

Vitenskapelig programmering

IMT3881

POISSON IMAGE EDITING



Shamil Khumparov - StudNr: 997722
Abdi Mohammad Bako - StudNr: 488104

24.Mars 2020 - 15.Mai 2020

Sammendrag

Denne rapporten inneholder dokumentasjon og informasjon om arbeidsprosess i prosjekta-
beid med redigering og manipulering av forskjellige type bilder. Oppgavene til dette prosjek-
tet er hentet fra oppgavetekst[1]. Planlegging, metode og resultat blir gjennomgått underveis
i denne rapporten. Avslutningsvis skal det være diskusjon og konklusjon av prosjektet, her
vil det være hva som kunne gjort annerledes og/eller forandres i prosjektet.

Innhold

1 Innledning	1
2 Metode	2
2.1 Planlegging	2
2.2 Matematisk Funksjon	2
2.3 Koding	3
2.3.1 Glatting	3
2.3.2 Inpainting	4
2.3.3 Kontrastforsterkning	4
2.3.4 Demosaicing	5
2.3.5 Sømløs kloning	6
2.3.6 Anonymisering av bilder med ansikter	7
2.3.7 GUI	8
3 Resultat	9
3.1 Glatting	9
3.2 Inpainting	10
3.3 Kontrastforsterkning	11
3.4 Demosaicing	12
3.5 Sømløs kloning	13
3.6 Anonymisering av bilder med ansikter	13
3.7 GUI	14
4 Diskusjon	15
5 Konklusjon	17

1 Innledning

Bildebehandling er en stor del av hverdagen til folk flest, det blir brukt på alt fra sosiale medier, reklamer og annonseringer. Det finnes mange forskjellige bildebehandlingsprogrammer og de kan gå fra å være gratis til å koste mange tusen kroner, eller helt enkle å bruke til å være veldig komplisert bygget. Om det er gratis, dyrt, enkelt eller vanskelig redigeringsprogram, så er alle disse programmene bygd opp på de samme prinsippene. I denne rapporten skal vi beskrive og vise hvordan et av disse prinsippene, Poisson Image Editing", kan tas i bruk.

I dette prosjektet har vi tatt i bruk forkjellige teknikker for redigere bilder. Glatting det vil si gjøre et bilde mer blurrig/uskarp, "Inpainting blir brukt for å reparere ødelagt bilde eller ting fra bildet, Kontraskforsterkning det vil si å gjøre området i bildet sterke og gjøre primære områder skarpere enn de egentlig er. Demosaicing er en type teknikk som brukes til å rekonstruere et fullfargebilde fra ufullstendige fargebilder som blir sendt ut fra en bilde-sensor som monokrom. Sømløs kloning del av et bilde skal over til et annet bilde uten skarpe kanter. "Anonymisering av bilder med ansikter skjuler kun ansikt området i et bilde. Det vil bli tatt med fremgangsmåte, utføring og resultat, og i tillegg lages en GUI som kan brukes for å teste de forskjellige teknikkene som er beskrevet over.

Git repository

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

2 Metode

2.1 Planlegging

For prosjektorganisering og ansvarsfordeling har vi valgt å bruke Issues liste på Gitlab. Denne funksjonen har hjulpet oss med å skape oversikt over hvilke oppgaver som det arbeides med, og hvilken status de har. Statusene var *Todo* for oppgavene som skulle gjøres, *Doing* for oppgavene som det jobbes eller *Close issues* for oppgaven som var ferdig. For mer effektivisering av arbeid, ble det valgt å gi forskjellige oppgaver til enkelte i gruppen.

2.2 Matematisk Funksjon

For å kunne løse Poisson Image Edition skal bildet fremstilles som en funksjon $u : \Omega \rightarrow C$, der $\Omega \subset R^2$ er det rektangulære området hvor bildet er definert. Her er C fargerommet, $C=[0,1]$ er for gråtonebilde og C er fargerommet og $C = [0, 1]^3$ er for fargebilder. Bildet $u(x, y)$ fremkommer dermed som en løsning av Poisson ligningen (1).

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

Når man velger å bruke denne metoden til å løse problemstillingen, vil man trenge randverdier. Der randverdien $\partial\Omega$ og funksjonen $h : \Omega \rightarrow R^{\dim(C)}$ spesifiseres avhengig av hvilket problem som skal løses. Her vil det være to mulige randverdier å velge mellom, Neumann eller Dirichlet-typen. Siden vi har allerede valgt denne fremgangsmåten er det også viktig initialverdi for bildet, $u(x, y, 0) = u_0(x, y)$.

Man kan løse Poisson ligning ved hjelp av gradientnedstigning ved å innføre en kunstig tidsparameter og la løsningen utvikle seg mot konvergens:

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

Vi har to numeriske skjemar for funksjonen (1), eksplisitt og implisitt. I dette prosjektet skal vi kun ta i bruk eksplisitt. Hvis vi løser for funksjonen for $u_{i,j}^{n+1}$ får vi :

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

$$\alpha = \frac{\Delta t}{\Delta x^2} \quad (4)$$

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

2.3 Koding

2.3.1 Glatting

Vi startet med å utvikle den enkleste delen av Glatting teknikken ved å sette $h=0$. For videre utvidelse av problemstillingen var det å sette $h = \lambda(u - u_0)$. Her er u som er forandret på, u_0 er originalbilde og λ er en konstant som skal styre hvor mye bildet skal glattes. På grunn av dette ser den nye funksjonen slik ut:

$$u_{i,j}^{n+1} = u_{i,j}^n + \alpha(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n) - \lambda(u - u_0) \quad (6)$$

Neste steg var å implementere dette i Python.

Listing 1: Glatting

```
1  def GlattingFargeBilde(image):
2      alpha=.25
3      lamda=0.001
4      image = image.astype(float) / 255
5      stage = 100
6      imageKopi=np.copy(image)
7      data=plt.imshow(image) # for
8      for i in range(stage):
9          image[1:-1, 1:-1] += alpha * funk.eksplisitt(image)
10         funk.neumann(image)
11         image -= lamda * (image - imageKopi)
12         data.set_array(image)
13         plt.draw()
14         plt.pause(1e-2)
15     #plt.pause(10)
16     #plt.close()
```

På linje 3 i funksjonen(1) er lamda en konstant som styrer glatting nivå av bildet, desto lavere verdien blir, desto mer utsydelig blir bildet. Lamda og antall steg er avhengig av hverandre. Når lamda øker i verdi, må for-løkken ta flere steg. Linje 13 og linje14 i funksjonen(1) gjør slik at man har mulighet til å se prosessen under glatting. Funksjonen Eksplisitt() på linje 9 i funksjonen(1) brukes på til å finne verdi på en gitt pixlene, med å ta de som er rundt. Randbetingelse ble valgt ut i fra problemstilling, Neumann randbetingelse blir brukt i de oppgavene som gikk ut på å korrigere ved enden av randen på bildet. Hvis problemstillingen i oppgaven gikk ut på å korrigere området som ikke berører kantene av bildet, ble det valgt Dirichlet randbetingelse.

2.3.2 Inpainting

Listing 2: Inpainting

```

1 def inpainting(img, broken_im):
2     x0 = 240
3     y0 = 250
4     x_n = 240
5     y_n = 250
6     alpha = 0.25
7     mask = np.zeros(img.shape)
8     mask[np.where(img == 1)] = 1
9     mask = mask.astype(bool)
10    omega_i = broken_im[x0:y0, x_n:y_n]
11    u = np.copy(broken_im)
12    data = plt.imshow(broken_im)
13    for i in range(100):
14        u[1:-1, 1:-1] += alpha * eksplisitt(u)
15        omega_i[:, :0] = u[x0, :0]
16        omega_i[:, :-1] = u[x0:y0, x0:y_n-1]
17        omega_i[0:, :] = u[x0:y0, x_n:y_n]
18        omega_i[-1:, :0] = u[-1, :0]
19        u[~mask] = broken_im[~mask]
20
21    data.set_array(broken_im)
22    plt.draw()
23    plt.pause(1e-4)

```

For å reparere et skadet område eller fjerne noe fra bildet, brukes Inpainting teknikken som nevnt i innledningen. Det lages først et bilde med en skade på et spesifisert område på bildet, deretter blir det laget en boolsk maske der 1(hvit område) er Ω_i og 0(svart område) er resten av bildet. På linje 10 blir det opprettet et ω_i bilde som er den skadde området. For å fulle ut det skadde området er det brukt Direchlet randbetingelse, som setter ytterste kanten på det skadde området Ω_i lik den innerste området av Ω , dette skjer da på linje 14 til linje 18. i funksjon2). Deretter jobber programmet inn over med å fylle det skadde området.

2.3.3 Kontrastforsterkning

Listing 3: Kontrastforsterkning

```

1 def ContrastColor(image):
2     image = image.astype(float) / 255
3     alpha=.25
4     k=1
5     dudx=np.zeros(image.shape) #Alloker plasser
6     dudx[1:-1, 1:-1] = image[2:, 1:-1] - image[1:-1, 1:-1]
7     dudy=np.zeros(image.shape) #Alloker plass
8     dudy[1:-1, 1:-1] = image[1:-1, 2:] - image[1:-1, 1:-1]
9     gradientIm=np.exp(dudx**2) + np.exp(dudy**2)
10
11    dudx2=np.zeros(dudx.shape)
12    dudx2[1:-1, 1:-1] = dudx[2:, 1:-1] - dudx[1:-1, 1:-1]

```

```

13 dudy2=np.zeros(dudy.shape)
14 dudy2[1:-1, 1:-1] = dudy[1:-1, 2:] - dudy[1:-1, 1:-1]
15 gradient2Im = dudx2+dudy2 #beregner divergenten til gradient
16 data=plt.imshow(image)
17 for i in range(50):
18     image[1:-1, 1:-1] += alpha * eksplisitt(image) -
19     k * gradient2Im[1:-1, 1:-1] * gradientIm[1:-1, 1:-1]
20     neumann(image) #neumann rand betingelse
21     image[image>1]=1
22     image[image<0]=0
23     data.set_array(image)
24     plt.draw()
25     plt.pause(1e-2)
26 plt.close()

```

For å finne en mer kontrastert utgave av originalbildet u_0 har vi satt $h = \nabla * (f(\nabla u_0))$.

I funksjonen(3), linje 5 til linje 8 er det gradient av orginalbildet u_0 og på linje 9 er det satt inn en ikke-linjær funksjon av gradienten. Fra linje11 til linje 14 i funksjonen(3) har det blitt tatt div av $dudx$ og $dudy$, disse har deretter blitt slått sammen. For å kunne styre kontrastforsterkningen på bildet, settes det en konstant k . Konstant k er da verdient som styrer nivået på kontrasten. Ikke linjære funksjon av $(f(\nabla u_0))$ har vi brukt enkel løsning, med å sette $\exp(dudx^{**2}) + \exp(dudy^{**2})$, linje 9 på Listing3. For at det ikke skal føre til løsning utenfor fargeområdet med $k \in [1, \infty)$, har det blitt satt inn kodesnutt på linje 21 og linje 22 som kutter verdiene som kommer utenfor 0 og 1.

2.3.4 Demosaicing

Listing 4: Demosaicing

```

1 def demosaicing(image):
2     image = image.astype(float) / 255
3     alpha = .25
4     mosaic = lagMosaic(image)
5     def inpaint(img, mask):
6         for i in range(1):
7             img[1:-1, 1:-1] += alpha * eksplisitt(img)
8             img[:, 0] = mosaic[:, 0] #dirklet betingelse
9             img[:, -1] = mosaic[:, -1]
10            img[0, :] = mosaic[0, :]
11            img[-1, :] = mosaic[-1, :]
12            img[~mask] = mosaic[~mask]
13     return img
14 def demo():
15     #flytter over til rette kanaler
16     demo = np.zeros((mosaic.shape[0], mosaic.shape[1], 3)) # Alloker plass
17     demo[::2, ::2, 0] = mosaic[::2, ::2] # -> R-kanal
18     demo[1::2, ::2, 1] = mosaic[1::2, ::2] # -> G-kanal
19     demo[::2, 1::2, 1] = mosaic[::2, 1::2] # -> G-kanal
20     demo[1::2, 1::2, 2] = mosaic[1::2, 1::2] # -> B-kanal
21
22     mask = np.ones(demo.shape) # Alloker plass
23     mask[::2, ::2, 0] = 0 # setter R-kanal til false

```

```

24     mask[1::2, ::2, 1] = 0 # setter G-kanal til false
25     mask[ ::2, 1::2, 1] = 0 # setter G-kanal til false
26     mask[1::2, 1::2, 2] = 0 # setter B-kanal til false
27     mask = mask.astype(bool)# setter amsken til boolean array
28     data = plt.imshow(demo)
29     for i in range(50):
30         inpaint(demo[ :, :, 0], mask[ :, :, 0]) #inpainter R-kanal
31         inpaint(demo[ :, :, 1], mask[ :, :, 1]) #inpainter G-kanal
32         inpaint(demo[ :, :, 2], mask[ :, :, 2]) #inpainter B-kanal
33         data.set_array(demo) #
34         plt.draw()
35         plt.pause(0.5)
36     return demo
37 demo()

```

Som nevnt tidligere i innledningen er demosaicing en teknikk for å rekonstruere fargebilder tatt av monokromt kamera sensor. Vi undersøker denne teknikken ved å simulere et fargebilde u som $M \times N \times 3$ en numpy array og sette opp en gråtonemosaiikk av den, slik det har blitt vist på lagMosaic i Felles funksjoner. For å kunne demosaicing av mosaikken flyttes informasjon over til de rette kanalene som det er gjort Listing(4) : Linje 16 til 20. For å kunne fylle opp den manglende informasjonen lages det en boolsk maske for hver kanal(Red-Green-Blue), den skal sette 0-er på områdene som mangler informasjon. Deretter sendes hver av kanalene til funksjonen inpaint som funker på samme måte som funksjonen fra 2.3 Inpainting, den setter Dirichlet randbetingelse og $img[mask] = demo[mask]$.

2.3.5 Sømløs kloning

Listing 5: Sømløs kloning

```

1 def somlosKloning(image1,image2):
2     xleng = 100
3     yleng = 200
4     x0 = 350
5     y0 = 320
6     x_n =100
7     y_n = 200
8     alpha = 0.25
9     image1 = image1.astype(float) / 255
10    image2 = image2.astype(float) / 255
11    im2Del = np.zeros((xleng, yleng)) #omega_i /del av bildet som skal
12        kopieres
12    im2Del = image2[x_n:x_n+xleng, y_n:y_n+yleng]
13    image3 = im2Del.view()
14    lap_image3 = eksplisitt(image3)
15    for i in range(200):
16        lap_im2Del= eksplisitt(im2Del)
17        im2Del[1:-1,1:-1] += alpha * (lap_im2Del - lap_image3)
18        im2Del[ :, 0] = image1[x0:x0+xleng , y0]
19        im2Del[ :, -1] = image1[x0:x0+xleng , y0+yleng]
20        im2Del[ 0, :] = image1[x0 ,y0:y0+yleng]
21        im2Del[ -1, :] = image1[x0+xleng , y0:y0+yleng]
22        im2Del[ im2Del < 0] = 0

```

```

23         im2Del[ im2Del > 1] = 1
24         image1[ x0:x0+xleng , y0:y0+yleng ] = im2Del
25         plt .imshow( image1 )
26         plt .show( )
27         return image1

```

Noen ganger ønsker man å kopiere deler av bildet over til et annet bilde. Vi har laget enkel kode av denne teknikken på Listing 8. Det tas først ut et lite del av et bildet u_0 , som blir Ω_i . Det lages en versjon 2 av Ω_i på linje 13, slik at den kan brukes til å trekke den fra Ω_i for hvert steg den kopierer og går mer mot midten av Ω_i . For at programmet skal klare å kopiere over området fra u til Ω_i brukes det Dirchlet Randbetingelse. Denne fungerer på samme måte som det har blitt forklarte i Inpainting, den store forskjellen i disse teknikkene er det området i Ω_i fylles ikke helt, kun ende i den kopierte delen. For å fullføre kloningen, blir Ω_i satt i bilde u , det skjer på linje 24 i Listing (5).

2.3.6 Anonymisering av bilder med ansikter

Listing 6: Anonymisering av bilder med ansikter

```

1 def anonymiser( img):
2     gray = cv2 .cvtColor( img , cv2 .COLOR_BGR2GRAY)
3     img = img .astype( float ) / 255
4     cv2 .imshow( 'Origional' ,img)
5     cv2 .waitKey(3000)
6     cv2 .destroyAllWindows()
7     face_cascade = cv2 .CascadeClassifier( 'haarcascade_frontalface_default .xml' ,
8                                         )
9     faces = face_cascade .detectMultiScale( gray , 1.3 , 6)
10    for (x,y,w,h) in faces:
11        img = cv2 .rectangle( img , (x,y) ,(x+w,y+h) ,(0,0,0) ,0)
12        omega_i = img [y:y+h , x:x+w]
13        alpha = .25
14        u = img .view()  #view av bilde/numpy array
15        for i in range(100):
16            omega_i[1:-1 , 1:-1] += alpha * func .eksplisitt( omega_i)
17            omega_i[ : , 0] = u[ x:(x+w) , y]      #Deriklet rand betingelse
18            omega_i[ : , -1] = u[ x:(x+w) , y+h]
19            omega_i[ 0 , : ] = u[ x,y:(y+h) ]
20            omega_i[ -1 , : ] = u[ (x+w) , y:(y+h) ]
21        cv2 .imshow( 'Anonym' ,img)  #viser det anonymiserte bildet
22        cv2 .waitKey()
23        cv2 .destroyAllWindows()
24        return img

```

Anonymisering av ansikter er en teknikk som ofte blir brukt for å skjule ansiktet på en person som ikke vil at andre skal se hvem han er. For å lage egen løsning av dette, er det brukt et ferdig utviklet bibliotek, CV2[2]. Dette biblioteket markerer ansiktene på et bilde med en firkant boks som blir Ω_i . Dette gjøres på linje 7 til 11 i koden Listing(6) . Ω_i er området som skal skjules. For å gjøre dette har det blitt brukt samme teknikk som tidligere i Glatting. Området Ω_i glattes, og for at kun området i Ω_i skal glattes har det blitt brukt Direcklet randbetingelse.

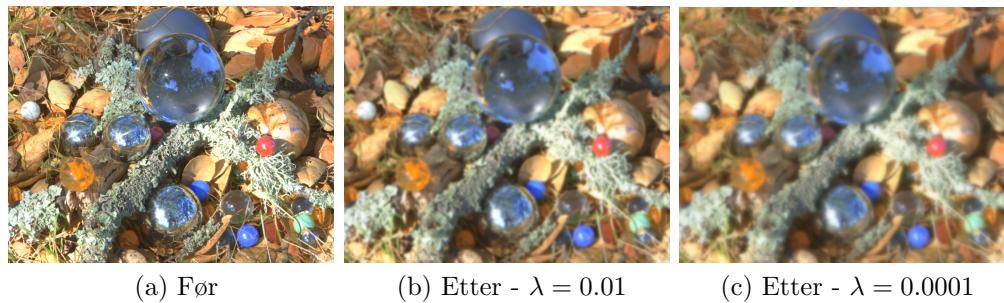
2.3.7 GUI

Det ble valgt å bruke Tkinter[3] pakke for å lage en GUI-applikasjon til dette prosjektet. Denne pakken ble valgt av den grunnen at den var enkel å implementere sammen med resten av koden. Denne GUI-applikasjon som heter Poisson Image Editinger laget med enkel brukergrensesnitt med enkle funksjoner. Applikasjonen vil gi brukeren mulighet å teste alle teknikkene vi har tatt med i prosjektet. GUI-applikasjonen skal bygges på *Button(Knapp)* funksjonen, det vil si at hele GUI-en blir bygget på at brukeren har kun mulighet å trykke *Knappene*. Parametrene i koden(6) styrer posisjonen til den knappene. Brukeren vil få mulighet å velge Last opp bildesom skal manipuleres med. For å kunne sentrere *Window(Vindu)* ble det laget en funksjon som sjekker brukerens skjerm og setter den midten av den.

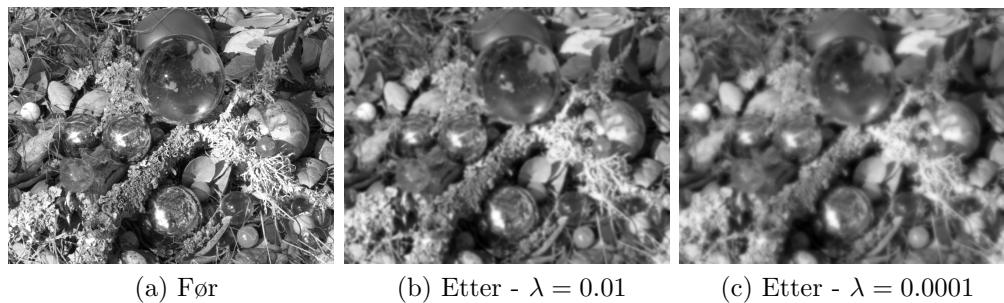
3 Resultat

3.1 Glatting

Bilde (a) på Figur (1) og Figur (2) er bildet før den har blitt kjørt med glatting/blurring programmet. Der kan vi se bildet med gode detaljer. Bilde (b) har blitt kjørt med programmet med $lamda = 0.01$ og antall $steg = 25$. Her ser vi at bildet har blitt litt utsynlig og mistet noen detaljer som for eksempel skarphet i lyset. På Bilde (c) har det blitt kjørt i programmet $lamda = 0.0001$ og antall $steg = 25$, her kan vi tydelig se at bildet har blitt enda mer utsynlig og mistet alle type lyseffekt som var på bildet. Ut ifra informasjonen resultatet har gitt, kan vi se at jo lavere konstanten $lamda$ blir, jo mer utsynlig/uskarp blir bildet. Det kom fram at kombinasjonen mellom $lamda = 0.01$ og $steg = 25 - 50$ passet best sammen, slik at det ikke ble for lite eller for mye av glattingen. Alt i alt så kan vi se at glatting virker likt på fargebildet og gråbilde.



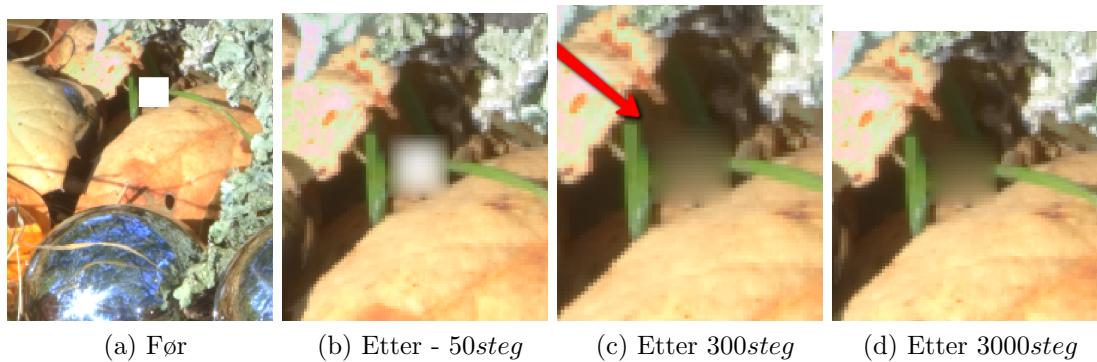
Figur 1: Fargebilde glatting



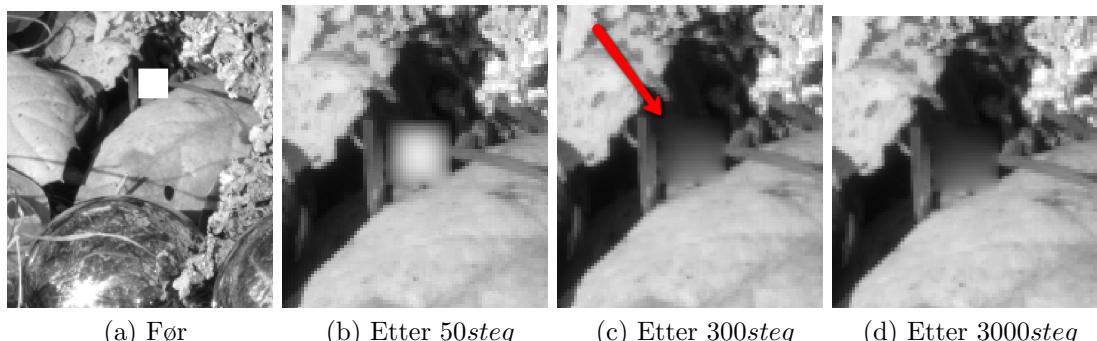
Figur 2: Gråbilde glatting

3.2 Inpainting

Inpainting går ut på å reparere en skadet del av bildet eller fjerne noe fra bildet. På bildene fra Figur (3) og Figur (4) kan vi se utviklingen av inpainting. Antall steg er avhengig av skaden det er på bildet, desto større skade det er, desto flere steg trengs det for å fylle opp det ødelagte området Ω_i . På Bilde (c) og Bilde (d) i Figur (3) og Figur (4) kan vi se at det ikke er så stor forskjell mellom disse bildene, til tross for at det har blitt utført ti ganger så mange steg på Bilde (d). Denne informasjonen gir oss et resultat på at det kommer et slutt punkt som ikke kan forbedres med dette programmet.



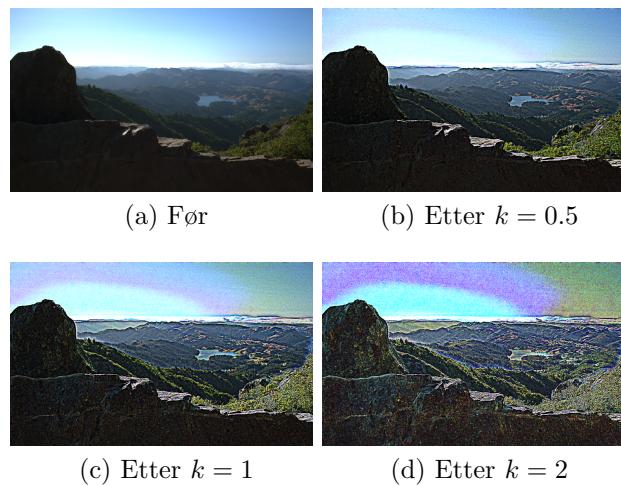
Figur 3: Fargebilde inpainting



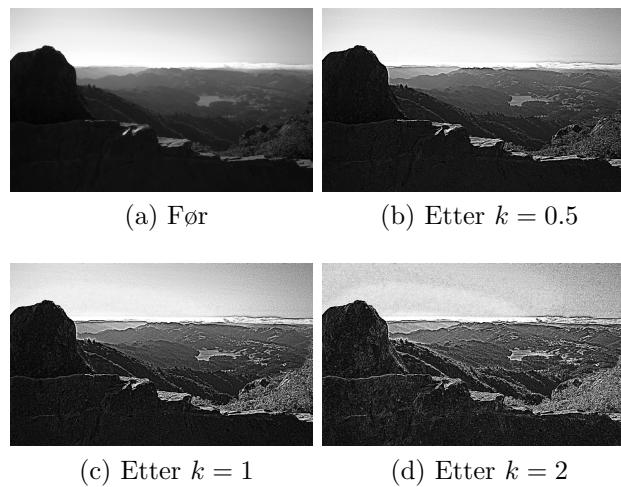
Figur 4: Gråbilde inpainting

3.3 Kontrastforsterking

For å løse problemstillingen med kontrastforsterking ble det satt opp $h = \nabla * (f(\nabla u_0))$. Her var det konstanten k som bestemmete hvor mye kontrast skulle tas på bildet skulle tas på. På Figur (5) og Figur (6) kan vi se forskjellige Bilder (a-d) med forskjellig kontrast. Når konstant k nærmet seg mot 0, var det mindre kontrastforskjell på bildet. På Figur (5) og Figur (6) Bilder (a) ser vi origalbildet som er litt mykere i farge, mens på Bildene(b,c,d) kan vi se, høyere konstant k er, mer og mer kontrastforsterkning det blir på bildene. Utfra resultatet kan vi se at beste konstanten for å få et stabilt kontrastforsterkning, så er det $k = 1$.



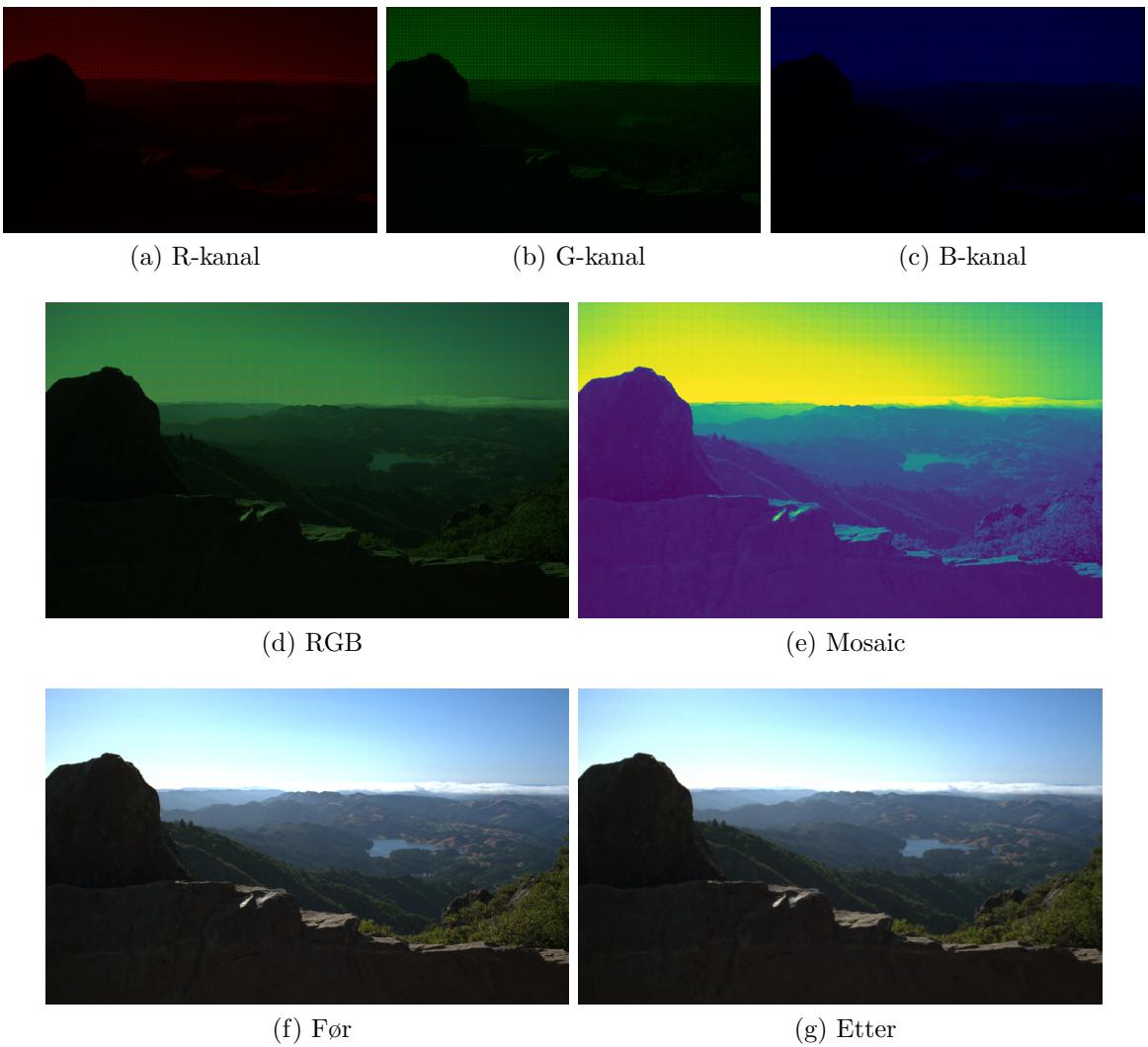
Figur 5: Fargebilde kontrastforsterkning



Figur 6: Gråbilde kontrastforsterkning

3.4 Demosaicing

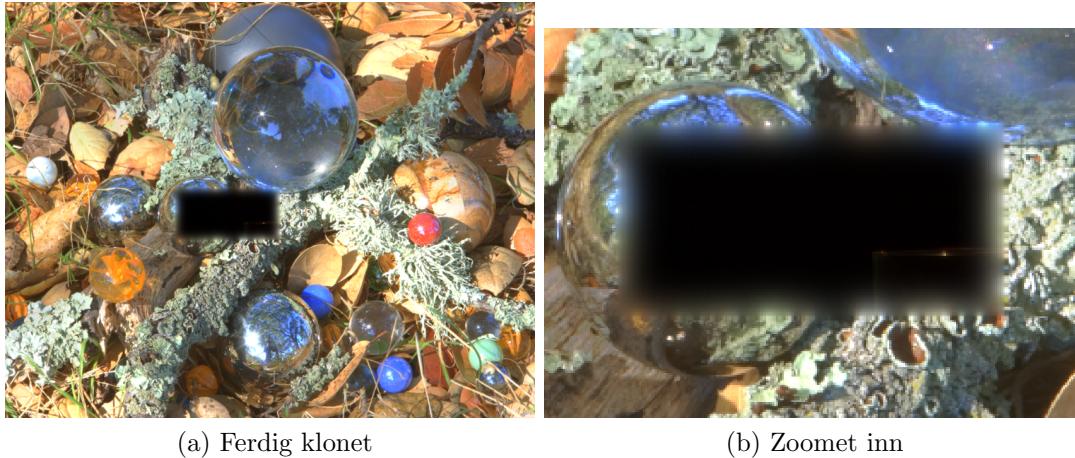
På Figur (7) ser vi en illustrasjon på hvordan en demosaicing fungerer. Bildene (a-c) har en illustasjon på hvordan et bilde ser ut i hvert av Rød-Grønn-Blå kanalene. Bilde (d) er når alle kanalene har blitt satt sammen å fylle inn den manglende informasjonen, det viser kun ren RGB bilde. Bilde (e) er en mosaic av originalbilde (f). Bilde (g) i Figur (7) er et sluttresultat av demosaicing med bruk av Bilde (f). Om originalbilde og sluttbilde av demosaicing blir sammenlignet er de nesten like, men om man ser nøyere på bildene kan vi se at originalbilde (f) er litt skarpere på fargene.



Figur 7: Demosaicing

3.5 Sømløs kloning

Bildene på Figur (8) viser hvordan Sømløs kloning fungerer, etter at den har flyttet over del av et bilde til et annet bilde. Tatt i betraktning at bilde (a) på Figur (8) har større oppløsning, kan man forsatt se at overgangen er sømløs. På bilde (b) kan man se en zoomet in versjon av bildet, og på denne kan man tydelig se en sømløs overgangen fra Ω_i til Ω



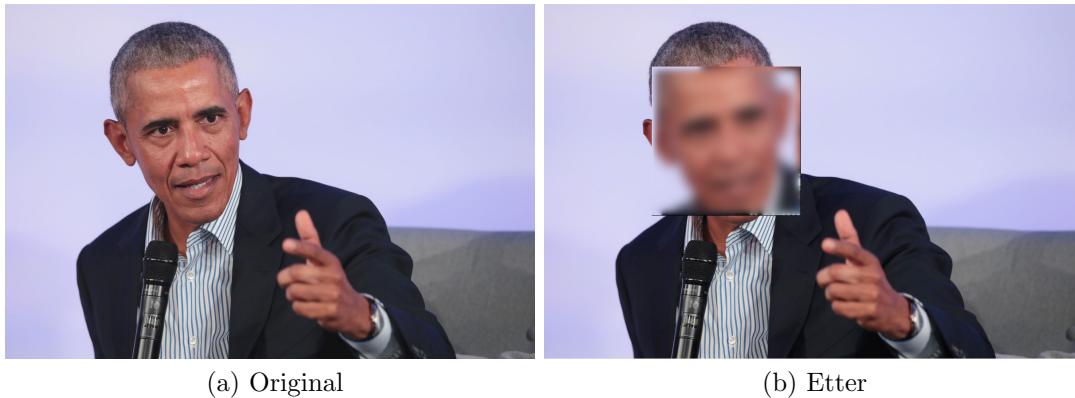
(a) Ferdig klonet

(b) Zoomet inn

Figur 8: Sømløs Kloning

3.6 Anonymisering av bilder med ansikter

Som nevnt i Section(2.3.6), går teknikken ut på å skjule ansiktet til person. På Figur (9) er det illustrert et bilde før og etter at ansiktet har blitt anonymisert. Størrelsen av det kvadratiske området Ω_i er bestemt av Python biblioteket CV2[2]. Utifra denne boksen, ble ansiktet glattet slik at det blir utsynlig.



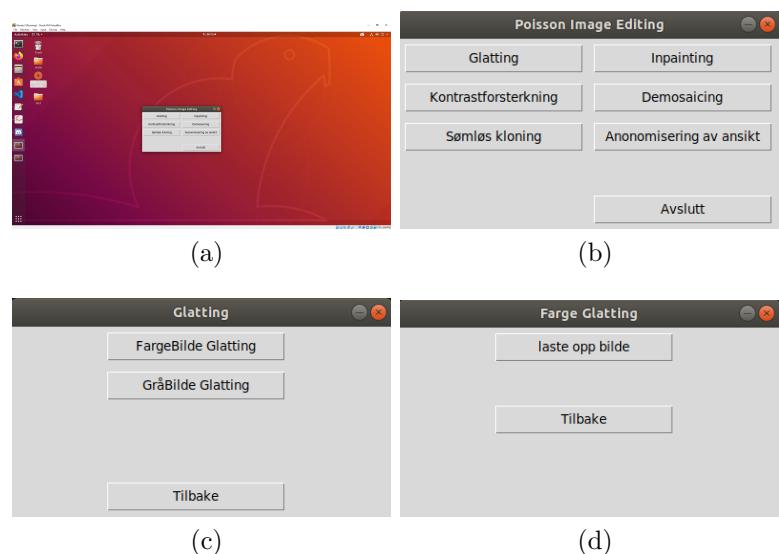
(a) Original

(b) Etter

Figur 9: Anonymisering av ansikt

3.7 GUI

Bildene i Figur (10) viser hvordan deler av GUI-applikasjonen ser ut. På bilde (a) i Figur (10) er det vist hvordan *Vindu* ser ut i midten av skjermen. Bilde (b) *PoissonImageEdition* er hovedvinduet når applikasjonen starter. Det er på denne delen av applikasjonen brukeren får valgt de forskjellige metodene. Hvis brukeren velger å trykke på *knappen* glatting, kan vi se hva som skjer på bildene (c-d) i Figur (10). På slutten av alle teknikkene er det en *knapp* “Last opp bilde”, som gir brukeren mulighet til å selv velge bildet som skal manipuleres. Siden det var valgt å lage en helt enkel GUI applikasjon, åpnes det det nytt *vindu* hver gang brukeren velger å gjøre noe med applikasjonen



Figur 10: GUI-applikasjon

4 Diskusjon

Fremgang

Underveis i prosjektet dukket det opp noen problemer og vanskeligheter. Problemene varierte, slik som tidsbruk på de forskjellige oppgavene, feil i kodingen og forståelse av oppgaveteksten. For å løse problemene som dukket opp, ble det tatt i bruk flere tiltak. Først og fremst ble det snakket internt i gruppen om hvordan vi skal komme fram til en løsning for disse problemene. Når det gjelder problemer med kodingen, valgte vi å ta i bruk internett. Selv om det ikke var direkte svar til oppgaveteksten, fantes det teoretisk forklaring på problemstillingene som var i oppgaveteksten. Hvis vi fortsatt ikke klarte å komme fram til en løsning for problemet, ble det spurt om hjelp av emneansvarlig eller lærerassistenene.

Selv om prosjektet ikke ble gjennomført helt etter planen nevnt i metoder tidligere, ble tidsrammen fulgt som planlagt. Istenfor at enkelte i gruppen jobbet individuelt, ble vi enige om å samarbeide med oppgavene. Det viste seg å være mer effektivt med tanke på tidsbruk på oppgavene. For bedre flyt i prosjektet kunne det ha vært gjort fra starten av, ettersom gruppen ikke var kjent, valgte vi å jobbe individuelt med oppgavene.

Gjennomføring

For at det ikke skal bli lange kode-linjer i rapporten, har vi valgt å kutte gjentagende kode i (2.3). Linje numrene og kodene er ikke eksakt kopi av filene, deler av de er kuttet, for å ikke bruke unødvendig plass på rapporten. Grunnen til at vi har valgt å bruke `data.set-array()` og `plt.draw` var for å se prosessen underveis av de forskjellige teknikkene. Vi kunne ha ersetret det med å sette `imshow()` og `show()` på slutten koden, men da kan man kun se resultaten og ikke selve prosessen. Denne løsningen har vi ikke tatt med i Sømløs kloning og anonymisering av ansikt. Grunnen til det var at oppsettet på Sømløs kloning ikke passet til det, og på anonymisering av ansikt så er det tatt i bruk en annen type pakke og derfor bestemte vi oss for å ikke bruke tid på det.

Med Glatting/Blurring delen av oppgaven fikk vi problemer med å implementere $h = \lambda(u - u_0)$. Når vi byttet ut $h = 0$, fikk vi melding om at det var shape forskjell på Listing(1), linje 8. Etterhvert fant vi ut at h skal trekkes ifra bilde- u som er ferdig å kjøre med eksplisitt. På Listing(1), kan $h = \lambda(u - u_0)$ settes enten foran Neumann eller etter, det påvirker ikke funksjonaliteten.

På inpainting startet vi med å lage en kode som selv finner den skadde delen på et bilde. Her fikk vi problemer med å skjønne hvordan det skulle gjøres. Etterhvert som vi ikke klarte å løse problemstillingen fikk vi hjelp av emneansvarlig. Vi fikk beskjed at det var i orden å skrive kode som lager skade på et bestemt område og løse det med ”hard kode”. Det gjøres med å sette parametre som skader på en bestemt posisjon og bruke samme parametrene til å fylle den opp.

Med kontrastforsterkningen startet vi å gjøre den basic delen av koden, med å sette $h = k\nabla^2u_0$. Denne delen hadde vi ikke problemer med fordi den er ganske lik Glattingkoden, forskjellen er at $h = k\nabla^2u_0$ og passer på at føring blir $u- > [0, 1]$. Vanskeligheter oppsto når vi gikk over til den mer avanserte delen. “En mer avansert form for kontrastforsterkning kan vi lage som beskrevet ved å innføre en ikke-lineær funksjon av gradienten, f.eks. $g = f(\nabla u_0)$, og så la $h = \nabla \cdot g = \nabla \cdot (f(\nabla u_0))$ i ligning (1). Randverdier og føringer ble som beskrevet over.” [1] Det viste seg at vi hadde løst oppgaven ((∇u_0)), og ikke tatt en funksjon av gradient ($f(\nabla u_0)$). Hvordan det har blitt implementert i koden er beskrevet i Section(2.3.3). Av en eller annen grunn virket ikke koden(3) på svart/hvit bilde, etter at vi byttet fram og tilbake mellom to ikke linjære funksjoner funket den. Når det gjelder demosaicing så fikk vi ikke noen store utfordringer med å kode den. Det eneste utfordrende under løsingen var å sette sammen de tre fra linje 26-28 i koden(4). Måten det har blitt løst på, kan man se i koden(4).

Når det gjelder den teoretiske delen, hadde vi ingen store utfordringer å skjønne dette. Når vi først begynte å implementere kodedelen, kom en del utfordringer. Vi hadde problemer med å sette opp riktig randbetingelse i forhold til kodingen. For å løse problemstillingen i denne oppgaven, fikk vi hjelp av emneansvarlig og noen forskjellige nettsider med forklaring av lignende type problemstilling.

Anonymisering av ansikt var annerledes fra andre teknikker vi har beskrevet over, for det ble brukt et annet type Python bibliotek [2] til denne oppgaven. Her var det viktig å sette seg inn i funksjonene i den nye pakken, for å kunne implementere det sammen med resten. Når man første skjønte den, var det bare å bruke glatting som i Section(2.3.1). Vi testet med et bilde som hadde flere ansikt, og det funket ikke pga forskjellige shape. Derfor valgte vi å bruke bilder med kun et ansikt.

GUI-applikasjonen kunne utvikles på mange forskjellige måter. Vi valgte satse på gjøre ferdig flere teknikker enn å lage en fin GUI-applikasjon. På grunn av tidsbegrensning så var applikasjonen bygd opp på en enkel måte. Det kunne ha vært lagt til flere muligheter for brukeren. Et av mulighetene vi prøvde å legge til var at brukeren selv bestemmer forskjellige parametrene i forskjellige teknikkene. For eksempel endre på λ i Section(2.3.1) eller konstant k i Section(2.3.3).

5 Konklusjon

Gjennom dette prosjektet fikk vi laget og gjennomført Poisson Image Editing med forskjellige teknikker innon for redigering og manipulering av bilder. Det er ting som kunne gjort annerledes i dette prosjektet, tatt med flere klasser som å gjøre kode delen bedre og enklere for struktur. Vi kunne tatt med test-filer for å teste de forskjellige delene av prosjekten. Prosjektet i sin heltet var grei å gjennomføre. Største utfordringen i prosjektet, var å samarbeide med prosjektet kun gjennom internett, og ikke ha fysiske møter.

Referanser

- [1] Farup Ivar. Poisson image edition, 2020. URL <https://git.gvk.idi.ntnu.no/course/imt3881/imt3881-2020-prosjekt/-/blob/master/oppgave/prosjekt.pdf>.
- [2] opencv-python 4.2.0.34, 2020-4-4. URL <https://pypi.org/project/opencv-python/>.
- [3] Tkinter - graphical user interface, 2019-12-06. URL <https://wiki.python.org/moin/TkInter>.

Figurer

1	Fargebilde glatting	9
2	Gråbilde glatting	9
3	Fargebilde inpainting	10
4	Gråbilde inpainting	10
5	Fargebilde kontrastforsterkning	11
6	Gråbilde kontrastforsterkning	11
7	Demosaicing	12
8	Sømløs Kloning	13
9	Anonymisering av ansikt	13
10	GUI-applikasjon	14

Listings

1	Glatting	3
2	Inpainting	4
3	Kontrastforsterkning	4
4	Demosaicing	5
5	Sømløs kloning	6
6	Anonymisering av bilder med ansikter	7