Vaja 2: Filtriranje, detekcija robov, kotov, črt, krogov

Robotika in računalniško zaznavanje

2019/2020

Za vajo ustvarite mapo `vaja2`. Sem si prenesite tudi razpakirano vsebino datoteke `vaja2.zip` s spletne strani predmeta. Rešitve nalog boste pisali v *Matlab/Octave* skripte in jih shranili v mapo `vaja2`. Da uspešno opravite vajo, jo morate predstavili asistentu na vajah. Pri nekaterih nalogah so vprašanja, ki zahtevajo skiciranje, ročno računanje in razmislek. Odgovore na ta vprašanja si zabeležite v pisni obliki in jih prinesite na zagovor. Deli nalog, ki so označeni s simbolom ★ niso obvezni. Brez njih lahko za celotno vajo dobite največ 75 točk (zgornja meja je 100 točk kar pomeni oceno 10). Vsaka dodatna naloga ima zraven napisano tudi število točk. V nekaterih vajah je dodatnih nalog več in vam ni potrebno opraviti vseh.

Naloga 1: Konvolucija in filtriranje

Na predavanjih ste se spoznali z linearnim filtriranjem, katerega osrednji del je konvolucija. Najprej si oglejmo implementacijo konvolucije na 1D signalu. Konvolucija jedra $g(x)$ preko slike $I(x)$ je definirana z naslednjim izrazom

$$I_g(x) = g(x) * I(x) = \int_{-\infty}^{\infty} g(u)I(x-u)du, \tag{1}$$

oziroma v primeru diskretiziranih signalov

$$I_g(i) = g(i) * I(i) = \sum_{-\infty}^{\infty} g(j)I(i-j)dj. \tag{2}$$

Zelo lepo vizualizacijo konvolucije si lahko ogledate na spletni strani Wikipedia[1]. Ker je ponavadi naše jedro končne velikosti, zgornja vsota teče samo od *levega roba jedra* do *njegovega desnega roba*. Na primer, recimo, da je naše jedro velikosti $N+1+N$ elementov. V i-ti točki signala $I(i)$ se vrednost konvolucije v *Matlab/Octave* izračuna kot `I_g(i) = sum(I(i-N:i+N).*g)`. To pomeni, da center jedra 'položimo' na signal v i-ti točki in seštejemo produkt istoležnih elementov.

(a) Na roke izračunajte konvolucijo spodaj podanih signala in jedra ($k \star f$).

$f =$

0	1	1	1	0	0.7	0.5	0.2	0	0	1	0

$k =$

0.5	1	0.3

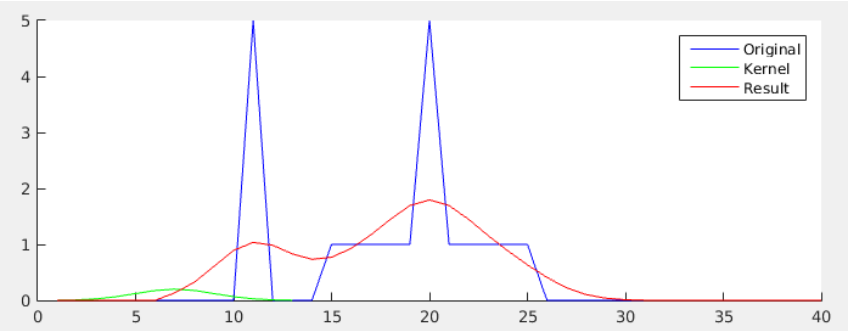
(b) Implementirajte funkcijo `simple_convolution.m`, ki za vhod vzame 1D signal I in simetrično jedro g velikosti $2N+1$, ter izračuna konvolucijo I_g. Zaradi enostavnosti

[1] `http://en.wikipedia.org/wiki/Convolution`

lahko začnete konvolucijo računati na mestu $i = N+1$ in končate na $i = length(I)-N$. To pomeni, da za prvih N elementov in zadnjih N elementov signala I kovolucije ne boste izračunali. Preverite implementacijo z uporabo skripte, ki naloži signal (datoteka `signal.txt`) in jedro (datoteka `kernel.txt`) ter naredi konvolucijo. Na isti sliki izrišite signal, jedro in rezultat konvolucije.

```
function Ig = simple_convolution(I, g)
N = (length(g) - 1) / 2;
Ig = zeros(1, length(I));
for i = N+1:length(I)-N
i_left = max([1, i - N]);
i_right = min([length(I), i + N]);
Ig(i) = sum(g .* I(i_left:i_right));
end
```

Vprašanje: Ali prepoznate obliko jedra g? Kakšna je vsota vseh elementov jedra?



(c) Ponovno izračunajte konvolucijo, vendar tokrat uporabite že vgrajeno funkcijo `Ig=conv(I, g, 'same')`.

Vprašanje: V dokumentaciji funkcije si poglejte kaj pomeni parameter `same`. V čem se rezultat razlikuje od `simple_convolution(I, g)`? Kaj je vzrok za to?

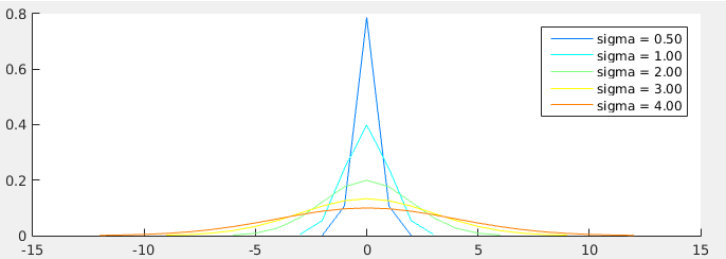
(d) Sedaj si oglejmo zelo pogosto uporabljano *Gaussovo jedro*, ki je definirano kot

$$g(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{x^2}{2\sigma^2}). \tag{3}$$

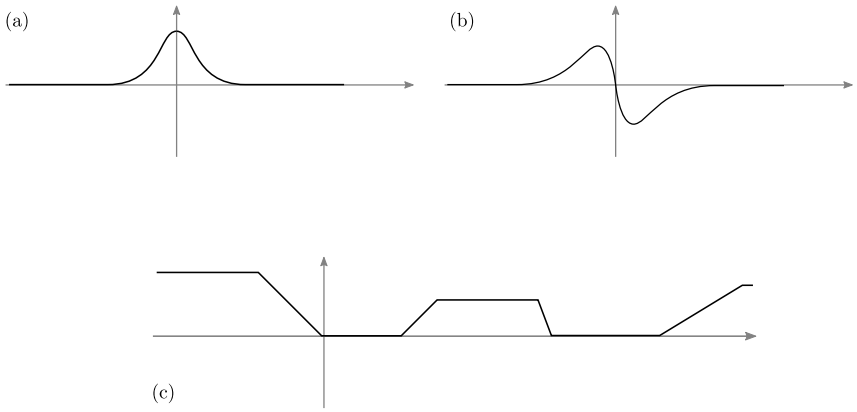
Pomembna lastnost Gaussovega jedra je, da njegova vrednost postane zelo majhna za $|x| > 3\sigma$. Zato je tudi jedro ponavadi velikosti $2 * 3\sigma + 1$. Naredite novo datoteko `gauss.m` in napišite funkcijo, ki ji podate parameter `sigma`, vrne pa Gaussovo jedro.

```
function [g, x] = gauss(sigma)
x = -round(3.0*sigma):round(3.0*sigma);
g = ... % <-- !!! calculate Gaussian kernel for values of x (in one line)
g = g / sum(g) ; % normalisation
```

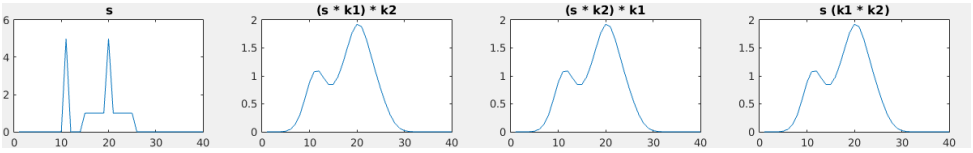
Generirajte jedro s $\sigma = 2$ in da je vsota njegovih elementov 1 in da je po obliki podoben jedru v datoteki `kernel.txt`. Na isto slko izrišite grafe za vrednost $\sigma = 0.5, 1, 2, 3, 4$.



(e) Na spodnji sliki sta prikazani dve jedri (a) in (b) ter signal (c). Skicirajte (ni se potrebno osredotočati na računsko natančnost, pomembno je razumevanje) rezultat konvolucje signala s posameznim jedrom. Po želji lahko konvolucijo s podobnimi dvemi signali tudi implementirate, vendar je pomembno predvsem razumevanje rezultata.



(f) Glavna prednost konvolucije napram korelaciji je asociativnost operacije. To nam omogoča, da več jeder najprej konvoluiramo med seboj, šele nato pa s sliko. Preverite to lastnost tako, da preberete signal iz datoteke `signal.txt` ter ga nato konvoluirate z dvema jedroma, najprej z Gaussovim jedrom k_1 s parametrom $\sigma = 2$, nato pa z jedrom $k_2 = [0.1, 0.6, 0.4]$. V drugem poskusu zamenjajte vrstni red jeder, nato pa preizkusite še najprej izračunati konvolucijo obeh jeder $k_1 * k_2$, rezultat pa konvoluirajte s signalom. Izrišite vse tri rezultate in jih primerjajte.



V drugem delu te naloge si boste pogledali, kako uporabiti konvolucijo nad 2-D signali za filtriranje slik. Na ta način lahko slike zgladimo, izostrimo ter odstranimo določene tipe šuma. Poleg tega si boste pogledali še filtre, ki niso implementirani kot konvolucija ter njihove značilnosti.

(g) Kot ste to slišali že na predavanjih, je pomembna lastnost Gaussovega jedra separabilnost v večih dimenzijah. To pomeni, da dobimo enak rezultat, če enkrat filtriramo z 2D jedrom ali če filtriramo najprej po eni in nato po drugi dimenziji. Torej lahko (počasno) nD filtriranje prevedemo na sekvenco hitrih 1D filtriranj.

Napišite funkcijo `gaussfilter.m`, ki generira Gaussov filter in ga nato aplicira na 2D sliko. Namesto funkcije `conv` uporabite njeno analogijo v 2D prostoru, funkcijo `conv2`. Upoštevajte dejstvo, da je jedro separabilno, zato generirajte 1D jedro, z njim najprej filtrirajte sliko po eni dimenziji (`Ib = conv2(I, g, 'same')`), nato pa po drugi dimenziji (`Ig = conv2(Ib, g' ,'same')`) preprosto tako, da jedro transponirate pred konvolucijo. Preizkusite filter v spodnji kodi, ki naloži sliko `lena.png`, jo spremeni v sivinsko in pokvari z Gaussovim šumom, in šumom sol-in-poper. Pokvarjene slike filtrira z vašim Gaussovim filtrom s $\sigma = 1$.

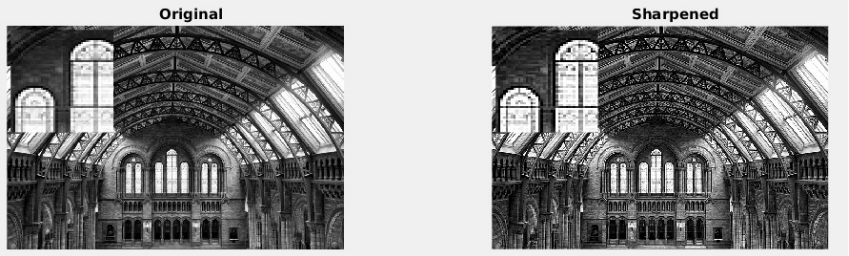
Vprašanje: Kateri šum Gaussov filter bolje odstrani?

```
A = rgb2gray(imread('lena.png'));
Icg = imnoise(A,'gaussian', 0, 0.01); % Gaussian noise
figure;
subplot(2,2,1); imshow(Icg); colormap gray;
axis equal; axis tight; title('Gaussian noise');
Ics = imnoise(A,'salt & pepper', 0.1); % Salt & pepper noise
subplot(2,2,2) ; imshow(uint8(Ics)); colormap gray;
axis equal; axis tight; title('Salt and pepper');
Icg_b = gaussfilter(double(Icg), 1);
Ics_b = gaussfilter(double(Ics), 1);
subplot(2,2,3) ; imshow(uint8(Icg_b)); colormap gray;
axis equal; axis tight; title('Filtered') ;
subplot(2,2,4) ; imshow(uint8(Ics_b)); colormap gray;
axis equal; axis tight; title('Filtered');
```

(h) ★ (5 točk) Še en uporaben filter je filter ostrenja, definiran je kot

$$k = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{4}$$

Implementirajte filter ter ga preizkusite na sliki iz datoteke `museum.jpg`. Kaj opazite? Poskusite filter na sliki zaporedno aplicirati večkrat ter opazujte spremembe.



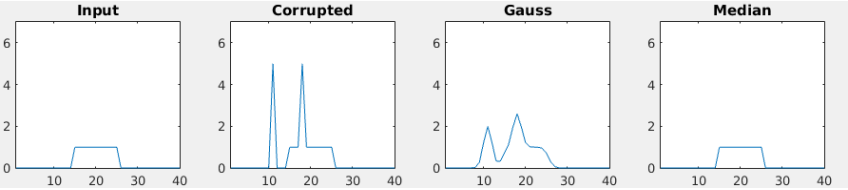
(i) Sedaj implementirajte nelinarni filter, ki ste ga obravnavali na predavanjih – *medianin filter*. Medtem, ko Gaussov filter izračuna lokalno uteženo povprečno vrednost v signalu, medianin filter lokalne vrednosti v signalu (t.j., vrednosti znotraj okna filtra) uredi po velikosti in vzame vrednost, ki je na sredini urejene množice (t.j., mediano). Implementirajte spodnji preprosti 1D medianin filter, ki za vhod vzame signal I in širino filtra W.

```
function Ig = simple_median(I, W)
 N = ceil((W-1)/2) ;
 Ig = zeros(1, length(I)) ;
 for i = N+1 : length(I)-N
    i_left = max([1, i-N]) ;
    i_right = min([length(I), i+N]) ;
    % sort the values and select the middle one
 end
```

(j) Uporabite spodnjo funkcijo, ki generira 1D stopnico, jo pokvari s šumom sol-in-poper, in nato zažene vaš Gaussov in medianin filter. Nastavite parametre filtrov tako, da bo rezultat filtriranja najboljši.

Vprašanje: Kateri filter se obnese bolje?

```
x = [zeros(1, 14), ones(1, 11), zeros(1, 15)]; % Input signal
xc = x; xc(11) = 5; xc(18) = 5; % Corrupted signal
figure;
subplot(1, 4, 1); plot(x); axis([1, 40, 0, 7]); title('Input');
subplot(1, 4, 2); plot(xc); axis([1, 40, 0, 7]); title('Corrupted');
g = gauss(1);
x_g = conv(xc, g, 'same');
x_m = simple_median(xc, 5);
subplot(1, 4, 3); plot(x_g); axis([1, 40, 0, 7]); title('Gauss');
subplot(1, 4, 4); plot(x_m); axis([1, 40, 0, 7]); title('Median');
```



(k) ★ (5 točk) Implementirajte 2-D verzijo medianinega filtra in jo preizkustite na sliki, ki je popačena z Gaussovim šumom ali šumom *sol in poper*. Primerjajte rezultate s filtriranjem z Gaussovim šumom za različne stopnje šuma, pa tudi za različne velikosti filtrov. Primerjajte (ocenite analitično) kolikšna je računska kompleksnost Gaussovega filtra in kakšna medianinega v $O(\cdot)$ notaciji, če v mediani uporabimo algoritem *quicksort* za sortiranje.

Zadnji segment v tej nalogi je posvečen odvajanju slik z uporabo konvolucije. Odvodi slike so pomembni kot indikator spremembe intenzitete, na njih temeljijo algoritmi za detekcijo črt in kotov. Numerično gre pri odvajanju za računanje razlike dveh zaporednih vrednosti signala, kar lahko zapišemo tudi kot konvolucijo z jedrom $[-1, 1]$. Ker jedro s sodim številom elementov povzroči premik signala za pol elementa, se v praksi uporablja liha verzija filtra $[-1, 0, 1]$.

(l) Slabost neposrednega odvajanja slike v neki točki je v tem, da zaradi prisotnosti šuma lokalne spremembe niso jasne in so tudi ocene odvodov šumne. V praksi zato sliko najprej zgladimo z majhnim filtrom, $I_b(x, y) = G(x, y) * I(x, y)$ in šele nato izračunamo odvod. Za glajenje pogosto uporabimo Gaussovo jedro, zaporedje odvajanja in konvolucije z Gaussovim jedrom pa lahko zaradi lastnosti konvolucije združimo v konvolucijo z odvodom Gaussovega jedra. Implementirajte funkcijo za izračun odvoda 1D Gaussovega jedra. Enačba odvoda Gaussovega jedra se glasi (za vajo lahko preverite analitično):

$$\frac{d}{dx}g(x) = \frac{d}{dx}\frac{1}{\sqrt{2\pi}\sigma}\exp(-\frac{x^2}{2\sigma^2}) \quad (5)$$

$$= -\frac{1}{\sqrt{2\pi}\sigma^3}x\exp(-\frac{x^2}{2\sigma^2}). \quad (6)$$

Jedro implementirajte v funkciji `gaussdx(sigma)`. Kot smo normalizirali diskretizirano Gaussovo jedro, je tudi priporočljivo, da normaliziramo odvod jedra. Ker je odvod Gaussovega jedra liha funkcija, ga normaliziramo tako, da je vsota absolutnih vrednosti elementov vsake od polovic enaka 1. Torej moramo jedro deliti z $0.5 \sum abs(g_x(x))$.

Slika je 2-D signal, zato se na njej lahko zračuna dva parcialna odvoda, odvod po x in odvod po y. Jedro odvoda po y je transponirano jedro odvoda po x.

(m) Lastnosti filtra lahko analiziramo preko tako imenovanega impulznega odziva filtra $f(x, y)$, ki je definiran kot konvolucija Dirac-ove $\delta(x, y)$ delte z jedrom $f(x, y)$: $f(x, y) * \delta(x, y)$. Zato najprej naredite sliko, ki ima vse vrednosti, razen centralnega elementa, enake nič:

```
I = zeros(25,25) ; imgImp(13,13) = 255 ;
```

Sedaj generirajte naslednji 1D jedri G in D:

```
sigma = 6.0;
G = gauss(sigma);
D = gaussdx(sigma);
```

Kaj se zgodi, če aplicirate naslednje operacije na sliko `I` (ali je zaporedje ukazov pomembno?):

(a) Najprej konvolucija z G in potem konvolucija z G'.

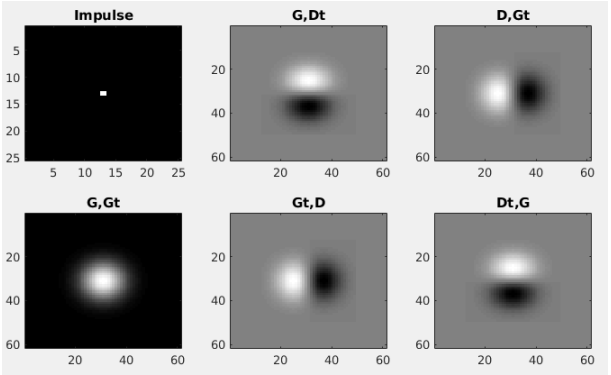
(b) Najprej konvolucija z G in potem konvolucija z D'.

(c) Najprej konvolucija D in potem konvolucija z G'.

(d) Najprej konvolucija G' in potem konvolucija z D.

(e) Najprej konvolucija D' in potem konvolucija z G.

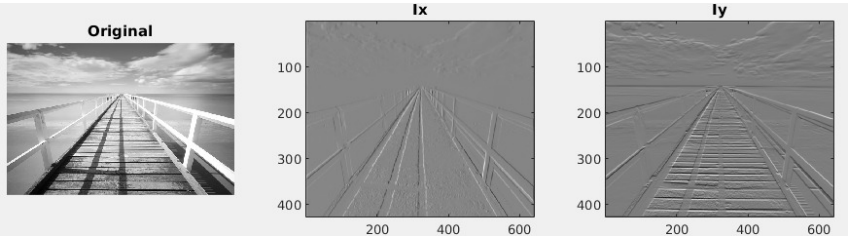
Izrišite si slike impulznih odzivov in uporabite `imagesc` za boljši prikaz.



(n) Implementirajte funkcijo, ki uporablja vaši funkciji `gauss` in `gaussdx`, in izračuna parcialni odvod slike po x in parcialni odvod po y. Funkcija naj zaradi morebitnega šuma v sliki za računanje odvoda po x uporabi kombinacijo filtra za odvod po x in glajenje po y, za odvod po y pa ravno obratno.

```
function [Ix, Iy] = image_derivatives(I, sigma)
 ...
```

Oba parcialna odvoda prikažite v oknu, pri tem pa bodite pozorni, da za prikaz uporabite funkcijo `imagesc`, saj so vrednosti odvodov lahko tudi negativne in je potrebno tako sliko pred prikazom najprej spraviti na ustrezen interval.



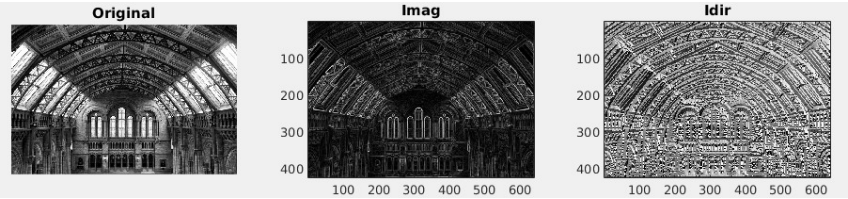
Naloga 2: Detekcija robov in kotov

Robove in kote iščemo tako, da analiziramo lokalne spremembe sivinskih nivojev v sliki, matematično to pomeni, da računamo *odvode slike*, kar ste si pogledali v prejšnji nalogi.

(a) Implementirajte funkcijo `gradient_magnitude`, ki za vhod vzame sivinsko sliko `I`, vrne pa matriko magnitud odvodov `Imag` in matriko kotov odvodov `Idir` vhodne slike. Magnitude izračunate po formuli $m(x,y) = \sqrt{(I_x(x,y)^2 + I_y(x,y)^2)}$, kote pa po formuli $\phi(x,y) = \arctan(I_y(x,y)/I_x(x,y))$. Zaradi učinkovitosti je pomembno, da uporabite matrično obliko operacij. Namig: Pri izračunu kotov se lahko ognete problemom deljenja z nič, če uporabite funkcijo `atan2`, ki sama poskrbi za take primere.

```
function [Imag, Idir] = gradient_magnitude(I, sigma)
 ...
```

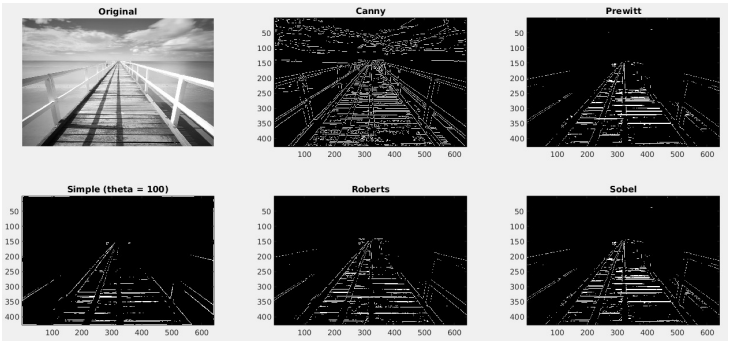
Rezultate vseh treh funkcij preizkusite na sliki `museum.jpg` in izrišite rezultate (uporabite funkcijo `imagesc`).



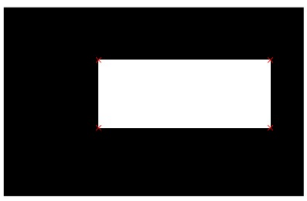
(b) Preprost detektor robov lahko dobimo že, če sliko magnitud gradientov `Imag` upragujemo s pragom `theta`. Napišite funkcijo `edges_simple`, ki za sivinsko vhodno sliko vrne binarno sliko robov `Ie`, ki označuje magnitude večje od predpisane pragovne vrednosti.

Implementirano funkcijo preizkusite na sliki `museum.jpg` in si izrišite rezultat za nekaj vrednosti parametera `theta`.

(c) Na predavanjih ste se spoznali s Cannyjevim detektorjem robov, ki je eden on najbolj razširjenih detektorjev robov v slikah. V okolju *Matlab/Octave* je v okviru funkcije `edge` poleg Cannyevega algoritma implementiranih še nekaj drugih algoritmov. Preizkusite jih na eni izmed priloženih slik.



(d) ★ (15 točk) Z odvodi lahko detektiramo tudi druge nizko-nivojske strukture v sliki, recimo kote. Na predavanjih ste obravnavali Harrisov algoritem, ki, podobno kot algoritmi za detekcijo črt, temelji na prvih odvodih slike. Implementirajte Harrisov algoritem v funkciji `harris`, pri tem pa bodite pozorni na učinkovitost implementacije (brez eksplicitnih zank). V pomoč naj vam bo priložena funkcija `nonmaxima_suppression`, ki služi za končno procesiranje odzivov in zaduši lokalno ne-maksimalne vrednosti. Pri testiranju funkcije si lahko pomagate tudi s primerjavo z rezultatom vgrajene funkcije `corners`, vendar vam čisto enakih rezultatov ni potrebno doseči.



Naloga 3: Detekcija črt in krogov

Ko imamo za sliko detektirane robove, lahko na podlagi jih detektiramo višje-nivojske strukture kot so črte/premice in krogi. Temu so namenjene različne izpeljanje Houghovega[2] algoritma. Najprej boste na kratko obnovili bistvo Houghovega pristopa za iskanje premic v sliki. Za več informacij poglejte zapiske s predavanj ter literaturo [2, 3], kakor tudi spletne aplikacije, ki demonstrirajo delovanje Houghovega transforma, npr., [1, 4].

[2]Hough se izgovori *Haf* in ne *Hug*.

Zamislimo si neko točko $p_0 = (x_0, y_0)$ na sliki. Če vemo, da je enačba premice $y = mx + c$, katere vse premice potekajo skozi točko p_0? Odgovor je preprost: vse premice, katerih parametra m in c ustrezata enačbi $y_0 = mx_0 + c$. Če si fiksiramo vrednosti (x_0, y_0), potem zadnja enačba opisuje zopet premico, vendar tokrat v prostoru (m, c). Če si zdaj zamislimo novo točko $p_1 = (x_1, y_1)$, njej prav tako ustreza premica v prostoru (m, c), in ta premica se seka s prejšnjo v neki točki (m', n'). Točka (m', c') pa ravno ustreza parametrom premice v (x, y) prostoru povezuje točki p_0 in p_1.

(a) ***Vprašanje:*** Na papirju rešite naslednji problem z uporabo Houghove transformacije: V 2D prostoru imamo podane štiri točke $(0, 0)$, $(1, 1)$, $(1, 0)$, $(2, 2)$. Določite enačbe premic, ki potekajo skozi vsaj dve točki.

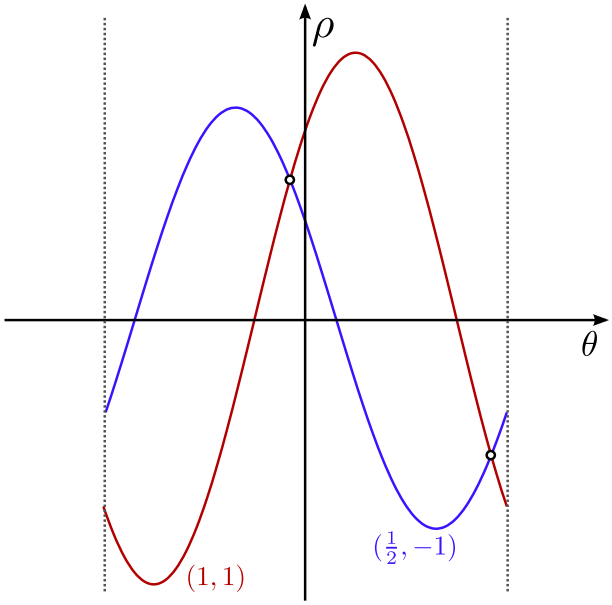
(b) ***Vprašanje:*** Na papirju z uporabo Houghovega algoritma določite parametre premice, ki poteka čez točki $(10, 1)$ in $(11, 0)$.

Če želimo poiskati vse premice v sliki z algoritmom, moramo naš postopek nekoliko preoblikovati, nimamo namreč na voljo neskončnega pomnilnika, ki bi ga potrebovali za izris premic, kot smo to naredi v zgornjih nalogah. Parametrični prostor (m, c) najprej kvantiziramo v matriko *akumulatorjev*. Za vsak slikovni element, ki je kandidat za rob v vhodni sliki, *narišemo* pripadajočo premico v prostoru (m, c) in povečamo vrednost akumulatorjev preko katerih ta premica poteka za 1. Vsi slikovni elementi, ki ležijo na isti premici v vhodni sliki bodo generirali premice v prostoru (m, c), ki se bodo sekale v isti točki in tako poudarile vrednost pripadajočega akumulatorja. To pomeni, da lokalni maximumi v (m, c) prostoru določajo premice, na katerih leži veliko slikovnih elementov v (x, y) prostoru.

V praksi je zapis premice v odvisnosti od m in n neučinkovit, še posebej, ko gre za navpične črte, saj takrat postane m neskončen. Temu problemu se preprosto ognemo tako, da premico parametriziramo s polarnimi koordinatami

$$x\cos(\theta) + y\sin(\theta) = \rho. \tag{7}$$

Postopek iskanja parametrov z akumulatorji je nespremenjen, razlika je samo v tem, da točka v (x, y) prostoru generira namesto premice sinusoido v prostoru (θ, ρ). Za točki $(1, 1)$ ter $(\frac{1}{2}, -1)$ sta ustrezni krivulji prikazani na spodnji sliki.

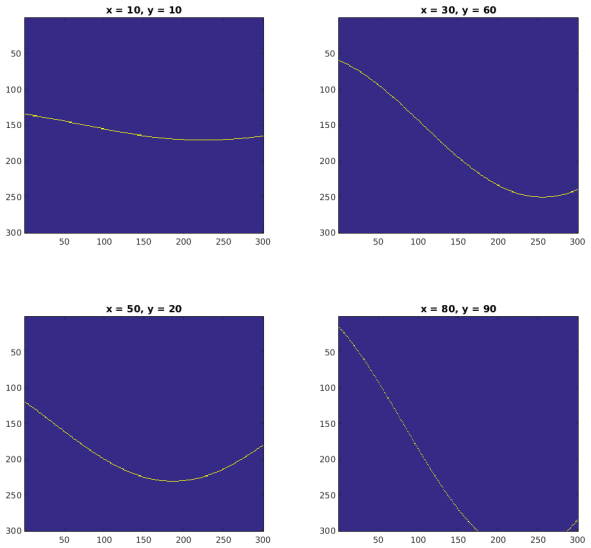


(c) Da bi bolje razumeli, kako se Houghov algoritem izvaja v praksi, ga boste delno implementirali. Pri tem vam bo v pomoč spodnja koda, ki za posamezno točko roba v akumulatorsko polje doda eno krivuljo.

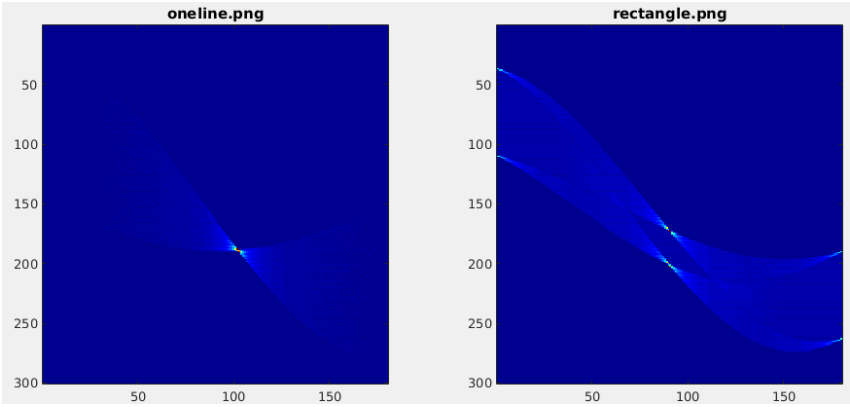
```
bins_theta = 300; bins_rho = 300; % Resolution of the accumulator array
max_rho = 100; % Usually the diagonal of the image
val_theta = (linspace(-90, 90, bins_theta) / 180) * pi; % Values of theta are known
val_rho = linspace(-max_rho, max_rho, bins_rho);
A = zeros(bins_rho, bins_theta);

% for point at (50, 90)
x = 50;
y = 90;
rho = x * cos(val_theta) + y * sin(val_theta); % compute rho for all thetas
bin_rho = round(((rho + max_rho) / (2 * max_rho)) * length(val_rho)); % Compute bins for rho
for i = 1:bins_theta % Go over all the points
    if bin_rho(i) > 0 && bin_rho(i) <= bins_rho % Mandatory out-of-bounds check
        A(bin_rho(i), i) = A(bin_rho(i), i) + 1; % Increment the accumulator cells
    end;
end;
imagesc(A); % Display status of the accumulator
```

Najprej v zgornji kodi spremenite položaj roba ter opazujte kako se spreminja končna krivulja.

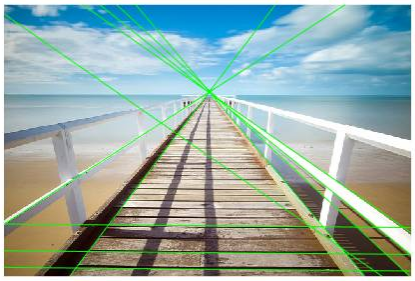


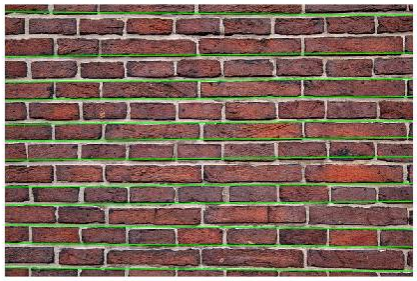
(d) Razširite zgornjo kodo, da iz slike robov (le-to dobite iz sivinske slike s funkcijo `edge`), inkrementalno izračunate vrednosti akumulatorske matrike tako, da za vsako točko ki je rob, dorišete ustrezno krivuljo. Končni akumulator nato prikažite na zaslonu. Za testiranje upoabite sintetični sliki iz datotek `oneline.png` in `rectangle.png`. Pri tem bodite pozorni na nastavitev vrednosti spremenljivke `max_rho`, ki se spreminja glede na velikost slike.



(e) Kot je bilo rečeno, algoritma za detekcijo premic ne boste implementirali v celoti, v nadaljevanju raje uporabite priloženo funkcijo `hough_find_lines`[3]. Naložite sliki iz datotek `skyscraper.jpg` in `pier.jpg`. Sliki spremenite v sivinski in na njej detekritajte robove z vgrajeno funkcijo `edges`. Prikažite rezultat ter preizkusite različne nabore parametrov algoritma detekcije robov ter detekcije črt, npr. spremenite parameter σ v detekciji črt ali število celic akumulatorja, da dobite rezultate ki so podobni ali boljši od rezultatov na spodnji sliki. Za izris rezultata uporabite priloženo funkcijo `hough_draw_lines`.

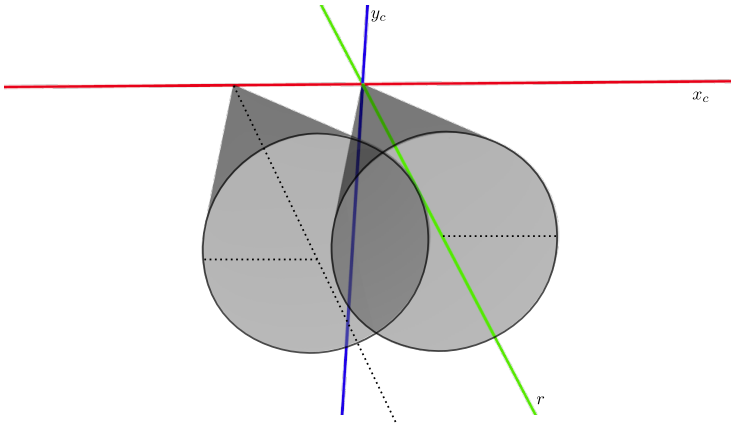
[3]V *Matlab/Octave* obstaja tudi vgrajena funkcija za detekcijo črt, vendar je njen rezultat drugače strukturiran in se ne more uporabiti skupaj z ostalimi funkcijami, ki so priložene.





Sedaj si bomo pogledali še, kako se Houghova transformacija uporabi za detekcijo krogov. Postopek je podoben, kot v primeru detekcije črt, le da imamo v splošnem pri krožnicah tri parametre: dva za center ter enega za radij.

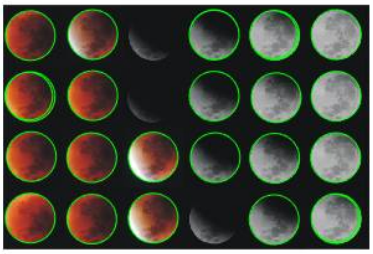
Zamislimo si neko točko $p_0 = (x_0, y_0)$ na sliki. Če vemo, da je enačba krožnice $r^2 = (x - x_c)^2 + (y - y_c)^2$, katere vse krožnice potekajo skozi točko p_0? Vse krožnice, katerih parametri x_c, y_c in r ustrezajo enačbi $r^2 = (x_0 - x_c)^2 + (y_0 - y_c)^2$. Če si fiksiramo vrednosti (x_0, y_0), potem zadnja enačba opisuje stožec v 3D prostoru (x_c, y_c, r). Ideja je prikazana na spodnji sliki.



(f) ***Vprašanje:*** Pogosto lahko obravnavamo problem iskanja krožnic, ko imamo radij že poznan. Kakšna je v tem primeru enačba, ki jo ena točka generira v parametričnem prostoru?

(g) ***Vprašanje:*** Na papirju rešite naslednji problem z uporabo Houghove transformacije: V sliki iščemo kroge s fiksnim radijem $r = 4$. Obravnavajte točki $A = (4, 8)$ in $B = (8, 4)$. Za vsako točko napišite enačbo v parametričnem prostoru in narišite pripadajočo krivuljo. Kaj lahko povemo o točkah A in B?

(h) Implementacija algoritma je podobna, kot pri detekciji premic, gre enostavno za drugačno enačbo. V nadaljevanju preizkusite priloženo funkcijo `hough_find_circles`, ki za podano sliko robov vrne koordinate in radie krogov. Poleg slike robov morate funkciji podati tudi enega ali več radiev. Algoritem preizkusite na slikah `eclipse.jpg` in `coins.jpg`. V prvem primeru eksperimentirajte z vrednostmi radija med 45 in 50 slikovnih elementov, v drugem pa preizkusite radije nekje med 85 in 90 slikovnih elementov. Za izris rezultata uporabite priloženo funkcijo `hough_draw_circles`.



Literatura

[1] C. Brechbühler. Interactive hough transform. `http://users.cs.cf.ac.uk/Paul.Rosin/CM0311/dual2/hough.htm`.

[2] W. Burger and M. J. Burge. *Digital Image Processing: An Algorithmic Introduction Using Java*. Springer, 2008.

[3] D. A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, 2002.

[4] M. A. Schulze. Circular hough transform. `http://www.markschulze.net/java/hough/`.