Obligatorisk innlevering 3, høsten 2014, INF3331

Kristoffer Brabrand <kristrek@student.matnat.uio.no>

November 13, 2014

Innledende beskrivelse

Oppgaven var grei nok å løse når det kom til prinsipper og bruk av python, men matematikken bød på noen utfordringer. Disse var primært knyttet til omregning mellom de ulike fargerommene, avrunding, grenseverdier mv.

De beskrevne oppgavene er løst, men jeg oppdaget for sent at denoise.c opererer på floats og konverterer til en unsigned char (0-255). Ettersom jeg opererer på heltall og runder av underveis har jeg noe større avvik i oppgaven. Dette er beskrevet under den aktuelle oppgaven.

Frontend for de ulike backendene er implementert i denoise.py i roten av oblig3-mappen, og er relativt enkel. Den gjør i korte trekk følgende;

1. Parsing av kommandolinjeparametre.
2. Validerer input.
3. Henter riktig backend ved bruk av den lokale symboltabellen.
4. Kaller `denoise_file` på den valgte backenden;

(a) Laster fil inn i liste/numpy.ndarray.

(b) Utfører denoising og/eller manipulasjon av bildet.

(c) Skriver bearbeidet fil til målfil.

Parametrene denoise.py tar vises enklest ved å kalle den med -h som parameter. Outputen som gis når man kaller `denoise.py -h` er vist nedenfor.

Terminal

```
$ python denoise.py -h
usage: denoise.py [-h] [--backend B] [--denoise D] [--kappa K] [--iter I]
                  [--verbose] [--lr N] [--lg N] [--lb N] [--lh N] [--ls N]
                  [--li N]
                  src dst

Image denoiser with image component manipulation support.

positional arguments:
  src            Path to source image
  dst            Destination for output image

optional arguments:
  -h, --help     show this help message and exit
  --backend B    What backend to use
  --denoise D    Perform denoising of image
  --kappa K      Kappa value. Allowed range [0.0, 1.0]
  --iter I       Number of iterations to run with the denoiser.
  --verbose, -v  Enable verbose script output
  --lr N         Amount to add to or remove from the R channel (RGB).
  --lg N         Amount to add to or remove from the G channel (RGB).
  --lb N         Amount to add to or remove from the B channel (RGB).
  --lh N         Amount to add to or remove from the H component (HSI).
  --ls N         Amount to add to or remove from the S component (HSI).
  --li N         Amount to add to or remove from the I component (HSI).
```

Oppgave 1: Denoise i Python og Numpy+Weave

Python-implementasjonen er basert på algoritmen som fantes i denoise.c. Algoritmen gjør som den skal, men er (som forventet) ganske treg.

Numpy-weave-implementasjonen derimot er ganske mye raskere. Jeg har delt opp implementasjonen av C-koden i to biter og lagt disse ut i en separat python-fil for å gjøre `numpy_weave.py` litt ryddigere. Koden finnes i src/denoise/weave_c.py, og er delt i denoise_c og support_c, der den siste inneholder typedefs og funksjoner som brukes i denoising og manipulering.

I oppgaveteksten var det nevnt at man skulle bruke timeit for å sammenligne de ulike backendene. Det sto ikke eksplisitt hvor det skulle gjøres, men ettersom flere backends skjeldent kjøres samtidig og man derfor ikke enkelt vil kunne få en sammenligning ved å kjøre det gjennom frontend implementerte jeg en sammenligningstest i `test_speed_test.py`. Testen sammenligner bare svart-hvit-denoising siden dette er den logikken som finnes i alle tre backendene.

Resultat fra kjøring

Terminal

```
$ python test_speed_test.py
Denoising assets/disasterbefore.jpg 10 times,
with kappa=0.100000 and 5 iterations per execution.

(execution times shown are totals)

pure_python: 7.525767 seconds
numpy_weave: 0.128361 seconds
denoise_c  : 0.117864 seconds
```

Det eneste overraskende med resultatet er at min numpy-weave-implementasjon ser ut til å være raskere enn denoise_c.

Det er egentlig ikke tilfelle, og skyldes at jeg i implementasjon av den rene C-backenden starter med å gjøre en import av bildet for å finne ut om det er et fargebilde (for å kunne gi feilmelding om at farger ikke støttes av den backenden).

Denoising-eksempler med ulike backends

Nedenfor vises originalbildet, og videre vises dette bildet denoiset med en kappa på 0.1 og 10 iterasjoner med de ulike backendene; numpy-weave, pure-python og til slutt fra denoise.c.

Figure 1: Original image

Terminal

```
python denoise.py disasterbefore.jpg out.jpg --kappa=0.1 --iter=10
```

Figure 2: Denoised with numpy-weave, kappa=0.1, iter=10

Terminal

```
python denoise.py disasterbefore.jpg out.jpg --kappa=0.1 --iter=10 --backend=python
```

Figure 3: Denoised with python, kappa=0.1, iter=10

Terminal

```
python denoise.py disasterbefore.jpg out.jpg --kappa=0.1 --iter=10 --backend=c
```

Figure 4: Denoised with denoise_c, kappa=0.1, iter=10

Oppgave 1: Profilering

Profileringen av de tre implementasjonene er gjort i `test_profiling.py`. Jeg implementerte oppgaven før jeg så kodeeksempelet som ble lagt ut og falt derfor ned på en litt mindre elegant løsning med splitting og offsetting for å finne riktig utvalg fra outputen.

```
s = StringIO.StringIO();
ps = pstats.Stats(pr, stream=s).sort_stats('cumulative');
ps.print_stats();

print "\n\n================================================";
print '{:=^50}'.format(" " + denoiser + " ");
print "================================================";

print "\n".join(s.getvalue().splitlines()[4:8]);
```

Resultat fra kjøring

Terminal

```
$ python test_profiling.py
Denoising assets/disasterbefore.jpg 10 times,
with kappa=0.100000 and 5 iterations per execution.



================================================
================= python_denoise =================
================================================
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.004    0.004    1.108    1.108 /INF3331/oblig3/src/denoise/pure_python.py:98(denoise_file)
        1    0.845    0.845    1.098    1.098 /INF3331/oblig3/src/denoise/pure_python.py:5(denoise_image_data)
```

```
   928778    0.086    0.000    0.086    0.000 {min}


================================================
============== numpy_weave_denoise ===============
================================================
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.018    0.018 /INF3331/oblig3/src/denoise/numpy_weave.py:152(denoise_file)
        1    0.000    0.000    0.015    0.015 /INF3331/oblig3/src/denoise/numpy_weave.py:12(denoise_image_data)
        1    0.000    0.000    0.015    0.015 /usr/lib/python2.7/dist-packages/scipy/weave/inline_tools.py:139(inline)


================================================
================== c_denoise ===================
================================================
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.012    0.012 /INF3331/oblig3/src/denoise/denoise_c.py:8(denoise_file)
        3    0.000    0.000    0.009    0.003 /usr/lib/python2.7/subprocess.py:473(_eintr_retry_call)
        1    0.000    0.000    0.008    0.008 /usr/lib/python2.7/subprocess.py:768(communicate)
```

Resultatet viser nokså tydelig hastighetsforskjellen mellom den rene python-implementasjonen og numpy-weave/denoise_c – som er tilnærmet like raske. `cProfile` er et fint verktøy for å profilere og tune python-kode, men er ikke spesielt hjelpsomt hvis målet er å optimalisere C-kode. Da er trolig `gprof` et mer passende verktøy.

Kommentarer til implementasjonen

Det er imidlertid verdt å nevne at svart-hvit-delen av koden, som er basert på denoise.c har lite rom for optimalisering ettersom den er såpass enkel og bare opererer på array-indekser, mens fargedelen med fordel kunne vært optimalisert.

Spesielt tenker jeg da på at den regner ut HSI-verdier fra RGB for omkringliggende punkter på nytt for hvert punkt. Den burde ha spart på, og gjenbrukt allokerte minneplasseringer for HSI og RGB-verdier og kunne nok med fordel også iterert over og konvertert alle piksler til HSI-verdier før manipuleringen begynte.

Oppgave 3: Utvidelse til farger

Utvidelsen til farger er implementert i numpy-weave og hoveddelen av denne logikken finnes i `support_c`-variablen i filen `weave_c.py`.

Metodene `createHSIFromRGB` og `createRGBFromHSI` inneholder utregningen av verdier basert på hhv. RGB og HSI.

Funksjonaliteten er en integrert del av numpy-weave-backenden og brukes dersom formen (shapen) på dataene som er importert med numpy er med tre dimmensjoner, à la (375, 500, 3). Siden arrayet blir gjort om til et endimmensjonalt array i weave brukes antallet componenter/kanaler til å beregne indexen for hver pixel i C-implementasjonen, slik som her;

```
// Calculate index of current pixel
current = (i * width + j) * channels;
```

(Variablene i og j svarer til hhv. raden og kolonnen i bildet.)

Eksemepel på denoising av fargebilde

Ovenfor vises et fargebilde med støy i, og nedenfor vises fargebildet denoiset med numpy-weave.

Figure 5: Original image

Terminal

```
python denoise.py assets/disasterbeforecolor.jpg \
report/images/nw-color-02-5.jpg --kappa=0.2 --iter=5
```

Figure 6: Denoised with numpy-weave, kappa=0.2, iter=5

Oppgave 4: Lineær manipulering

Oppgaven som gikk på lineær manipulering var ikke så godt beskrevet i oppgaven, men jeg har tatt utgangspunkt i at det er et positivt eller negativt tall som adderes til den aktuelle komponenten/kanelen på alle punkter i bildet.

Parametrene for å manipulere verdiene i bildet sendes med til frontend som vist i hjelp for denoise.c;

–lr N Amount to add to or remove from the R channel (RGB). –lg N Amount to add to or remove from the G channel (RGB). –lb N Amount to add to or remove from the B channel (RGB). –lh N Amount to add to or remove from the H component (HSI). –ls N Amount to add to or remove from the S component (HSI). –li N Amount to add to or remove from the I component (HSI).

Manipulering av flere kanaler kan gjøres samtidig og all manipulering skjer etter evt. denoising. Ettersom konvertering til/fra HSI er krevende sjekkes det om det skal gjøres manipulering av noen av HSI-komponentene, og evt. konvertering gjøres da. RGB krever ikke noen konvertering ettersom verdiene i arrayet er RGB-verdier.

Det er implementert sjekker for å sørge for at ingen manipulering fører til verdier som er utenfor grensene som gjelder for de ulike komponentene.

Eksempel på lineær manipulering

Figure 7: Original image

Oppgave 5: Frontend

Output fra frontend er vist allerede, så jeg gjentar ikke det, men et par andre ting er verdt å nevne.

Terminal

```
python denoise.py assets/disasterbeforecolor.jpg \
report/images/nw-color-02-5-100-02.jpg --kappa=0.2 --iter=5 --lr=100 --ls=-0.2
```

Figure 8: Denoised with numpy-weave, kappa=0.2, iter=5, r=100, s=-0.2

Timeit i frontend

Det står nevnt i opgaveteksten at det skal være mulig å skru `timeit`-modulen av/på i frontend. Med en tanke om fordeling av ansvar (separation of concerns) i bakhodet mener jeg at det tilfører lite verdi å ha timeit-funksjonalitet

i frontenden ettersom man skjeldent bruker mer enn én backend samtidig likevel.

Jeg valgte derfor, som tidligere beskrevet, å implementere hastighetssammenlikning i en separat fil; `test_speed_test.py` og output fra denne er vist tidligere.

eps-parameter til frontend

Jeg forsto rett og slett ikke hvorfor denne skulle være et parameter til frontend og valgte til slutt å ta den bort. Den er tatt med som første og eneste parameter til sammenlikningstesten som genererer og sammenlikner verdier i bilder (`test_file_comparison.py`).

Oppgave 7: Rapport

Du har lest rapporten nå. Hvis du har lyst til å generere den kan du kjøre kommandoen nedenfor fra roten av oblig3;

Terminal

```
python ../oblig2/prepro.py --pretty report/oblig3.tex oblig3.tex
python ../oblig2/compile.py oblig2.tex
```