STJERNEOPPGAVE 1

Tema: bildebehandling med prosedyre-basert programmering

Resultat av prosessering av et bilde (fra venstre: originalt bilde, enkel blur, sharp, smart blur).

I denne oppgaven skal det implementeres et program som laster inn et bilde til minne og gjør ulike prosesseringer på det. Til slutt, skal de resulterende bildene skrives til fil.

Forkunnskapskrav: typer, kontrollstrukturer, prosedyre-basert programmering, tabeller og pekere.

Til støtte for denne oppgaven har det blitt utviklet en støtteklasse for dere. Denne klassen følger stilen prosedyre-basert programmering uten bruk av objekter. Det vil si at den tilbyr en rekke prosedyrer som dere kan bruke. I tillegg, lagrer den et par globale variabler som gjør bruken av prosedyrene noe enklere.

Les dokumentasjonen for klassen Imaging (Imaging.html). Se over de ulike prosedyrene (method summary) og forstå hva som kreves av input data og hva prosedyrene returnerer.

Opprett en java-fil med main i mappen der jar-filen (imaging.jar) ligger. Du kan teste om biblioteket fungerer ved å laste inn et bilde, for deretter å lagre filen til filsystemet:

```
import programmering.*; // importer biblioteket vårt

public class MyImagingProgram {

	public static void main(String[] args) {

		if(args.length < 1)
			System.err.println("Please provide an image path");

		int[] img = Imaging.readImageFromFile(args[0]);
		if(img == null) // fikk ikke lastet inn fil
			System.exit(-1);

		Imaging.writeImageToFile(img, "output.jpg", "jpg");

	}

}
```

Du må inkludere jar-filen i kompileringen for å korrekt kompilere programmet. Følgende konsoll-kommandoer oppnår dette:

```
javac –cp ".;imaging.jar;" MyImagingProgram.java
java –cp ".;imaging.jar;" MyImagingProgram "Lenna.png"
```

Etter kjøring skal det ha blitt opprettet en ny fil (output.jpg) i mappen programmet ditt ligger i. Du har nå implementert et enkelt program som konverterer en bildefil til et bilde av formatet "jpeg".

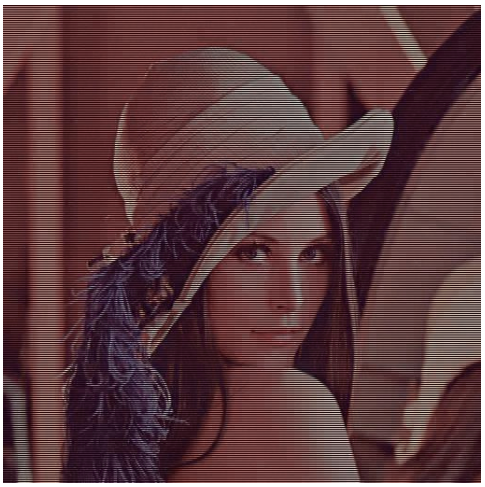
Biblioteket imaging.jar er skrevet i Java. Du trenger ikke å kjenne til implementasjonen til prosedyrene. Dette er en av fordelene med prosedyre-basert programmering. Du må bare kjenne til hva som kreves av parametere og hva prosedyrene returnerer, men du må ikke kjenne til implementasjonsdetaljer. Hvis du imidlertid er interessert i hvordan Imaging klassen er implementert, er kildekoden vedlagt i jar-filen. Denne kildekoden kan leses ved å åpne jar-filen i et program som støtter lesing av zip-filer (for eksempel 7-zip eller keka).

Kom i gang!

Et bilde består konseptuelt av et todimensjonalt nett, der hvert punkt i nettet representerer en pixel (picture element). En pixel består av tre heltallsverdier mellom 0 og 255. Den første verdien representerer bidraget til fargen rød, den andre verdien representerer grønn, og den tredje verdien representerer blå. Selv om dette er alt du trenger å vite, kan du gjerne lese om representasjonen av bilder her: https://en.wikipedia.org/wiki/Raster_graphics

Datastrukturen du skal arbeide med representerer bilder som en enkel tabell med int-verdier. Tabellen inneholder alle pixlene til bildet og hver heltallsverdi representerer en kodet verdi av fargen.

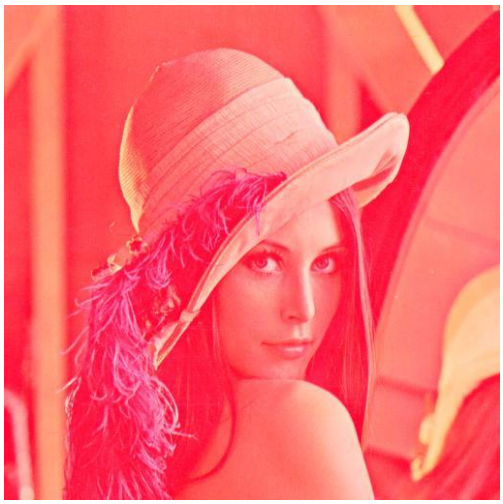
For å bli kjent med dette formatet, kan du iterere igjennom tabellen og angi hvert andre element verdien 0 (svart). Korrekt utførelse av denne operasjonen vil produsere følgende bildet:



Imaging-klassen tilbyr prosedyrer for å få tak i RGB (Red-Green-Blue) verdiene til hver heltallsverdi og konvertering av tabellindekser til todimensjonalt punkter. I tillegg, lagrer Imaging klassen en global variabel Imaging.currentImageWidth. Etter at en fil av blitt lastet korrekt inn, vil denne variabelen inneholde bredden til det innlastede bildet. Prosedyren Imaging.getImgHeight kan brukes til å få tak i høyden for bildet. Følgende kode itererer, ved hjelp av en nøstet for-løkke, igjennom alle pixlene til bildet og henter ut RGB-verdien til hver pixel. RGB verdien er representert som en heltallstabell med tre verdier.

```
for(int x = 0; x < Imaging.currentImageWidth; x++) {
	for(int y = 0; y < Imaging.getImgHeight(img); y++) {
		int index = Imaging.convert2DTo1D(x, y);
		int[] RGB = Imaging.getRGBArray(img[index]);
	}
}
```

For å teste om oppsettet over fungerer, sett rød-fargen til maksimal verdi (255), konverter RGB tabellen tilbake til et heltall, og overskriv nåværende farge i img-tabellen med den nye fargen. Du kan konvertere RGB tabellen til et heltall ved hjelp av Imaging.getIntColour prosedyren. Korrekt utførelse av dette steget vil produsere følgende resultat:



Filter 1: enkel blur (box blur)

Du skal nå implementere et *filter*, en operasjon innen bildebehandling som prosesserer til gitt bilde og produserer et nytt (ofte forbedret) bilde. Filteret du skal implementere utfører en *blur* operasjon, en filtreringsoperasjon som ofte brukes for å redusere støy i et bilde (støy er ofte et problem i eldre, spesielt skannede, bilder).

Før du går videre, kan det være hensiktsmessig å implementere en prosedyre som kopierer heltalls-tabeller. En slik kopieringsoperasjon er nødvendig for å produsere to bilder i programmet ditt: det originale bildet og det produserte (filtrerte) bildet. Merk at en slik prosedyre implementeres i programmeringsoppgave 2, oppgavesett 5.

Box blur er en filtreringsoperasjon som erstatter fargen til en pixel med fargegjennomsnittet til de ni pixlene rundt pixelen (inkludert pixelen selv). Implementer en prosedyre som utfører en slik box blur operasjon.

Tips:

- Pixlene på kantene (for eksempel koordinater (0,0), (0,1), (1,0)) har ikke åtte nabopixler. Det er flere løsninger på dette problemet. En enkel løsning er å ekskludere pixlene på kantene i kalkuleringen (dvs. starte på indeks 1 istedenfor 0 i for-løkkene).
- Gjennomsnittet skal kalkuleres separat for hver RGB fargekanal. For hver pixel, er det naturlig å opprette en heltallstabell med tre verdier og akkumulere RGB verdiene til alle nabopixlene (8 pixler) til nåværende pixel. For kanalen rød, skal dermed summen av alle rød-verdiene til alle nabopixler kalkuleres og legges til rødfargen til nåværende pixel. Endelig farge er dermed denne summen delt på 9.

- Unngå duplikat kode. Hvis du kopierer en kodesekvens, så vurder om kodesekvensen kan flyttes til en egen prosedyre.
- Angående lesbarhet og høy kvalitet på kode, så er en tommelfingerregel å bruke maksimalt to nøstede (til nød tre) konstruksjoner i en prosedyre. For eksempel, tre til fire nøstede for-løkker i en prosedyre fører nesten alltid til kode som er svært vanskelig å forstå for andre programmerere. Hvis du befinner deg i en slik situasjon, er det naturlig å vurdere å opprette en hjelpeprosedyre.

Merk at iterativ filtrering av box blur på et bilde er en tilnærmelse på det mer sofistikerte Gaussian blur filteret. Implementer en prosedyre som iterativt utfører box blur på et bildet et gitt antall ganger. Utfør operasjonen på bildet "noise.png" og finn, ved å eksperimentere, den beste innstillingen for å fjerne støy fra bildet.

Filter 2: skarpere bilder (unsharp mask)

En annen typisk operasjon innen bildebehandling er å gjøre bildet skarpere. *Unsharp mask* er et filter som, i motsetning til det navnet tilsier, gjør bilder skarpere ved å elegant mikse det originale bildet med et bildet filtrert med et blur filter (som box eller Gaussian blur).

Unsharp mask er definert slik:

$$S = O + (O - B) * a,$$

der S er det produserte skarpere bildet, O er det originale bildet, B er den filtrerte versjonen av O (med box eller Gaussian blur), og a er en variabel som varierer skarpheten til S. Standard verdi til a er 2.0. Lavere verdi gjør bildet mindre skarpt, mens høyere verdi gjør bildet skarpere.

Lag en prosedyre som utfører unsharp mask operasjonen. Hvis du ikke har implementert kopiering av heltallstabeller, må du nå gjøre dette for å kunne lagre de to bildene O og B i hovedminne.

Hint:

- Som med kalkulering av gjennomsnitt, skal regneoperasjonene utføres separat for hver RGB kanal. For hver pixel, må du hente RGB tabellene (fra O og B) fra heltallsverdiene for pixelen, utføre regneoperasjonen, og konvertere den resulterende RGB tabellen tilbake til en heltallsverdi som plasseres i S.
- Regneoperasjonen kan føre til av fargeverdien går under 0 eller over 255. Dette kan oppstå for mørkere eller lysere deler av bildet. Konverteringsfunksjonen getIntColour tillater ikke ugyldige fargeverdier og programmet krasjer hvis fargeverdier ikke er mellom 0 og 255. Implementer derfor en funksjon som setter alle negative heltalsverdier til 0 og alle verdier over 255 til 255. Denne operasjonen kalles for colour clamping eller clamping.

Filter 3: smart blur (bilateral filter)

Både box og Gaussian blur oppnår støyreduksjon, men samtidig gjør kanter mer utydelige. Grunnen til dette er at disse filtrene er konstante over hele bildet (de kalles for *lineære filtre*). Du skal nå implementere et filter som tar hensyn til kanter (et såkalt ikke-lineært filter). Bilateral filter oppnår dette ved å variere sitt filter for hver pixel ved å ta hensyn til fargeverdier og distanse til nabopixler når man kalkulerer gjennomsnittet. Dette filteret kalkulerer dermed ikke et matematisk gjennomsnitt av naboområdet til pixelen, men kalkulerer istedenfor et mer representativt gjennomsnitt relativt til den originale fargen til pixelen. Farger som likner mer på pixelen blir vektet høyere enn farger som er forskjellige.

For å kunne implementere dette filteret trenger vi to matematiske funksjoner: Gauss funksjonen og Euklidsk lengde (begge funksjonene oppkalt etter to rimelige kjente matematikere).

Gauss funksjonen gir høye verdier desto nærmere man er et "sentrum" (i vårt tilfelle $x = 0$). Den implementeres i Java slik:

```
public static double GaussianFunction(double x, double variance) {
	return Math.exp(-Math.pow(x, 2.0)/(2.0*variance));
}
```

Euklidsk lengde gir distansen langs en rett linje mellom to punkt plassert i et Eulidsk rom (noe pixler er plassert i). Java implementasjon (tabellene a og b må være av samme lengde):

```
public static double EuclideanDistance(int[] a, int[] b) {
	double sum = 0.0;
	for(int i = 0; i < a.length; i++) {
		sum += Math.pow(a[i]-b[i], 2.0);
	}
	return Math.sqrt(sum);
}
```

Med disse funksjonene kan du implementere bilateral filter, som er definert slik for hver pixel x:

$$F(x) = \frac{1}{W} \sum_{x_i \in N} O(x_i) w_f\left(d\big(O(x_i), O(x)\big)\right) w_p\big(d(x_i, x)\big),$$

der W er en normaliseringsfaktor for å påse at de produserte verdiene er gyldige fargeverdier:

$$W = \sum_{x_i \in N} w_f\left(d\big(O(x_i), O(x)\big)\right) w_p\big(d(x_i, x)\big)$$

De andre faktorene er:

- N er et definert naboområdet rundt pixelen x. Du kan starte med de nærmeste 8 pixlene, som med box filter, men det anbefales at du utvider implementasjonen til å inkludere virkårlig størrelse av naboområdet (nabostørrelse burde lagres som en variabel). For eksempel, vil et såkalt "2-ring" området inkludere to "steg" fra x (5*5 pixler), mens et 3-ring området inkluderer 49 (7*7) pixler. Pixelen x_i representerer den nåværende nabopixelen i iterasjonen av pixlene i N. **Det vil si**: for hver pixel x, må du iterere gjennom alle nabopixlene til x (inkludert selve pixelen x) og kalkulere det som står inne i summeringstegnene.
- $O(x)$ er RGB fargen til pixel x i det originale bildet O. **Husk**: RGB fargen, ikke det kodete heltallet.
- Faktorene w_f og w_p er vekten for hver nabopixel. Faktoren w_f representerer likhet i farge mellom x og x_i, mens faktoren w_p representerer distansen mellom pixlene x og x_i. Funksjonen d er den Euklidske lengden, mens funksjonen w (dvs. både w_f og w_p) er Gauss funksjonen.

Merknader:

- Vektene w_f og w_p er felles for de to utrykkene over (for $F(x)$ og W). I kode, kan dette tas hensyn til ved å kalkulere og summere vektene separat fra multiplikasjonen mellom vektene og den originale fargen $O(x_i)$. Kort sagt: man trenger ikke å iterere naboområdet for en pixel x to ganger.

- RGB fargene er representert ved int-heltall. Kalkuleringen burde for nøyaktighetens skyld utføres som flyttall (double anbefales). Dette burde gå greit, da den Euklidske funksjonen returnerer en double-verdi, men det er noe du allikevel burde være klar over i implementasjonen. Du konverterer bare tilbake til int etter at kalkuleringene er ferdige (sannsynligvis etter divisjon på W).
- Gauss funksjonen brukes to ganger: en gang for å kalkulere likhet mellom farger (w_f) og en gang for lengde (w_p). Parameteren variance (varians) brukes for å kontrollere likhetsverdiene som produseres av funksjonen og er relativ til verdiområdet til parameteren til x. For eksempel, er farger definert mellom 0 og 255, mens lengde mellom nabopixler ligger mellom 0 og ca. 20. Høyere varians vil resultere i mindre forskjeller mellom to ulike punkter (mer blur). Du burde opprette to konstante variabler for varians og teste ulike verdier. Standardverdier kan være 100 for variansen til w_f og 10 for variansen til w_p.
- Kalkuleringen inneholder en rekke matematiske operasjoner på tabeller, som multiplikasjon, divisjon, og summering/akkumulering. Det anbefales på det sterkeste at du implementerer hjelpefunksjoner som utfører disse operasjonene. Ellers vil antall kodelinjer blåses opp, du vil lage kode som er vanskelig å lese, og du vil sannsynligvis bruke lengre tid på å løse oppgaven.
- Husk at $O(x_i)w_f\left(d\big(O(x_i),O(x)\big)\right)$ er en forkortelse for $O(x_i) * w_f\left(d\big(O(x_i),O(x)\big)\right)$, der $*$ er gangetegnet.

Test implementasjonen på en rekke bilder. Støyreduksjon burde med bilateral filter være bedre enn box filter, da kanter i bildet vedlikeholdes.

Innlevering

Inkluder bilderesultater i din innlevering, samt Java kildekode. Lag gjerne et dokument med kommentarer. Pakk inn filene i en ZIP fil og last filen opp til OneDrive. Lever oppgaven ved å sende en epost til henrik.lieng@hioa.no med lenke til OneDrive filen (senest en uke før semesterslutt). Besvarelsen blir da sett over av en studentassistent og du vil få nyttige og konstruktive tilbakemeldinger.