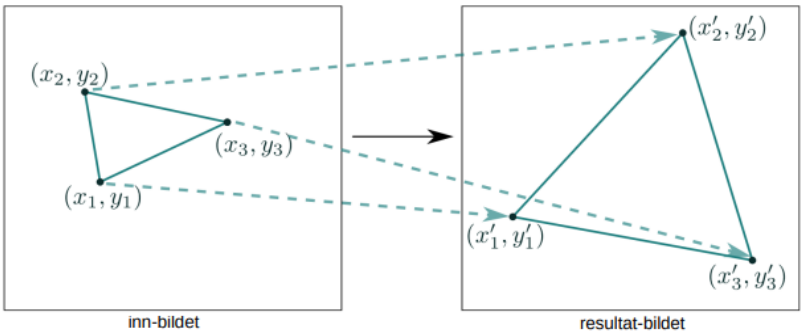
Preprosessering av portrett for ansiktsgjenkjenning

1. Mellom-resultat-bildet etter gråtonetransformen

Resultatbildet er vedlagt som ‘portrett_standardisert.png’.

2. Forklaring på hvordan jeg fant koeffisientene til den affine transformen

Jeg fant koeffisientene ved å spesifisere tre (x,y)-koordinater fra ‘portrett.png’, henholdsvis venstre øye, høyre øye og munn. Deretter spesifiserte jeg hvor jeg ønsket at disse punktene skulle være i ‘geometrimaske.png’ etter transformasjonen.



For å finne de 6 ønskede koeffisientene, a0, a1, a2, b0, b1, b2 løste jeg følgende lineære system, på formen Ax = b. Punktparene er kjent, som betyr at vi kan finne de 6 ukjente.

$$Ax = b$$

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} x_1' \\ y_1' \\ x_2' \\ y_2' \\ x_3' \\ y_3' \end{bmatrix}$$

Det ga meg mulighet til å utføre både forlengs- og baklengstransformasjoner. For å utføre baklengstransformasjoner brukte inverse koeffisientmatrisen og matrisemultipliserte med x’ og y’ for å finne pikselverdier i originalbildet.

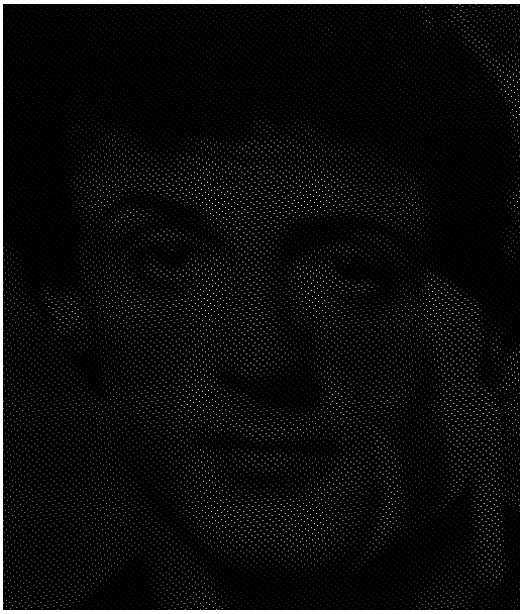
Forlengstransform

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 \\ b_0 & b_1 & b_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Baklengstransform

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 \\ b_0 & b_1 & b_2 \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

3. Kommentar på forskjeller til de ulike transformasjonene



Forlengs-mapping:
Ved forlengs-mapping så regnes (x', y')-koordinatet i resultatbildet ut basert på (x,y) fra originalbildet, gjennom en affin transformasjon.
Dersom (x',y') er innenfor piksel-området avgrenset av NxM, vil piksel-intensiteten i (x',y') settes lik piksel-intensiteten i (x,y) fra originalbildet. En slik mapping vil dermed kunne gi et resultat der ikke alle piksler i resultatbildet er blitt gitt en piksel-intensitet.
Det er tilfellet her, og man kan se at resultatbildet mangler en god del piksel-intensiteter..

Nærmeste-nabo:
Denne metoden bruker baklengs-mapping og en enkel form for "approksimasjon" av piksel-intensitet. Algoritmen finner piksel-intensiteter basert på (x',y')-koordinater. Det gjøres ved at vi regner den inverse affine transformasjonen, det vil si at alle (x',y')-koordinater i resultatbildet blir invers transformert til (x,y) koordinater i originalbildet.

Disse koordinatene er ikke heltall, og vi må derfor hente piksel-intensiteten fra den nærmeste naboen til (x,y). Dette gjøres enkelt ved å runde (x,y).
Det betyr at alle piksler i resultatbildet få en piksel-intensitet, men siden intensiteten blir bestemt av nærmeste nabo vil piksel-skiller kunne oppfattes "kornete".

Bilinær interpolasjon:

I denne algoritmen beregnes pikselintensiteten basert på interpolasjon av de 4 nærmeste pikslene i en inverstransformasjon. Dette betyr at vi får en glattere overgang når pikselintensiteter endres, sammenlignet med Nærmeste-nabo

4. Programkode

Koden for å kjøre oppgave 1 heter oppgave1.py. For å kjøre programmet skriver man

```
$ python3 oppgave1.py
```

Først vil poppe opp visualiseringen for oppgave 1.1, som er det standardiserte bildet. For å fortsette til oppgave 1.2 må disse vinduene lukkes. Bilde fra oppgave 1.1 lagres i mappen som “portrett_standardisert.png”.

Deretter vil programmet utføre de ulike affine transformasjonene. Disse vil også poppe opp og deretter lagres når vinduene lukkes. De ulike bildene heter “forlengsmapping.png”, “nærmeste-nabo.png” og “bilinær-interpolasjon”. Det er forøvrig disse bildene som er limt inn i forklaringen ovenfor.

Detektering av cellekjerner

1. Tekstlig beskrivelse av implementeringene i a) og b)

a)

average_kernel(n)

Jeg valgte å bruke et gjennomsnitt filter i denne oppgaven. For å lage det oppretter jeg en matrise fylt med 1'ere som deretter deles på filterstørrelsen. Dette gjøres i "average_kernel(n)" metoden. Denne brukes videre som argument i convolution() metoden

padding(img, kernel_size, zeros)

Denne funksjonen utvider originalbildet med kolonner og rader tilsvarende k. K regnes ut basert på størrelsen til filteret. Dersom man ønsker en bilderand med bare 0-ere så returneres bildet med en gang. Men oppgaven ba om nærmeste-nabo, det løses med å bruke arrayindeksering. Gjennom arrayindekseringen settes bilderanden til nærmeste nabo.

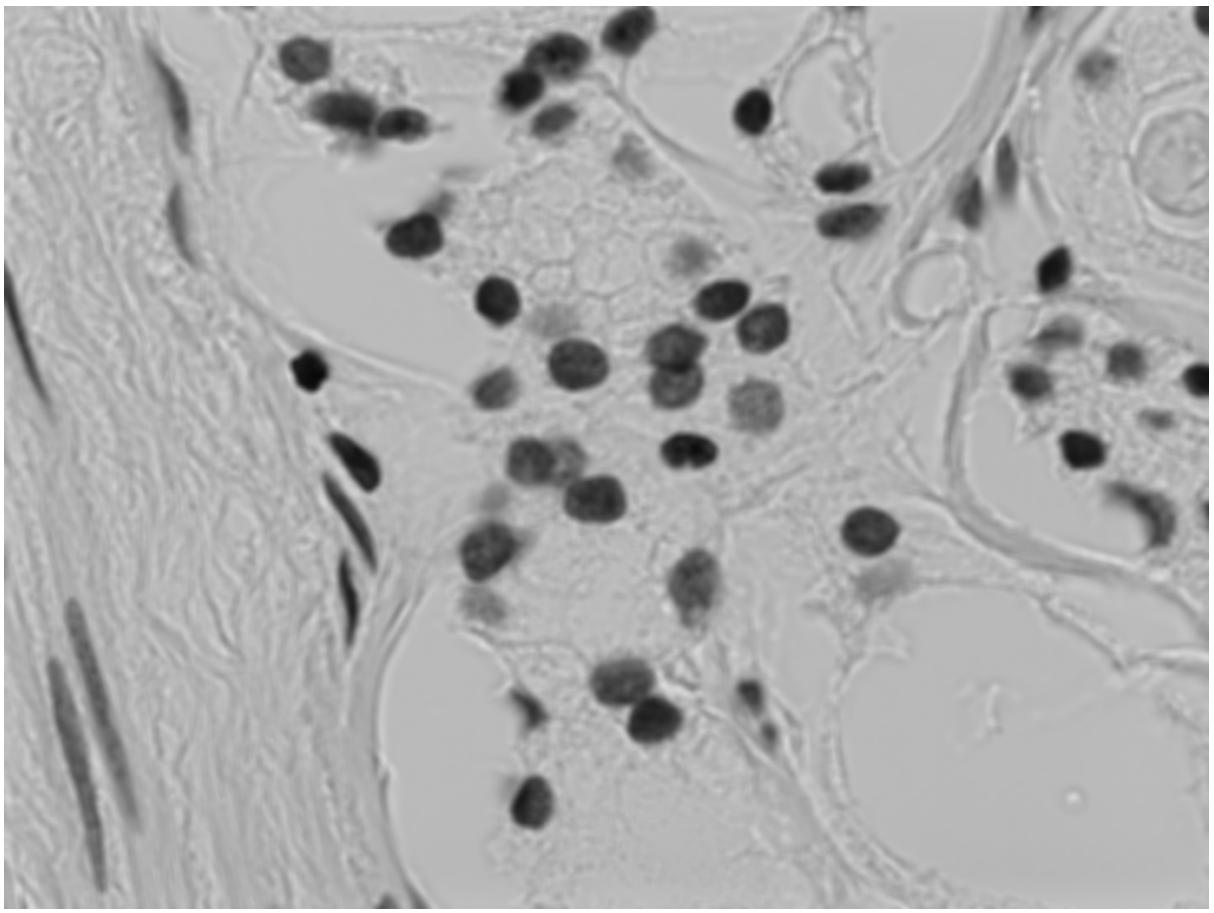
convolution(img,kernel,zeroes,conv_1D)

For å oppnå kravet at utbildet skal være samme dimensjon som innbildet opprettet jeg en padding funksjon, som utvider bildet iht størrelsen til filteret vi skal konvolvere med, denne funksjonen blir forklart under. Videre har man mulighet for å kjøre konvolusjon med den symmetriske 1D-operatoren vha separasjon av f.eks sobel-operatoren, men det blir ikke brukt i denne delen av oppgaven. Denne metoden er generell og kan brukes for alle videre filter i obligen.

Filteret som brukes blir rotert 180 grader. Deretter kommer det en nøstet for-løkke som sjekker alle piksel i originalbildet. For hver piksel lages det en matrise av pikslene i originalbilde, i samme størrelsen som filteret. Origo i denne matrisen tilsvarer senterpikselen de resterende pikslene er dens naboer. For hver iterasjon utfører vi en elementvis matrisemultiplikasjon av filteret og senterpikselen med naboer. Disse multiplikasjonene summeres til en verdi. Denne verdien blir dermed den nye pikselintensiteten i det filtrerte bildet.

oppgave2_1(filename, n)

Denne metoden strukturerer selve oppgave a). Først leses bilde inn som et gråskalertbilde. Etter det utføres det et kall på "average_kernel(n)" som returnerer et gjennomsnittsfilter med ønsket størrelse. Videre sendes dette filteret inn i "convolution(img,average)" som returnerer det filtrerte bildet med samme dimensjoner som originalbildet. Til slutt lagres filtrerte bildet lagres som 'cellekjerner_blur.png'. Har vedlagt det her også, i dette bildet ble det brukt et 9x9 gjennomsnittsfilter.



b)

gaussian_kernel(sigma)
Denne metoden returnerer et gaussian filter der brukeren kan justere sigma selv. Siden vi ønsker at senterpikselen skal være i origi av filteret må man starte iterasjonen i negativ "retning". Dette løses ved å heltallsdele filterstørrelsen på 2, slik at dersom filteret er et 3x3 filter, der x=0,y=0 er senter av filteret, tilsvarer x=-1,y=-1 pikselposisjonen oppe til venstre og x=1,y=2 er pikselposisjonen nede til høyre. Jeg hadde problemer med å regne A slik som forelesningsnotatet oppga, derfor definerte jeg A slik som det ble gjort i oblig 2017.

sobel(img, sigma, zeros, conv_1D)
I denne metoden blir defineres sobel-operatorene for hx og hy. Videre finner vi gradientene for vertikal og horisontal retning, Gy og Gx. Sobel-operatorene blir separert i selve "convolution()" metoden, med at vi sender inn "conv_1D=True" som argument. Separeringen skjer ved arrayindeksering, der vi også roterer 1D-operatorene før konvolusjonen. Ved å bruke 1D-operatorene utfører vi 6 operasjoner for hver piksel istedenfor 9. Etter Gy og Gx er kalkulert beregnes selve gradientmagituden til innbildet med å bruke formelen oppgitt i forlesningsnotatet. Deretter finner jeg de tilhørende gradientvinklene (i radianer) som blir konvertert og rundet til nærmeste 45 graders multiplum/vinkel, der 0 grader tilsvarer en horisontalkant, 45 og 135 en skråkant og 90 en vertikalkant. Videre skaleres gradientmagituden til gråskala. Til slutt returneres gradientmagnituden og gradientvinklene til bildet.

thinning_edges(gradient,theta_rounded)
I denne metoden finner man først ut i hvilken retning pikselen vi ser på er rettet. Deretter sjekkes det om de tilhørende 8-nabopikslene i samme retning har en høyere magitude. Dersom det er tilfellet endres senterpikselen til 0, dersom det ikke er tilfellet beholdes pikselintensiteten. Dette gjøres for alle pikselintensiteter i gradientmagnitude bildet. Ved å utføre en slik endring vil kantene som vi fant i sobel() bli tynnere. Det som gjenstår nå er å fjerne de pikselintensitene som ikke lenger er relevante.

hysterese_8connectivity(strong,weak,thin_gradient)
Først splittes gradientmagnitudebildet inn i 3 deler; piksler som ikke skal tas med videre (discard), piksler som muligens skal tas med videre (weak) og piksler som definitivt skal være med videre (strong). "strong" piksler må være større eller lik den brukersatte grensen (strong) er pikselintensiteter. Alle disse piksel lagres i et array med verdien 255 (maks intensitet).

Deretter sjekker man for piksler som er under den brukersatte grensen til "weak"-piksler. Disse lagres i et array som heter discard, men trengs i grunnen ikke bli lagret.

Alle andre pikselverdier faller innenfor et område av pikselintensiteter som vi må sjekke om skal være med i sluttbildet. Pikselintensiteteten til disse pikslene er høyere enn weak-grensen, men lavere enn strong-grensen.

For alle weak-piksler ønsker vi å identifisere om dens 8-naboer tilhører en strong-piksel. Dersom det er tilfellet ønsker vi at denne weak-pikselen skal inkluderes i sluttbildet. Det

gjøres ved å løpe over alle pikslene i weak-arrayet som er større enn 0. Derettes sjekkes dens 8 naboer, og dersom en av naboene har en verdi > 0 betyr det at der en strong-piksel. Jeg valgte å lagre weak-pikslermed verdien 25.

Etter denne metoden har vi et gradientmagnitude bilde bestående av kun 3 ulike pikselverdier, 0, 25 og 255.

oppgave2_2(filname, sigma, threshold_H, threshold_L)
Først finner jeg gradientmagnitudene og gradientvinklene ved å kalle på sobel. Her har jeg muligheten til å justere på klarheten i bildet jeg skal analysere ved å endre på sigma (standardavviket til gaussian filteret). Etter bildet er justert blir kantene tynnet. Til slutt fjerner vi piksel vi ikke lenger mener er nødvendige for oppgaven. Pikslene som blir valg bort/tatt med er bestemt av grensene satt, henholdvis threshold_H og threshold_L.

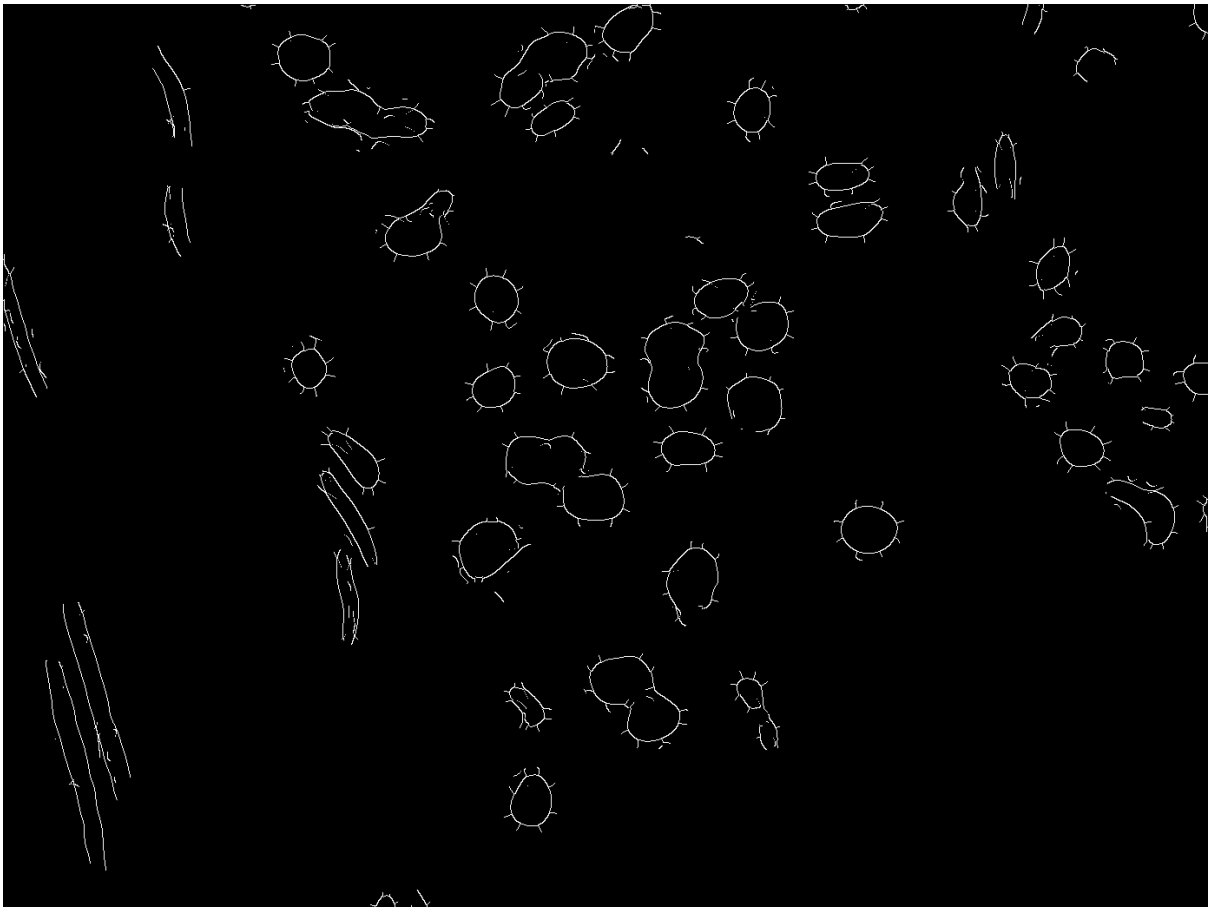
Disse funksjonene bruker en del til på å kjøre, forventet kjøretid er ca 60s. Resultatet blir lagret som “canny_high[threshold_H]_low[threshold_L]_sigma[sigma].png”

2. Resultatbilde med tilhørende parameterverdier

Jeg utførte flere tester med ulike parametere. Den jeg synes var best hadde følgende parametre:

- Threshold high = 80
- Threshold low = 30
- Sigma = 7

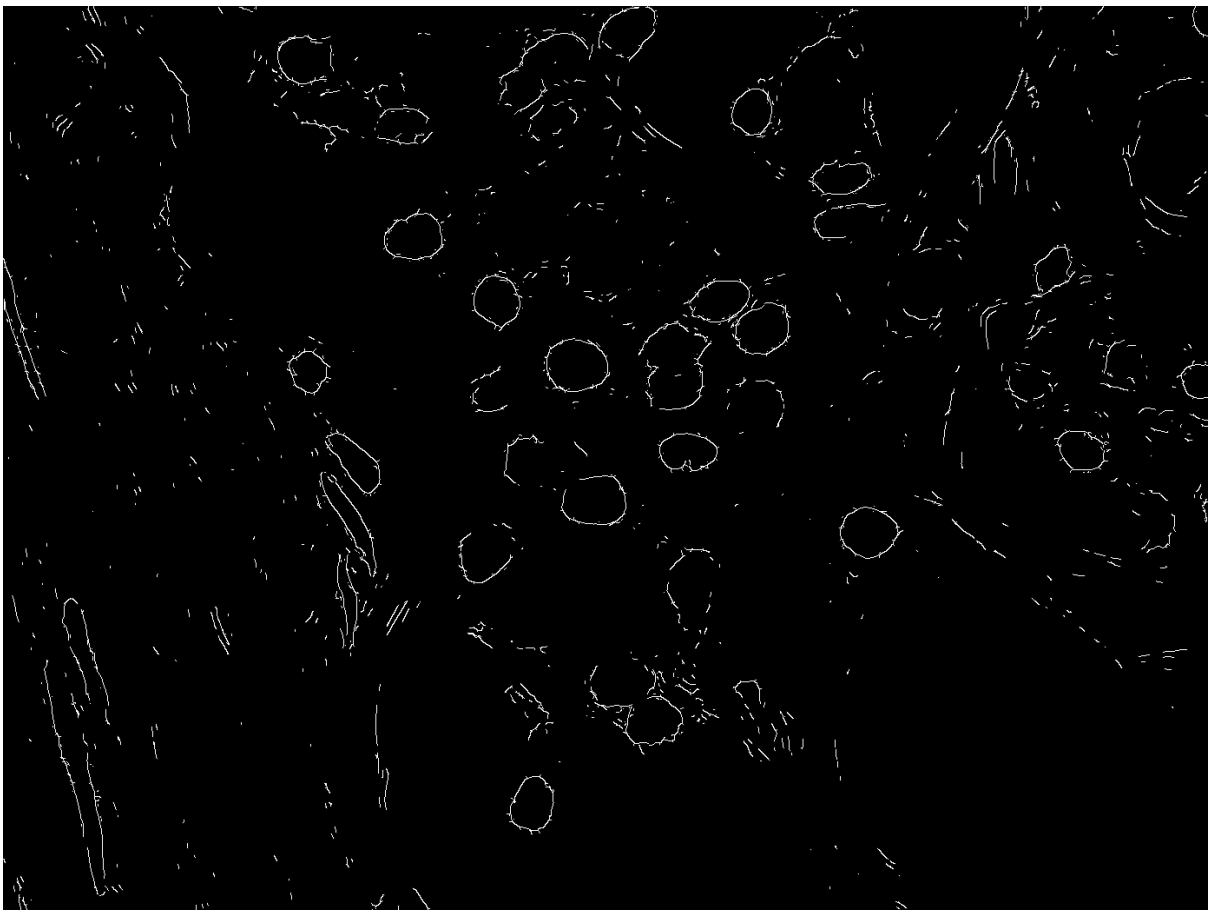
Bilder er lagret i obligmappen med også vedlagt under.



3. Drøfte resultatbildet

Jeg ble relativt fornøyd med sluttresultatet. Det jeg ikke helt forstod var hvorfor kantene fikk små streker som strakk seg ut. Det jeg observerte var at dersom jeg økte sigmaen, som igjen økte glattingen av orignalbildet så ble disse lengre. Samtidig ga en økning av sigma mindre støy i bilde og dermed klarere kanter.

I starten hadde jeg lave sigma verdier og fikk følgende resultat:



Det som gjør at canny er en god metode er fordi vi lett kan fjernes støy fra bildet ved å bruke et gaussian filter med høyere standardavvik. Videre tynner vi kantene i forhold til f.eks sobel, som lager et bedre bilde av hvor kantene er.

Samtidig kan det å redusere støy virke negativt på filtreringen, fordi dersom man glatter bilde for mye er det fare for å minste kanter, som igjen gjør at vi ikke får en riktig illustrasjon av kantene etter canny's filtreringen.

4. Programkode

For å kjøre koden til oppgave2 skriver man følgende i terminalen:

```
$ python3 oppgave2.py
```

Da kjøres metodene oppgave2_1 og oppgave2_2. Etter de er kjørt lagres tilhørende bilder i programmappen. Dersom det er ønskelig å endre på innbilde, sigma, grenseverdier eller filterstørrelse kan dette gjøres i main() metoden.