiglaget kall fra et eksisterende bibliotek. Prosesen endte med at vi benyttet oss av et bilateralt filter fra OpenCV<sup>9</sup>.

Bilateral filtrering er en metode som vil forminske problemet med haloer. I utgangspunktet ligner denne teknikken på Gaussisk blur som ble benyttet i 3.3.3.2. Forskjellen er at det enkle sigma-parameteret endres til to parametere; et for bilderommet og et for fargerommet. I tillegg kommer det et justeringsledd. Dette vises i ligning 5. Et bilateralt filter vil filtrere ved hjelp av et vektet gjennomsnitt av nabopixlene. Dette resulterer i et utglattet bilde med beholdte kanter.

$$BF[I]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(|p - q|)G_{\sigma_r}(|I_p - I_q|)I_q \quad (5)$$

#### 3.3.4.3 Kvalitetssikring

Ettersom denne delen gjenbruker mye kode fra 3.3.3 benytter den også mange av de samme testene. I tillegg er det laget en test som tester ikke-lineær uskarping (blur). *test\_blur\_image\_nonlinear* tester om ikke-lineær uskarping (blur) gir forventede verdier.

#### 3.3.4.4 Sluttresultat

Funksjonen kalles ved *localHDR.filter\_image(image, filters)*. Merk at dette gjøres identisk til 3.3.3.4. Den eneste forskjellen er verdien på Linear-parameteret. Redusering av  $\sigma_r$  vil redusere haloer. En økning i *diameter*-parameteret vil øke mengden støy som blir redusert på bekostning av hvor lang tid funksjonen bruker.

I figur vises bilder hvor den uskarpe delen av bildet er rendret globalt med ulike funksjoner nevnt i 3.2.3.2. Det skal nevnes at her er det uskarpe bildet laget med et kantbevarende filter, i motsetning til i 3.3.3.4.

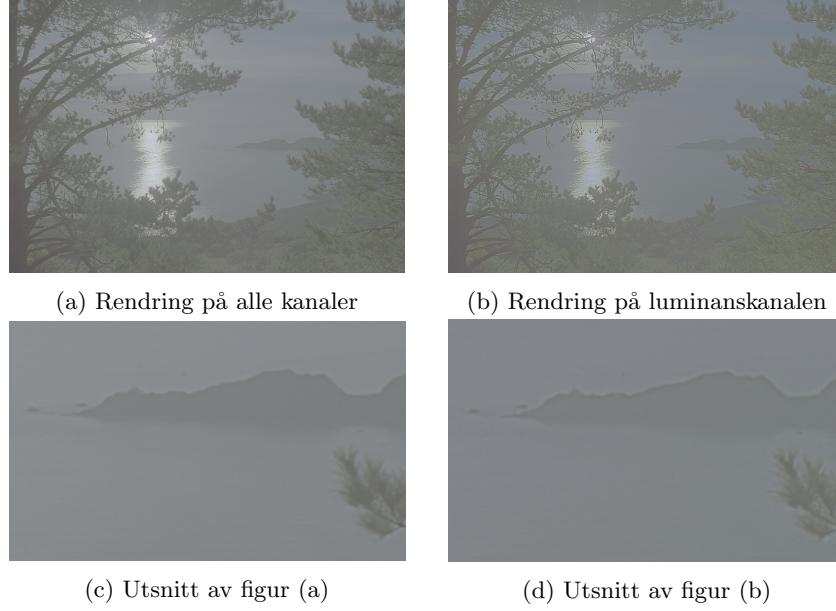
Ved inspirering av figur 12 ser man at fjellet har en negativ haloeffekt på den mørke siden av kanten i (a). Denne har blitt forbedret med bilateral filtrering, som vist i (b).

#### 3.3.5 Forbedringspotensiale

Ettersom Canny edge detection både skal gi et bedre resultat, og siden vi virkelig ble fascinert over denne metoden, hadde det vært en god idé å implementere denne. I ettertid ser at vi kunne benyttet oss av en ferdiglagd versjon Canny

---

<sup>9</sup>[https://docs.opencv.org/3.1.0/d4/d13/tutorial\\_py\\_filtering.html](https://docs.opencv.org/3.1.0/d4/d13/tutorial_py_filtering.html)



Figur 11: Bilateral filter med ulik render konfigurasjon

edge detection. Dette hadde redusert nødvendig arbeidsmengde og gitt et bedre sluttresultat.

### 3.3.6 Gradient komprimering

#### 3.3.6.1 Problemstilling

Som nevnt i oppgaveteksten er en teknikk å komprimere gradienten eller endringen i pixelverdien. Denne teknikken har utviklet seg fra at vi mennesker tolker endring i farger og ikke den absolutte verdien.

#### 3.3.6.2 Fremgangsmåte

For å finne en løsning må vi først finne gradienten,  $\nabla u_0$ . Vi ønsker å beholde vektorretningen og bare komprimere lengden. Når vi har gjort dette lager vi en ny  $\nabla u$ , som igjen blir brukt til å finne  $\nabla^2 u$ . Når vi har  $\nabla^2 u$  kan vi finne det bildet som har samme Laplacian ved å bruke en gradient nedstigningsmetode, som baserer seg på en partiell differensialligning. For å gjøre det enkelt for oss selv, valgte vi å bruke en eksplisitt løsning med randverdier  $\partial\Omega = 0$ . Deretter itererte vi oss til en løsning med formel (6).

$$u_{i,j}^{n+1} = \frac{\Delta t}{\Delta x^2} (u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n) - \Delta t \nabla^2 u + u_{i,j}^n \quad (6)$$

Dette vil konvergere til en verdi ettersom  $u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n$  i formel (6) er et estimat for  $\nabla^2 u_n$ . Etter en stund vil dette gi  $\nabla^2 u_n -$



(a) Gaussisk filtrering

(b) Bilateral filtrering

Figur 12: Sammenligning mellom Gaussisk og Bilateral filtrering

$\nabla^2 u \approx 0$ . I tillegg vil  $\Delta x^2 = 1$  ettersom vi regner med pixelverdier som har  $\Delta x = 1$  i begge retninger.

Siden metoden i formelen (6) kan føre til haloer rundt sterke kanter som nevnt i artikkelen til Fattal[9], foreslås en metode som bruker en Gaussisk pyramide. Denne metoden vil se på hvordan luminansen endrer seg på større deler av bildet og vil derfor glatte ut detaljene litt. Her vil man bruke samme prosess som i formelen (6), men for hvert nivå av pyramiden oppskaleres resultatet og brukes som initialverdiene i den neste iterasjonen. For at oppskaleringen skal fungere på alle nivåene må vi reskalere bildet siden de kan ha ulike dimensjoner. Dette er fordi dimensjonene kan være udelig med 2.

Alt dette utføres på logaritmen av luminansen,  $\ln L$ . Dette betyr at man ikke får et fargebilde tilbake i formel (6). For å rekonstruere dette har vi valgt å følge Fattal sin foreslalte metode og bruke formel (7) hvor  $C$  er et fargeblide og  $s$  er en konstant for å forandre på fargemetningen.

$$C_{out} = \left( \frac{C_{in}}{L_{in}} \right)^s L_{out} \quad (7)$$

### 3.3.6.3 Kvalitetssikring

For å sikre oss at koden fungerer som den skal har vi laget tre tester.

- *test\_gradient\_vectors*: Tester om gradient-vektorene og deres lengder er kalkulert riktig basert på en kjent funksjon.
- *test\_div\_matrix*: Tester om Laplacian-matrisen  $\nabla^2 u$  er kalkulert riktig med bakoverdifferanse basert på en kjent funksjon.
- *test\_compress\_gradient\_image*: Tester at hele prosessen returnerer et bilde som har samme dimensjon som orginalbildet for monokrome bilder.

### 3.3.6.4 Sluttresultat

Disse funksjonene kan brukes ved å kalle *gradient\_compression.gradient\_compress\_image(image, config)*. Her kan man sende med et config objekt, *GradientFilterConfig*, hvor

man kan sette parameter som sier om man skal bruke en Gaussisk pyramide, fargemetning, og verdiene  $n_{max}$  og  $\Delta t$  i formel (6).

Etter å ha implementert det som står i 3.3.6.2 ser vi at vi har glemt å endre  $\Delta x$  ved bruk av en Gaussisk pyramide. Dette gjør av vi får veldig sterke haloer på store deler av bildet, som vist øverst til venstre i figur 14b. Dette går litt i mot poenget med en Gaussisk pyramide når den egentlig skulle redusere sterke haloer rundt kantene.

En liten observasjon med den nåværende implementasjonen er at vi kan gjøre gradient eksplandering, noe som fører til en uskarpings-effekt (blurring). Dette gjøres ved å sette *GradientFilterConfig.func* til en funksjon som  $|\nabla^2 u|^4$ . Dette vil føre til at verdiene i bildet smitter over i andre pixler og vil derfor føre til uskarping, som vist i figur 13f.

### 3.3.7 Forbedringspotensiale

Siden vi bruker randbetingelsen  $\partial\Omega = 0$  som nevnt i 3.3.6.2, vil resultatet være noe unøyaktig. Dette betyr at vi kunne utvidet til en litt mer nøyaktig Neumannbetingelse. I tillegg kunne vi prøvd å bruke en implisitt model av formel (6).

Som nevnt i 3.3.6.4 burde vi prøvd å vekte de forskjellige verdiene i de forskjellige pyramide nivåene, noe som er nevnt i Fattal sin artikkel med formelene (8) og (9).

$$\nabla H_k = \left( \frac{H_k(x+1, y) - H_k(x-1, y)}{2^{k+1}}, \frac{H_k(x, y+1) - H_k(x, y-1)}{2^{k+1}} \right) \quad (8)$$

$$\varphi_k(x, y) = \frac{\alpha}{\|\nabla H_k(x, y)\|} \left( \frac{\|\nabla H_k(x, y)\|}{\alpha} \right)^\beta \quad (9)$$

Ved bruk av disse to likningene ville resultatet forhåpentligvis blitt enda bedre.

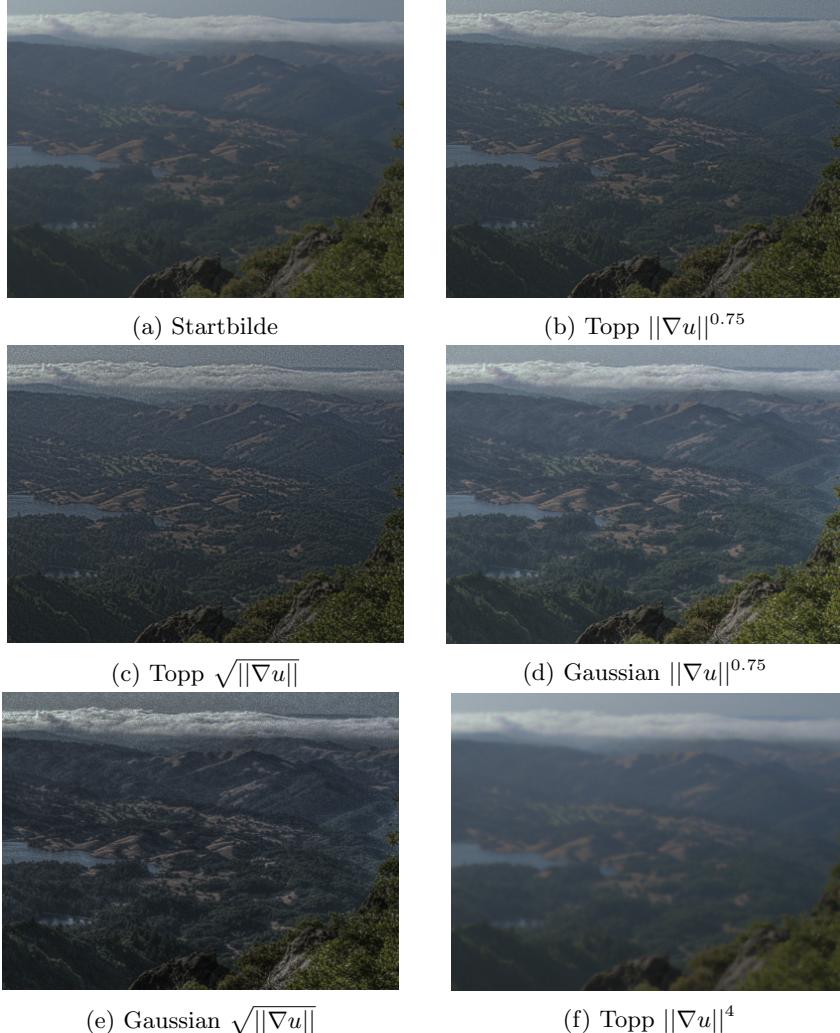
## 3.4 GUI

### 3.4.1 Problemstilling

For å få tillgang til de forskjellige funksjonalitetene som vi har implementert, må man først og fremst kunne kode, laste ned koden og skrive sin egen kode. For det første er dette en veldig tungvint, og i tillegg er det ikke sikkert at alle vil ha mulighet til å bruke arbeidet vårt; ettersom det forutsetter kodekompetanse.

### 3.4.2 Framgangsmåte

Løsningen på dette problemet ble å lage vårt eget grafiske brukergrensesnitt som inneholder det aller meste av funksjonene våre og tillater endring av parametre.



Figur 13: Gradientkomprimering oppskalert



Figur 14: Gradientkomprimering

Rammeverket vi endte opp med å bruke for å lage GUI-en er PyQt5<sup>10</sup>. Dette er et rammeverk som klarer å tegne GUI på Mac, Windows og Linux, noe som forminsker arbeidsmengden.

### 3.4.2.1 Struktur

All GUI-kode er skrevet i ren kode uten noen form for XML. Dette er fordi vi følte det ble lettere å modularisere koden med forskjellige komponenter, men også slik vi har bedre kontroll på hva som faktisk skjer med GUI-en.

For å vise bilder har vi valgt å bruke Matplotlib. Grunnen til dette er fordi vi er kjent med rammeverket, men også slik at vi har muligheten til å vise grafer som fargetethet og eventuelt HDR-grafer. Disse grafene ble aldri lagt til, men det er satt opp slik at det kun er en liten utvidelse.

I tillegg har vi laget våre egne grafiske komponenter som *FilterWidget*. Denne komponenten arver fra *QWidget* og viser de forskjellige matematiske funksjone ne som er beskrevet i 3.2.3. I tillegg viser den en slider for å endre på verdiene der det er mulig.

For å holde det ryddig har vi også en egen klasse for hver type operasjon. Dette har ført til at vi har *FilterWidget*, *LuminanceFilterWidget*, *GaussianFilterWidget*, *BilateralFilterWidget* og *GradientCompressionFilterWidget* hvor alle arver fra *FilterWidget* og har en funksjon som heter *apply\_filter(self, image)*. Denne bruken av strategy pattern vil gjøre ting lettere når vi skal redigere bildet.

Vi har også valgt å lage en egen klasse for å endre tallverdier. Her får man en tekst som sier hva verdien er, men den inneholder også to knapper for finpresisjon og en slider for større endringer. Alt dette ligger i klassen *SliderWidget*. Denne slideren er bygd på en responsiv måte, hvor man sender inn en funksjon som sier når brukeren har endret på verdien.

For å holde styr på alle disse *QWidget*-ene har vi en egen klasse *App*. Den inneholder de øverste nivåene av GUI-en, pluss alle forskjellige handlinger som kan gjøres. I tillegg inneholder den verdiene *hdr\_image*, *edited\_image* og *original\_image\_set*, som beskriver forskjellige variasjoner av det innlastede bilde.

Filterene som er lagt til i GUI-en vil også bli lagt til i en variabel som heter *filter\_widgets*. Som nevnt bruker vi et strategy pattern på filterne. Dette fører til at når vi skal redigere bildet, så kan vi gå gjennom alle filterne og kalle funksjonen *apply\_filter(image)* med originalbildet. Denne strukturen gjør at vi kan legge på så mange filter vi ønsker, og vi kan legge dem på i den rekkefølgen vi ønsker.

Siden vi har gjort det mulig å legge til uendelig mange filter, kan det fort bli lite plass på skjermen. På grunn av dette har vi endt opp med å legge alle filterne inn i et skrollbart vindu som kan bli uendelig stort.

---

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

### 3.4.3 Brukerveiledning og beskrivelse

#### 3.4.3.1 HDR-rekonstruksjon

For å benytte seg av HDR-rekonstruksjonsfunksjonaliteten trykker man på knappen '*Last inn bilde*'. Her har du mulighet til å laste inn et sett med .png bilder eller et .exr bilde. Hvis man velger et sett med .png bilder vil den automatisk generere HDR-bildet. Denne vil anta at bildene er opplinjert perfekt.

Ovenfor innlastningsknappen befinner det seg en statusbanner som sier hvordan operasjonene gikk. Dette betyr at den vil vise om det kommer en statusmelding om operasjonen gikk bra eller feilet.

#### 3.4.3.2 Bildeopplinjering

Ettersom bildene i 3.4.3.1 ikke opplinjer bildene automatisk, finnes det en egen knapp '*Opplinjer Bildesett*'. Denne funksjonen må gjøres manuelt ettersom algoritmen som nevnt i 3.1.6 ikke alltid klarer å gjøre en vellykket jobb.

#### 3.4.3.3 Global HDR rendering

For å benytte seg av denne funksjonaliteten i GUI-en trykker man på knappen '*Legg til globalt filter*'. Denne oppretter et tomt filter i det dynamiske sidelementet nede til høyre. Ved redigering av filteret kan man velge mellom de ulike funksjonene (kurvene) bildet skal rendres ved.

Om man ønsker å rendre bildet på luminanskanalen alene, velges '*Legg til luminans filter*'. Resten av prosessen utføres likt.

#### 3.4.3.4 Lineær spatiell filtrering

En lineær spatiell filtrering vil utføres når brukeren trykker på knappen '*Legg til gaussian filter*'. Da presenteres brukeren en meny som gir tilgang til å endre hvilket filter som skal rendre bildet og om det skal gjøres globalt eller kun i luminanskanalen. Naturligvis kan man også endre på sigma-verdien som endrer hvor mye bildet skal korrigere støy. Gamma-verdien kan endres om man ønsker å vektlegge detaljene mer eller mindre enn den uskarpe bakgrunnen.

#### 3.4.3.5 Ikke-lineær spatiell filtrering

For å utføre en ikke-lineær spatiell filtrering trykker man på knappen '*Legg til bilateral filter*'. Her kan farge- og bilderommet endres. Resten av mulighetene er like som i 3.4.3.4. Den eneste endringen er at diameter-parameteret endrer hvor kraftig støy skal korrigeres.

#### 3.4.3.6 Gradient komprimering

For å benytte seg av denne funksjonaliteten må man trykke på knappen '*Legg til gradient comp. filter*'. Denne fungerer rimelig likt som brukeropplevelsen i 3.4.3.3. Derimot kommer det også en boks man kan skru av og på som endrer på om man bruker en Gaussian pyramide. Man har også muligheten til å gjøre gradient komprimering kun på øverste nivå, se 3.3.6.2 for mer info.

### 3.4.3.7 Lagring av bilder

Ettersom man har begynt å endre på bildene, kan det være interessant å lagre resultatet. Derfor har vi lagt til en '*Lagre bilde*' knapp. Denne kan lagre .exr til et nytt .exr evt til et .png bilde. Har man lastet inn et sett med .png bilder har vi bare fått til å lagre til .png-format. Her vil man få opp et vindu som spør etter et filnavn å lagre til og hvilket format det skal være. Når brukeren har valgt filnavn, vil programmet ta det nåværende *edited\_image* og lagre bildet til fil. Brukeren har også mulighet å avbryte hvis det er ønsket.

### 3.4.4 Forbedringspotensial

Vi ser i ettertid at GUI-koden har blitt større enn vi forventet. Dette har ført til at koden i *App* har blitt veldig lang, noe linteren i 1.2.1.1 klagere over. En god idé ville vært å dele opp de forskjellige knappene til en egen klasse, som kunne blitt kalt *ToolbarWidget*.

I tillegg burde vi gitt enda mer utfyllende feilmeldinger. For øyeblikket er de veldig generelle. I noen scenarioer gir programvaren beskjed om at feilen skjedde ved innlasting når det egentlig er rendring som feiler.

Vi ser også at det burde vært mulig å endre på rekkefølgen til filterene. Dette ville gjort brukeropplevelsen mye mindre frustrerende når man ønsker å sette et av filterene før et annet. Vi burde også legge til muligheten for å skru av og på et filter i stedet for å måtte slette det helt. Dette ble ikke prioritert i prosjektarbeidet.

## 4 Oppsummering

Begge gruppemedlemmene har opplevd at prosjektet har vært morsomt og det er en faktor som motiverer. Dette har hjulpet oss stort ved at vi gjerne legger inn noen arbeidstimer på kveldstid i hverdagen eller trår til med noen økter i helgene. Ved å benytte *AwesomeGraphs*-tillegget<sup>11</sup> i Bitbucket kommer dette tydelig frem.

Underveis i utviklingen skulle vi ønske at vi brukte ferdiglagde biblioteker tidligere der det var tillatt, eksempelvis i ikke-lineær spatiell filtrering og i opplinjeringen.

### 4.1 Læringsutbytte

Hovedsaklig har vi lært hvordan et bilde er bygd opp og hvordan man kan behandle et bilde for å fremheve egenskaper som kan skjule seg i bildets data. Vi kjenner nå til de teoretiske metodene bak bildebehandlingen, og vi kan si noe om hvordan bilde vil endre seg ved bruk av ulike teknikker.

Prosjektet har også ført til bedre kjennskap til Python og L<sup>A</sup>T<sub>E</sub>X. Før vi startet arbeidet med prosjektet tok det tid å gjøre det man ønsker; nå går mye på automatikk.

---

<sup>11</sup><https://marketplace.atlassian.com/apps/1210934/awesome-graphs-for-bitbucket-server>

Strukturering av kode, bruk av forhåndsregler og oppsett av automatiske tjenester er også noe vi har lært bruken av. Dette er noe som er veldig aktuelt til senere prosjekter og i arbeidslivet. Spesielt oppsett av pipeline og automatisk testing har lært oss om konseptet Infrastructure as Code og hvordan det benyttes.

## Figurer

1	Coverage report . . . . .	3
2	Oversikt over git branches . . . . .	3
3	Utvikling i Trello over tid . . . . .	4
4	HDR-rekonstruksjon av monokromebilder . . . . .	8
5	HDR-rekonstruksjon av fargebilder . . . . .	9
6	HDR-rekonstruksjon av ulinjerte bilder . . . . .	12
7	Tre globale renderinger . . . . .	15
8	Tre globale renderinger på luminanskanalen . . . . .	16
9	Sammenligning for global rendering med $\sqrt{u}$ . . . . .	17
10	Gaussisk filter med ulik render konfigurasjon . . . . .	19
11	Bilateral filter med ulik render konfigurasjon . . . . .	21
12	Sammenligning mellom Gaussisk og Bilateral filtrering . . . . .	22
13	Gradientkomprimering oppskalert . . . . .	24
14	Gradientkomprimering . . . . .	24

## Listings

1	Sammenligning av to bilder . . . . .	4
2	Før Vektorisering . . . . .	10
3	Etter Vektorisering . . . . .	10
4	Datautsnitt . . . . .	11

## Referanser

- [1] Farup I. HDR-avbildning: rekonstruksjon og rendring; 2019. Available from: <https://bitbucket.org/ifarup/imt3881-2019-prosjekt/src/master/oppgave/oppgave.pdf>.
- [2] Debevec PE, Malik J. Recovering High Dynamic Range Radiance Maps from Photographs. In: Proceedings of SIGGRAPH 97. Computer Graphics Proceedings; 1997. p. 369–378.
- [3] Tomasi C, Manduchi R. Bilateral filtering for gray and color images. In: Iccv. vol. 98; 1998. p. 2.
- [4] Wang W, Xu Z, Lu W, Zhang X. Determination of the spread parameter in the Gaussian kernel for classification and regression. Neurocomputing. 2003;55(3-4):643–663.
- [5] Gupta S, Mazumdar SG. Sobel edge detection algorithm. International journal of computer science and management Research. 2013;2(2):1578–1583.
- [6] Perona P, Malik J. Scale-space and edge detection using anisotropic diffusion. IEEE Transactions on pattern analysis and machine intelligence. 1990;12(7):629–639.
- [7] Canny J. A computational approach to edge detection. In: Readings in computer vision. Elsevier; 1987. p. 184–203.
- [8] Neuback A, Van Gool L. Efficient non-maximum suppression. In: 18th International Conference on Pattern Recognition (ICPR’06). vol. 3. IEEE; 2006. p. 850–855.
- [9] Fattal R, Lischinski D, Werman M; ACM. Gradient domain high dynamic range compression. ACM Transactions on Graphics (TOG). 2002;21(3):249–256.