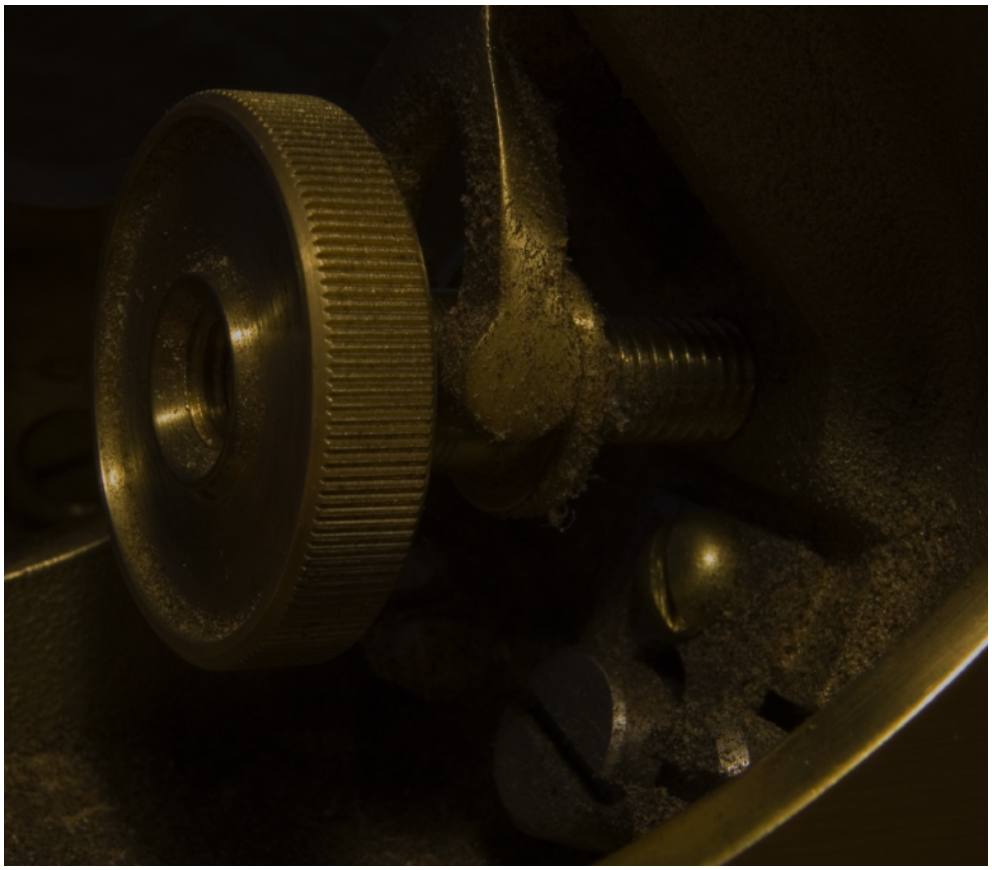
Vitenskapelig programmering
IMT3881

Poisson Image Editing



Mustafa Abdullah
StudNr: 472447
Våren 2021
GIT URL:
https://git.gvk.idi.ntnu.no/mustafaa/imt3881-2020-prosjekt.git

Innhold

1 Innledning **1**

2 Metode **1**

3 Glatting **2**
3.1 Glatting Metode 2
3.2 Glatting Resultat 2

4 Inpainting **3**
4.1 Inpainting Metode 3
4.2 Inpainting Resultat 4

5 Kontrast **5**
5.1 Kontrast Metode 5
5.2 Kontrast Resultat 5

6 Demosaicing **7**
6.1 Demosaicing Metode 7
6.2 Demosaicing Resultat 7

7 Sømløs **8**
7.1 Sømløs Metode 9
7.2 Sømløs Resultat 9

8 Anonymisering **10**
8.1 Anonymisering Metode 10
8.2 Anonymisering Resultat 11
8.3 GUI 11

9 Diskusjon **12**

10 Konklusjon **12**

1 Innledning

I dette prosjektet har vi fått oppgaven av å bruke forskjellige teknikker for behandling av bilder som heter Poisson image Editing [1]. Teknikken som blir gått gjennom er Glatting, Inpainting, Kontrastforsterkning, Demosaicing, sømløs koding og Anonymisering. Nedover så går vi gjennom koden og resultatene som ble fått av å kjøre koden på disse.

2 Metode

Metoden for å løse Poisson Image Edition, går ut på at man et bilde som en funksjon: $u : \Omega \to C$ der $\Omega \subset R^2$ er det rektangulære bilde område. C Representerer fargerommet, der C=[0,1] er for gråtone-bilder og $C = [0, 1]^3$ er fargerommet for bilder med farger. Bildet $u(x, y)$ fremkommer dermed som en løsning av Poisson Ligningen:

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

Der $\partial\Omega$ er randverdien, og funksjonen $h : \Omega \to R^{\dim(C)}$ spesifiseres avhengig av oppgaven som løses. Poisson ligninger løses ofte ved å iterere seg til svaret ved å sette inn en tidsparameter for å få en konvergen:

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

Funksjonen (2) har to numeriske skjemaer explisitt og implisitt der explisitt blir brukt gjennom dette prosjektet. Løser det:

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

$$\alpha = \frac{\Delta t}{\Delta x^2} \tag{4}$$

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

3 Glatting

Glatting går ut på å gjøre gjøre et blide ved å glatte ut fargene for å gjøre bilde utydelig, eller ved bruk av engelske ord “blurry” eller “Smoothing”.

3.1 Glatting Metode

```
6   def FGlatting():
7       alpha = 0.1
8       lamBda = 0.001
9       stages = 10000
10      image = Funksjoner.importImg()
11      image = image.astype(float) / 255
12      mirrorImage = np.copy(image)
13
14      plt.imshow(image)
15      plt.show(block=False)
16      plt.close()
17
18      BufferData = plt.imshow(image)
19      for x in range(stages):
20          plt.axis('off')
21          image[1:-1, 1:-1] += alpha * Funksjoner.eksplisitt(image)
22          Funksjoner.neumann(image)
23          image = image - (lamBda * (image - mirrorImage))
24          BufferData.set_array(image)
25          plt.draw()
26          plt.pause(1e-2)
27
```

Figur 1: Glatting kode

. Koden for glatting i dette prosjektet (Figur: 1), begynner ved å definere variablene lambda og Stages. Lambda blir skrevet som LamBda som definert på linje 8, dette fordi lambda i python er en pre-definert variabel for anonyme funksjoner. Lambda er en konstant-variabel som styrer hvor mye bilde glattes, desto mindre lambda verdi desto mer glattes blide. Dette vil da medføre at vi trenger flere gjennomganger gjennom “for” løkken i linje 19. I Linje 21 prøver å få tak i verdien til en gitt pixel, med pixelene rundt den spesifiserte pixelen.

3.2 Glatting Resultat

Kjøring av koden viste til at at koden for å glatte fargede bilder fungere like bra på grå bilder (Figur: 2). Det viste seg også at for mange steg på løkken var unødvendig. For eksempel

Lambda verdi på 0.01 viste seg til å være unødvendig etter 30 steg.

(a) Orginal bilde (b) Farget Glatting (c) Grå Glatting

Figur 2: Glatting Resultat

4 Inpainting

Inpainting går ut på å fikse/reparere deler av et bilde eller legge til et bilde. For å demonstrere dette i dette prosjektet så blir det brukt en funksjon for å skade et spesifik område i et bilde, og gitt til koden for å kunne demonstrere Inpainting.

4.1 Inpainting Metode

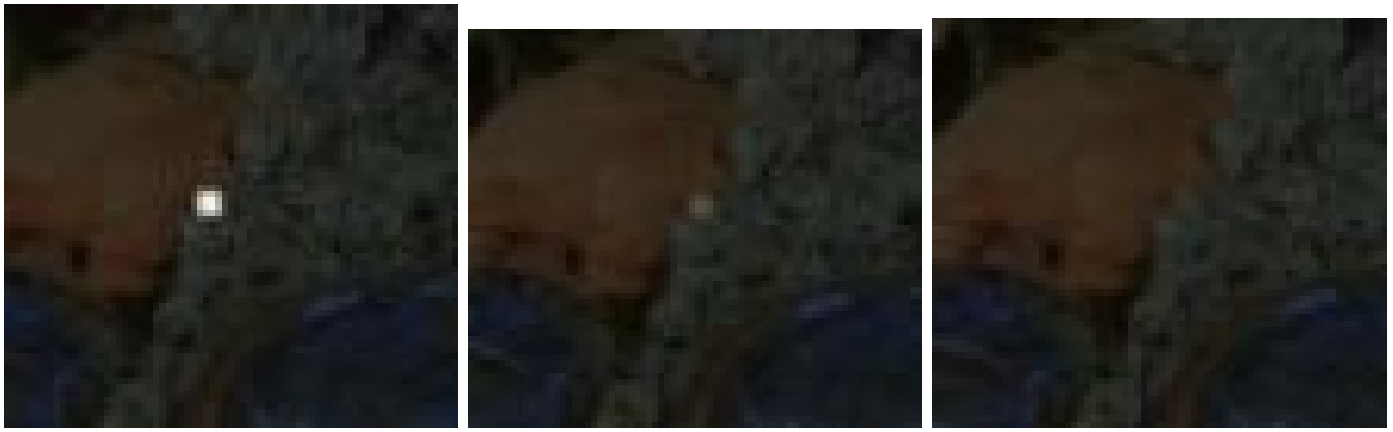
I denne koden går ut på å bruke Direchlets randbetingelse for å hente verdien til ytterste områdene til det skadete område og setter det lik det innerste område, dette skjer på flere steg for å bearbeide skadet område innover. Dette skjer ved hjelp av omega of mirrorimage i for løkken i Figur 3 i linje 17.

```
6   def inpainting(image, DmgImage):
7       x = 300
8       y = 310
9       alpha = 0.25
10
11      OverLay = np.zeros(image.shape)
12      OverLay[np.where(image == 1)] = 1
13      OverLay = OverLay.astype(bool)
14      Omega = DmgImage[x:y, x:y]
15
16      MirrorImage = np.copy(DmgImage)
17      for i in range(100):
18          MirrorImage[1:-1, 1:-1] += alpha * Funksjoner.eksplisitt(MirrorImage)
19          Omega[:, :0] = MirrorImage[x, :0]
20          Omega[:, :-1] = MirrorImage[x:y, x:y - 1]
21          Omega[0:, :] = MirrorImage[x:y, x:y]
22          Omega[-1:, :0] = MirrorImage[-1, :0]
23          MirrorImage[~OverLay] = DmgImage[~OverLay]
24
25          plt.imshow(DmgImage).set_array(DmgImage)
26          plt.draw()
27          plt.pause(1e-4)
```

Figur 3: Inpainting kode

4.2 Inpainting Resultat

Under i Figur 4 så kunne vi se hvordan bilde utviklet seg underveis som bilde blir utviklet ettersom bilde blir utviklet.



(a) Skadet Bilde (b) Bilde under reperasjon (c) Bilde Reparert

Figur 4: Inpainting resultat

5 Kontrast

Kontrastforsterkningsmetoden går ut på å manipulere fargene på et bilde.

5.1 Kontrast Metode

For kunne finne en mer kontrastert bildeutgave av det originalbildet $u0$ er det satt $h = \nabla * (f(\nabla u0)$. For å kunne justere kontrastforsterkning på et gitt bilde, blir det satt inn en konstant k, denne k er det som styrer nivået på kontrasten. For at løsningen ikke går utenfor fargeområdet med k>1, har det blitt satt inn metode som kutter verdiene som vil komme utenfor 0 og 1.

```
def ContrastColor(image):
    image = image.astype(float) / 255
    alpha = 0.25
    k = 1

    derivx = np.zeros(image.shape)
    derivx[1:-1, 1:-1] = image[2:, 1:-1] - image[1:-1, 1:-1]
    derivy = np.zeros(image.shape)
    derivy[1:-1, 1:-1] = image[1:-1, 2:] - image[1:-1, 1:-1]
    gradient = np.exp(derivx ** 2) + np.exp(derivy ** 2)

    derivx2 = np.zeros(derivx.shape)
    derivx2[1:-1, 1:-1] = derivx[2:, 1:-1] - derivx[1:-1, 1:-1]
    derivy2 = np.zeros(derivy.shape)
    derivy2[1:-1, 1:-1] = derivy[1:-1, 2:] - derivy[1:-1, 1:-1]
    gradient2 = derivx2 + derivy2

    BufferData=plt.imshow(image)
    for i in range(50):
        image[1:-1, 1:-1] += alpha * Funksjoner.eksplisitt(image) - k * gradient2[1:-1, 1:-1] * gradient[1:-1, 1:-1]
        Funksjoner.neumann(image)
        image[image > 1] = 1
        image[image < 0] = 0
        BufferData.set_array(image)
        plt.draw()
        plt.pause(1e-2)
    plt.close()
```

Figur 5: Kontrast kode

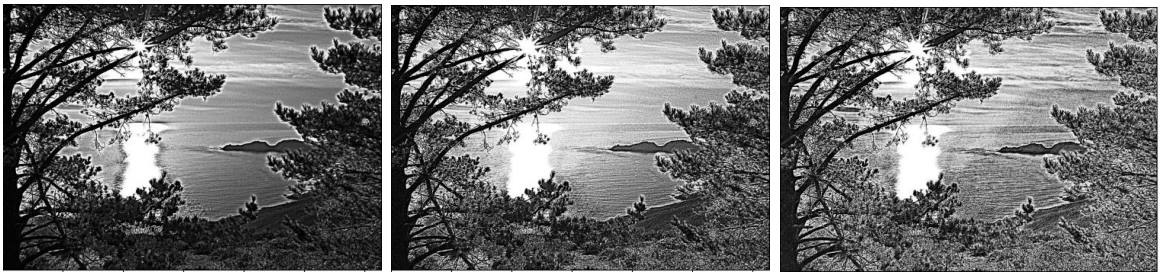
5.2 Kontrast Resultat

Figur(6), er originalbildet som vil bli brukt for å manipulere forsterkningen på. På figur(7) ser vi at, høyere konstant k er, sterkere farge kontrasten på bildet blir.

Figur 6: Kontrast Orginal

(a) K=1 (b) K=2 (c) K=3

Figur 7: Farget Kontrast



(a) K=1 (b) K=2 (c) K=3

Figur 8: Grå Kontrast

6 Demosaicing

Demosacing er et rekonstruksjons algoritme for rekonstruksjon av monokrome kamerasensorer, der man kan konstruere bildene fra 4 forskejllige kanaler.

6.1 Demosaicing Metode

```
20    def inpainting(image, mask):
21        for i in range(1):
22            image[1:-1, 1:-1] += alpha * Funksjoner.eksplisitt(image)
23            image[:, 0] = mosaicdemo[:, 0]
24            image[:, -1] = mosaicdemo[:, -1]
25            image[0, :] = mosaicdemo[0, :]
26            image[-1, :] = mosaicdemo[-1, :]
27            image[~mask] = mosaicdemo[~mask]
28        return image
```

Figur 9: Demosaicing Inpainting Funksjon

```
30    def demosaicing():
31        demo = np.zeros((mosaicdemo.shape[0], mosaicdemo.shape[1], 3))
32        demo[::2, ::2, 0] = mosaicdemo[::2, ::2]  # --> R-kanal
33        demo[1::2, ::2, 1] = mosaicdemo[1::2, ::2]  # --> G-kanal
34        demo[::2, 1::2, 1] = mosaicdemo[::2, 1::2]  # --> G-kanal
35        demo[1::2, 1::2, 2] = mosaicdemo[1::2, 1::2]  # --> B-kanal
36
37        mask = np.ones(demo.shape)
38        mask[::2, ::2, 0] = 0  #R-Kanal -> false
39        mask[1::2, ::2, 1] = 0  #G-Kanal -> false
40        mask[::2, 1::2, 1] = 0  #G-Kanal -> false
41        mask[1::2, 1::2, 2] = 0  #B-Kanal -> false
42        mask = mask.astype(bool)
43
44        data = plt.imshow(demo)
45
46        for i in range(50):
47            inpainting(demo[:, :, 0], mask[:, :, 0])  #R-kanal inpainting
48            inpainting(demo[:, :, 1], mask[:, :, 1])  #G-kanal inpainting
49            inpainting(demo[:, :, 2], mask[:, :, 2])  #B-kanal inpainting
50            data.set_array(demo)
51            plt.draw()
52            plt.pause(1)
```

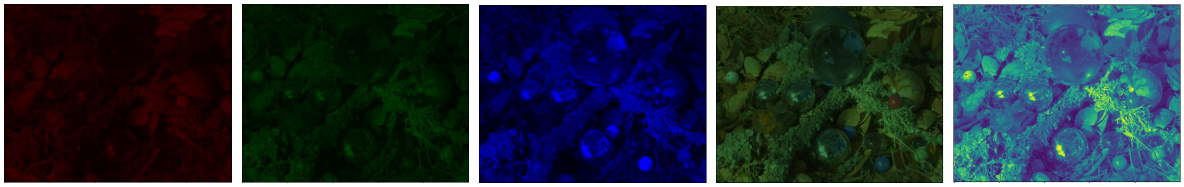
Figur 10: Demosaicing Funksjon

For å vise Demosaicing så ble det brukt 4 funksjoner, der en av funksjonene inneholdte 2 andre funksjoner “Nested funktions”. Dette fordi to av funksjonene trengte tillgang til felles variable som mosaic og orginale bilde. Siste funksjonen er det den som inpainter kanalene seinere (Figur 9). På (Figur 10) så har vi ned brekkingen av bilde til forskjellige kanaler fra linje 30-37. Rekonstruksjonen skjer i for løkken i linje 46-54 på (Figur 10).

6.2 Demosaicing Resultat

I Figur 11 Så ser vi at (d) er et resultat av kombineringen av bildene (a, b og c), men man får fortsatt ikke tilbake orginal bilde. Dette fordi en mengde data mangler, det er her

mosaic bildet (e) som kombineres med RGB bilde og vi får tilbake orginal bilde når alle bilde kanalene er satt sammen.



(a) Rød kanal (b) Grønn kanal (c) Blå kanal (d) RGB (e) Mosaic

Figur 11: Demosaicing prossess

Figur 12: Ferdig DemoSaicing

7 Sømløs

Sømløs Koding går ut på at et bilde blir satt inn på et annet bilde ved å passe på at kantene til bilde som blir satt inn ikke er skarpe og at de blir glattet ut.

7.1 Sømløs Metode

```
6   def somlos(image1, image2):
7       xlengde = 100
8       ylengde = 100
9       x0 = 100
10      y0 = 100
11      xn = 100
12      yn = 100
13      alpha = 0.25
14
15      image1 = image1.astype(float) / 255
16      image2 = image2.astype(float) / 255
17      omega = image2[xn:xn + xlengde, yn:yn + ylengde]
18
19      image3 = omega.view()
20      lap_image3 = Funksjoner.eksplisitt(image3)
21      for i in range(100):
22          lap_omega = Funksjoner.eksplisitt(omega)
23          omega[1:-1, 1:-1] += alpha * (lap_omega - lap_image3)
24          omega[:, 0] = image1[x0:x0 + xlengde, y0]
25          omega[:, -1] = image1[x0:x0 + xlengde, y0 + ylengde]
26          omega[0, :] = image1[x0, y0:y0 + ylengde]
27          omega[-1, :] = image1[x0 + xlengde, y0:y0 + ylengde]
28          omega[omega < 0] = 0
29          omega[omega > 1] = 1
30
31      image1[x0:x0 + xlengde, y0:y0 + ylengde] = omega
32      plt.imshow(image1)
33      plt.show()
```

Figur 13: Sømløs kode

Koden i sømløs koding er ganske lik inpainting der Dirchlets randbetingelse for å klone en liten del av bilde over til et annet bilde. Forskjellen mellom sømløs og inpainting er at bilde som blir overført ikke blir totalt overført. Bilde blir kopiert over i linje 31 i Figur 13 over før det bliur vist til brukeren.

7.2 Sømløs Resultat

Under på (Figur 14) forstørret del av en sømløs kloning av et bilde inn på et annet bilde. Som viser overgangen mellom de to bildene.

Figur 14: Sømløs Resultat

8 Anonymisering

8.1 Anonymisering Metode

```
4   def anonymiser(image):
5       alpha = 0.1
6       recolor = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
7       image = image.astype(float) / 255
8       cv2.imshow('Original', image)
9       cv2.waitKey(1000)
10      cv2.destroyAllWindows()
11
12      face = cv2.CascadeClassifier('haarcascade_frontalface_default.xml').detectMultiScale(recolor, 1.3, 6)
13
14      for (top, down, height, width) in face:
15          image = cv2.rectangle(image, (top, down), (top + width, down + height), (0, 0, 0), 0)
16          omega = image[down:down + height, top:top + width]
17
18      buffer = image.view()
19
20      for x in range(100):
21          omega[1:-1, 1:-1] += alpha * Funksjoner.eksplisitt(omega)
22          omega[:, 0] = buffer[top:(top + width), down]
23          omega[:, -1] = buffer[top:(top + width), down + height]
24          omega[0, :] = buffer[top, down:(down + height)]
25          omega[-1, :] = buffer[(top + width), down:(down + height)]
26
27      cv2.imshow('Anon', image) #viser bilde
28      cv2.waitKey(10000)
29      cv2.destroyAllWindows()
```

Figur 15: Anonymisering kode

For å løse oppgaven av anonymisering av ansikt krevde at programmet kunne identifisere hvor ansiktet på bilde er, for dette ble biblioteket CV2 i linje 12 i Figur 15. CV2 ble brukt for

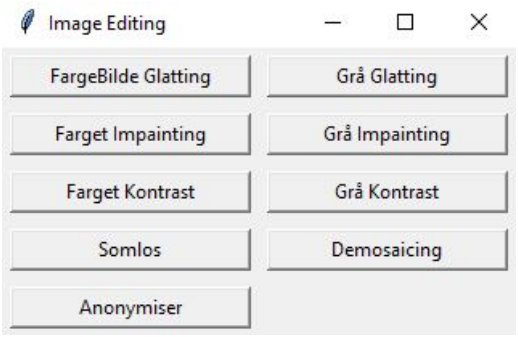
å lage en firkantet omega område som videre ble brukt for å bruke Direcklet randbetingelse for å glatte.

8.2 Anonymisering Resultat

(a) Orginal bilde (b) Anonymisert

Figur 16: Anonymisering

8.3 GUI



Figur 17: GUI

Graphical user interface(GUI) som ble laget for dette prosjektet brukte et bibliotek som heter “TKinter”. Eneste hensikten med GUI er at brukeren kan velge hvilken funksjonalitet

de vil kjøre også velge ønskede bilde de vil kjøre programmet på. Det ble ingen gui funksjon for å styre Variablene brukt i forskjellige teknikker, dette medfører til at brukeren må selv gå inn for å endre på variablene i koden om ønsket.

9 Diskusjon

Hoved problemet med utførelsen av prosjektet var mangel av tid, selv om bachelor oppgaven ikke er direkte koblet til dette prosjektet så var det vanskelig å balansere arbeids mengden mellom dette prosjektet og bachelor prosjektet. Tidsklemmen gjorde det vanskelig å søke opp matriell på nett. Noe som hadde vært morsomt å nevne er at jeg ikke kunne se forskjellen mellom RGB og Green-channel på grunn av synet mitt brukte en del tid helt til jeg ble fortalt at de så helt forskjellige.

10 Konklusjon

I dette prosjektet gikk vi gjennom forskjellige algoritmer av Poisson Image Editing. Det mange ting som kunne blitt gjort annerledes. For det første er det gjenbruk av kode. Noe annet som kunne blitt gjort anderledes er digital planlegging på GITLAB, dette skjedde fordi utvikling aleine føltes det som ekstra arbeid som var vanskelig å holde seg til selv om man mister en del dokumentasjon på arbeidet. Til slutt så hadde jeg lyst til å implementere mer gui funksjomnaliteter der brukeren selv kan inputte variabel verdi enn å gå inn i programmeringen.

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.

Figurer

1 Glatting kode 2
2 Glatting Resultat 3
3 Inpainting kode 3
4 Inpainting resultat 4
5 Kontrast kode 5
6 Kontrast Orginal 6
7 Farget Kontrast 6
8 Grå Kontrast 6
9 Demosaicing Inpainting Funksjon 7
10 Demosaicing Funksjon 7
11 Demosaicing prossess 8
12 Ferdig DemoSaicing 8
13 Sømløs kode 9
14 Sømløs Resultat 10
15 Anonymisering kode 10
16 Anonymisering 11
17 GUI 11

Listings