Vaja 2: Detekcija robov in vogalov

Vsako od vaj zagovarjate pri asistentu. Za uspešen zagovor morate rešiti vse naloge, razen tistih, ki so označene z ★. Če katera od nalog zahteva skiciranje ali ročno računanje, morate to pokazati na zagovoru. Programerske naloge morate rešiti samostojno in izvorno kodo ter rezultate predstaviti na zagovoru. Če kode, ki rešuje vaše naloge, ne boste poznali v detajle, lahko asistent vaš zagovor zavrne. V primeru zavrnitve zagovora, ga lahko ponovno opravljate z zamudo, a bo zato vaše končno število točk kaznovano s faktorjem **0.75**.

Za rešen obvezen del nalog lahko dobite največ **75** točk, dodatne naloge pa vam lahko prinesejo do **25** dodatnih točk. Za pomanjkljive rešitve ali slabo teoretično znanje vam asistent lahko odšteje točke.

Pogoste napake in nasveti
- Zamenjava x in y osi
- Napačen podatkovni tip: `float`, `uint8`
- Napačno definicijsko območje podatkov: [0,255], [0,1]
- Rešitev razvijajte na enostavnih primerih (lahko so sintetični)

---

1. Konvolucija in filtriranje

Operacija, ki se uporablja za linearno filtriranje signalov, se imenuje konvolucija. Za enostavnejše razumevanje bomo osnove obravnavali na 1D signalih, principi pa so večinoma podobni tudi za več dimenzij. Zvezna konvolucija slike $I(x)$ z jedrom $g(x)$ je definirana z naslednjim izrazom:

$$I_{g(x)} = g(x) \star I(x) = \int_{-\inf}^{\inf} g(u)I(x-u)du$$

v primeru diskretiziranih signalov pa:

$$I_{g(x)} = g(i) \star I(i) = \sum_{-\inf}^{\inf} g(j)I(i-j)$$

Vizualizacijo konvolucije si lahko ogledate na spletni strani https://en.wikipedia.org/wiki/Convolution. Ker je naše jedro ponavadi končne velikosti, zgornja vsota teče samo od *levega roba* do *desnega roba* signala. Denimo, da je naše jedro g velikosti $N + 1 + N$ elementov. V i-ti točki signala $I_{g(i)}$ se vrednost konvolucije v Pythonu izračuna kot:

```
I_g[i] = np.sum(I[i-N:i+N+1] * g)
```

To pomeni, da center jedra *položimo* na signal v i-ti točki in seštejemo produkt istoležnih elementov (isto operacijo bi lahko izvedli tudi s funkcijo `np.dot()`, ki implementira skalarni produkt).

a) Na roke izračunajte konvolucijo spodaj podanega signala in jedra $(k * f)$
$f = [0, 1, 1, 1, 0, 0.7, 0.5, 0.2, 0, 0, 1, 0]$
$k = [0.5, 1, 0.3]$

b) Implementirajte funkcijo `simple_convolution()`, ki za vhod vzame 1D signal I in simetrično jedro g velikosti $2N + 1$, ter izračuna konvolucijo I_g. Zaradi enostavnosti lahko začnete konvolucijo računati na mestu $i = N$ in končate na $i = \text{len}(I) - N - 1$. To pomeni, da za prvih N elementov in zadnjih N elementov signala I konvolucije ne boste izračunali. Preverite implementacijo z uporabo spodaj definiranega signala in jedra. Na isti sliki izrišite signal, jedro in rezultat konvolucije.

```
signal = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 5.0, 0.0,
0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 5.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

kernel = [0.0022181959, 0.0087731348, 0.027023158, 0.064825185,
0.12110939, 0.17621312, 0.19967563, 0.17621312, 0.12110939, 0.064825185,
0.027023158, 0.0087731348, 0.0022181959]
```

Ali prepoznate obliko jedra `kernel`? Kakšna je vsota vseh elementov jedra in zakaj je to pomembno?

c) Ponovno izračunajte konvolucijo in izrišite rezultat, vendar tokrat uporabite numpy funkcijo `np.convolve()`. V dokumentaciji poglejte, kaj pomeni parameter `'same'`. V čem se ta funkcije razlikuje od vaše implementacije `simple_convolution()`? Kaj je vzrok za razliko?

V praksi zelo pogosto uporabljano jedro je *Gaussovo jedro*, ki je definirano kot:

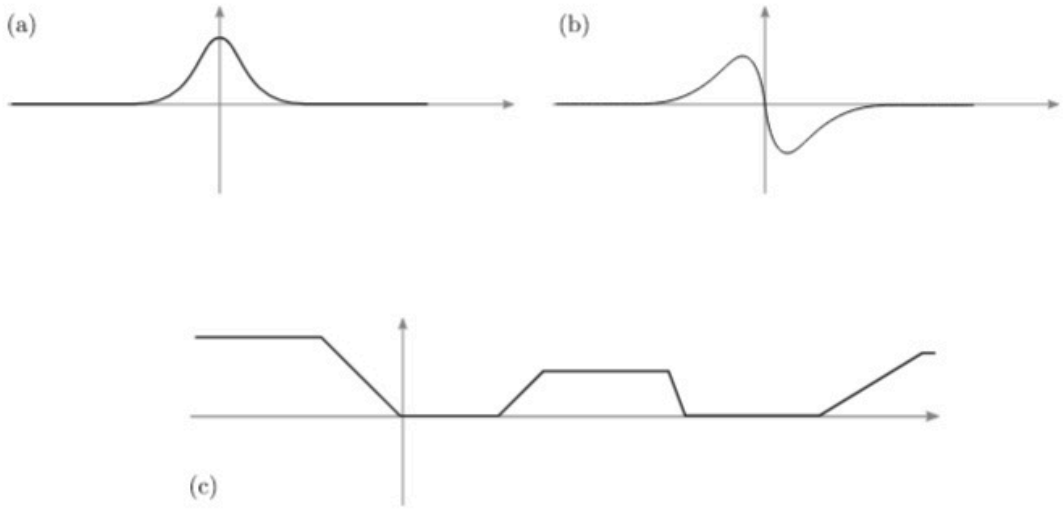
$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2}\right)$$

Gaussovo jedro je simetrično, njegova vsota pa je 1. Parameter σ določa *sploščenost* jedra, večje vrednosti σ proizvedejo bolj sploščena jedra. Pomembna lastnost Gaussovega jedra je, da za $|x| > 3\sigma$ njegova vrednost postane zelo majhna. Zato je tudi jedro ponavadi velikosti $2 \cdot 3\sigma + 1$.

d) Napišite funkcijo `simple_gauss`, ki ji podate parameter `sigma`, vrne pa Gaussovo jedro. Razpon vrednosti torej definirajte med -3σ in 3σ, središčni element je torej enak 0. Ker so vrednosti, ki so oddaljene več od 3σ, zelo majhne, omejite število elementov jedra na $2\lceil 3\sigma \rceil + 1$. Jedro normalizirajte, tako da ga delite z vsoto vseh elementov.

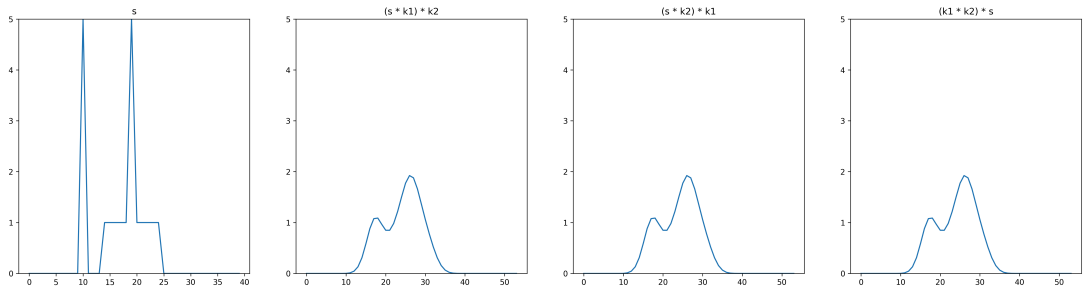
e) Izrišite jedro s $\sigma = 2$ in preverite, da je vsota njegovih elementov enaka 1 ter, da je po obliki podoben zgoraj definiranemu jedru `kernel`. Na isto sliko izrišite še Gaussova jedra z vrednostjo $\sigma = 0.5, 1, 2, 3, 4$.

f) 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 konvolucije signala s posameznim jedrom. Po želji lahko konvolucijo s podobnima dvema signaloma tudi implementirate, vendar je pomembno predvsem razumevanje rezultata.



g) Konvolucija je zelo podobna operaciji korelacija, ima pa konvolucija lepše matematične lastnosti, kot je na primer asociativnost. To nam omogoča, da več jeder najprej konvoliramo med seboj, nato pa rezultat apliciramo na signal. Preverite to lastnost tako, da uporabite zgoraj definirani signal `signal` ter ga nato konvolirate 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 konvolirajte s signalom. Izrišite vse tri rezultate in jih primerjajte.

Namig: Primerjavo signalov lahko izvedete s kombinacijo funkcij `np.all(np.isclose(a,b))`. Zaradi numeričnega računanja rezultati niso nujno točno enaki, jih pa znotraj neke tolerance (na primer $1e-8$) lahko obravnavamo kot identične.



Če želimo konvolucijo izvesti na 2D signalih, lahko to naredimo z ustreznim 2D filtrom. Obstaja pa tudi skupina jeder, ki so separabilna (ločljiva), kar pomeni, da lahko pridobimo isti rezultat tudi z zaporednimi 1D konvolucijami. Torej lahko počasnejše nD filtriranje prevedemo na sekvenco hitrih 1D filtriranj.

h) Napišite funkcijo `gauss_filter()`, ki generira Gaussov filter in ga nato aplicira na 2D sliko. Z uporabo funkcije `cv2.filter2D()` lahko na sliko apliciramo 1D ali 2D filter.

Preizkusite separabilnost Gaussovega jedra tako, da na sliki najprej uporabite rezultat funkcije `simple_gauss()`, nato pa na rezultatu še enkrat uporabite isto jedro, le da ga prej transponirate. Poskrbite, da ima jedro pred transpozicijo dve dimenziji, ne samo ene. Za to lahko uporabite funkcijo `np.expand_dims()`.
Nato izračunajte 2D obliko Gaussovega jedra tako, da 1D obliko matrično zmnožite z njegovo transponirano obliko (v numpy je to lahko `k_2d = k@k.T`). Uporabite 2D Gaussovo jedro na izhodiščni sliki in primerjajte oba rezultata.
Filter lahko preizkusite tako, da naložite dve pokvarjeni varianti slike `lena.png` – `lena_gauss.png` in `lena_sp.png`, kjer je bila prva varianta pokvarjena z Gaussovim šumom, druga pa s šumom sol-in-poper. Kateri šum Gaussov filter bolje odstrani?

Z ustreznim filtrom lahko slike tudi izostrimo. Primer takega filtra je na primer:

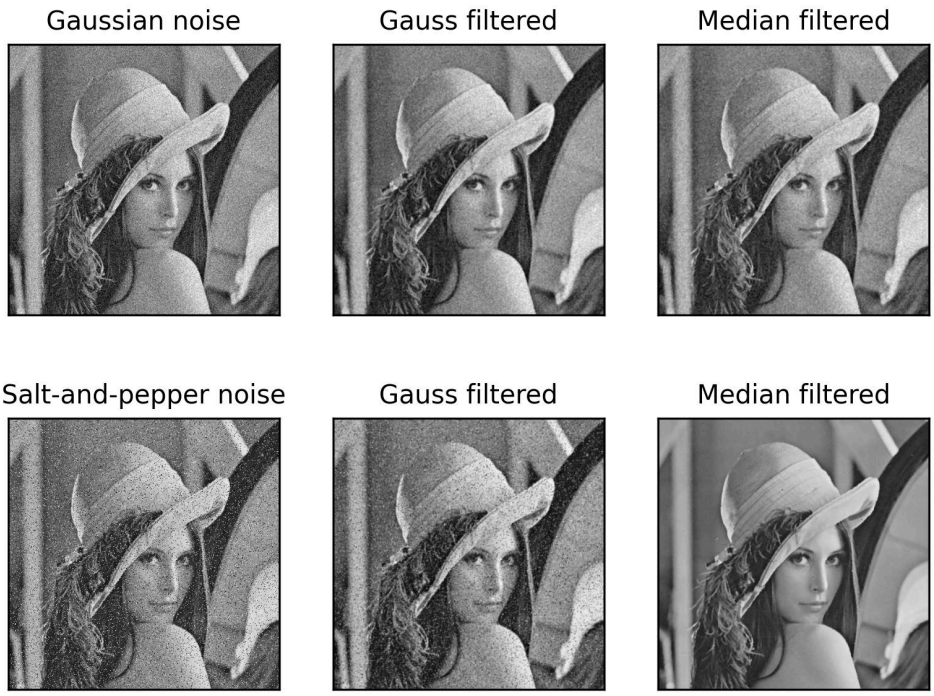
$$k = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{pmatrix} - \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

★ i) (5 točk) Implementirajte filter za ostrenje ter ga preizkusite na sliki iz datoteke `fox.jpg`. Kaj opazite? Poskusite filter na sliki zaporedno aplicirati večkrat ter opazujte spremembe.

j) Implementirajte funkcijo, ki uporabi medianin filter na 1D signalu. 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).
Uporabite spodnjo kodo, ki generira 1D stopnico, jo pokvari s šumom sol-in-poper, in nato ločeno uporabite Gaussov in medianin filter. Nastavite parametre filtrov tako, da bo rezultat filtriranja najboljši. Kateri filter deluje bolje in zakaj?

```
x = np.concatenate((np.zeros(14), np.ones(11), np.zeros(15)))
 # pokvarjeni signal
xc = np.copy(x)
xc[11] = 5
xc[18] = 5
```

★ k) (10 točk) Implementirajte 2D verzijo medianinega filtra in jo preizkusite na sliki, ki je popačena z Gaussovim šumom (`lena_gauss.png`) ali šumom sol-in-poper (`lena_sp.png`). Primerjajte rezultate filtriranja z Gaussovim jedrom in medianinim filtrom za različne velikosti filtrov. Primerjajte (ocenite analitično) kolikšna je računska kompleksnost Gaussovega filtra in kolikšna je kompleksnost medianinega filtra v $O(\cdot)$ notaciji, če v mediani za urejanje elementov uporabimo algoritem `quicksort`.



2. Odvodi slik

Za nadaljnjo analizo vsebine slik lahko uporabimo odvode. Odvodi nam služijo kot indikator hitre spremembe intenzitete, kar pogosto pomeni prisotnost robov v sliki. Algoritmi za detekcijo črt in vogalov temeljijo na odvodih. Numeričen približek odvoda na posamezni lokaciji v sliki je razlika med zaporednima vrednostima signala, kar lahko zapišemo kot jedro $[-1, 1]$. Zaradi simetrije pa se v praksi uporablja jedro z lihim številom elementov: $[-1, 0, 1]$.

Direktno računanje odvoda je lahko nenatančno zaradi prisotnosti šuma, ki ga operacija odvajanja dodatno poudari. Zaradi tega slike pred odvajanjem navadno zgladimo z manjšim filtrom in šele nato izračunamo odvod. Za glajenje po navadi uporabljamo Gaussovo jedro, odvod pa lahko izračunamo z uporabo odvoda Gaussovega jedra.

a) Implementirajte funkcijo za izračun odvoda 1D Gaussovega jedra. Enačba odvoda Gaussovega jedra je:
$\frac{d}{dx}g(x) = \frac{d}{dx}\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{x^2}{2\sigma^2}\right) = -\frac{1}{\sigma^3\sqrt{2\pi}}x\exp\left(-\frac{x^2}{2\sigma^2}\right)$

   Jedro implementirajte v funkciji `simple_gaussdx(sigma)`. Tako kot za navadno Gaussovo jedro, je priporočljivo normalizirati tudi odvod Gaussovega 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 $\frac{1}{2}\sum|g_x(x)|$.

Slike so 2D signali, kar pomeni, da lahko za njih izračunamo dva parcialna odvoda, torej odvod po x in odvod po y. Jedro odvoda po y je transponirano jedro odvoda po x. Svojo implementacijo razširite še na 2D signal.

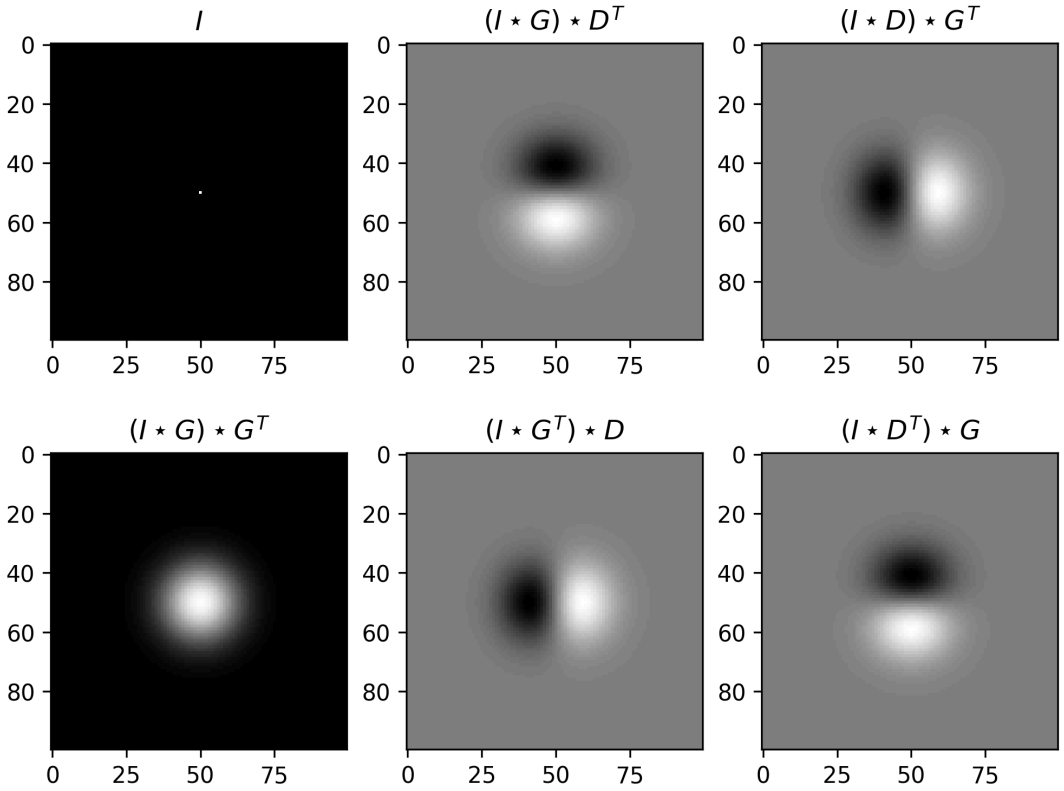
Lastnosti filtra lahko analiziramo preko tako imenovanega impulznega odziva filtra $f(x, y)$, ki je definiran kot konvolucija Dirac-ove $\delta(x, y)$ delte z jedrom $f(x, y)$: $f(x, y) * \delta(x, y)$.

b) Sestavite sliko, ki ima vse vrednosti, razen centralnega elementa, enake nič. Sedaj generirajte Gaussovo jedro G in njegov odvod D.

Kaj se zgodi, če aplicirate naslednje operacije na sliko I? Ali je zaporedje ukazov pomembno?

- Najprej konvolucija z G in potem konvolucija z G^T.
- Najprej konvolucija z G in potem konvolucija z D^T.
- Najprej konvolucija z D in potem konvolucija z G^T.
- Najprej konvolucija z G^T in potem konvolucija z D.
- Najprej konvolucija z D^T in potem konvolucija z G.

Izrišite si slike impulznih odzivov.



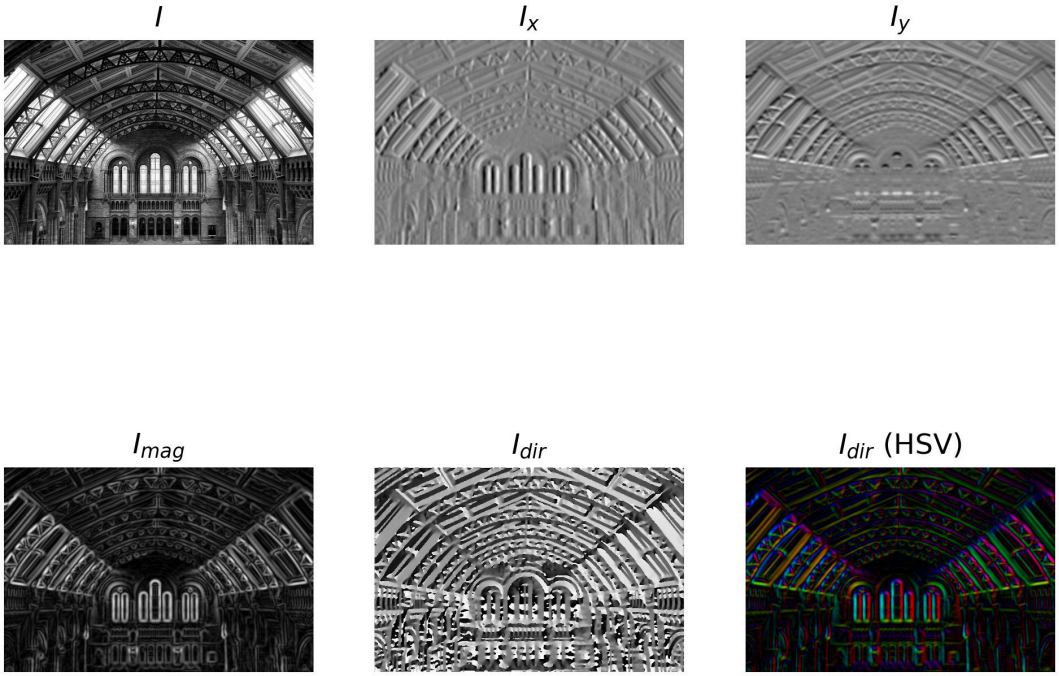
3. Detekcija robov in vogalov

Robove in vogale iščemo tako, da analiziramo lokalne spremembe sivinskih nivojev v sliki, matematično to pomeni, da računamo *odvode* slike.

★ a) (10 točk) Če želite bolje spoznati delovanje algoritma, implementirajte funkcijo `gradient_magnitude`, ki za vhod vzame sivinsko sliko I, vrne pa matriko magnitud odvodov I in matriko kotov odvodov I_{mag} vhodne slike. Magnitude izračunamo po formuli $m(x, y) = \sqrt{I_x(x, y)^2 + I_y(x, y)^2}$, kote pa po formuli $\Theta(x, y) = \frac{I_y(x,y)}{I_x(x,y)}$.

Zaradi učinkovitosti je pomembno, da uporabite matrično obliko operacij in ne `for` zanke. Namig: Pri izračunu kotov se lahko ognete problemu deljenja z nič, če uporabite funkcijo `np.arctan2()`, ki sama poskrbi za take primere. Rezultate vseh treh funkcij preizkusite na sliki `museum.jpg` in izrišite rezultate. Primer rezultatov je prikazan na Sliki 5.

Namig: Za boljšo predstavo si lahko kote izrišete tudi z barvami, primerna barvna preslikava (angl. *colormap*) je v tem primeru npr. HSV.



Slika 5: Prikaz odvodov slike `museum.jpg`. Desno spodaj so prikazani robovi, prebarvani preko prostora HSV, uteženi z magnitudo roba.

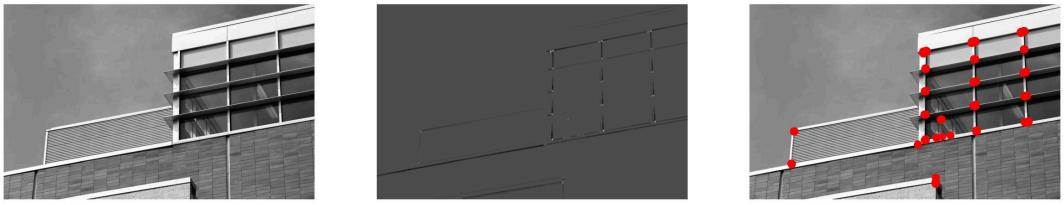
★ b) (5 točk) Preprost detektor robov lahko dobimo že, če sliko magnitud gradientov I_{mag} upragujemo s pragom ϑ. Napišite funkcijo `edges_simple`, ki za sivinsko vhodno sliko vrne binarno sliko robov I_e, ki označuje magnitude večje od predpisane pragovne vrednosti. Implementirano funkcijo preizkusite na sliki `museum.jpg` in si izrišite rezultat za nekaj vrednosti parametra ϑ.

c) Na predavanjih ste spoznali Cannyjev detektor robov, ki je eden on najbolj razširjenih detektorjev robov v slikah. V OpenCV knjižnici je implementiran v sklopu funkcije `cv2.Canny()`. Preučite nabor njegovih vhodnih parametrov in preizkusite delovanje na slikah zajetih s pomočjo spletne kamere ali v video posnetku. Izberite si poljuben predmet in nastavite parametre tako, da se bo njegov obris najbolje videl.

d) Z odvodi lahko detektiramo tudi druge nizko-nivojske strukture v sliki, recimo vogale. Na predavanjih ste obravnavali Harrisov algoritem, ki, podobno kot algoritmi za detekcijo črt, temelji na prvih odvodih slike. V OpenCV knjižnici je implementiran v sklopu funkcije `cv2.cornerHarris()`. Preučite nabor njegovih vhodnih parametrov in preizkusite delovanje na slikah zajetih s pomočjo spletne kamere ali v video posnetku.

Poskusite nastaviti parametre tako, da boste detektirali čimveč vogalov na npr. šahovnici, sudoku polju, itd.

Namig: Ker funckija `cv2.cornerHarris()` vrne tudi negativne in zelo velike vrednosti, je upragovanje enostavnejše, če vrednosti prej normaliziramo na interval $[0, 1]$.



Slika 6: Prikaz rezultatov funkcije `cv2.cornerHarris()` na sliki `building.jpg`.

4. Detekcija črt in krogov

Ko imamo za sliko detektirane robove, lahko na podlagi njih detektiramo višje-nivojske strukture kot so črte in krogi. Temu so namenjene različne izpeljanke Houghovega algoritma. Najprej boste na kratko obnovili bistvo Houghovega pristopa za iskanje premic v sliki. Za več informacij poglejte zapiske s predavanj, kot tudi spletni aplikaciji Hough transform demo, Circular Hough transform, ki demonstrirata delovanje Houghove transformacije.

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: vse premice, katerih parametra m in c ustrezata enačbi $y_0 = mx_0 + c$. Če fiksiramo vrednosti (x_0, y_0), potem zadnja enačba zopet opisuje 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', c'). Točka (m', c') zato ustreza parametrom premice v (x, y) prostoru in povezuje točki p_0 in p_1.

a) Na papir 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.

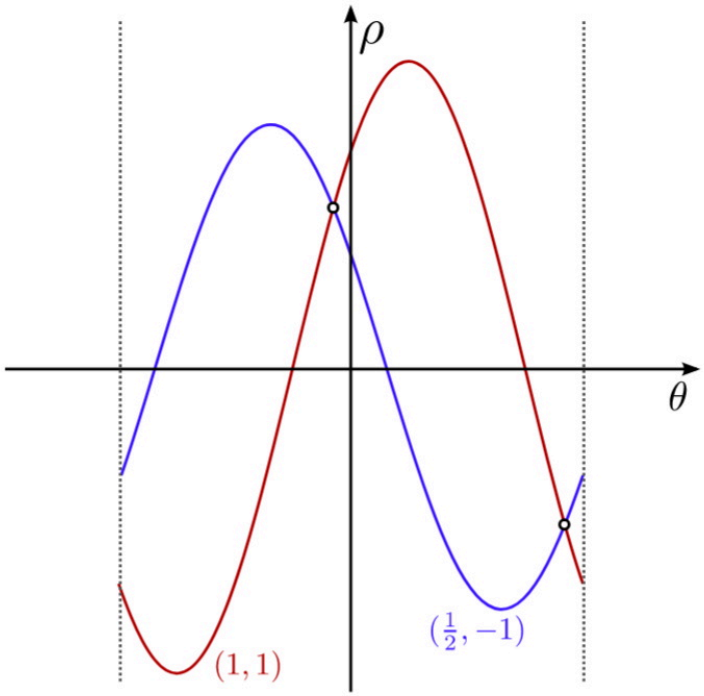
b) Na papirju z uporabo Houghovega algoritma določite parametre premice, ki poteka skozi točki $(10, 1)$ in $(11, 0)$.

V praksi je zapis premice v odvisnosti od m in c 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:

$$x \cos(\vartheta) + y \sin(\vartheta) = \rho$$

Parametra ρ in ϑ v polarnem prostoru opisujeta razdaljo do koordinatnega izhodišča in kot. Če želimo v naši sliki poiskati vse premice, pa moramo naš postopek še preoblikovati v diskretizirano obliko. To naredimo tako, da parametrični prostor (ρ, ϑ) kvantiziramo v t.i. akumulatorsko matriko. To pomeni, da prostor parametrov razdelimo na celice in nato povečujemo vrednosti celic, skozi katere potekajo krivulje, ki ustrezajo našim točkam.

V polarnem prostoru je postopek iskanja premic nespremenjen, razlika je le v tem, da točka iz slikovnega prostora (x, y) ne predstavlja več premice, ampak sinusoido. Za točki $(1, 1)$ ter $\left(\frac{1}{2}, -1\right)$ sta ustrezni krivulji prikazani na spodnji sliki.

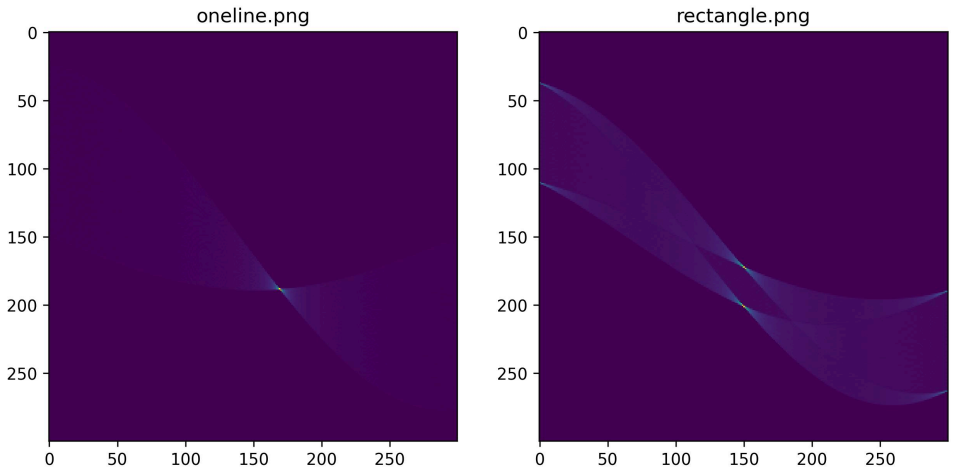


Vsi slikovni elementi, ki ležijo na isti premici v vhodni sliki bodo generirali krivulje v prostoru (ρ, ϑ), ki se bodo sekale v isti točki in tako poudarile vrednost pripadajoče celice v akumulatorski matriki. To pomeni, da lokalni maksimumi v prostoru (ρ, ϑ) določajo premice, na katerih leži veliko slikovnih elementov v (x, y) prostoru.

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

```python
# Resolucija akumulatorskega polja:
bins_theta = 300
bins_rho = 300

max_rho = 100  # Navadno je to diagonala slike

val_theta = np.linspace(-90, 90, bins_theta) / 180 * np.pi # vrednosti
theta
val_rho = np.linspace(-max_rho, max_rho, bins_rho)
A = np.zeros((bins_rho, bins_theta))

# Primer za točko (50, 90)
x = 50
y = 90
rho = x * np.cos(val_theta) + y * np.sin(val_theta) # Izračunamo rho za
vse vrednosti theta
bin_rho = np.round((rho + max_rho) / (2 * max_rho) * len(val_rho))

for i in range(bins_theta):
    if bin_rho[i] >= 0 and bin_rho[i] <= bins_rho - 1:
        A[int(bin_rho[i]), i] += 1
```

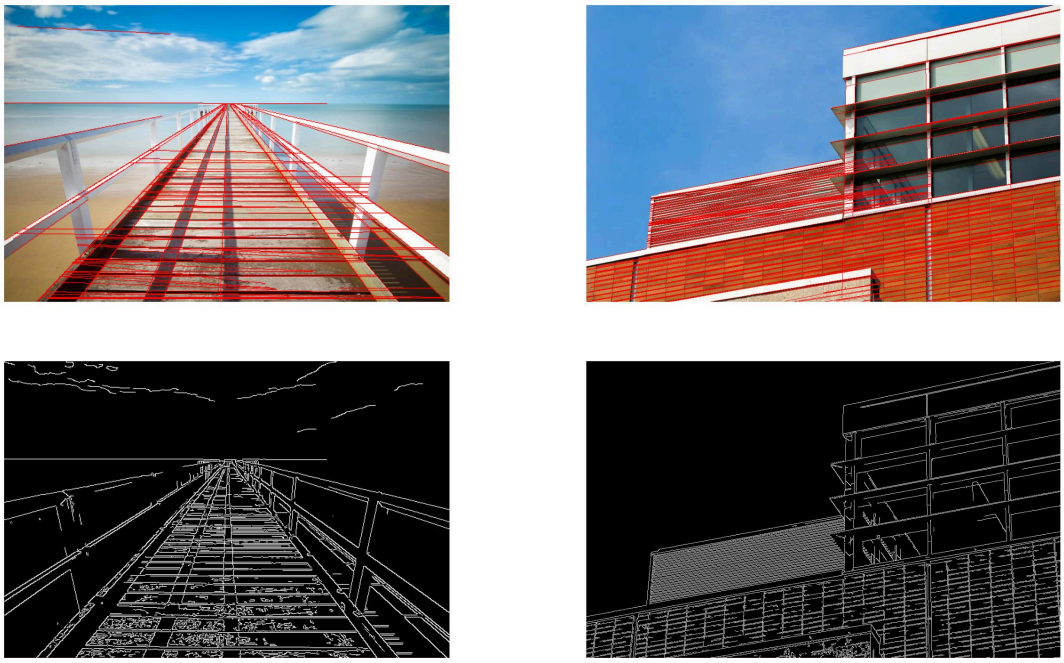
c) Razširite zgornjo kodo tako, da iz slike robov (le-to lahko pridobite z uporabo Cannyjevega detektorja robov), inkrementalno izračunate vrednosti akumulatorske matrike tako, da za vsako točko, ki predstavlja rob, dorišete ustrezno krivuljo (t.j. povečate ustrezne celice za 1). Končni akumulator nato prikažite na zaslonu. Za testiranje uporabite 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. Slik ne smete uporabiti direktno, ampak morate na njih najprej pognati detekcijo robov, npr. `cv2.Canny()`. Primer rezultata je prikazan na Sliki 8.



Slika 8: Akumulatorski matriki za sliki `oneline.png` in `rectangle.png`.

Algoritma za detekcijo premic ne boste implementirali v celoti, zato nadaljevanju uporabite že implementirano OpenCV funkcijo `cv2.HoughLinesP()`. Ta funkcija implementira probabilistično izvedbo detekcije črt, ki ne vrača premic, ampak segmente črt.

d) Naložite sliki iz datotek `building.jpg` in `pier.jpg`. Sliki spremenite v sivinski in na njej najprej detektirajte robove s pomočjo funkcije `cv2.Canny()`, nato pa detektirajte črte z uporabo funkcije `cv2.HoughLinesP()`. Prikažite rezultate 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 rezultatom na spodnji sliki. Za izris črt uporabite funkcijo `cv2.line()` znotraj `for` zanke.



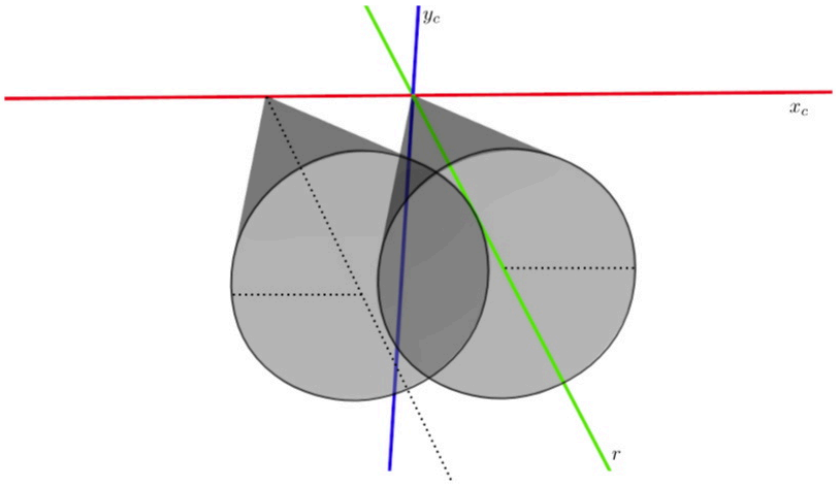
Slika 9: Zgoraj: rezultati funkcije `cv2.HoughLinesP()` na slikah `pier.jpg` in `building.jpg`. Spodaj: rezultati Cannyjevega algoritma na istih slikah.

e) Ko dosežete zadovoljive rezultate, preizkusite delovanje svojega algoritma še na slikah, pridobljenih iz spletne kamere ali video posnetka. Delovanje detekcije črt preizkusite s pomočjo npr. sudoku polja, črt v zvezku itd.

5. Hough transform za iskanje krogov

Koncept Hough transforma lahko uporabimo tudi za detekcijo drugih oblik, ne samo črt. Postopek je zelo podoben kot za črte, se pa spremeni število parametrov. V primeru 2D krogov moramo določiti dva parametra za center in enega za radij.

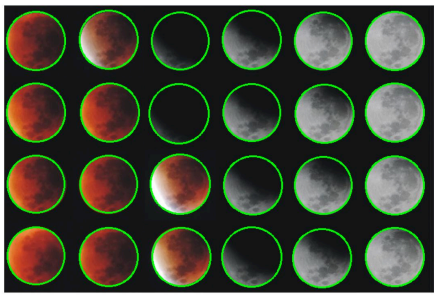
Zamislimo si neko točko $p_0 = (x_0, y_0)$ na sliki. Če vemo, da je enačba krožnice $r^2 = (x - x_c)^2 + (y - y_c)^2$, katere krožnice potekajo skozi točko p_0? Vse krožnice, katerih parametri x_c, y_c in r ustrezajo enačbi $r^2 = (x_0 - x_c)^2 + (y_0 - y_c)^2$. Če fiksiramo vrednosti (x_0, y_0), potem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



f) 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) 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) Preizkusite OpenCV implementacijo `cv2.HoughCircles()` s pomočjo katere detektirajte kroge 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 krogov uporabite funkcijo `cv2.circle()`.



i) Preizkusite delovanje svojega algoritma še na živi sliki, t.j. sliki pridobljeni s pomočjo spletne kamere. Delovanje detekcije krogov preizkusite s pomočjo npr. kovancev na mizi.