

# Opplegg 2 - Digitale Bilder, Farger og Filtere

Bilder, video og datagrafikk er en av de store grunnene til at vi bruker datamaskiner. Men hvordan fungerer egentlig digitale bilder? I dette opplegget går vi nærmere inn på hvordan digitale bilder er bygget opp og hvordan de lagres som informasjon på en datamaskin. Dette er et veldig godt grunnlag for å gå inn på en bred rekke med temaer som for eksempel fargeteori, menneskesynet, kombinatorikk og masse annet. Dette er muligens det aller bredeste opplegget vi presenterer, og hvilke deler du velger å bruke vil avhenge av hva du vil at elevene skal sitte igjen med.

## Plan

Dette opplegget begynner med å gå gjennom hvordan et digitalt bilde er bygget opp av piksler, og hvordan informasjonen i bildet kan lagres som en lang rekke tall på datamaskin. Deretter begynner man å gå inn på farger og ser på hvordan man kan manipulere fargene i et digitalt bilde matematisk. Opplegget går også inn på mer av teorien bag farger og menneskelig syn. Iløpet av opplegget kan man lage enkle filtere på egenhånd, men vi viser også hvordan man kan bruke innebygde filtere til å jobbe med digitale bilder der elevene kan lage egne verk.

## Kompetanse mål

- **Matematikk**

1. drøfte og løyse enkle kombinatoriske problem Kunst og Håndverk
2. Undersøke, forklare og bruke tallsystemer på forskjellige måter

- **Kunst og håndverk**

1. Bruke ulike funksjoner i bildebehandlingsprogram
2. Bruke ulike materialer og redskaper i arbeid med bilder ut fra egne interesser

- **Naturfag**

1. Gjennomføre forsøk med lys, syn og farger, og beskrive og forklare resultatene

## Piksler

Et digitalt bilde er bygget opp av *piksler*. Ordet kommer fra det engelske *pixel* som er en sammenslåing av *picture element*. Et digitalt bilde er som et puslespill, der puslespillbrikene er pikslene, det er de minste bitene av bildet. Ulikt puslespillbrikker dreimot, så er hver piksel ensfarget.

Når vi ser på et bilde på mobilskjermen vår, så ser vi ikke de enkelte pikslene i bildet, fordi de er så små og det er så utrolig mange av dem, som gjør at de smelter litt sammen i synet vårt. Kameraer oppgir ofte hvor mange *megapiksler* de har. Mega betyr million, så et bilde fra et 10 megapiksel kamera har 10 millioner piksler!

Piksler er også viktig når vi snakker om skjermer, fordi digitale skjermer består også av piksler som kan endre farge utifra hva som skal vises frem. Her gis antall piksler ofte i form av *oppløsning*. En *full HD* skjerm for eksempel, har en oppløsning på "1920x1080", dette betyr at det er 1920 piksler i bredden, og 1080 i høyden. Antall piksler kan vi da regne oss frem til, på samme måte vi finner arealet av et rektangel

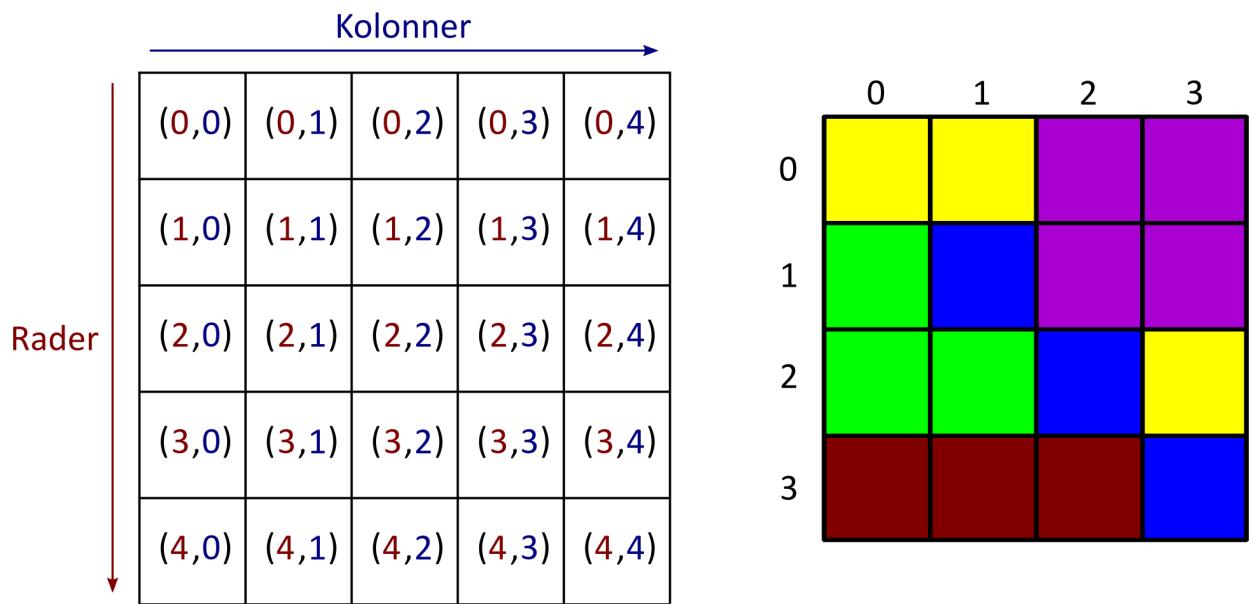
$$1920 \cdot 1080 = 2,073,600.$$

En *full HD* skjerm har altså ca 2 millioner piksler, så det er kanskje ikke så rart at du ikke klarer å se de individuelle pikslene i en dataskjerm eller mobilskjerm, men om du ser en gammel TV, eller har et bra forstørrelsesglass, så skal du kunne se de enkele pikslene.

## Koordinatsystemet i bildet

Et bilde består av piksler, og for hver piksel må vi ha informasjon om hvilken farge pikselen skal ha. Et digitalt bilde er egentlig bare en lang liste med farger på pikslene. Men for å vite hvilken farge som hører til hvilken piksel er det viktig at vi har et ryddig system, så det ikke blir noe surr.

Digitale bilder er alltid rektangler, som gjør ting veldig mye lettere for oss, for da vet vi at pikslene ligger i et helt vanlig rutenett. Det er derfor vanlig å bruke et koordinatsystem, tilsvarende det vi bruker i mattetimen. Når vi snakker om koordinatsystemet til bilder er det vanlig å tenke på det som å lese en tekst, så vi starter i hjørnet øverst til venstre. Vi lar denne pikselen få koordinatene  $(0, 0)$ . Når vi beveger oss mot venstre langs den øverste linjen med pixler lar vi den andre koordinaten øke, så vi får  $(0, 1)$ , så  $(0, 2)$  osv. Når vi beveger oss ned en rad hopper vi også tilbake til venstre side, så der finner vi først  $(1, 0)$ , så  $(1, 1)$  osv.



## Eksempeloppgaver

- a) På bildet til venstre. Hvilke farger har pikslene  $(1, 0)$ ,  $(2, 0)$  og  $(2, 1)$ ?
- b) Hvilke piksler er det som er blå?
- c) Hvilke piksler er det som er gule?

## Bildefiler

Et digitalt bilde er egentlig bare en lang liste med farger for hver piksel.

- $(0, 0) \rightarrow$  rød
- $(0, 0) \rightarrow$  lyserød
- $(0, 1) \rightarrow$  blå
- og så videre

Denne listen skrives til en fil på maskinen der informasjonen lagres. Når du vil se på bildet, eller for eksempel skrive det ut, så lester maskinen informasjonen og fargelegger skjermen eller arket utifra informasjonen lagret i filen. Digitale bildefiler er altså litt som oppskrifter for bildene.

Når et bilde typisk består av millioner med piksler er det en ganske lang liste med farger som må lagres! For å spare litt plass kan vi droppe å skrive koordinatene, og bare la dem være gitt i samme rekkefølge som vi leser i. Så det holder at vi bare lister fargene: "rød, lysrød, blå, ..."

## Gråtoner

På en datamaskin lagres all informasjon som tall, inkludert bilder. Dette betyr at listen med fargeinformasjon må være en lang liste med tall. Hvordan kan vi få farger fra tall? Det skal vi se på snart, men la oss starte med å se på bilder uten farger. Et svart-hvit bilde har ikke farger, men består av *gråtoner*. La oss se litt nærmere på et digitalt gråtonebilde.

Vi kan enkelt importere bilder fra filer på datamaskinen, eller fra nett. Men for vårt første eksempel tar vi et testbilde bygget inn pakken `skimage`.

In [1]:

```
1 from pylab import *
2 gray() # Nødvendig for at gråtonebilder skal vises riktig
```

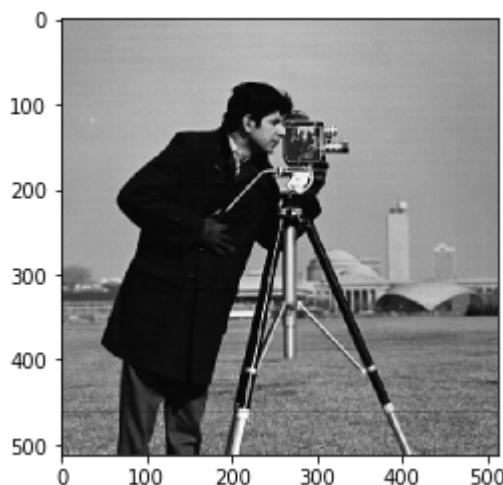
In [2]:

```
1 from skimage import data
2
3 img = data.camera()
```

Vi har nå importert et bilde og lagret det i variabelen `img`. Vi kan vise frem bilder med kommandoen `imshow`, som vanlig bruker vi også `show` til vise det på skjermen etterpå.

In [3]:

```
1 imshow(img)
2 show()
```



I tillegg til å vise frem selve bildet ser vi at det er akser på sidene som viser pikselkoordinatsystemet. Om du ikke ønsker å ha med dette kan du bruke funksjonen `axis('off')` før du bruker `show`, da vises bare selve bildet.

La oss prøve å printe ut bildet, istedet for å vise det frem, hva skjer da?

In [4]:

```
1 print(img)
```

```
[[156 157 160 ... 152 152 152]
 [156 157 159 ... 152 152 152]
 [158 157 156 ... 152 152 152]
 ...
 [121 123 126 ... 121 113 111]
 [121 123 126 ... 121 113 111]
 [121 123 126 ... 121 113 111]]
```

Vi ser at bildet består av masse tall, i lister. Disse tallene er selve informasjonen som bildet består av. Her er `img` objektet det vi kaller et *array*, som er som et rutenett. Vi kan sjekke dimensjonene til et array som følger

In [5]:

```
1 print(img.shape)
```

```
(512, 512)
```

Så vi ser at bildet vårt består av 512x512 tall. Vi ser også at tallene er heltall, og ikke desimaler. De få tallene vi ser skrevet ut over ligger på litt over 100, men la oss sjekke de største og minste tallene i bildet

In [6]:

```
1 print(img.min())
2 print(img.max())
```

```
0
```

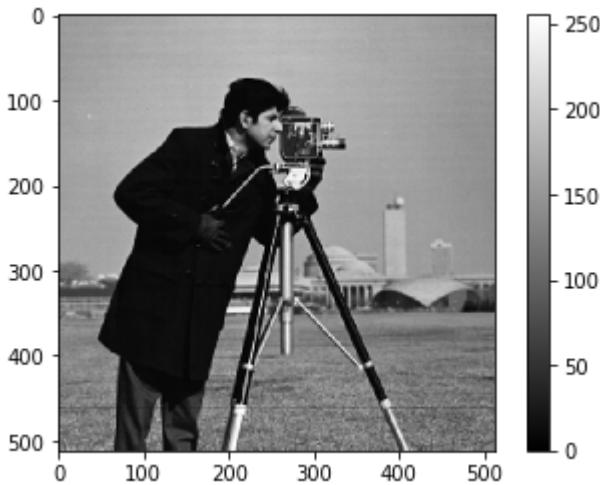
```
255
```

Så vi ser at bildet består av heltall mellom 0 og 255. Tallene angir mengden lysintensitet som skal være i hver piksel. Det betyr at 0 betyr ingen lys, og en piksel med verdi 0 blir altså helt svart. På den andre siden er 255 maks intensitet, og en piksel med verdi 255 blir helt hvit. I mellom disse to finner vi et spekter av gråtoner. Siden fargeverdiene oppgis som heltall ser vi at det er 256 ulike farger pikslene i bildet kan ta.

Vi kan se selve fargespekteret ved å kalle på `colorbar` før `show`.

In [7]:

```
1 imshow(img)
2 colorbar()
3 show()
```



Du lurer kanskje på hvorfor det er akkurat 256 mulige gråtoner i bildet? Hvorfor ikke 300? Eller 500? Eller 1000? Informasjon på en datamaskin lagres ikke bare som tall, men som en rekke "0" og "1". Dette kalles *totalssystemet*, for det finnes bare to siffer: 0 og 1. Dette er i motsetning til det vanlige tallsystemet vi bruker, som har 10 siffer: 0, 1, 2, ..., 9, og som vi derfor kaller *titallssystemet*.

Totalssystemet fungerer helt likt som titallssystemet: når vi går over det største sifferet, så 'ruller' denne rundt, og tallet til venstre øker med 1: for eksempel  $9 + 1 = 10$  og  $39 + 1 = 40$ . Sånn er det i totalssystemet, men vi ruller rundt mye før. Ta for eksempel  $1 + 1$ . Det finnes ingen 2-er vi kan bruke, så vi må rulle rundt og få 10, så  $1 + 1 = 10$ . Dette ser kanskje veldig rart ut, men det er bare fordi tallet til venstre er ikke på "10-er plassen", men på "2-er plassen".

Når vi lagrer informasjon om bilder i en fil lønner det seg om hver piksel bruker like mange siffer i informasjonen sin, sånn at datamaskinen skjønner hvor en piksel slutter og den neste begynner. I bildet vi ser på bruker hver piksel 8 siffer, eller 8 *bits*, som vi kaller dem i dataverden. Det betyr at hver piksel består av åtte 0-ere og 1-ere på rad. For eksempel 01001101 og 11010101. Hvor mange forskjellige gråtoner kan vi lage av disse åtte bitsene? For hver av dem, så kan vi velge enten 0 eller 1, så vi har 2 valg for hvert tall, da gir kombinatorikk oss

$$2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = 2^8 = 256.$$

Og det er grunnen til at det er 256 mulige gråtoner. Hvis vi ønsker mer mulige nyanser i et svart-hvitt bilde er det fullt mulig å bruke flere bits per piksel. Da vil bildefilen ta større plass, men vi får mer detaljer. Man liker å bruke antall bits som selv er 2-er potenser, så hvis man ønsker å gå opp fra 8 bits er det vanligst å bruke 16, 32 eller 64 bits.

I vårt tilfelle er bildet 8 bits informasjon, dette kan vi dobbeltsjekke ved å skrive ut datatypen til bildet vårt, som gjøres som dette

In [8]:

```
1 print(img.dtype)
```

uint8

Her er `uint8` en forkortelse for "unsigned integer 8-bit". Dette betyr at hver piksel i bildet er et heltall (integer), det er ikke et negativt tall (unsigned), og det består av 8-bit, som vil si det har verdier mellom 0 og 255.

## Hvor mye plass tar bildet?

La oss kjapt se på hvor mye plass bildet tar på datamaskinen. Vi så over at bildet består av 512x512 piksler, så det er totalt

$$512 \cdot 512 \text{ piksler} = 262144 \text{ piksler.}$$

Hver piksel bruker 8 bit på maskinen, så det blir

$$262144 \text{ piksler} \cdot 8 \text{ piksler per piksel} = 2,097,152.$$

Bildet består altså av over to millioner 0-ere og 1-ere på rad. Det høres kanskje ut som ekstremt mye, men i datasammenheng er det egentlig ganske lite: en datamaskin kan rett og slett lagre enorme mengder informasjon.

Det er vanligere å oppgi lagringskapasitet i *bytes*. En byte er 8 bits, så hver piksel i bildet vårt bruker 1 byte. Som betyr at bildet derfor har 262144 bytes, eller 0.26 megabytes (mega betyr million). Du har kanskje hørt om megabyte før, det er en ganske liten enhet. En vanlig harddisk pleier å ha lagringskapasitet på hundrevis eller tusenvis av *gigabytes*, der giga betyr milliard.

Dette bildet bruker såpass lite lagringskapasitet fordi det er forholdsvis lav oppløsning, og det er bare gråtoner, som tar lite plass.

## Farger og RGB

Vi har sett hvordan en datamaskin lagrer et gråtonebilde som tall mellom 0 og 255. Men hva med fargebilder? Vi ønsker å ha et system som kan bruke veldig mange forskjellige typer farger i bildene våre, men hvordan kan vi lage farger fra tall? Den vanligste måten å gjøre dette på er med et system som kalles RGB.

RGB står for Rød-Grønn-Blå (eller Red-Green-Blue på originalspråket). Disse tre fargene er *primærfargene* i fargesystemet, og ved å blande sammen de tre primærfargene i forskjellige forhold kan vi lage nye farger og nyanser.

### Litt Historiske Fargebilder

Det første fargefotografiet i verden, ble tatt allerede i 1861 av James Clerk Maxwell, en kjent naturvitenskaper og fysiker. Bildet til Maxwell ble laget ved å ta tre bilder med forskjellige fargefilteret foran kameraet, for så å sette sammen de forskjellige bildene etterpå.



I perioden 1909-1915 ble det vanlig blant noen, spesielt interesserte fotografer å bruke den samme teknikken til å ta fargefotografier. Dette var en dyr og komplisert prosess, og det var ikke før i 1960 at det begynnte å bli enklere og vanligere å ta fargefotografier.

Under er et fotografi fra 1911, av Muhammed Alim Khan, den siste emiren av Bukhara. På høyre side kan dere se at fotografen har tatt tre forskjellige bilder, hvert med et farget filter foran kameraet. Hvert bilde har da registrert mengden lys som kommer igjennom kameraet, og om bildene blir fremstilt som vanlig vil de se svart-hvit ut. Men det fotografen har gjort er å fremstille de tre bildene med forskjellige kjemikalier som produserer forskjellige farger, og de tre fargede komponentene kan nå kombineres til ett fargefotografi, som vist til venstre.



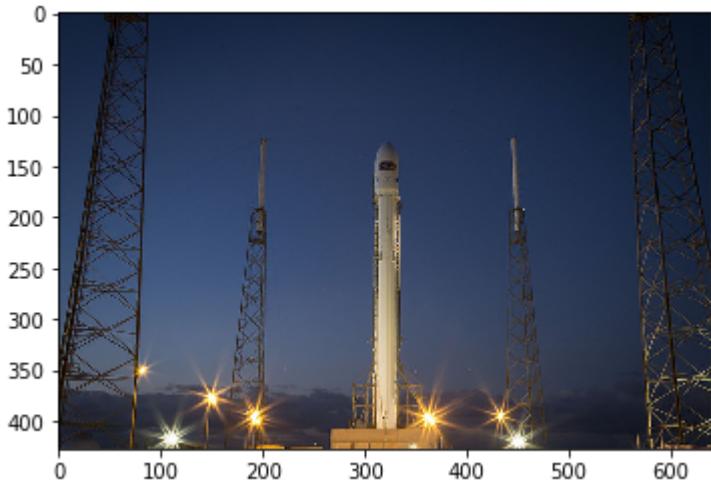
## RGB i Digitale bilder

Eksemplene over er ikke digitale bilder, men de samme idéene brukes for digitale bilder. Når vi skal skrive hvilken farge en piksel har, så gjør vi ikke dette med ett enkelt tall, men istedet med 3 tall. Vi lar hvert av de tre tallene bestå av 8 bit. For hver piksel oppgir vi tallene i rekkefølgen RGB, så om vi ønsker en veldig rød piksel kan vi for eksempel si (255, 0, 0). Om vi ønsker en lilla piksel må vi blande rød og blå, så vi kan for eksempel si (200, 0, 200).

La oss se på et fargebilde som eksempel. Denne ganger laster vi inn et annet av testbildene i skimage:

In [9]:

```
1 img = data.rocket()
2 imshow(img)
3 show()
```



In [10]:

```
1 print(img.shape)
2 print(img.dtype)
```

(427, 640, 3)  
uint8

Når vi skriver ut `img.shape` for bildet av raketten ser vi at det står `(427, 640, 3)`, dette betyr at bildet er  $427 \times 640$  pixler. Tretallet til slutt betyr at det for hver piksel er tre tall, en for hver av de tre fargene.

Vi ser også at datatypen til bilder igjen er `uint8`, som betyr at hvert tall igjen er et heltall mellom 0 og 255. Forskjellen er nå at hver piksel består av tre slike tall, én for hver farge, som betyr at hver piksel egentlig nå består av  $8 + 8 + 8 = 32$  piksler. På grunn av dette kalles av og til RGB-system for "32-bits farger".

Vi kan skrive ut fargene til én piksel som følger

In [11]:

```
1 print(img[0, 0])
```

[17 33 58]

## Dele opp et bilde i de ulike fargene

Siden det er et tall assosiert med hver av de tre primærfargene kan vi tolke fargebildet litt som tre separate gråtonebilder lagt over hverandre, slik som vist i bildet av emiren over. Disse tre bildene kalles *fargekanalene* til bildet.

Vi skal nå ta et eksempelbilde og dele det opp i fargekanalene sine. Denne gangen tar vi et eget eksempel bilde. Denne er lagret i bildefilen `peppers.png`, og ligger i en mappe som heter `fig`, denne mappen ligger ved siden av notebooken vår. Da kan vi laste inn bildet som følger:

In [12]:

```
1 img = imread("fig/peppers.png")
2
3 imshow(img)
4 show()
```



For å trekke ut fargekanalene bruker vi *indekssering*. De to første indekssene angir hvilken piksel det er snakk om, men vi vil ha alle piksler, så vi skriver `: , :` for de to første indeksene, som betyr *alle*. For den tredje indeksen velger vi fargekanal, det er tre kanaler: RGB. Siden Python begynner å telle på 0 blir det da slik at vi har indeks 0 (rød), 1 (grønn) og 2 (blå).

In [13]:

```
1 red = img[:, :, 0]
2 green = img[:, :, 1]
3 blue = img[:, :, 2]
4
5 print(red.shape)
6 print(green.shape)
7 print(blue.shape)
```

(384, 512)

(384, 512)

(384, 512)

Vi ser at hver av fargekanalene er et bilde med samme oppløsning. Vi kan nå plotte dem og se hvordan de ser ut. Siden vi nå bruker `imshow` på bilder med en tallverdi per pixel tolkes og vises det automatisk som gråtoner. Jo mørkere tonen er, jo mindre av den gitte fargen er i originalbildet - jo lysere tonen er, jo mer er det av fargen.

In [14]:

```
1 imshow(red)
2 axis('off')
3 title('Red')
4 show()
5
6 imshow(green)
7 axis('off')
8 title('Green')
9 show()
10
11 imshow(blue)
12 axis('off')
13 title('Blue')
14 show()
```

Red



Green



Blue



### Eksempelspørsmål

- a) Hvorfor er den røde kanalen så mye lysere enn de to andre?
- b) Hvorfor er den blå så mørk?
- c) Hvilen farge er det mest av i originalbildet?
- d) Det eneste som er lyst i alle bildene er løkene. Hvorfor tror du hvitløken ser hvit ut på alle tre kanalene?

Det finnes et triks vi kan bruke for å vise de ulike kanalene separat, men i farger istedenfor gråtoner. Dette gjør vi ved å sette de to andre kanalene til 0, istedet for å trekke de spesifikke kanalene ut.

In [15]:

```
1 red = img.copy()
2 red[:, :, 1] = 0
3 red[:, :, 2] = 0
4
5 green = img.copy()
6 green[:, :, 0] = 0
7 green[:, :, 2] = 0
8
9 blue = img.copy()
10 blue[:, :, 0] = 0
11 blue[:, :, 1] = 0
```

In [16]:

```
1 imshow(red)
2 axis('off')
3 title('Red')
4 show()
5
6 imshow(green)
7 axis('off')
8 title('Green')
9 show()
10
11 imshow(blue)
12 axis('off')
13 title('Blue')
14 show()
```

Red



Green



Blue



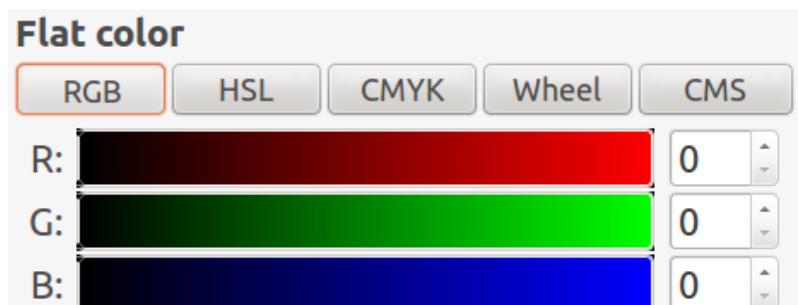
## Antall mulige farger

I RGB-systemet har hver piksel tre farger som alle kan være 0-255, antall mulige kombinasjoner blir da  
 $256 \cdot 256 \cdot 256 = 16,777,216$ .

Det finnes altså godt over 16 millioner forskjellige farger i 32-bits RGB bilder. Menneskeøynner kan skille mellom omrent 10 millioner farger, så over 16 millioner er veldig mange! Det er altså mange forskjellige farger i RGB som vil se helt like ut for oss.

Det finnes derimot noen andre begresninger i RGB som vi skal gå inn på etterhvert.

For å se mulighetene i RGB kan du bruke en [fargeplukker](https://www.w3schools.com/colors/colors_rgb.asp) ([https://www.w3schools.com/colors/colors\\_rgb.asp](https://www.w3schools.com/colors/colors_rgb.asp)), disse finnes det mange av på nett, og de fleste tegneprogram har dem innebygd.

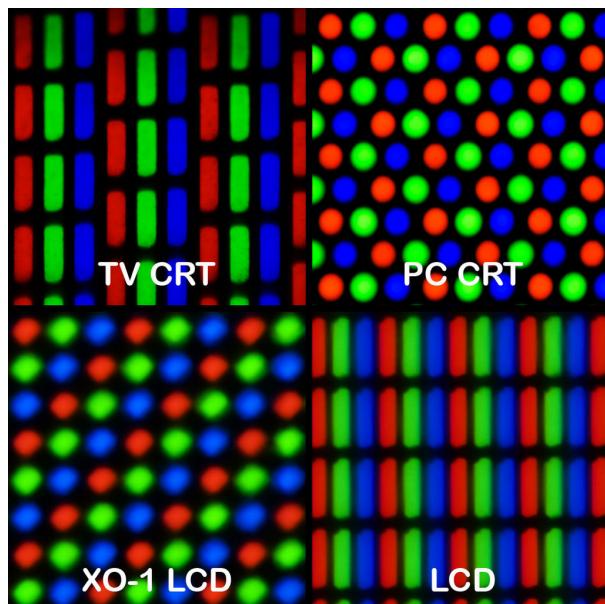


Bilde: RGB-Fargeplukkeren i tegneprogrammet *Inkscape*, de andre knappene på toppen endrer til andre fargesystemer, som man kan bruke istedenfor RGB om ønskelig.

## RGB i skjermpiksler

En piksel i en dataskjerm, mobilskjerm eller på en TV består av tre subpiksler, én i hver farge. Når skjermen skal vise noe får pikselen beskjed om hvilke farger den skal vise som en RGB kode, deretter justeres lysstyrken til de tre subpikslene til de riktig verdiene. På avstand smelter de tre fargene sammen, og vi oppfatter resultatet som en enkelt piksel av den riktige fargen.

En piksel i en skjerm trenger forresten ikke bestå av faktiske små firkanter, de kan for eksempel være runde, eller tynne streker. Se på bilder under, hvor det er zoomet inn på skjermer laget med forskjellige teknologier. Fra avstand ser de nok helt like ut.



(Bildet er tatt fra [Wikimedia Commons](#)

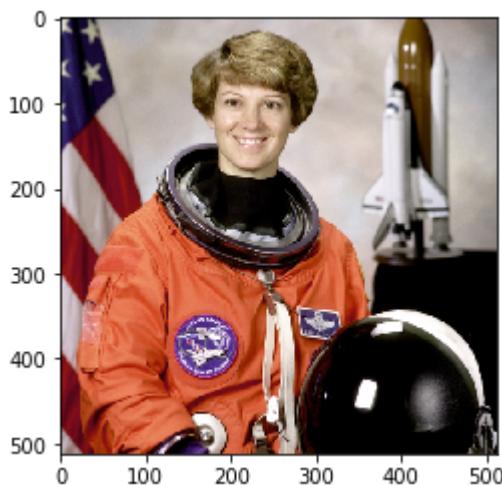
([https://upload.wikimedia.org/wikipedia/commons/thumb/4/4d/Pixel\\_geometry\\_01\\_Pengo.jpg/1024px-Pixel\\_geometry\\_01\\_Pengo.jpg](https://upload.wikimedia.org/wikipedia/commons/thumb/4/4d/Pixel_geometry_01_Pengo.jpg/1024px-Pixel_geometry_01_Pengo.jpg)) under [CC BY-SA 3.0](https://creativecommons.org/licenses/by-sa/3.0/) (<https://creativecommons.org/licenses/by-sa/3.0/>) lisens.)

## Konvertere fra farger til gråtoner

Nå som vi begynner å skjønne hvordan farger i digitale bilder fungerer, la oss se om vi klarer å konvertere et fargebilde til gråtoner. Vi bruker et nytt eksempelbilde fra skimage

In [17]:

```
1 img = data.astronaut()
2
3 imshow(img)
4 show()
```



Istad så vi hvordan vi kan trekke ut de enkelte fargekanalene og vise frem kun disse

In [18]:

```
1 red = img[:, :, 0]
2 green = img[:, :, 1]
3 blue = img[:, :, 2]
4
5 imshow(red)
6 axis('off')
7 title('Red')
8 show()
9
10 imshow(green)
11 axis('off')
12 title('Green')
13 show()
14
15 imshow(blue)
16 axis('off')
17 title('Blue')
18 show()
```

Red



Green



Blue



Disse tre bildene er jo gråtoner, men hvilken av dem er 'riktig'? Den midterste ser jo kanskje mest naturlig ut, så kanskje vi skal velge den? Vel, det finnes en bedre løsning: vi kan ta gjennomsnittet av de tre! Ved å ta gjennomsnittet får med informasjon fra alle tre fargekanalene.

In [19]:

```
1 imshow(red/3 + green/3 + blue/3)
2 show()
```



Vi ser at gjennomsnittet av de tre fargene lager et bilde som ser veldig naturlig i gråtoner, dette er fordi vi har fått med informasjon fra alle tre primærfargene i sluttresultatet.

**Eksempeloppgave:** Lag en funksjon som tar et vilkårlig fargebilde og gir tilbake et gråtonebilde. Kall funksjonen din for `rgb2gray`.

**Fasit:**

In [20]:

```
1 def rgb2gray(img):
2     red = img[:, :, 0]
3     green = img[:, :, 1]
4     blue = img[:, :, 2]
5     gray = red/3 + green/3 + blue/3
6     return gray
```

**Eksempeloppgave:** Test funksjonen din med et bilde du velger helt selv.

Fasit:

In [21]:

```
1 img = imread("fig/jonas.jpg")
2
3 gray = rgb2gray(img)
4
5 imshow(img)
6 axis('off')
7 show()
8
9 imshow(gray)
10 axis('off')
11 show()
```



## Eksempel: Skru av farger i et bilde

Vi har nå sett hvordan vi kan gjøre et fargebilde om til gråtoner. La oss se hvordan vi kan bruke det vi har lært til å skru av visse farger i et bilde. Som eksempel bruker vi et bilde fra filmen Matrix. I denne scenen er poenget at Neo skal legge merke til personen i rød kjole, fordi hun er den eneste som går i farger, mens alle de andre går i svarte klær. La oss prøve å forsterke denne effekten enda mer, ved å la resten av bilde være i gråtoner, mens kjolen forblir rød.

In [22]:

```
1 figure(figsize=(12, 8))
2 img = imread("fig/matrix_red_dress.jpg")
3 imshow(img)
4 show()
```



Vi har lyst å gjøre alt i bildet svart-hvitt, unntatt kjolen. Vi prøver først å gjøre om den blå og den grønne kanalen til gråtoner, men lar den rød kanalen være. Hva skjer?

In [23]:

```
1 # Laster inn bilde fra fil
2 raw_img = imread("fig/matrix_red_dress.jpg")
3
4 # Henter ut gråtoner og fargekanaler
5 gray = rgb2gray(raw_img)
6 red = raw_img[:, :, 0]
7 green = raw_img[:, :, 1]
8 blue = raw_img[:, :, 2]
9
10 # Lag en ny variant av bildet, så vi ikke endrer originalen
11 img = raw_img.copy()
12
13 # "Skru av" grønn og blå kanal
14 img[:, :, 1] = gray
15 img[:, :, 2] = gray
16
17 # viser bildet
18 figure(figsize=(12, 8))
19 imshow(img)
20 axis('off')
21 show()
```

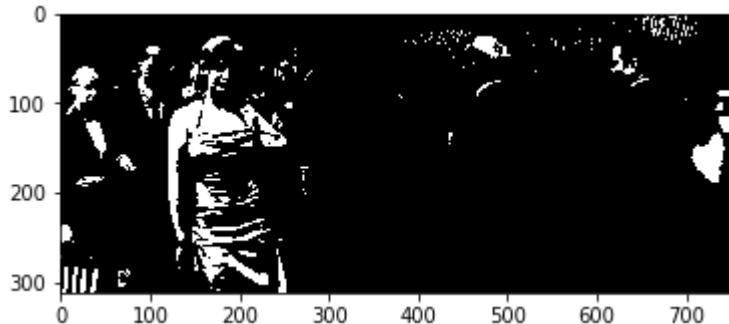


Vi ser at en del av fargene er forsvunnet, men bildet er ikke helt i gråtoner, for det er fortsatt rødskjær i for eksempel hudfargene. Samtidig er kjolen blitt noe vasket ut og den har blitt litt blass. Dette er fordi vi har endret på de grønne og blå fargene i kjolen, som er en viktig del av helhetsinntrykket.

La oss prøve noe annet. Det vi egentlig ønsker å gjøre, er å la alt bli gråtoner, bortsett fra kjolen, som vi skal la være som originalt. Så det vi egentlig trenger å gjøre er å finne en måte å trekke ut kjolen fra resten av bildet. Vi kan først prøve å gjøre dette ved å plukke ut alle piksler som har mye rødt i seg

In [24]:

```
1 dress = red > 150
2 imshow(dress)
3 show()
```

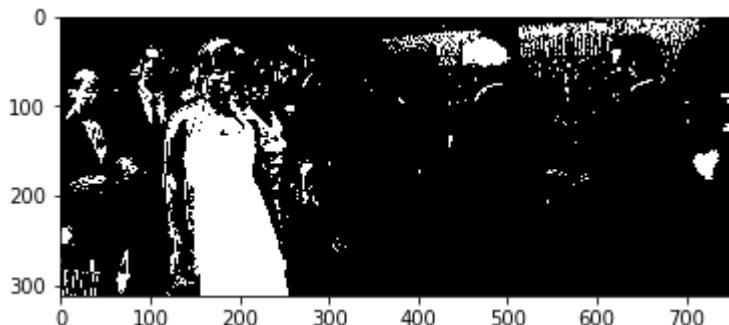


Vi ser at denne taktikken ikke fungerer så bra. Vi ser for eksempel at den lyse huden slår ut, fordi den inneholder mye rødt (men har samtidig mye grønnt og blått, så den ser ikke så rød ut på bildet), samtidig er deler av kjolen ikke innafor, fordi den blir for mørk, og mørk farge tilsvarer lite rødt (og enda mindre blått og grønnt, som er grunnen til at det ser rødt ut).

Det vi heller bør gjøre, er å se etter piksler som er *mye mer røde enn de er blå eller grønne*

In [25]:

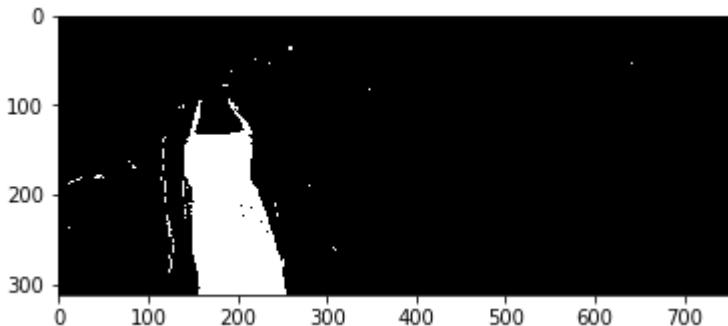
```
1 dress = red > (green + blue)
2 imshow(dress)
3 show()
```



Dette fungerer langt bedre, for nå har vi fått med hele kjolen. Derimot har vi fortsatt litt mye annet med på kjøpet. La oss prøve å dele de opp og si at pikslene må være både mye mer rød enn grønn, og mye mer rød enn blå.

In [26]:

```
1 dress = (red > 2.5*green)*(red > 2.2*blue)
2 imshow(dress)
3 show()
```



Der sittern! Nå gjør vi først hele bildet grått, men så gjør vi pikslene som svarer til kjolen om til originalbildet.

In [27]:

```
1 img = raw_img.copy()
2 img[:, :, 0] = gray
3 img[:, :, 1] = gray
4 img[:, :, 2] = gray
5
6 img[dress] = raw_img[dress]
7
8 figure(figsize=(12, 8))
9 imshow(img)
10 axis('off')
11 show()
```



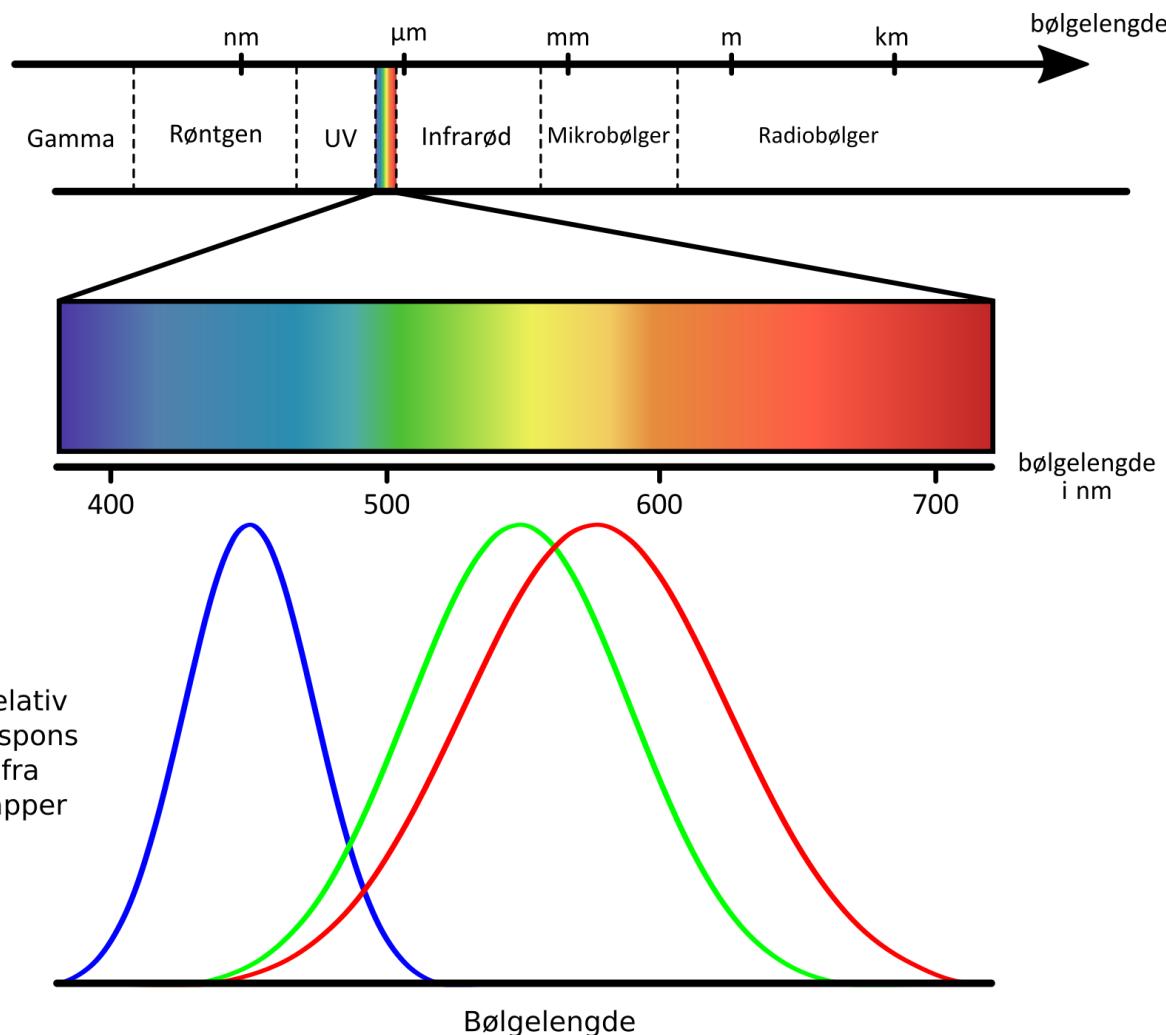
## Mennesklig syn

Vi har nå sett litt på hvordan digitale farger kan bygges opp som en kombinasjon av primærfargene i digitale bilder. Det som er litt stilig er at RGB-systemet ligner veldig på mennesklig syn!

Når vi ser med øynene våre er det fordi lys går igjennom øyet og treffer *netthinnen*. Netthinnen er et omeråde mot baksiden av øyet fyllt med celler som er sensitive for å ta imot lys. Disse cellene mottar lyset og sender signaler til hjernen som tolker signalene. Disse kaller vi lysreseptorer.

I netthinnen har vi to typer lysreseptorer: staver og tapper. Disse reseptorcellene er utrolig små: et vanlig menneskeøye har omrent 6 millioner tapper og 130 millioner staver. Tappene er ansvarlig for fargesyn, mens stavene ikke bryr seg om farge, de ser bare lysintensitet, det vil si, lys og mørke.

Tappene, som gir oss fargesynet, kan igjen deles i tre reseptorer, som svarer på forskjellige farger. Som du kanskje kan gjette på nå er disse reseptorene responsive på rødt, grønnt og blått lys. Vi sier at mennesker er *trikromatiske*, siden vi har reseptorer som ser tre forskjellige farger ("kroma" er gresk og betyr farge). Hvis det er mørkt blir tappene mindre aktive, og stavene tar mer over. Når man har godt mørkesyn ser man altså også automatisk mindre farger og synet blir mer svart-hvitt.



Det er ikke tilfeldig at digitale skjermer og bilder er laget for å gi rødt, grønnt og blått lyst, når det er dette menneskesynet svarer best på. Skjermene er spesialdesignet på denne måten for å gjengi flest mulig av de fargene mennesker kan se. Noen dyr, for eksempel mange fugler, er *tetrakromater* (tetra betyr 4), som betyr at disse fuglene trenger 4 primærfarger for å gjenngi alle farger de kan se (den fjerde primærfargen er UV lys, som mennesker ikke kan se). For en fugl er nok fargene på en dataskjerm ganske tamme og litt kjedlige, for de mangler UV delen av fargene.

## Fargeblindhet

Det finnes mange som ikke ser farger på samme måte som andre mennesker, og disse kalles gjerne for *fargeblinde*. Ordet *fargeblinde* får det til å høres ut som man ikke kan se farger i det hele tatt, og ser alt i gråtoner, slik er det ikke. Det finnes noen som ser i gråtoner, men det er *ekstremt* sjeldent. De fleste som er

fargeblinde ser fortsatt farger, men de ser noe mindre farger enn vanlig.

Den vanligste fargeblindheten er rød-grønn fargeblinde, vi kaller det *rød-grønn* fargeblind, fordi man ikke klarer å skille røde og grønne farger. Når vi ser på bildet over ser vi at rød og grønn ligger veldig nærmee hverandre i måten tappene våre ser farger. Hos noen ligger disse enda mer over hverandre, og reseptorene i øynene svarer helt likt på rød og grønne farger - derfor kan man ikke skille på dem lenger.

En annen mulighet er at man kan mangle tapper av en type, om man mangler de røde eller de grønne tappene vil man heller ikke kunne skille rød og grønn (det går også ann å mangle de blå tappene, men dette er mer sjeldent). Det å ha en form for fargeblindhet er egentlig ganske vanlig, på verdensbasis er 8% av alle menn fargeblinde, men bare 0.3% kvinner.

La oss prøve å se om vi kan gjennskape det rød-grønn fargeblinde ser ved hjelp av programmering.

### **Simulere Rød-Grønn fargeblindhet**

Siden rød-grønn fargeblinde ikke klarer å skille på røde og grønne farger kan vi tenke oss at vi kan simulere effekten ved å blande sammen den røde og den grønne kanalen i et bilde. La oss prøve med et testbilde av en hage full av blomster (bildet er hentet fra [colorblindawareness.org\\_\(http://www.colourblindawareness.org/colour-blindness/colour-blindness-experience-it/\)](http://www.colourblindawareness.org/colour-blindness/colour-blindness-experience-it/), slik at vi kan sammenligne resultatet vårt med den mer fancy simulatoren deres).

In [28]:

```
1 # Laster inn filen flowers.jpg
2 img = imread("fig/flowers.jpg")
3
4 # Vis originalbilde
5 imshow(img)
6 axis('off')
7 show()
8
9 # Plukk ut de forskjellige fargekanalene
10 rød = img[:, :, 0]
11 grønn = img[:, :, 1]
12 blå = img[:, :, 2]
13
14 # Lag en ny fargekanal som er en blanding av rød og grønn
15 rødgrønn = 0.5*rød + 0.5*grønn
16
17 # Lag en kopi av originalbilde
18 fargeblind = img.copy()
19
20 # Bytt ut både den røde og grønne kanalen med den nye blandingen vi har laget
21 fargeblind[:, :, 0] = rødgrønn
22 fargeblind[:, :, 1] = rødgrønn
23
24 # Vis det simulerte fargeblinde bildet
25 imshow(fargeblind)
26 axis('off')
27 show()
```





Vi ser at når vi blander den røde og grønne kanalen får vi et bilde som ikke inneholder noe særlig grønt og rødt! Gressfargen er blitt mer gulaktig, og de røde blomstene ser vi ikke lenger, for de har samme farge som bakgrunnen. De blå og gule blomstene ser vi fortsatt godt. De rosa og lilla blomstene er blitt mer blå eller hvite, for vi klarer ikke å se rødfargen så godt lenger - og slik er det faktisk, rød-grønn fargeblinde sliter samtidig å se forskjellen på blå og lilla.

**Eksempeloppgave:** Skriv en funksjon som tar et vilkårlig fargebilde og gir et bilde tilbake som simulerer rød-grønn fargeblindhet.

**Fasit:**

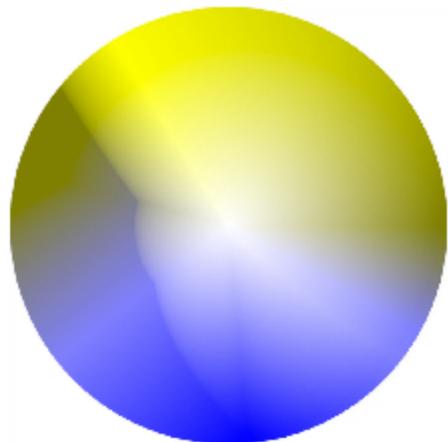
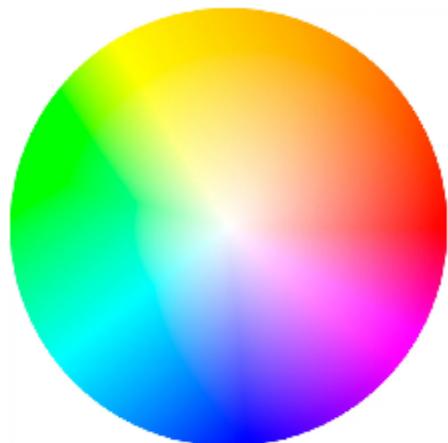
In [29]:

```
1 def fargeblindhet_simulator(img):
2     rød = img[:, :, 0]
3     grønn = img[:, :, 1]
4     blå = img[:, :, 2]
5
6     rødgrønn = 0.5*rød + 0.5*grønn
7     fargeblindt_bilde = img.copy()
8     fargeblindt_bilde[:, :, 0] = rødgrønn
9     fargeblindt_bilde[:, :, 1] = rødgrønn
10
11 return fargeblindt_bilde
```

Nå som vi har en funksjon kan det være interessant å prøve den ut på litt forskjellige bilder. La oss første prøve å ta et bilde av et helt fargehjul

In [30]:

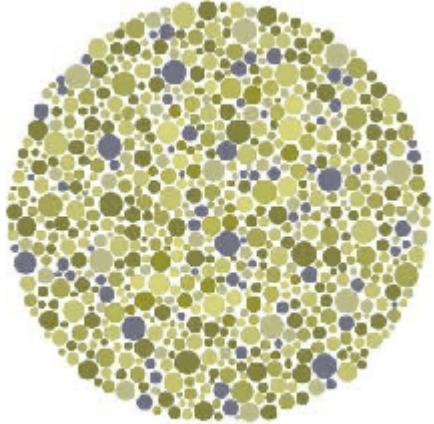
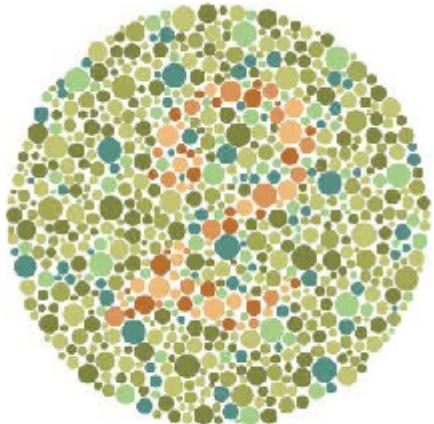
```
1 # Last inn bildet
2 img = imread("fig/fargehjul.png")
3
4 # Vis originalen
5 imshow(img)
6 axis('off')
7 show()
8
9 # Vis rød-grønn fargeblind versjon
10 imshow(fargeblindhet_simulator(img))
11 axis('off')
12 show()
```



La oss også teste en fargeblindhetstest

In [31]:

```
1 img = imread("fig/fargeblind_test.jpg")
2
3 # Vis originalen
4 imshow(img)
5 axis('off')
6 show()
7
8 # Vis rød-grønn fargeblind versjon
9 imshow(fargeblindhet_simulator(img))
10 axis('off')
11 show()
```



Måten vi simulerer rød-grønn fargeblindhet her er noe overforenklet, og det er smånyanser vi ikke inkluderer. Om du vil se bedre simuleringer kan du ta en titt på nett, der finnes det mange simulatorer på nett, for eksempel på [colorblindawareness.org](http://www.colourblindawareness.org/colour-blindness/) (<http://www.colourblindawareness.org/colour-blindness/>).

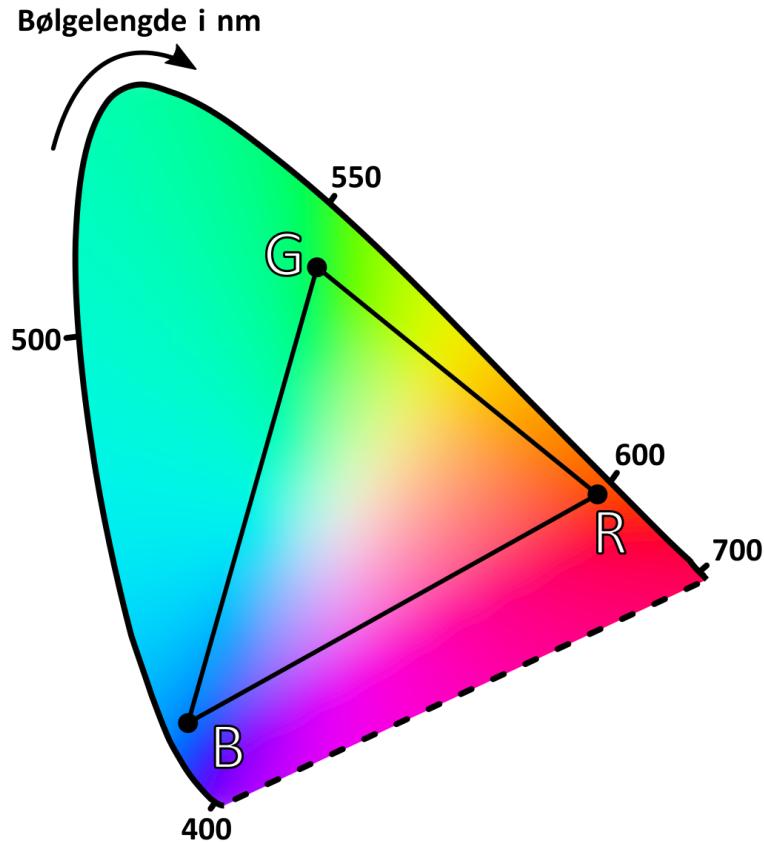
## Fargehesteskoen

Lys er elektromagnetisk stråling med bølgelengder mellom omtrent 400 og 700 nm. Hver bølgelengde er knyttet til en bestemt farge, der hele spekteret ser ut som regnbuen. Det er dette som vises i figuren lenger oppe. Det er ganske vanlig å tegne det elektromagnetiske spekteret langs en rett linje på denne måten, der aksen bortover er økende bølgelengde.

En annen måte å tegne spekteret på, er som en hesteskoform, der bølgelengdene øker med klokka rundt kanten på hesteskoen. Grunnen til at vi tegner den på denne måten er at vi kan nå også forstå hvordan farge det blir på lys dersom vi blander forskjellige bølgelengder! Måten vi gjør det på er å trekke en strek mellom bølgelengdene som blandes, og der de krysser finner vi fargen vi ser.

En dataskjerm har lysdioder som sender ut rødt, grønt og blått lys. Så hvis vi setter en prikk på disse tre fargene i hesteskoen lager vi et trekant. En dataskjerm kan lage alle farger innenfor denne trekanten. Ved å endre hvor kraftig de tre forskjellige lysdiodene lyser så flytter man seg rundt inne i trekanten.

En dataskjerm kan derimot ikke lage noen av fargene *utenfor* trekanten. I bildet under av en slik fargehestesko ser det ikke ut som vi går glipp av så mye, men det er jo fordi bildet vises frem på en dataskjerm!



## Dataskjerm eller Fargeprinter

Vi har snakket mye om RGB fargesystemet, og dette er veldig mye brukt og veldig viktig, men det er ikke det eneste som finnes. Om du for eksempel åpner bildehandlingsprogramvare kan du ofte se lange lister over ulike fargesystemer som brukes til diverse formål. De fleste av disse formatene er forkortelser, for eksempel CMYK og YCbCr. Forskjellige fargesystemer er laget for forskjellige formål og teknologier. La oss se på CMYK som et eksempel.

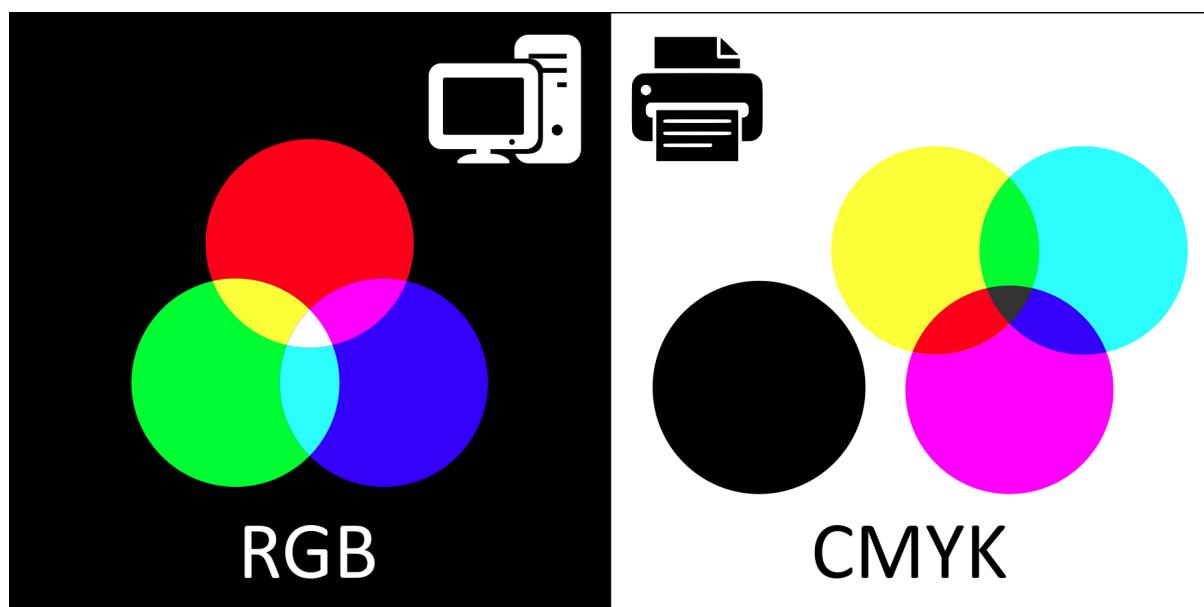
Som vi har forklart fungerer RGB bra for å forklare lys, for eksempel det lyset dataskjermen vår sender ut. Det er slik at om vi blander lys av forskjellig farge sammen får vi et sterkere og hvitere lys. Om vi blander alle fargene sammen får vi helt hvitt lys. Dette er grunnen til at sollys er hvitt, solen sender ut lys med masse forskjellige bølgelengder, så vi tolker denne blandingen som hvit.

Om vi skal skrive ut noe på et ark, eller male et maleri, fungerer det derimot annerledes. Da er det ikke lenger slik at å blande sammen farger gjør dem lysere, istedet blir de mørkere! Dette er fordi maling og blekk er laget for å absorbere lys, det er det de *ikke* absorberer som vi ser. Så rødt blekk absorberer alle farger *unntatt* rødt.

Når vi blander mange farger sammen absorberer de mer og mer farge til vi ikke får noe lys igjen i det heletatt, og det tolker vi som svart.

Vi kan si at RGB er *additiv*, fordi å legge sammen farger gjør dem lysere/sterke (som ved addisjon av to tall  $a+b$ ), mens fargene i en printer er *subtraktive*, fordi å legge sammen farger gjør dem svakere/mørkere (som ved subtraksjon  $a-b$ ). En fargemodel som beskriver den *subtraktive* fargeblandingen for en printer er CMYK. Her står CMY for "Cyan" (turkis), "Magenta" (rosa) og "Yellow" (gul). Dette er primærfargene i modellen, og ved å blande dem kan vi få alle andre farger. Til slutt har vi "k", som står for "key", som egentlig bare er svart.

Grunnen til at vi inkluderer svart, er at det ville vært altfor dyrt å blande sammen masse fargeblekk hver gang vi skulle hatt svartfarge, da er det billigere å ha svart blekk, som koster mye mindre enn fargeblekk. I tillegg er det vanskelig å lage en fin svartfarge ved å blande farger. Du har kanskje prøvd dette selv med maling, om man blander sammen masse farger får man gjerne en mørk brunaktig farge eller en dyp grå, men det blir liksom aldri helt svart, så da er det lettere å bare bruke svart maling



**Tankespørsmål:** Hvordan får vi hvit i CMYK modellen? Kan en printer skrive ut hvit? På samme måte, hvordan lager en projektor svarte farger? Kan en projektor projisere "svart" lys?

## Filtere

Vi skal nå snakke litt om bildefiltere. Vi har nok alle hørt om filtere, ihvertfall om man bruker instagram eller snapchat! Men hva er egentlig et bildefilter? Et bildefilter er ganske enkelt en funksjon som tar et bilde inn, gjør noen endringer på det og sender et annet bilde tilbake. I programmering vil altså et filter kunne lages som en funksjon og brukes som `filter(bilde)`. Men dette har vi jo gjort mange ganger så langt. Det stemmer, vi har laget flere bildefiltere i dette opplegget allerede. For eksempel er `rgb2gray()` og `fargeblindhet_simulator` funksjonene vi lagde eksempler på bildefiltre, og vi lagde disse helt fra bunnen av.

Vi skal ikke lage noen flere filtre fra bunn av, men nå skal vi istedet bruke et par innebygde filtre i `skimage`. Disse filtrene er laget for å brukes til å analysere og forstå bilder, ikke nødvendigvis for at bilder skal se mer fancy eller morsomme ut - så disse filtrene er litt annerledes enn de filtrene vi finner i insta/snap. De er også gjerne oppkalt av de som fant dem opp, så navnene på dem er kanskje litt kryptiske.

## Chelsea

Som eksempelbilde bruker vi et av de innebygde filtrene i skimage pakken, denne gangen av katten *Chelsea*

In [32]:

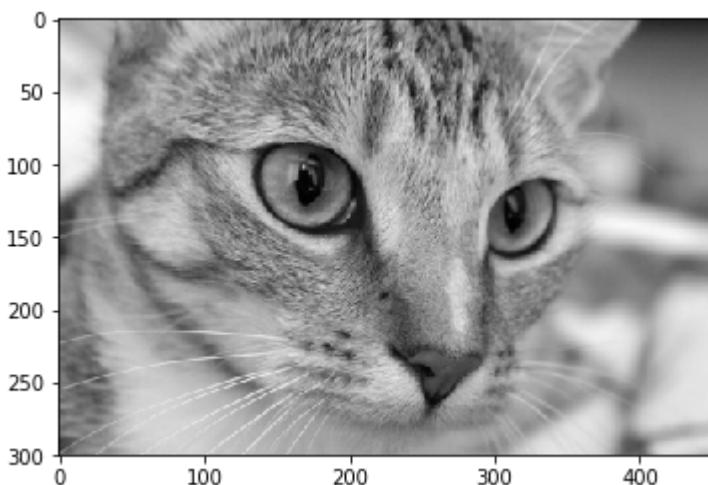
```
1 chelsea = data.chelsea()
2 imshow(chelsea)
3 axis('off')
4 show()
```



Filtrene vi skal se på fungerer kun på gråtonebilder, så vi bruker `rgb2gray()` funksjonen du lagde tidligere for å konvertere bildet til gråtoner. (Hvis du ikke har laget denne selv kan du bruke `skimage.color.rgb2gray()` som gjør dette for deg.

In [33]:

```
1 chelsea_gray = rgb2gray(chelsea)
2 imshow(chelsea_gray)
3 show()
```

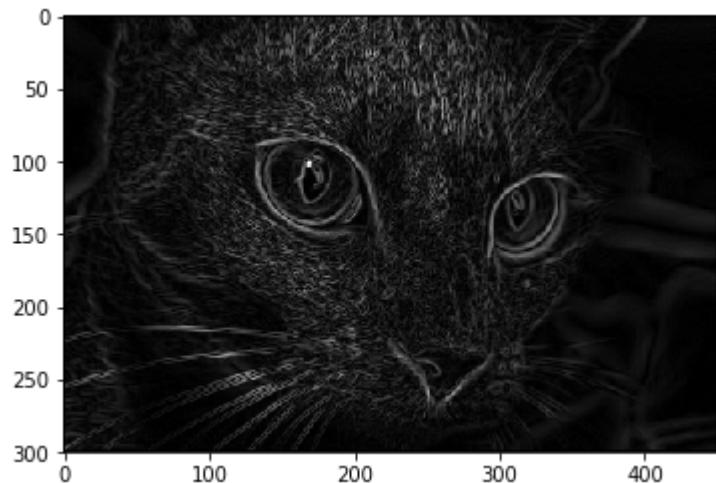


## Sobelfilteret finner konturer

Det første filteret vi skal se på heter ett *Sobel*-filter. Et Sobel filter finner områder i et gråtonebilde der fargene endrer seg raskt, det vil si der det går fra svart til hvitt eller motstatt. Filteret er altså god på å finne overganger og konturer i et bilde. For eksempel kanter og mønstre.

In [34]:

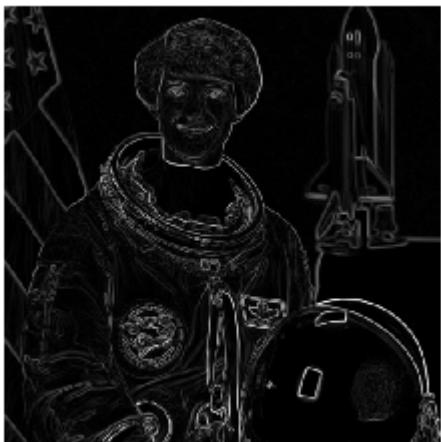
```
1 from skimage.filters import sobel
2
3 konturer = sobel(chelsea_gray)
4 imshow(konturer)
5 show()
```



Vi ser på bildet av katten at øynene, nesa og deler av værhårene er det som kommer ut av Sobel-filteret, dette er fordi det er der det er tydligst skiller i originalbildet. La oss bruke et annet eksempelbilde også

In [35]:

```
1 img = data.astronaut()
2 gray = rgb2gray(img)
3
4 imshow(gray)
5 axis('off')
6 show()
7
8 imshow(sobel(gray))
9 axis('off')
10 show()
```



Sobelfilteret er veldig viktig, for konturer gjør det lett å dele ett bilde opp i forskjellige objekter. Å tolke bilder er veldig vanskelig for en datamaskin, men vi ser at ved å bruke Sobel kan det være enklere å plukke ut interessante objekter.

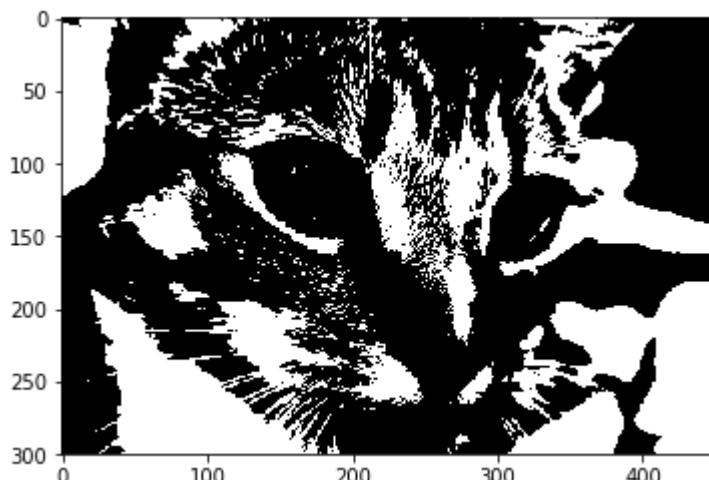
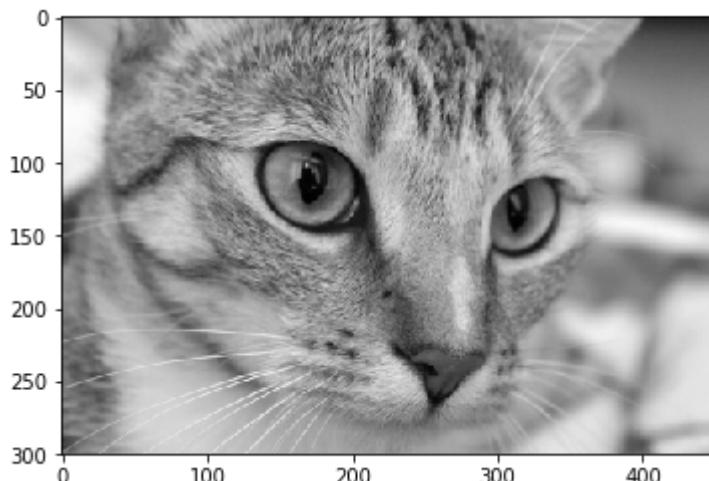
## Otsu-filter deler et bilde i forgrunn og bakgrunn

Vi har sett hvordan vi kan gjøre fargebilder om til gråtonebilder, men nå skal vi gå ett steg til å gjøre dem helt svart-hvitt. Det vil si, vi skal kun bruke helt hvit, eller helt svart. Dette kalles et *binærbilde* fordi vi bare har 2 farger, og *binært* betyr at noe er laget av to ting/komponenter.

Det er veldig enkelt å lage et binærbilde om vi har et gråtonebilde, vi bare bestemmer oss for en grense og sier at alle farger som er mørkere enn grensen skal være svarte, og alle farger som er lysere enn grensen, skal være hvite. La oss ta et eksempel. Husk at bilder ofte har tallverdier mellom 0 og 255. Så la oss prøve å dele bildet helt på midten, dvs, ved 128. Da skriver vi bare `chelsea_gray > 128`. Når vi skriver dette går Python igjennom hver piksel i bildet og sjekker, er verdien større enn 128 ( $>128$ ) så blir resultatet "1", som betyr *sant*, og om verdien er mindre enn 128 så blir resultatet "0", som betyr *falskt*. Resultatet lagrer vi i en ny variabel.

In [36]:

```
1 imshow(chelsea_gray)
2 show()
3
4 svarthvitt = chelsea_gray > 128
5
6 imshow(svarthvitt)
7 show()
```

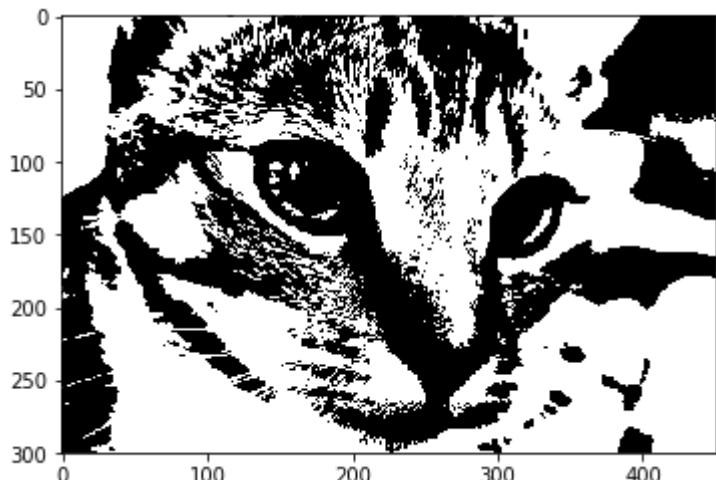


Vi ser at Binærbilde av Chelsea ble ganske bra, vi ser ihvertfall fortsatt at det er en katt! Men kanskje bildet er litt vel mørkt, så vi bør redusere grensen vår litt. Kanskje vi skulle prøvd med en grense på 100?

Her er det et filter som gjør jobben for oss. *Otsu*-filteret velger automatisk grensen man bør bruke for å lage et binærbilde. Den gjør dette ved å først regne seg frem til hvilken grense som gir oss mest detaljer. La oss prøve på to forskjellige bilder. Funksjonen `filters.threshold_otsu()` tar et bilde inn, og gir oss grenseverdien tilbake, ikke et ferdig bilde. Så vi må først bruke Otsu for å finne grensen, så lage binærbildet selv. Ordet `threshold` er engelsk og betyr *terskel* eller *grense*

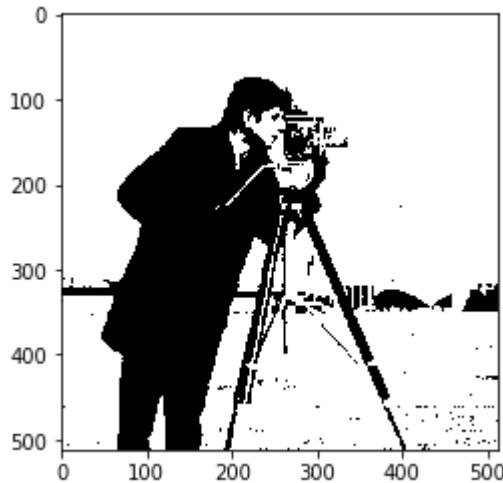
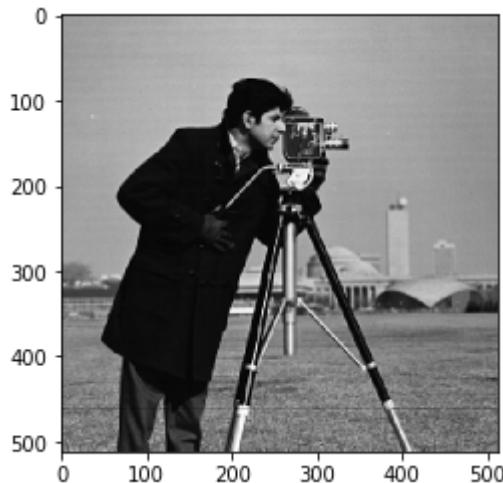
In [37]:

```
1 from skimage.filters import threshold_otsu
2
3 grense = threshold_otsu(chelsea_gray)
4 svarthvitt = chelsea_gray > grense
5
6 imshow(svarthvitt)
7 show()
```



In [38]:

```
1 # Dette er også et gråtone bilde, så vi trenger ikke konvertere det til gråtone
2 img = data.camera()
3
4 # Vis originalbildet
5 imshow(img)
6 show()
7
8 # Bruk Otsu til å finne grense og lag et binærbilde med grensen
9 grense = threshold_otsu(img)
10 svarthvitt = img > grense
11
12 # Vis frem binærbildet
13 imshow(svarthvitt)
14 show()
```



Binærbilder, spesielt funnet ved hjelp av Otsu, brukes også masse i digital bildegenkjennig. På samme måte som konturer kan det hjelpe for å plukke ut interessante objekter eller mønstre. Samtidig blir bildet mye enklere, som gjør at det kan være lettere for datamaskinen å analysere eller endre på det etterpå.

## Gaussisk filter gjør ting uklare

De neste filtrene vi skal se på heter *blurs* på engelsk. *Blur* betyr uskarpt, så disse filtrene gjør bilder mindre skarpe. På TV og film, spesielt i detektivserier, ser man gjerne at folk bruker filtre som gjør bilder *skarpere* enn de er, de kaller de sier gjerne "enhance", eller noe lignende. Men dette går egentlig ikke. Vi kan ikke få *mer* detaljer ut av et uklart bilde. Men *blurs* gjør altså ting *mindre skarpe*, og det er fullt mulig.

For å forstå hvordan blurs fungerer må vi først tenke oss hvordan vi kan ta "gjennomsnittet" av et bilde. Vi har jo sett at et bilde bare er en lang rekke tallverdier, ett tall for hver piksel (i et gråtonebilde) Så vi kan ta gjennomsnittet av et helt bilde ved å summere alle tallene og dele på antall piksler. Først finner vi antall piksler, det gjorde vi med `.shape()`

In [39]:

```
1 print(chelsea_gray.shape)
```

(300, 451)

Bildet er altså 300 piksler i høyden, og 451 i bredden. Eller totalt  $300 \times 451 = 135300$  piksler. For å finne summen av alle pikselverdiene kan vi bruke `sum()`.

In [40]:

```
1 antall_piksler = 135300
2 gjennomsnitt = chelsea_gray.sum() / antall_piksler
3
4 print(int(gjennomsnitt))
```

115

Så gjennomsnittet er en gråfarge omrent på midten av (0, 255). Vi kunne funnet dette enda enklere med `.mean()`. ("mean" er engelsk og betyr gjennomsnitt)

In [41]:

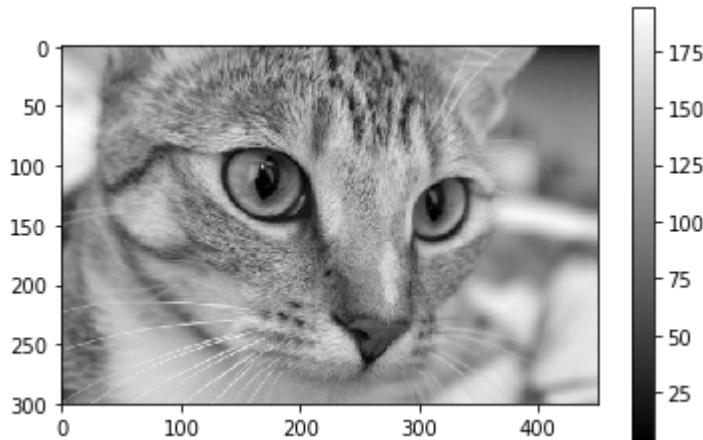
```
1 print(int(chelsea_gray.mean()))
```

115

Siden dette bare er ett tall, kan vi ikke vise det frem som et bilde, men vi kan skrive ut gråtoneskalaen ved siden av originalbildet og vise den

In [42]:

```
1 imshow(chelsea_gray)
2 colorbar()
3 show()
```

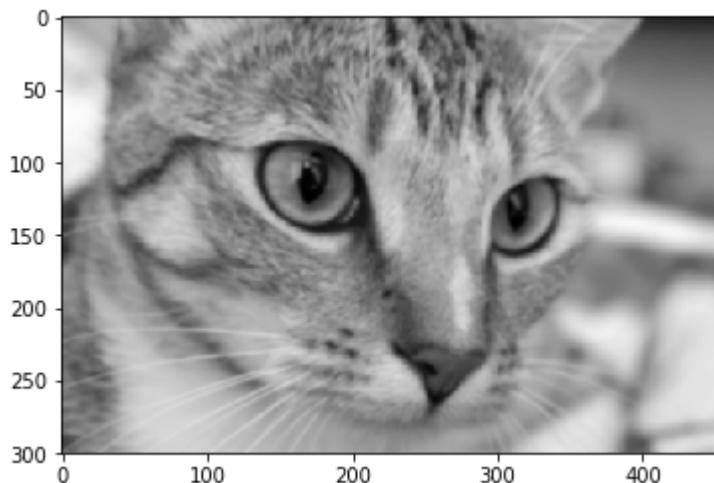
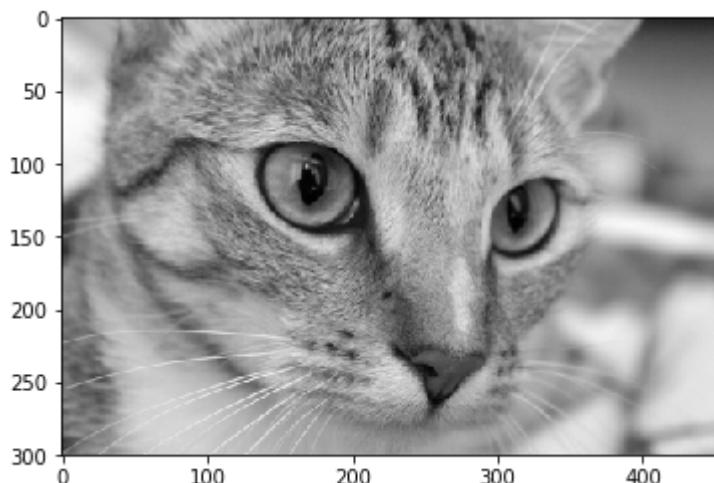


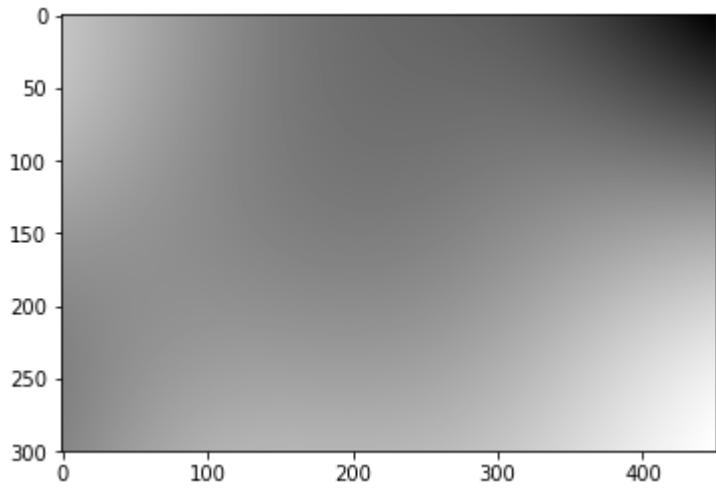
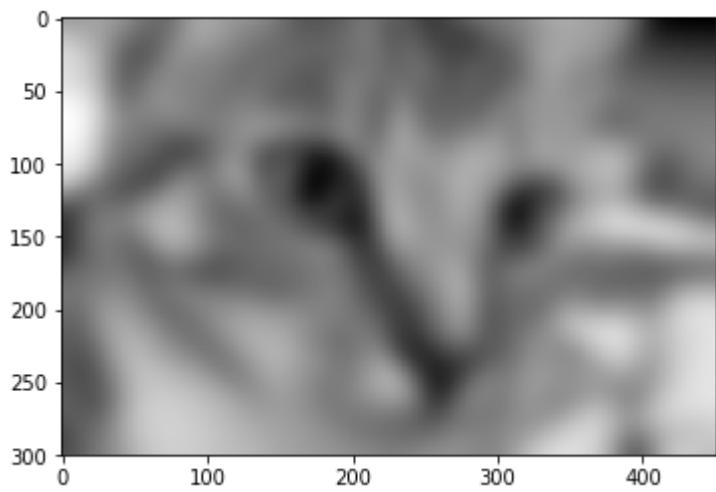
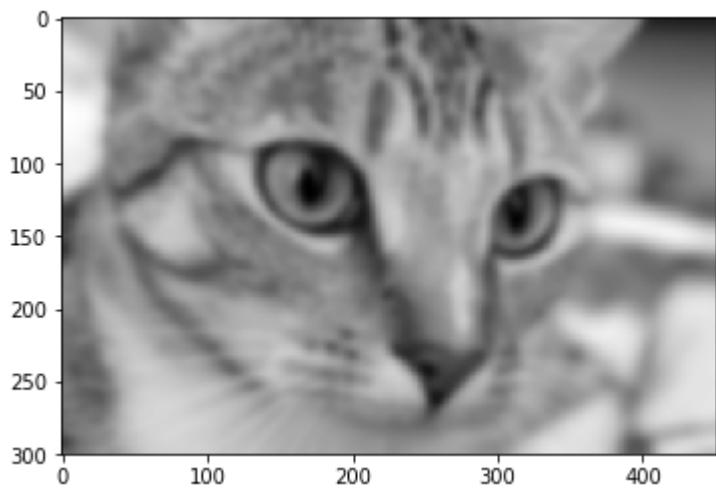
Så hvorfor snakker vi så mye om å ta gjennomsnittet av bilder? Vel, det å bruke en *blur*, eller å gjøre det mindre skarpt, er egentlig bare å ta gjennomsnitt av pikselverdiene, men man gjør det ikke med hele bildet, man gjør det med små biter av bildet. Du kan tenke på det litt som om det var et maleri der fargene ikke hadde tørket helt enda, det *blur*-filtre gjør er å gå over bildet med en liten pensel og blande ut små områder litt granne. Piksler vil altså ligne mer på nabopikslene sine etter et blur-filter er brukt.

Det finnes flere *blur*-filtere, men resultatene blir ganske like. Vi bruker `filters.gaussian` som eksempel. Vi sender inn et tall i tillegg til bildet som skal endres. Tallet som sendes inn er er størrelsen på området gjennomsnittet tas over: jo større tall, jo mer uskarpt blir bildet. Du kan tenke på det som å bruke en større pensel i malerimetaforen vår - jo større pensel, jo mer farger blir blandet sammen og jo mer uklart blir bildet.

In [43]:

```
1 from skimage.filters import gaussian
2
3 imshow(chelsea_gray)
4 show()
5
6 blurred = gaussian(chelsea_gray)
7 imshow(blurred)
8 show()
9
10 blurred3 = gaussian(chelsea_gray, 3)
11 imshow(blurred3)
12 show()
13
14 blurred10 = gaussian(chelsea_gray, 10)
15 imshow(blurred10)
16 show()
17
18 blurred100 = gaussian(chelsea_gray, 100)
19 imshow(blurred100)
20 show()
```

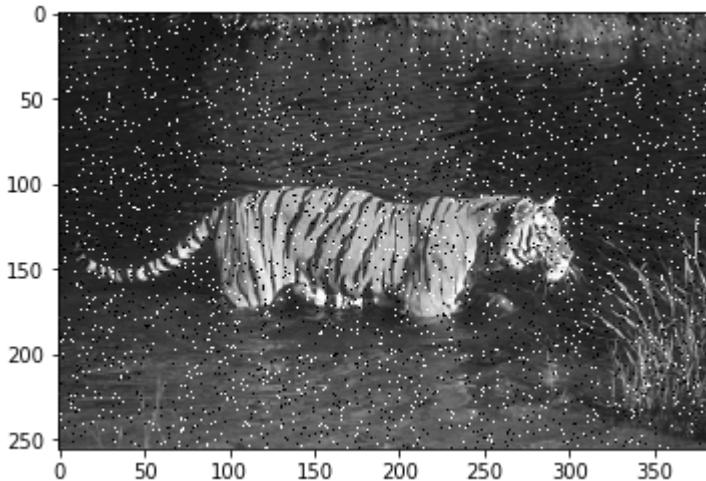




Men hvorfor er vi egentlig interessert i å gjøre et bilde *mer uskarpt*, altså *mindre skarpt*. Vi ødelegger jo kvaliteten på bildet! Om originalbildet vi begynner med er veldig klart og tydelig er det sånn ja, og da bruker vi gjerne ikke filtere. Men mange bilder vil ofte inneholde det vi kaller *støy*, som ikke egentlig skal være med i bildet, men de sniker seg inn på grunn av for eksempel dårlig lysforhold eller dårlig kamerautstyr, la oss se på et eksempel.

In [44]:

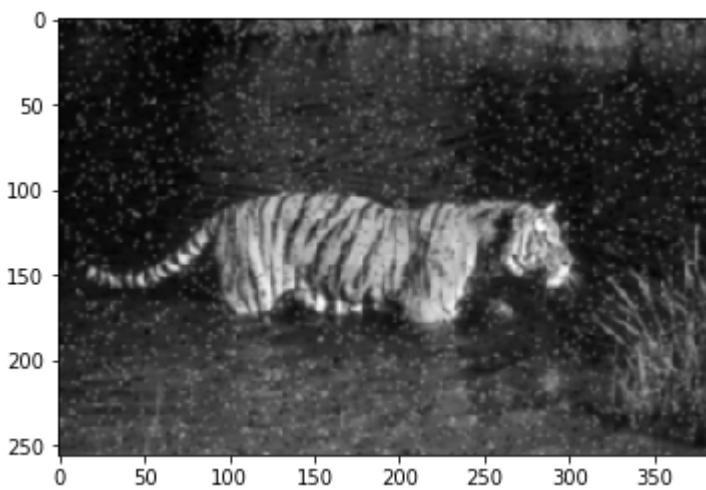
```
1 img = imread("fig/tiger.png")
2 imshow(img)
3 show()
```



Dette bildet av en tiger er egentlig skarpt og fint, men det har masse støy over seg. Denne typen støy kalles *salt-og-pepper-støy*, fordi det ser ut som noen har spredd salt og pepper utover bildet. Når vi ser på bildet ser det ut som det er masse støy, men merk at støyen stort sett er enkelte piksler, som betyr at dersom vi tar gjennomsnittet hviskes denne støyen ut. La oss prøve

In [45]:

```
1 blurred = gaussian(img, 1)
2 imshow(blurred)
3 show()
```



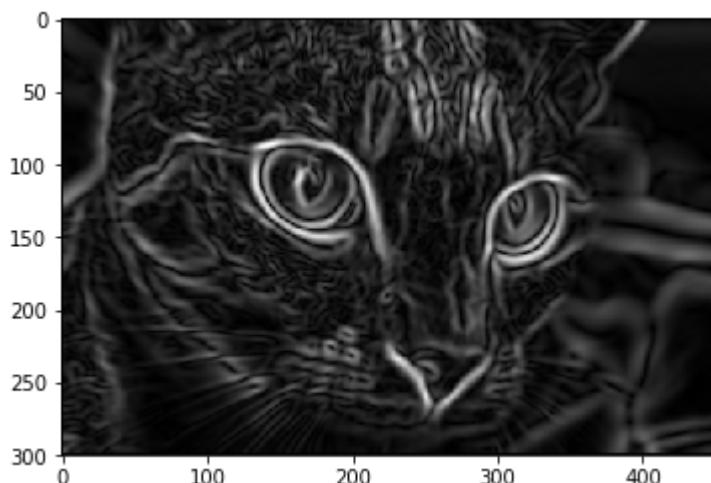
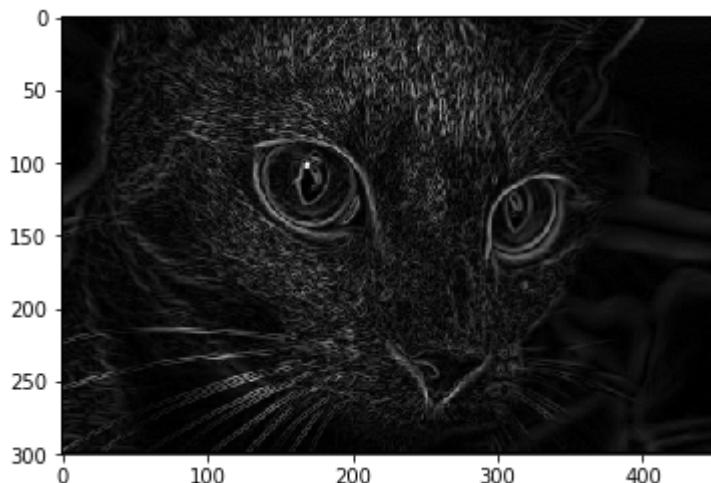
Vi ser at filteret fjerner støyen ganske bra. Om vi øker til 2 eller 3 blir støyen mer eller mindre helt borte, men samtidig blir resten av bildet veldig uklart. Det er altså en balanse mellom hvor mye støy vi vil fjerne og hvor klart bilde vi vil ha. Det finnes også andre måter å fjerne støy på fra et bilde, og hvilken metode vi bruker avhenger av hva slags bilde og hva slags støy vi har.

Et *blur-filter* brukes gjerne også som første steg i digital bildeanalyse, fordi de fleste bilder er altfor kompliserte for datamaskiner å forstå uansett, så vi kan like greit forenkle bildet litt og fjerne litt støy. Ta for eksempel bildet

av katten Chelsea. Når vi finner konturene finner vi masse små forandringer i pelsen til katten, men om først bruker en *blur* og så en *Sobel*, så ser vi mindre av disse små detaljene, mens de sterkere konturene rundt øynene og nesen blir tydligere.

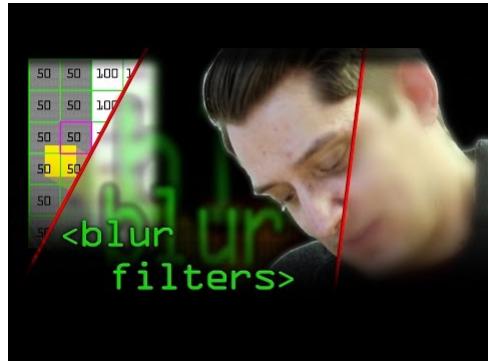
In [46]:

```
1 # Bruk bare Sobel filter
2 konturer = sobel(chelsea_gray)
3 imshow(sobel(chelsea_gray))
4 show()
5
6 # Bruk først Gaussisk, så Sobel filter
7 blurred = gaussian(chelsea_gray, 2)
8 konturer = sobel(blurred)
9 imshow(konturer)
10 show()
```



## Mer detaljer om filtere

Om du ønsker å gjøre et lite dypdykk i hvordan filtrene vi nå har gått igjennom fungerer, så er det ikke supervanskelig. *Computerphile* på Youtube, har laget en del videoer som beskriver disse i litt mer detalj. Du kan for eksempel begynne med denne videoen:



([http://www.youtube.com/watch?feature=player\\_embedded&v=C\\_zFhWdM4ic](http://www.youtube.com/watch?feature=player_embedded&v=C_zFhWdM4ic) ).

Link til videoen: How Blurs & Filters Work - Computerphile ([https://www.youtube.com/watch?v=C\\_zFhWdM4ic](https://www.youtube.com/watch?v=C_zFhWdM4ic))