UNIVERZA V LJUBLJANI

FS Fakulteta za strojništvo

Strojni vid

Drago Bračun

Ljubljana, 2025

STROJNI VID

Drago Bračun

Laboratorij za mehatroniko, proizvodne sisteme in avtomatizacijo
Fakulteta za strojništvo, Univerza v Ljubljani Ljubljana, 2025

Recenzenta: prof. dr. Primož Podržaj in prof. dr. Roman Kamnik

Lektorirala: Simona Špolad, prof. slov. in univ. dipl. lit. komp.

Grafična obdelava besedila in slik: avtor

© Univerza v Ljubljani, Fakulteta za strojništvo, 2025

Brez soglasja založnika in avtorja je prepovedano vsakršno razmnoževanje ali prepis v katerikoli obliki.

Kataložni zapis o publikaciji (CIP) pripravili v Narodni in univerzitetni knjižnici v Ljubljani

COBISS.SI-ID 253603331
ISBN 978-961-7187-24-3 (PDF)

Kazalo

Simboli

Spremenljivka	Enota	Opis
a	mm	razdalja med točko optične preslikave v objektivu in objektom
B	byte	modra barva (ang. blue)
b	mm	razdalja med točko optične preslikave v objektivu in sliko
C	se	število stolpcev (ang. columns) na sliki
C_{PSF}	μm	premer PSF
c_u	se	center slike v u-smeri (presek optične osi s slikovnim zaznavalom)
c_v	se	center slike v v-smeri (presek optične osi s slikovnim zaznavalom)
d	mm	premer objektiva
Δ	/	odstopek
f	mm	goriščna razdalja objektiva
f_u	se	goriščna razdalja v u-smeri
f_v	se	goriščna razdalja v v-smeri
f_N	Hz	mejna frekvenca vzorčenja; Nyquistova frekvenca
f_s	Hz	frekvenca vzorčenja
φ	rad	kot objekta glede na optično os
G	byte	zelena barva (ang. green)
$k_{1,2}$	/	parametri radialne distorzije
k_c	/	vektor parametrov distorzije (k_1, k_2, p_1, p_2)
K	se	matrika kamere (3 x 3), notranji parametri
λ	nm	valovna dolžina svetlobe
O	/	objekt
p	$*$	kazalec na podatke v spominu (ang. pointer)
r	se	radij glede na center slike
r_{ij}	rad	elementi rotacijske matrike
R	rad	rotacijska matrika
R	se, byte	število vrstic na sliki; tudi rdeča barva (ang. red)
$\vec{r}_{\mathrm{c}}$	mm	koordinate točke v kamerinem koordinatnem sistemu, vektor $(x_{\mathrm{c}}, y_{\mathrm{c}}, z_{\mathrm{c}})$
$\vec{r}_{\mathrm{g}}$	mm	koordinate točke v globalnem koordinatnem sistemu, vektor $(x_{\mathrm{g}}, y_{\mathrm{g}}, z_{\mathrm{g}})$
S	/	slika
σ_{u}	/	standardni odklon v u-smeri
σ_{v}	/	standardni odklon v v-smeri
T	mm	premik izhodišč koordinatnih sistemov, vektor $(t_{\mathrm{x}}, t_{\mathrm{y}}, t_{\mathrm{z}})$
u	se	u-koordinata na sliki; tudi indeks smeri
U	se	U-os slike (v smeri stolpcev)
v	se	v-koordinata na sliki; tudi indeks smeri
V	se	V-os slike (v smeri vrstic)
x	mm	x-koordinata v 3D-kartezičnem koordinatnem sistemu; tudi indeks smeri
y	mm	y-koordinata v 3D-kartezičnem koordinatnem sistemu; tudi indeks smeri
z	mm	z-koordinata v 3D-kartezičnem koordinatnem sistemu; tudi indeks smeri
X	mm	X-os v 3D-kartezičnem koordinatnem sistemu
Y	mm	Y-os v 3D-kartezičnem koordinatnem sistemu
Z	mm	Z-os v 3D-kartezičnem koordinatnem sistemu

Indeksi

$_c$	indeks, ki označuje, da se navedeni parameter nanaša na kamero
$_d$	indeks, ki označuje prisotnost distorzije
$_i$	indeks zanke for
$_j$	indeks zanke for
$_n$	normirana koordinata
$_u$	indeks u-smeri na sliki (po stolpcih)
$_v$	indeks v-smeri na sliki (po vrsticah)

Okrajšave

AD	analogno digitalna pretvorba
API	programska oprema, ang. application programming interface
B	modra barva
CCD	tehnologija izdelave slikovnih zaznaval, ang. charge-coupled device
CMOS	tehnologija izdelave integriranih vezij, ang. complementary metal–oxide–semiconductor
CPU	računalniški procesor
DC	enosmerni tok, ang. direct current
FOV	vidni kot kamere, ang. field of view
FFT	hitra Fourierjeva transformacija
GigE	mrežna digitalna komunikacija, ang. gigabit ethernet
GPU	grafični procesor
GKS	globalni koordinatni sistem
GUI	grafični uporabniški vmesnik
HD	format slike (okrajšava "high definition", tj. 1080p oz. 1920 x 1080 se)
KKS	kamerin koordinatni sistem
KTL	vrsta barve in posebna tehnologija nanosa, elektroforetski katodni kovinski premaz
LED	polprevodniško svetilo, ang. light-emitting diode
MOD	najkrajša razdalja med objektom in objektivom, kjer sliko še izostrimo, ang. minimum object distance
PLC	industrijski krmilnik, ang. programmable logic controller
PSF	porazdelitev svetlosti po optični preslikavi točkastega vira svetlobe, ang. point spread function
ROI	območje interesa, ang. region of interest
SDK	programsko razvojno okolje, ang. software development kit
se	slikovni element, ang. pixel
SKS	slikovni koordinatni sistem
USB	digitalna komunikacija, ang. universal serial bus
VDC	napetost enosmernega toka
VGA	format slike (640 x 480 se), grafični standard v računalništvu

Poglavje 1

Uvod

Tematike, ki jih obravnava področje strojnega vida (ang. machine vision, computer vision), so obsežne in odvisne od področja uporabe. V tem delu se omejimo na področje avtomatizacije in naloge, ki iz njega izhajajo. Primer teh je npr. lokalizacija, kjer ugotavljamo, kje se nahaja neki sestavni del ter kako je zasukan, da ga nato s kakšnim aktuatorjem, npr. robotom, poberemo (slika 1.1). Pogosto gre za naloge preverjanja prisotnosti izdelka, ali je pravilen izdelek v pravem trenutku vstavljen v ustrezno gnezdo. Poseben sklop nalog predstavlja kontrola kakovosti, kjer preverjamo, ali so kontrolirani izdelki znotraj specifikacij. Primer tega je kontrola dimenzij, kontrola napak barvanja, kontrola zvarov itd. Slikovni sistemi imajo veliko dodano vrednost, kjer nadomestijo človeka v monotonih kontrolnih nalogah, kot je primer vizualne kontrole, kjer mora operater pregledati več tisoč izdelkov na izmeno pod neonskimi lučmi.

Osnova strojnega vida so slikovni sistemi, torej sistemi, v katerih je glavni nosilec informacije slika. Verjetno je že vsak slišal pregovor "Ena slika je vredna tisoč besed". Slikovni sistem je v osnovi merilni sistem, ki je sestavljen iz več sklopov. Prvi in najpomembnejši je svetlobni del, kjer je glavni nosilec informacije zakodiran v svetlobni vzorec – sliko. Temu sledi električni del, kjer sliko digitaliziramo in jo s pomočjo sodobnih digitalnih komunikacij prenesemo v spomin računalnika. Zadnji del predstavlja programska oprema za obdelavo slike, kjer s slike izluščimo iskano informacijo in jo ovrednotimo. Za uspešnost in zanesljivost delovanja so pomembni vsi gradniki, še najmanj dela je z digitalizacijo in s prenosom slike v računalnik, saj to kot gotovo rešitev ponujajo proizvajalci kamer. Naša naloga je, da domislimo svetlobni del slikovnega sistema, da uporabimo različne svetlobne efekte, tako da je v zajeti sliki iskana informacija in nič drugega. To je prvi in ključni korak v razvoju slikovnega sistema. Če je na zajeti sliki poleg opazovanega objekta v ozadju vidno še pol delavnice, potem je obdelava slike izredno težavna in nezanesljiva. V nasprotnem, če je na sliki samo opazovani objekt in ozadje uniformne in bistveno drugačne barve od objekta, potem smo že v fazi razvoja slikovnega dela naredili tudi pol obdelave slike.

V praksi se pogosto srečamo z vprašanjem, ali je mogoče za praktičen problem razviti slikovni sistem in ali bo ta zadovoljivo deloval. Hitro oceno lahko pripravimo na osnovi vizualne presoje z lastnimi očmi. Ko pogledamo opazovani objekt in njegovo ozadje, je človeku v trenutku jasno, ali je izdelek ustrezen ali ni. Ali je nalepka nalepljena postrani, je izdelek opraskan, zvar ni prevarjen, ali je pravilen sestavni del na tekočem traku itd. Če nam vizuelno takoj "pade v oči", ali je vse ustrezno ali ne, potem bomo takšno nalogo tudi enostavno izvedli s slikovnim sistemom. Zavedati pa se moramo, da vse naloge niso tako enostavne ter zahtevajo veliko znanja in izkušenj, da jih izvedemo. Primer tega so natančne dimenzijske meritve izdelkov, pri katerih preverjamo, ali je izdelek znotraj kontrolnih meja, zagotavljamo sledljivost, preverjamo merilno negotovost itd.

Zelo pomembna naloga je strega izdelkov v vidno polje kamere. Pogosto gre za sočasen razvoj avtomata, ki izvaja delovne operacije, in slikovnih sistemov, ki skrbijo za kontrolo izdelkov in pravilnost izvajanja delovnih operacij. Če gre za avtomat z več gnezdi, na katerih se izvajajo delovne operacije v nekem zaporedju, je to relativno jasna naloga, saj imajo avtomati vgrajene mehanizme, ki premikajo izdelke med gnezdi in lahko vizualni kontroli namenimo posebno gnezdo. Mnogokrat kamero namestimo nad tekoči trak itd. V mnogo primerih razvijamo podajalni mehanizem, ki vključuje zalogovnik, podajanje in sortiranje izdelkov ter je bistveno bolj kompleksen in dražji za izvedbo kot sam slikovni sistem. Pomembni razmisleki pri načrtovanju podajalnega sistema so ponovljivost pozicioniranja izdelkov, kako naj bodo izdelki vpeti, da ne bodo motili slikovnega sistema, čistost okolja, vibracije, svetlobne motnje iz ozadja in varnost.

Slika 1.1: Lokalizacija. Slika prikazuje primer robotskega pobiranja izdelkov z obešal na vozičku. Kljub natančnemu pozicioniranju vozička in natančni izdelavi nosilcev vsak izdelek na nosilcih stoji malo drugače. Pri robotskem pobiranju bi s prijemalom podrsali po izdelku in ga opraskali. Z lokalizacijo natančno določimo lego izdelka in popravimo točko robotskega prijemanja (vir: TPV, d. d.).

Z vidika organizacije proizvodnje moramo premisliti, kje v proizvodnem procesu bo umeščena kontrolna naprava s slikovnim sistemom, kako bo umeščen transport izdelkov do nje in od nje, kakšni so razpoložljivi časi za transport in kontrolo ter kdo bo odgovorni skrbnik naprave, ki presoja pravilnost delovanja in popravlja nastavitve, če so spremembe v procesu ali izdelku. Zagotoviti moramo izobraževanje oseb za vzdrževanje slikovnih sistemov ter jih opremiti z ustreznimi odgovornostmi in s pooblastili. Zagotoviti moramo tudi nadomestne dele, ki morajo biti na zalogi v primeru odpovedi komponent, kot so osvetljevala, kamere in komunikacijske enote. To je pomembno predvsem tam, kjer okvara slikovnega sistema pomeni zastoj celotnega proizvodnega procesa in s tem povezane stroške.

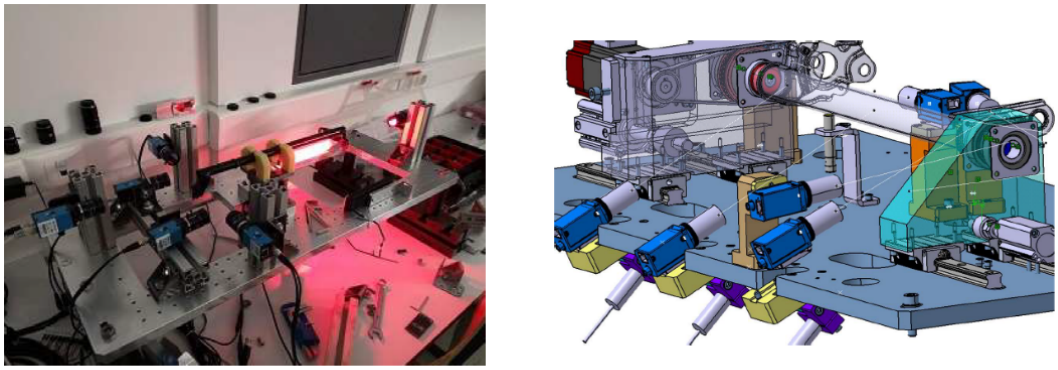
Sodobni trendi vodijo v smeri povezovanja vseh naprav v industrijska omrežja. Z vidika sledljivosti proizvodnje se na podatkovne strežnike podjetja tipično prenašajo statistični podatki o rezultatih kontrole, kot je število pregledanih kosov na izmeno, število dobrih ali slabih. Mnogokrat se celo za vsak izdelek zapisujejo serijska številka, datum in čas kontrole ter izid kontrole ali celo izvorne slike. Z vidika vzdrževanja sistema pogosto zagotovimo oddaljen dostop, tako da se odgovorna oseba ali vzdrževalec iz kontrolne sobe ali svoje pisarne poveže na računalnik, ki izvaja obdelavo slike, in preveri delovanje sistema ali popravi kakšno nastavitev. To je še posebej pomembno tam, kjer je slikovni sistem vgrajen v avtomatizirano celico in bi fizični dostop do njega pomenil zaustavitev celotne celice.

Razvoj začnemo s pridobivanjem ključnih specifikacij o namenu strojnega vida, kje bo vgra-

jen, kakšen bo transport izdelkov itd. Organiziramo tim strokovnjakov, ki nam pomagajo pri razvoju – od konstrukterja, programerja, električarjev itd. Pogosto naredimo laboratorijski prototip, na katerem testiramo optični sistem, zajem in obdelavo slike. Če smo uspešni na laboratorijskem prototipu, potem tim kot celota zasnuje in detajlira celoten sistem, ki gre v nadaljevanju v izdelavo, montažo in nastavljanje (slika 1.2).

Razvoju sledita preizkušanje in validacija delovanja sistema kot celote. V tej fazi nastavljamo tako optični in mehanski sistem kot programsko opremo. Pogosto moramo tudi kaj predelati. Za namen testiranja programske opreme je praktično shraniti veliko slik, programsko opremo pa napisati tako, da bere slike iz nekega direktorija in izvaja obdelavo. Tako lahko na velikem številu slik v kratkem času testiramo vpliv variacije nastavitev sistema.

Če povzamemo, razvoj sistemov strojnega vida zahteva znanja različnih področij. Najprej s proizvodnega inženirstva in kakovosti ter kam in kako umestiti slikovni sistem. Z navedenih področij izhajajo tudi ključne specifikacije in zahteve za slikovne sisteme. Potrebujemo znanja mehatronike, s poudarkom na optiki, za pravilen razvoj optičnega dela slikovnega sistema ter računalništva in programiranja obdelave slike. V tem učbeniku bomo obravnavali predvsem slednje: načrtovanje svetlobnega dela slikovnega sistema, izbiro optike in programiranje obdelave slike. Privzamemo, da študenti dobijo osnovna znanja organizacije in kakovosti pri drugih, temu namenjenih predmetih. Kdor bo želel pridobiti poglobljena znanja s področja strojnega vida, lahko poseže po tuji literaturi, npr. *Handbook of Machine Vision* [1] ali pa *Robotic Vision and Control* [2]. Prva podaja poglobljeno znanje s področja strojnega vida, druga pa vsebuje tudi teme s področja krmiljenja, industrijske in mobilne robotike ter je primerna izbira za vse, ki načrtujete integracijo slikovnih sistemov na različnih področjih robotike.



Slika 1.2: Testno preizkuševališče, na katerem testiramo optični sistem, zajem in obdelavo slike (a). Izdelano je iz univerzalnih komponent, s katerimi enostavno spreminjamo in nastavljamo lego kamere in osvetlitve glede na objekt. Če smo uspešni na laboratorijskem prototipu, potem zasnujemo in detajliramo celoten sistem (b).

Poglavje 2

Zgradba slikovnih sistemov

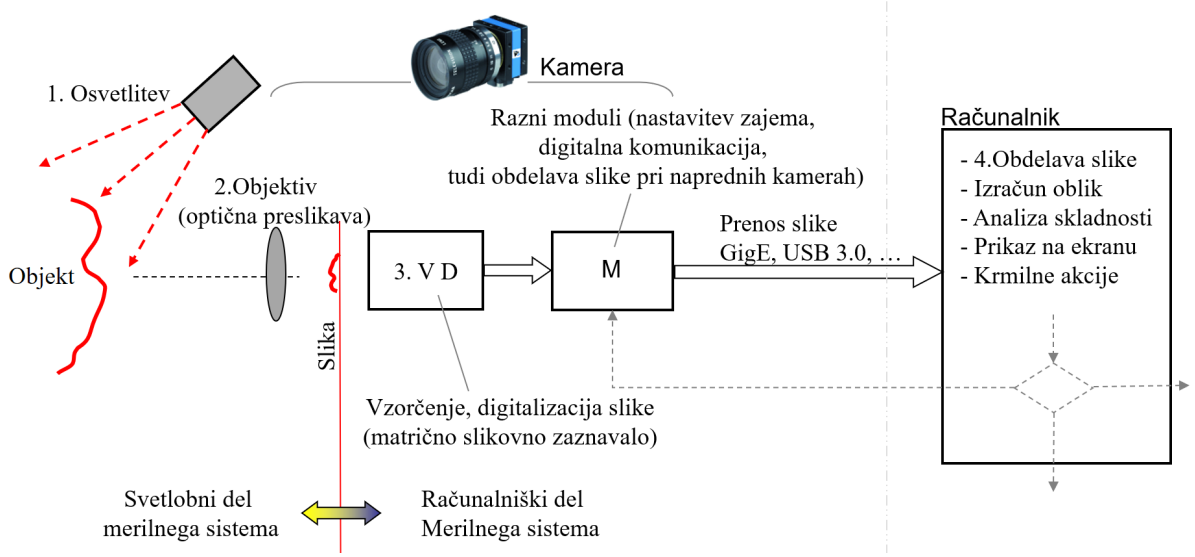
Ključni gradnik slikovnih sistemov je svetlobni (optični) del, ki vsebuje osvetlitev objekta in objektiv za zajem svetlobe, odbite z objekta, ter ustvarjanje slike na slikovnem zaznavalu kamere (glej sliko 2.1). Z osvetlitvijo je podobno kot z oljem pri hidravliki – če ga ni, ne deluje. Tudi slikovni sistemi brez svetlobe ne delujejo. Izbira vira osvetlitve in osvetljevalnih efektov je ključni korak v načrtovanju slikovnega sistema in mu bo v nadaljevanju posvečena posebna pozornost. Pod osvetljevalne efekte mislimo predvsem, kam postaviti vir osvetlitve glede na objekt in kam kamero glede na osvetlitev in objekt. S svetlobnimi efekti dosežemo vidnost opazovanih detajlov, predvsem pa to, da je na sliki tista informacija, ki je za nas pomembna.

Drugi pomemben gradnik je zajem slike in prenos v spomin računalnika. Tu je ključni gradnik kamera z matričnim slikovnim zaznavalom, ki vpadli svetlobni vzorec – sliko digitalizira in pretvori v numerični zapis svetlosti. Tukaj je pomembno razumeti zgradbo slikovnega zaznavala, ključne parametre, na osnovi katerih izračunamo dimenzije vidnega polja, najmanjše vidne detajle, določene z ločljivostjo, ter bitnost digitalizacije.

Sodobne kamere imajo vgrajene module za korekcijo napak pri zajemu slike, razne filtre, imajo napredno proženje zajema slike, predvsem pa vgrajeno digitalno komunikacijo za prenos slike v spomin računalnika za obdelavo v naslednjem koraku. Sodobne digitalne komunikacije omogočajo velike količine prenosa podatkov. Primer takšne komunikacije je gigabitna mrežna komunikacija (GigE), ki omogoča prenos do 60 slik visoke ločljivosti (HD) v sekundi. Mrežne komunikacije omogočajo povezavo večjega števila kamer preko mrežnih stikal na en računalnik, kar je zelo pomembno pri razvoju kompleksnih kontrolnih naprav z veliko kamerami. Podobne hitrosti prenosa dosegamo tudi z USB3.0-digitalno komunikacijo. Na strani računalnika ima vsaka kamera pripadajoči gonilnik, ki sprejme podatke in jih ustrezno strukturira ter zapiše v spomin računalnika. Zelo pomembno je, da do teh podatkov (slik) enostavno dostopamo z različnimi programskimi orodji, kot so OpenCV ali gstreamer v operacijskih sistemih Linux. Večinoma proizvajalci kamer nudijo tudi razvojna okolja (SDK), kjer v C++ ali Pythonu dostopamo do slik, naredimo obdelavo in programsko opremo po želji.

Na tem mestu omenimo še napredne kamere, ki imajo že vgrajen procesor za obdelavo slik in naloženo programsko opremo za enostavno obdelavo slike. Takšna kamera ima signalno vhodno/izhodno enoto, preko katere komunicira z drugimi napravami, npr. s PLC-jem, ki upravlja z delovanjem celotne kontrolne naprave in mu podaja informacijo o izidu kontrole (dober/slab).

V končni fazi sledi obdelava slike, kjer izluščimo iskane informacije in jih ustrezno interpretiramo. V ta namen imamo na voljo mnogo različnih orodij. Kdor ima veselje do programiranja, lahko uporabi OpenCV – odprtokodno knjižnico za obdelavo slik in v C++ razvije programsko opremo za obdelavo slike in vse, kar je še dodatno potrebno. Lažja pot je uporaba OpenCV v Pythonu ali Image processing toolbox knjižnice v Matlabu. Kdor se želi izogniti neposrednemu

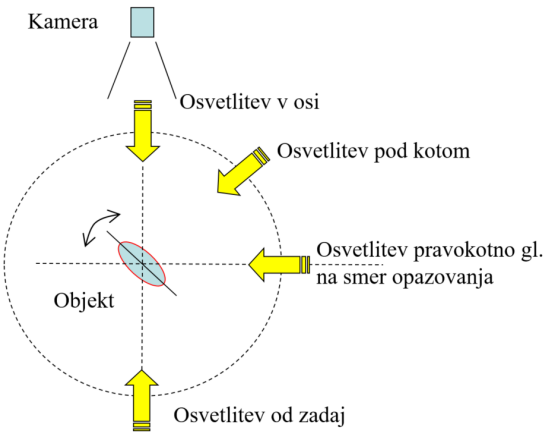


Slika 2.1: Splošna struktura slikovnega sistema

programiranju, lahko uporabi že gotove visokonivojske programske pakete za obdelavo slike. Ti močno skrajšajo čas razvoja, omogočajo velike možnosti testiranja, predvsem pa ne zahtevajo programerskega znanja. Primer takšnega je npr. RoboRealm, ki nudi veliko število funkcij obdelave in zgolj s klikanjem po funkcijah gradi proces, tj. zaporedje operacij obdelave slike, in takoj vidimo tudi rezultat obdelave. Vgrajeno imajo tudi veliko možnosti povezovanja in integracije v kompleksne sisteme.

2.1 Osvetlitev

Osvetlitev objekta ter prostorska umestitev osvetlitve in kamere glede na opazovani objekt sta ključni za doseganje svetlobnih efektov, s katerimi dosežemo dobro vidnost vsega, kar nas zanima, nepomembno in kar nas moti pri obdelavi slike, pa zatemnimo, tako da ni vidno. Dodatno si pri tem pomagamo še z izbiro ustreznih barvnih kombinacij, kjer barvo osvetlitve in ozadja prilagodimo barvi objekta, na kamero pa namestimo ustrezne barvne filtre.



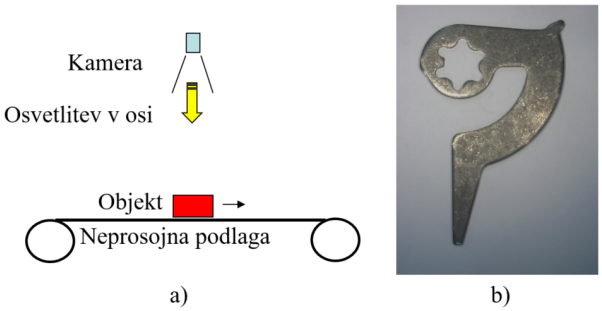
Slika 2.2: Možne postavitve kamere in osvetlitve glede na objekt. Omogočajo izbrane svetlobne efekte in s tem doseganje dobre vidnosti vsega, kar nas zanima.

Osvetlitev lahko postavimo pod različnimi koti glede na kamero in objekt. Slika 2.2 prikazuje ključne postavitve: v osi s kamero, pod kotom glede na kamero in objekt, pravokotno na kamero ali za objekt proti kameri, tako da je objekt med osvetlitvijo in kamero. Vsaka ima specifične svetlobne efekte, ki se prvenstveno izrazijo igri svetlobe in senc, kjer tisto, kar nas zanima, osvetlimo, in kar nas ne zanima, zatemnimo. V nadaljevanju analiziramo posamezne konfiguracije.

2.1.1 Osvetlitev od spredaj sovpadajoče z optično osjo kamere

Osvetlitev od spredaj v osi s kamero je najpogosteje uporabljana, saj luč v praksi najlažje vgradimo tik ob kameri. Če od blizu pogledamo površino poljubnega objekta, vidimo, da je hrapava ter da je sestavljena iz množice raz in mikrohribčkov in dolin, ki so posledica tehnologije izdelave oziroma nastanka objekta. Če takšno površino osvetljujemo pravokotno, vpadla svetloba enakomerno z vseh strani osvetljuje mikroraze. Zaradi enakomerne osvetlitve se sence ne pojavijo, vse je enakomerno svetlo in hrapavost ni vidna. V tej konfiguraciji hrapavost, mikrorazpoke, majhne udarnine, majhni kovinski delci na površini NISO vidni.

Ker je površina enakomerno osvetljena, dobro vidimo barve in objekt kot celoto. Tovrstna konfiguracija je primerna za naloge analiziranja barvnih vzorcev, prepoznave teksta, iskanje obrisa objekta, torej za probleme, kjer nas vzorci hrapavosti motijo in jih ne želimo, želimo pa opazovati objekt kot celoto.

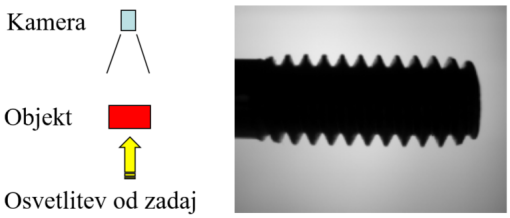


Slika 2.3: Osvetlitev v osi. a) Objekt na tekočem traku. b) Slika, zajeta v tej konfiguraciji.

Slika 2.3a prikazuje pogost industrijski primer, kjer izdelek potuje po tekočem traku, kamera in luč pa sta nameščeni nad trakom v predhodno opisani konfiguraciji. Opazovati želimo objekt kot celoto, nočemo pa motenj mikrotopografije in hrapavosti. Tipična naloga bi bila ugotoviti lego in zasuk izdelka, saj ga v naslednjem koraku poberemo. Lahko bi imeli nalogo prepoznati, ali je na montažni liniji pravilen sestavni del, ali pa izvesti kontrolo oblike ali barve, prepoznati napis itd. Kadar za transport izdelkov uporabljamo tekoči trak, je praktično poskrbeti, da je ta uniformne in bistveno drugačne barve od opazovanega izdelka. Pri načrtovanju sistema kot celote lahko izbiramo standardne barve trakov, kot so črna, modra, siva, in s tem močno poenostavimo obdelavo slike. Slika 2.3b prikazuje primer slike odkovka na tekočem traku, zajete v opisani konfiguraciji z vsemi predhodno opisanimi značilnostmi.

2.1.2 Osvetlitev od zadaj

Osvetlitev od zadaj ali senčna fotografija (ang. shadow photography) je zelo uporabna, kadar je pomemben obris objekta in kar je s tem povezano, npr. natančne dimenzijske meritve, kontrola oblike navoja itd. Slika 2.4 prikazuje konfiguracijo, kjer objekt zastre pot svetlobnim žarkom (podobno kot pri sončnem mrku) in dobimo zelo kontrastno sliko, pri kateri je objekt povsem temen, ozadje pa svetlo.

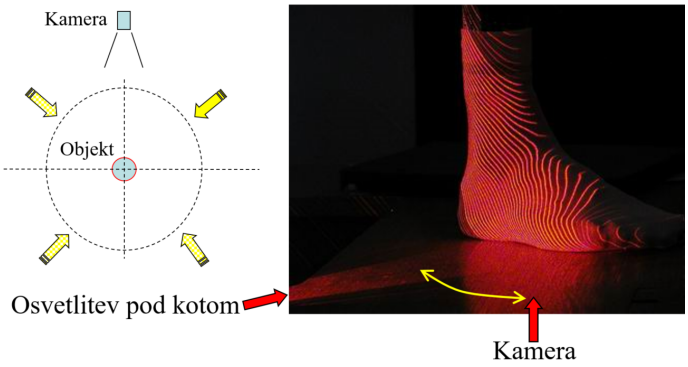


Slika 2.4: Osvetlitev od zadaj

Če želimo dobiti ostro mejo med temnim objektom in svetlim ozadjem, moramo zagotoviti vir osvetlitve, ki oddaja vzporedne svetlobne žarke. Popolnoma vzporedne svetlobne žarke pa dobimo le, kadar je vir svetlobe točkovni (npr. LED-dioda), nameščen v gorišče kolimatorja (skupek leč, podobno kot objektiv kamere). Takšna osvetljevala so dimenzijsko velika in tudi draga, zato jih v praksi zamenjujejo površinska LED-osvetljevala, kjer ostro mejo dobimo zgolj v primeru, da je osvetlitev daleč za objektom, kamera pa blizu. Problem je v tem, da površinska osvetljevala delujejo kot velika množica virov svetlobe.

2.1.3 Osvetlitev pod kotom

Osvetlitev pod kotom razkrije tridimenzionalno (3D) obliko objekta. Zelo nazoren primer, kako je 3D-oblika vidna, prikazuje slika 2.5. Na sliki je prikazano stopalo, ki je osvetljeno z množico vzporednih svetlobnih ploskev. Kamera je posnela sliko v smeri pravokotno na ekran, laserska osvetlitev pa je pod kotom postrani z leve glede na kamero. Zaradi tega so na telesu vidne svetle krivulje, podobno kot izohipse na zemljevidih. Vsaka krivulja tvori presek svetlobne ploskve z objektom – stopalom. Če bi bila osvetlitev npr. v osi s kamero, bi videli zgolj ravne vzporedne črte.



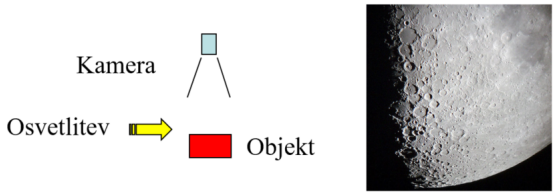
Slika 2.5: Osvetlitev pod kotom

Osvetljevanje pod kotom se uporablja pri merilnikih 3D-oblike teles (3D-skenerji), kjer se za osvetljevanje lahko uporabi laserska svetloba – podobno, kot je prikazano na sliki 2.5, ali pa različni svetlobni vzorci, ki jih projeciramo s projektorji (podobno projektorjem v predavalnicah). Tako zajete slike se nato obdelajo, iz njih se izluščijo projecirani vzorci, npr. izohipse, nato pa se na principu triangulacije ob poznavanju množice parametrov kamere in osvetlitve ter njune medsebojne lege rekonstruira 3D-oblika.

Če povzamemo: osvetljevanje pod kotom uporabimo v primeru, kadar moramo določiti obliko površine opazovanega objekta in tudi dejansko lego v prostoru. Primer uporabe bi bil sistem za sledenje zvarne reže pri avtomatiziranem robotskem varjenju.

2.1.4 Osvetlitev tangencialno na površino objekta

Kadar osvetljujemo objekt skoraj vzporedno (tangencialno) s površino, potem če se vrnemo na hrapavost, osvetljujemo zgolj boke, obrnjene naproti osvetlitvi (prisojna stran), boki, obrnjeni vstran, pa so v senci (osojna stran). Hrapavost in mikrostruktura postaneta vidni. Lep primer tega najdemo v naravi, če ponoči z daljnogledom opazujemo Luno, kot to prikazuje slika 2.6. Na terminatorju, kjer je meja med svetlim in temnim delom Lune, so kraterji lepo vidni, drugod pa ne. Na tem področju svetloba pada tangencialno na površino in opisani efekt pride do izraza.

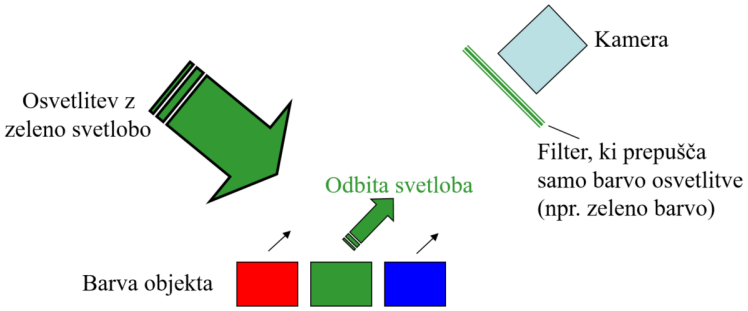


Slika 2.6: Osvetlitev tangencialno na površino objekta

Tovrstno konfiguracijo uporabimo v primerih, kadar preiskujemo površino objekta za praske, razpoke, udarnine in podobne poškodbe.

2.1.5 Barvne kombinacije

Poleg prostorske umestitve luči in kamere je smiselno uporabiti še kombiniranje barv za izboljšanje vidljivosti zgolj opazovanih objektov. Načeloma vsak objekt sestoji iz neke kombinacije osnovnih barv. Primer uporabe barvnih kombinacij prikazuje slika 2.7. Predvidimo, da je objekt zelene barve. To pomeni, da se z njegove površine odbija zelena barva. Če takšen objekt osvetlimo z zeleno svetlobo, se bo ta z njega močno odbijala. Sedaj lahko uporabimo optični filter, nameščen na objektiv kamere, ki do slikovnega zaznavala v kameri prepušča zgolj zeleno barvo. Kamera v takšni postavitvi bo zaznavala objekte zgolj v zelenem delu spektra. Če pri tem poskrbimo, da je ozadje bistveno drugačne barve od objekta, npr. moder tekoči trak, potem bo na sliki viden samo opazovani objekt, ozadje pa bo temno. S takšnim pristopom precej poenostavimo obdelavo slike. Žal so kovinski izdelki v strojništvu tipično sive barve, kjer so enakomerno zastopane vse barve, zato je ta pristop omejeno uporaben. Kar lahko naredimo, je, da opisani pristop uporabimo na ozadju. Primer tega bi bil tekoči trak modre barve, na katerem so sivi kovinski izdelki. Če trak osvetlimo z modro barvo, bo trak močno svetel, objekt pa temen. Slika bi izgledala podobno kot pri osvetlitvi od zadaj, kar bi bilo primerno za zaznavo lege, rotacije, obrisa in meritev dimenzij.



Slika 2.7: Uporaba barvnih kombinacij za izboljšanje vidljivosti opazovanih objektov

Naj še omenimo, da pri merjenju 3D-oblike teles, ki sloni na principu osvetljevanja pod kotom, skoraj vedno uporabimo tudi barvne kombinacije za povečanje zanesljivosti delovanja

merilnikov. Predvidimo, da površino osvetljujemo z lasersko svetlobo, kot je to prikazano na sliki 2.5. Za lasersko svetlobo je značilno, da sestoji zgolj iz ene valovne dolžine. Če pred objektiv kamere namestimo ozkopasovni filter, ki prepušča zgolj lasersko svetlobo, bo na zajeti sliki vidna zgolj laserska osvetlitev, ozadje pa bo temno. Takšna je tudi prikazana slika.

Vprašanja

1. Kako postaviti kamero in osvetlitev glede na objekt v primeru, da želimo kontrolirati:
   - praske, udarnine, hrapavost,
   - meriti zunanje dimenzije,
   - brati in prepoznati tekst,
   - meriti 3D-profil površine,
   - pobirati izdelke s tekočega traku,
   - kontrolirati nivo tekočine v stekleni posodi,
   - kontrolirati poškodbe embalaže?
2. Kako bi zasnovali svetlobni del sistema strojnega vida (osvetlitev, postavitev, ozadje, barve, filtri) v primeru, da:
   - kontroliramo višino kartonskih škatel, ki potujejo mimo kamere na tekočem traku,
   - prepoznavamo lego in obliko hlebcev kruha (še testo) na tekočem traku,
   - iščemo rjo na kovinski pločevini,
   - ugotavljamo, ali je izdelek pravilno vstavljen v gnezdo pri montaži,
   - sledimo zvarni reži pri varjenju,
   - iščemo napake barvanja (praske, vodni pocedki, nepobarvani deli površine)?

2.2 Objektiv in nastanek slike

V tem poglavju govorimo o tem, kako nastane slika. Pod pojmom slika obravnavamo dvodimenzionalni svetlobni vzorec, ki nastane po optični preslikavi osvetljenega objekta. Slika 2.8 prikazuje primer, kako leča naredi sliko na zaslonu. Objekt je v tem primeru okno z zunanjim virom svetlobe (okno je v ozadju in ni vidno), slika pa je vidna kot svetlobni vzorec levo od leče. Zaslon je v tem primeru stena. Slika je na zaslonu (steni) ostra zgolj, kadar je leča na točno določeni oddaljenosti od stene. To lahko ugotovimo s premikanjem leče proti steni ali vstran od nje, lahko pa jo tudi izračunamo s pomočjo enačb optične preslikave.

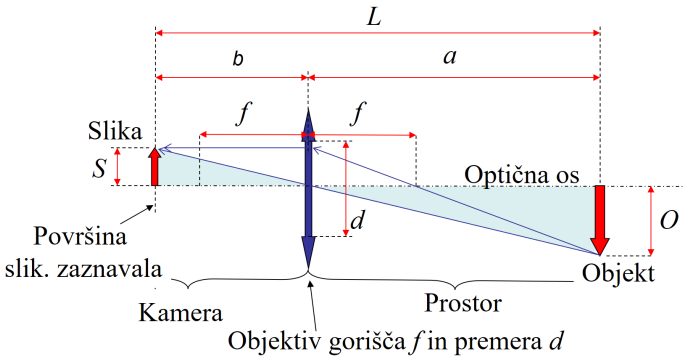
Slika 2.8: Primer, kako leča naredi sliko na zaslonu

Slika je zrcalno obrnjena glede na objekt, njena velikost pa odvisna od izbire objektiva in oddaljenosti od objekta. Pri večini industrijskih aplikacij strojnega vida je slika mnogo manjša od osvetljenega objekta. Le pri mikroskopskih konfiguracijah je slika večja od objekta. Optično preslikavo izvede objektiv, ki je skupek leč v mehanskem ohišju. Ohišje omogoča manjše premikanje leč vzdolž optične osi in s tem nastavljanje ostrine slike na zaslonu (digitalizatorju slike v kameri). V analizah bomo objektiv obravnavali poenostavljeno, kot zgolj eno tanko lečo, in uporabili poenostavljen model optične preslikave za tanke leče. V resnici je vsak objektiv zaporedni sestav več leč, katerih oblika in materiali so izbrani tako, da v največji meri odpravijo barvne napake ter minimizirajo geometrijske popačitve slike (npr. distorzijo). Za resno analizo optične preslikave skozi skupek leč običajno uporabimo programske pakete za simulacijo širjenja svetlobnih žarkov skozi lečja. Gre za ekspertno znanje s področja optike, ki pa na področju strojnega vida načeloma ni potrebno, saj zadostuje zgolj poenostavljen model obravnave objektiva kot tanke leče. Proizvajalci objektivov poskrbijo za to, da objektiv naredi kvalitetno sliko. Izračuni, ki jih izvedemo, so izračuni potrebne goriščne razdalje objektiva za določeno oddaljenost kamere od objekta.

2.2.1 Optična preslikava

O modelu optične preslikave za tanke leče ste med šolanjem že večkrat slišali. Če povzamemo na kratko (slika 2.9), se svetloba z osvetljenega objekta na desni odbija proti objektivu. Ta je simbolično prikazan na sredini slike s poudarjeno dvosmerno puščico. Svetloba potuje skozi lečje (slika 2.11b), ki spremeni smer njenega gibanja tako, da se svetlobni žarki po izhodu iz lečja sestavijo na določeni oddaljenosti od objektiva, kjer nastane slika (na levi strani). Ključni parametri sistema so goriščna razdalja objektiva f, oddaljenost objekta od točke optične preslikave v središču objektiva a, oddaljenost slike od točke optične preslikave b ter velikost objekta O in slike S.

Osnovna enačba optične preslikave 2.1 povezuje oddaljenosti objekta in slike od točke optične preslikave (v objektivu), v odvisnosti od goriščne razdalje objektiva.



Slika 2.9: Model optične preslikave za tanke leče

$$\frac{1}{a} + \frac{1}{b} = \frac{1}{f}. \tag{2.1}$$

Enačba za povečavo m povezuje velikosti objekta in slike

$$m = \frac{b}{a} = \frac{S}{O}. \tag{2.2}$$

Če ti dve enačbi sestavimo, dobimo enačbo za hitro oceno potrebne goriščne razdalje objektiva, če poznamo velikost objekta in velikost slike ter oddaljenost objekta od objektiva:

$$f = \frac{a \cdot S}{O + S}. \tag{2.3}$$

Pri izbiri objektiva najprej uporabimo enačbo 2.3. Velikost objekta O v praksi predstavlja območje, ki ga opazujemo s kamero, npr. širina tekočega traku, po katerem prihajajo izdelki pod kamero. Oddaljenost a izberemo glede na možnost vgradnje kamere na delovno mesto. Velikost slike S predstavlja velikost slikovnega zaznavala v kameri v milimetrih (torej moramo predhodno izbrati kamero in s podatkovnega lista odčitati velikost slikovnega zaznavala v milimetrih; pozor, ne v slikovnih elementih oz. pikslih).

Primer izračuna: Predpostavimo, da s kamero opazujemo tekoči trak širine 250 mm. Na voljo imamo kamero z 2/3-velikim slikovnim zaznavalom, kar v praksi znaša 8,8 mm x 6,6 mm. Kamero lahko namestimo na konstrukcijo približno 500 mm nad tekočim trakom. Kolikšno gorišče naj ima objektiv?

Predpostavimo, da kamero namestimo tako, da je daljša stran slikovnega zaznavala pravokotno na trak. Najprej uporabimo enačbo 2.3 za hitro oceno potrebnega gorišča

$$f = \frac{a \cdot S}{O + S} = \frac{500 \text{ mm} \cdot 8{,}8 \text{ mm}}{250 \text{ mm} + 8{,}8 \text{ mm}} = 17 \text{ mm}. \tag{2.4}$$

Teoretično potrebujemo objektiv z goriščem 17 mm. Proizvajalci objektivov izdelujejo objektive z zgolj določenimi goriščnimi razdaljami, npr. 12 mm, 16 mm, 25 mm ... 50 mm. Najbližji dosegljivi objektiv ima gorišče 16 mm. Če takšen objektiv namestimo na kamero, bo vidno polje nekaj večje od 250 mm. Če nas to ne moti, je naloga končana, v nasprotnem pa moramo spremeniti oddaljenost kamere a od tekočega traku.

Za začetek izračunajmo, kolikšna je dejanska širina vidnega polja v primeru uporabe objektiva z goriščem 16 mm. Iz enačbe (2.1) izračunamo, kolikšna je oddaljenost slike od točke optične preslikave v objektivu b.

$$\frac{1}{b} = \frac{1}{f} - \frac{1}{a} = \frac{1}{16 \text{ mm}} - \frac{1}{500 \text{ mm}} \rightarrow b = 16{,}5 \text{ mm}. \tag{2.5}$$

Izračunamo novo širino merilnega območja

$$m = \frac{b}{a} = \frac{S}{O} \rightarrow O = \frac{S \cdot a}{b} = \frac{8{,}8 \cdot 500 \text{ mm}}{16{,}5} = 266 \text{ mm}. \tag{2.6}$$

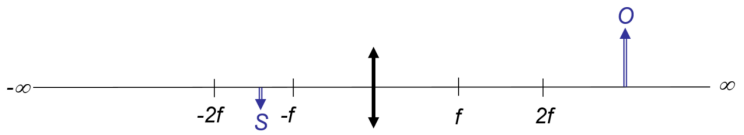
Merilno območje je za 16 mm širše od tekočega traku. Kot rečeno, če to ni problem, je izračun objektiva zaključen. Če pa je specifikacija, da mora biti vidno polje natančno 250 mm, potem moramo kamero z izbranim objektivom 16 mm približati tekočemu traku, torej spremenimo a. Izračun nove vrednosti a se začne pri enačbi za povečavo (2.2), kjer razmerje b/a vstavimo enačbo za optično preslikavo (2.1), iz te pa izračunamo nov a.

$$m = \frac{b}{a} = \frac{S}{O} \rightarrow m = \frac{8{,}8}{250} = 0{,}035 \rightarrow b = 0{,}035 \cdot a \tag{2.7}$$

$$\frac{1}{a} + \frac{1}{b} = \frac{1}{f} \rightarrow \frac{1}{a} + \frac{1}{0{,}035 \cdot a} = \frac{1}{f} \rightarrow \frac{1}{a} \cdot 29{,}40 = \frac{1}{f} \rightarrow a = 29{,}40 \cdot f = 29{,}40 \cdot 16 \text{ mm} = 470 \text{ mm}. \tag{2.8}$$

Za enako vidno polje moramo kamero približati tekočemu traku na oddaljenost 470 mm.

Komentar lege in velikosti slike: V prejšnjem preračunu smo kamero približali objektu in s tem vplivali na velikost vidnega polja. Zelo pomembno je, da razumemo, kje nastane slika v odvisnosti od tega, kako daleč pred objektivom se nahaja opazovani objekt ter v kolikšni meri je slika povečana oziroma pomanjšana. Za lažje razumevanje preučimo skico na sliki 2.10.



Slika (S)	Objekt (O)	
-f	∞	Daljnogled (O>S; m<1)
-2f < S < -f	2f < O < ∞	
-2f	2f	1:1
-∞ > S > -2f	f < O < 2f	Mikroskop (O<S; m>1)
-∞	f	

Slika 2.10: Lega slike v odvisnosti od oddaljenosti objekta od objektiva

Objektiv je predstavljen z odebeljeno dvosmerno puščico na sredini skice, horizontalna črta predstavlja optično os, ki sovpada z osjo lečevja v objektivu na sliki 2.11. Pri razumevanju preslikav sta pomembni razdalji f in $2f$ pred objektivom in za njim. Pred objektivom je v desno proti objektu, za objektivom je proti sliki v levo.

Če bi se objekt nahajal zelo daleč stran od objektiva (v ∞), bi slika nastala na levi v gorišču objektiva. (Komentar: vsakdo je že lečo s sončno svetlobo poskusil kaj žgati. Sonce je v neskončnosti, slika v gorišču. Kamere zato nikoli ne usmerite naravnost v sonce, ker boste zažgali slikovno zaznavalo in s tem uničili kamero.) Za objektive s kratkim goriščem je "neskončnost" že nekaj metrov pred kamero oziroma slika nastane v gorišču za objekte, ki so od nekaj metrov stran od objektiva, pa do ∞. Slika je mnogo manjša od objekta.

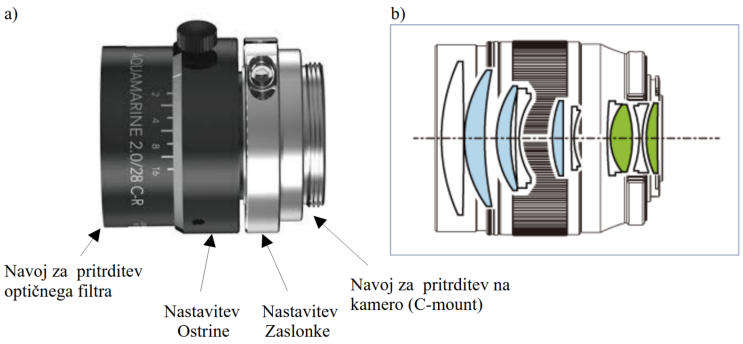
Ko se objekt iz ∞ približuje $2f$, se slika na levi odmika iz $-f$ in postaja vse večja. Iz tega razloga so objektivi mehansko izdelani tako, da z vrtenjem dela objektiva premikamo lečje vzdolž optične osi. S tem nastavljamo lego slike na slikovno zaznavalo. Večina primerov strojnega vida deluje v tej konfiguraciji.

Če se objekt približa na oddaljenost $2f$, potem se slika na levi odmakne v $-2f$. To je preslikava 1 : 1, kjer sta objekt in slika enako velika ter enako odmaknjena od točke optične preslikave v objektivu. Uporablja se bolj redko, v npr. fotokopirnih strojih, za obračanje slike v lovskih daljnogledih ali kot vmesna leča med objektivom in okularjem daljnogleda.

Kadar se objekt nahaja med f in $2f$ pred objektivom, slika nastane za $-2f$ na levi. Bolj kot je objekt blizu f, bolj se slika odmika proti neskončnosti in v skrajnem primeru, ko je objekt v f, je slika v $-\infty$. V vseh primerih je slika večja od objekta in narašča s pomikanjem proti neskončnosti. Tovrstna konfiguracija se uporablja pri mikroskopih za doseganje velikih povečav.

2.2.2 Izvedbe objektivov

Primer tipičnega objektiva za industrijske kamere je prikazan na sliki 2.11. Gre za manjše objektive premera med 30 in 50 mm ter dolžinami do 50 mm. Cene se gibljejo med 100 in 300 eur (l. 2023), odvisno od modela (standard, megapixel). Ostrino nastavljamo z vrtenjem dela objektiva, poleg tega pa je še dodaten obroč na nastavljanje zaslonke. V praksi je kamera nameščena fiksno, zato ni potrebe po spreminjanju nastavitev objektiva. Ko enkrat nastavimo ostrino in zaslonko, blokiramo nastavljanje, tj. vrtenje obročev s stranskimi vijaki na obročih. Objektiv se na kamero pritrdi s pomočjo finega navoja (C-mount, premer 25,4 mm, 32 navojev/colo), zato se objektiv poimenuje kar "C-mount" objektiv. Obstaja še CS-izvedba objektivov, kjer je navoj enak, razlika je le v oddaljenosti, kjer slika nastane gledano od prirobnice navoja. Pri C-izvedbi je ta razdalja 17,5 mm, pri CS pa 12,5 mm. Kamere večinoma podpirajo CS-izvedbo (slikovno zaznavalo se nahaja vzdolž optične osi 12,5 mm od prirobnice, na katero se nasloni objektiv). CS-objektiv privijačimo neposredno na kamero, pri C-izvedbi objektivov pa moramo uporabiti še 5-milimetrski distančni obroč med objektivom in kamero.

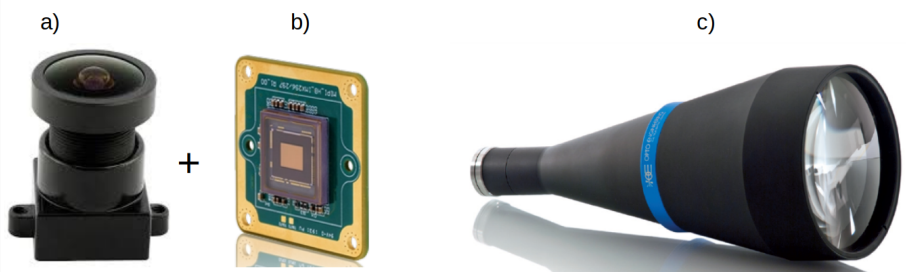


Slika 2.11: Izgled tipičnega objektiva (a); lečje v objektivu (b)

Pri vgradnih sistemih, kjer kamero tvori zgolj tiskanina s slikovnim zaznavalom in z najnujnejšim čipovjem (glej sliko 2.12b), se na tiskanino privijači nosilec z majhnim objektivom, kot to prikazuje slika 2.12a. Gre za majhne objektive, kjer se ostrina nastavlja zgolj s privijanjem objektiva, nastavljive zaslonke pa ni. Takšni objektivi imajo zunanji navoj M12 x 0,5 in so površinsko prilagojeni majhnim slikovnim zaznavalom. Dobijo se z različnimi gorišči, prevladujejo pa širokokotni s kratkimi goriščnimi razdaljami. Pogosto se uporabljajo na mobilnih robotskih platformah za vizualno navigacijo, kjer je potreba po majhnih dimenzijah in teži ter čimvečjem vidnem polju.

Obstaja še mnogo posebnih izvedb objektivov. Od teh bi izpostavili telecentrične (slika 2.12c). Njihova posebnost je, da vidno polje ne narašča z oddaljevanjem od objektiva, ampak je vseskozi enako. S tem dosežejo, da je opazovani objekt na sliki enake velikosti, in sicer ne glede na oddaljenost. Gre za dimenzijsko velike objektive, od 100 do 300 mm po dolžini, katerih optični sistemi so izredno kompleksni, temu pa je primerna tudi nakupna cena. Premer vidnega

polja je enak premeru vstopne leče.



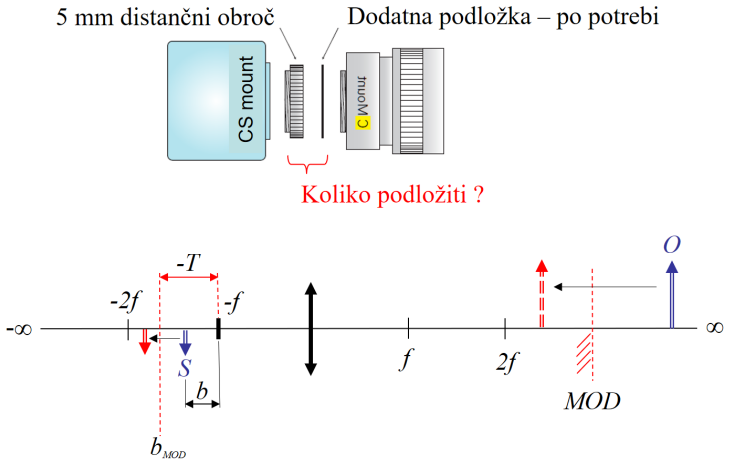
Slika 2.12: Miniaturni objektiv M12 z nosilcem za vgradne kamere (a); primer vgradne kamere (b); telecentrični objektiv (c)

Ključne specifikacije pri izbiri objektiva so sledeče:

- Goriščna razdalja, npr. 16 mm;
- Format objektiva npr. 2/3". Ta parameter govori o tem, kako veliko slikovno zaznavalo objektiv v celoti pokrije z ostro sliko. Objektiv s formatom 2/3" je tako primeren za 2/3" in manjša slikovna zaznava. Pri večjih slika na robovih ne bo ostra.
- Največja odprtina f/d npr. 1 : 2,8. Ta podatek govori o svetlobni moči objektiva oziroma o premeru vstopne leče. Večja kot je vstopna leča, več svetlobe bo objektiv zbral, bolj svetla bo slika v slabših svetlobnih pogojih. Torej manjša kot je številka, več svetlobe objektiv zbere, boljše bo.
- Zaslonka 2,8-22. Gre za območje nastavljanja zaslonke. Številka podaja vrednost f/d. Zaslonka se obnaša kot "ventil", kjer nastavljamo pretok oz. količino svetlobe, ki potuje skozi objektiv. Gre za mehanizem več tankih lističev, povezanih z vodilnim obročem, ki deluje tako, da z vrtenjem vodilnega obroča spreminja premer odprtine, ki jo tvorijo lističi, kar se obnaša, kot če bi spreminjali premer leč d, skozi katere potuje svetloba.
- Priključni navoj, npr. C, CS, M12.
- Kvaliteta izvedbe. Pogosto se kaže v optičnih napakah in ostrini slike. Plastične ali steklene leče.
- Zunanje dimenzije.
- MOD (ang. minimum object distance) ali najbližji odmik ostrine; npr. 500 mm. Če objektiv privijemo na kamero, lahko sliko ostrimo od MOD-razdalje naprej (z vrtenjem dela objektiva). Če želimo opazovati objekte bližje od te razdalje, potem moramo med kamero in objektiv vgraditi dodaten distančni obroč.

Dodaten distančni obroč: Kadar kamero namestimo v bližino objekta, na razdalji, krajši od MOD, moramo med kamero in objektiv dati dodaten distančni obroč. Razlog za to je v tem, da se razdalja b poveča več od pričakovanega v mehanski izvedbi objektiva. Kadar nastavljamo ostrino slike, vrtimo del objektiva in s tem premikamo lečevje vzdolž optične osi. Slika je ostra, kadar nastane na slikovnem zaznavalu. Vrtenje dela objektiva je možno znotraj nekih meja, npr. 1.5 obrata. Ena skrajna lega ustreza primeru, kadar je objekt v neskončnosti, slika pa v gorišču objektiva f. Druga skrajna lega ustreza primeru, kadar je objekt na oddaljenosti MOD izpred objektiva, slika pa se odmakne iz gorišča za $-T$, $b = -f - T$.

Če se objekt nahaja bližje objektivu, kot je MOD (glej rdečo črtkano puščico na desni strani slike 2.13), se b še bolj poveča in slika nastane bolj v levo od b_{MOD} proti $-2f$. To je simbolično prikazano z manjšo rdečo puščico na levi. Z distančnim obročem kamero premaknemo v novo lego slike, tako da sredina območja T sovpada z novo lego slike. Širino distančnega obroča izračunamo kot seštevek povečanja razdalje b glede na b_{MOD} in polovico T.



Slika 2.13: Izračun distančnega obroča

Primer izračuna distančnega obroča: Predpostavimo, da kamero namestimo bližje objektu, kot je MOD objektiva. Podatki so naslednji $a = 300$ mm, $f = 25$ mm, MOD $= 500$ mm. Najprej se vprašajmo, kje bi nastala slika, če bi se objekt nahajal točno v MOD. Uporabimo enačbo za optično preslikavo (2.1)

$$\frac{1}{b_{\mathrm{MOD}}} = \frac{1}{f} - \frac{1}{MOD} \rightarrow \frac{1}{25} - \frac{1}{500} \rightarrow b_{\mathrm{MOD}} = 26{,}32 \text{ mm}. \tag{2.9}$$

Podobno izračunamo, kje nastane slika, če se objekt nahaja na razdalji $a = 300$ mm:

$$\frac{1}{b_a} = \frac{1}{25} - \frac{1}{300} \rightarrow b_a = 27{,}27 \text{ mm}. \tag{2.10}$$

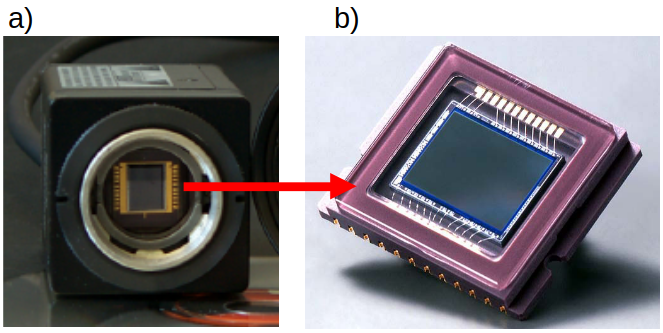
Vidimo, da se b poveča za 0,96 mm. Če bi med kamero in objektiv vstavili distančni obroč navedene debeline, potem bi objektiv sliko izostril zgolj, kadar je obroč za ostrenje na objektivu zasukan v skrajno lego. Praktično je, da je obroč za nastavljanje ostrine nekje na sredini območja, tako da ostrino natančno nastavimo z vrtenjem v obe smeri. V ta namen moramo izračunani razdalji prišteti še polovico T. Razdalja T med b_{MOD} in f znaša 1,32 mm. Končna vrednost distančnega obroča tako znaša 0,96 mm + 0,66 mm = 1,62 mm. Gre za majhno razdaljo, zato distančni obroč izdelamo s struženjem, lahko pa tudi kupimo standardne distančne obroče, kjer je najbližja standardna mera 1,5 mm. Ostrina v tem primeru ne bi bila čisto točno na sredini območja nastavljanja ostrine slike na objektivu.

2.3 Slikovno zaznavalo v kameri

Bistvo sodobnih kamer je slikovno zaznavalo (slika 2.14 a in b), ki svetlobni vzorec, ustvarjen z objektivom, vzorči in digitalizira. Na slikovnem zaznavalu se torej zgodi pretvorba svetlobnega vzorca v številčni zapis slike. Številčni zapis slike digitalno (električno) prenašamo na večje razdalje (s kamere v računalnik), zapišemo v spomin računalnika in tam naprej po potrebi numerično obdelujemo (obdelava slike) do želenih informacij.

Bistvo slikovnega zaznavala je velika množica majhnih fotocelic (angleško poimenovanje takšne fotocelice je pixel – slikovni element), ki vpadlo svetlobo pretvorijo najprej v električni naboj, električno vezje v ozadju fotocelice naboj pretvori v napetost, to pa digitalizira (glej analogno digitalna "AD"-pretvorba), da dobimo številčni zapis električne napetosti oziroma svetlosti. AD-pretvorba je tipično 8-bitna, kar pomeni, da je zaloga vrednosti števil med 0 in 255. Vrednost 0 ustreza povsem črni barvi, vrednost 255 pa povsem beli barvi. Vmesne vrednosti ustrezajo različnim stopnjam svetlosti oziroma odtenkom sivine. Z 8-bitno digitalizacijo tako dosežemo 256 odtenkov sivine. Sodobne kamere omogočajo izbiro stopnje digitalizacije, kjer tipično izbiramo med 8 ali 16 biti. Poleg digitalizacije je izredno pomembno še prostorsko in časovno vzorčenje, kar si bomo podrobneje ogledali v nadaljevanju.

Predhodno opisan primer, pri katerem imamo za vsak slikovni element 8-bitni podatek o svetlosti, je tipičen za črno-bele kamere. Podobno je pri barvnih kamerah, le da so na slikovne elemente naparjeni barvni filtri (slika 2.15), ki do fotocelice prepuščajo zgolj eno od osnovnih barv, tj. rdečo (R), modro (B) ali zeleno (G). Slikovni element torej zaznava zgolj eno barvo. Manjkajoči barvi se naknadno izračunata kot povprečje sosednjih slikovnih elementov v ustrezni manjkajoči barvi. Filtri so namreč na slikovne elemente naparjeni v takšnem vzorcu, da so sosedje vedno druge barve (Bayer vzorec).



Slika 2.14: Kamera (a); slikovno zaznavalo (b)

2.3.1 Digitalizacija

Bistvo digitalizacije je v pretvorbi svetlosti, ki jo zaznava slikovni element v številčni zapis – v številko. Pri 8-bitni digitalizaciji digitalizirano vrednost (številko med 0 in 255) zapišemo v 8-bitno spremenljivko tipa *BYTE*. To je zelo pomembno z vidika količine spomina, potrebnega za shranjevanje digitalizirane slike. Sodobne kamere imajo tipično več milijonov slikovnih elementov, kar pomeni, da slika v spominu računalnika zavzame temu ustrezno več MB (MegaBytov). Pri razvoju sistemov strojnega vida za razliko od klasične digitalne fotografije velikost slike je pomembna, saj mnogokrat zajemamo in obdelujemo veliko število slik v sekundi (npr. sistemi za sledenje). Hitro pridemo do razmišljanja o tem, koliko slik v sekundi prenese celotna veriga od zajema do obdelave in krmilnih akcij. Iz tega razloga pogosto izberemo kamero z manjšo sliko in raje domislimo optični del merilnega sistema do te mere, da slika vsebuje samo tisto, kar je pomembno, in čim manj nepomembnega balasta (zanimiva razmišljanja na to temo podaja teorija vsebnosti informacije v sliki).

Za primerjavo pri 16-bitni digitalizaciji je zaloga vrednosti števil med 0 in 65535. Načeloma bi 16-bitno številko lahko shranili v 16-bitno spremenljivko tipa *short*, vendar sodobni programski jeziki kot tudi 32(64)-bitni procesorji tovrstni podatkovni tip opuščajo, zato imamo več težav, kot če za shranjevanje uporabimo 32-bitno celoštevilčno spremenljivko tipa *int*. Vredno si je zapomniti, da 16-bitno digitalizirana slika posledično zavzame 4-krat več spomina kot 8-bitna

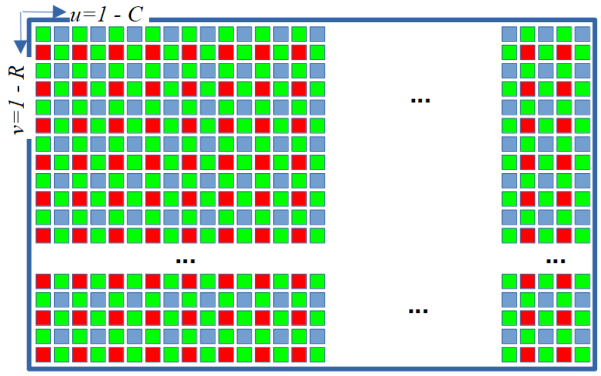
slika.

Za večino aplikacij strojnega vida je 8-bitna digitalizacija zadostna. 16-bitna digitalizacija pride do izraza v primeru, kadar se v odtenkih svetlosti nahaja ključna informacija, kot je to primer pri analizi rentgenskih slik.

2.3.2 Prostorsko vzorčenje

Slikovno zaznavalo tvori velika množica matrično razporejenih slikovnih elementov (fotocelic, ang. pixels), ki vzorčijo svetlobni vzorec, ustvarjen s strani objektiva (slika 2.15). Tipične velikosti slikovnih elementov so 3–7 μm. Več kot je slikovnih elementov, bolj podrobne detajle lahko zaznamo. Podatek o številu stolpcev in vrstic matrike pogosto označujemo z **resolucijo** in jo podajamo kot R x C, kjer je R število vrstic in C število stolpcev.

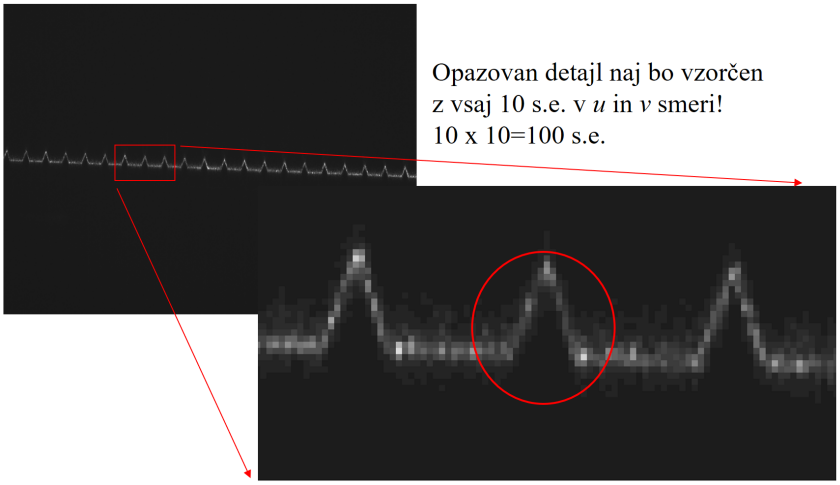
Trend sodobnih fotoaparatov gre v smeri zajema čim bolj podrobnih detajlov, zato imajo ti zelo veliko slikovnih elementov (npr. Canon EOS Ra jih ima 30 milijonov razporejenih v matriko velikosti 6720 x 4480). Pri strojnem vidu je slika s tako velikim številom slikovnih elementov samo veliko breme, predvsem zaradi velikosti slike, ki obremenjuje celotno verigo od zajema do obdelave. Iz tega razloga imajo industrijske kamere slikovna zaznavala z manjšim številom slikovnih elementov: tipično od VGA (640 x 480) do HD (1920 x 1200).



Slika 2.15: Prostorsko vzorčenje svetlobnega vzorca z množico matrično razporejenih slikovnih elementov (ang. pixels). Slikovni elementi so prikazani s kvadratki, katerih tipična velikost se giblje med 3–7 μm. Pri zajemu barvne slike so na slikovne elemente nanešeni barvni filtri v osnovnih RGB-barvah in vzorcu, kot prikazuje slika.

Po teoriji vzorčenja je potrebno opazovane pojave vzorčiti z minimalno 2-krat višjo frekvenco, kot je frekvenca opazovanega pojava. Pri prostorskem vzorčenju slike bi teoretično to pomenilo, da bi morali opazovani detajl vzorčiti z minimalno 2 slikovnima elementoma, kar pa je pri vzorčenju slik veliko premalo. Praktične izkušnje kažejo, da je potrebno mejno vzorčenje vsaj 10 slikovnih elementov na opazovanem detajlu. Primer mejnega vzorčenja prikazuje slika 2.16. Objekt s trikotnimi utori osvetlimo pravokotno na površino s tanko svetlobno ploskvijo, tako da se na površini vidi svetlobni profil (konfiguracija, podobna kot na sliki 2.5). Pri kameri sta objektiv in slikovno zaznavalo izbrana tako, da je trikotni utor vzorčen z 10 x 10 slikovnimi elementi. Če od blizu pogledamo povečan detajl utora, vidimo, kako je v resnici svetlobni profil grobo vzorčen. Iz takšne slike lahko ocenimo, ali gre za trikotni profil in ali je prisoten. Za natančne dimenzijske meritve globine utora, kota, za kontrolo poškodb itd. pa je vzorčenje premajhno in meritve niso možne. Povečati bi ga morali na vsaj 100 x 100 slikovnih elementov na

utor. Primer kaže, v kolikšni meri je pomembno zadosti visoko vzorčenje. Seveda pa ni smisla pretiravati zaradi povečanja velikosti slike. Vzorčenje mora biti prilagojeno potrebam aplikacije.



Slika 2.16: Minimalno vzorčenje.
Opazovani detajl naj bo vzorčen z vsaj 10 slikovnimi elementi ali več tako v u- kot v v-smeri.

Nalaganje (ang. aliasing)

Zelo pogosta težava pri vzorčenju slike je nalaganje (ang. aliasing), ki se kaže v obliki "fantomskih" kolobarjev, ki se pojavijo na sliki, če s kamero slikamo vzorec z drobnimi detajli (npr. karirast tekstil, glej sliko 2.17a). Tovrstni vzorci so poimenovani moire vzorci. Nastanejo kot posledica interference med sliko, ki jo ustvari objektiv (slika vsebuje periodični vzorec), in matričnim vzorcem slikovnih elementov. Gre za škodljiv pojav, ki spremeni dejansko sliko, zato in ga želimo preprečiti.

Ključni podatek pri vzorčenju slike je frekvenca vzorčenja f_s in polovična frekvenca, imenovana Nyquistova frekvenca $f_N = f_s/2$. Navedena parametra lažje razumemo s pomočjo slike 2.17(b). Vzorčenje izvajajo slikovni elementi. f_s je povezana z razdaljo med slikovni elementi. Predstavljajmo si valovanje, katerega perioda bi ustrezala razdalji med sosednjima slikovnima elementoma. Frekvenca tega valovanja je f_s. Pol nižja frekvenca f_N pomeni valovanje z 2-krat daljšo periodo, kar ustreza razdalji med vsakim drugim slikovnim elementom. Po teoriji vzorčenja se nalaganje pojavi za vse frekvence, ki so višje od f_N. Višje frekvence se preslikajo (naložijo) z območja nad f_N, v območje frekvenc nižjih od f_N.

Če povzamemo: če s kamero opazujemo periodične vzorce (npr. sinusni, krožni ali karirasti vzorci) in je perioda vzorcev manjša od treh slikovnih elementov na slikovnem zaznavalu, potem nastanejo moire vzorci, ki spremenijo sliko.
Pojav nalaganja preprečimo na več načinov. Ključno je, da s slike pobrišemo (optično filtriramo) periodične vzorce, katerih frekvence so višje od f_N, tj. pobrišemo detajle, manjše od treh slikovnih elementov.

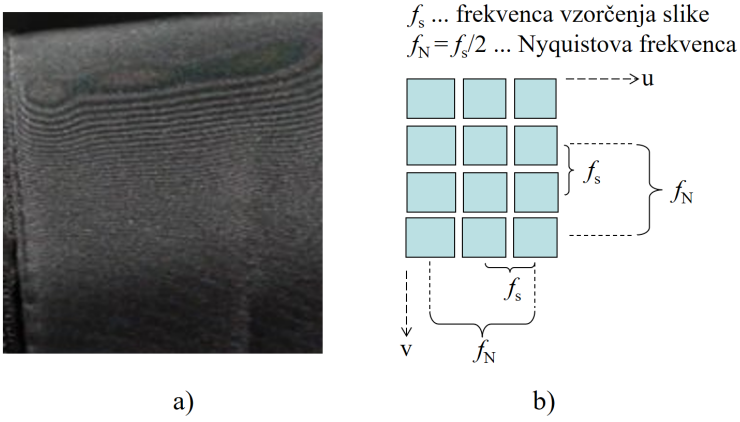
Najbolj preprost pristop optičnega filtriranja je v primerni izbiri objektiva. Večina objektivov se obnaša kot nizkopasovni optični filter, ki zamegli višje frekvence, tj. ostre detajle. Objektivi za strojni vid, npr. slika 2.12, spadajo med objektive, ki so optično slabše kvalitete od fotografskih objektivov in niso zmožni izostriti detajlov pod neko mejo. V optiki se to vrednoti z "ang. point spread function (PSF)". (Za lažje razumevanje PSF se najprej spomnimo, kako testiramo dinamske lastnosti mehanskih sistemov. Sistem vzbudimo s skočno funkcijo; ta se odzove z lastnimi nihanji. PSF si lahko predstavljamo kot nekaj podobnega v optični domeni.) Za vzbujanje uporabimo točkovni vir svetlobe. Ko objektiv ustvari sliko točkovnega vira sve-

tlobe, bo ta zaradi uklonskih efektov in optičnih napak lečevja (lastnost sistema) zamegljen disk določenega premera. Idealno bi bilo, če bi to bila spet točka. PSF-graf podaja porazdelitev svetlosti nastalega vzorca po optični preslikavi in se uporablja kot merilo pri določanju optične ločljivosti in ostrine slike. Ozek PSF pomeni ostro sliko z visoko optično ločljivostjo, medtem ko širok PSF vodi do zameglitve in izgube podrobnosti. Izkustvena enačba za hitro oceno premera PSF je:

$$C_{PSF} = 2{,}5\ \lambda\ \frac{f}{d}. \tag{2.11}$$

Za svetlobo z valovno dolžino npr. 650 nm, objektiv s $f = 25$ mm in $d = 10$ mm znaša $C_{PSF} = 4$ μm. S slike so filtrirani detajli pod 4 μm. Velikost slikovnih elementov v industrijskih kamerah 3–6 μm je že približno prilagojena omenjenim objektivom. Če zapremo zaslonko na npr. $d = 5$ mm premera ($f/d = 5$), se C_{PSF} poveča na 8 μm, s čimer je problem nalaganja že rešen, če so slikovni elementi velikosti npr. 3 μm.

Pri kvalitetnih objektivih z majhnim C_{PSF} tako potrebujemo majhne slikovne elemente $< 3\mu m$. Dodatno k temu se za optično filtriranje uporabijo tudi posebni optični filtri, katerih delovanje temelji na dvolomnih kristalih. Takšne rešitve so dražje in se uporabljajo v digitalnih fotoaparatih.



Slika 2.17: Nalaganje v obliki moire vzorcev (ang. aliasing) se pojavi, kadar slika vsebuje periodične svetlobne vzorce, manjše od treh slikovnih elementov (a); razlaga Nyquistove frekvence (b).

2.3.3 Časovno vzorčenje

Bistvo časovnega vzorčenja je v številu slik, ki jih zajamemo v sekundi oz. kolikokrat v sekundi ponovimo prostorsko vzorčenje slike. Časovno vzorčenje se označuje s kratico FPS, kar je okrajšava angleškega "Frames Per Second". Koliko slik v sekundi zajamemo, je pomembno samo pri sistemih, kjer v realnem času spremljamo neko dogajanje.

Primer takšnega sistema je sledenje zvarne reže pri robotskem varjenju. Gorilnik varilnega aparata se giblje z določeno hitrostjo. S kamero slikamo zamik gorilnika od centra zvarne reže, s slike izračunamo zamik, ga umerimo in pošljemo na krmilnik robota za popravek poti gibanja. Večkrat kot to v sekundi ponovimo, bolj zvezno sledimo zvarni reži.

Ločiti moramo FPS slikovnega zaznavala od FPS celotne merilne verige, ki vključuje tudi digitalno komunikacijo, obdelavo slike in krmilne akcije. FPS slikovnega zaznavala (kamere) je podatek, ki ga pridobimo iz specifikacij kamere. Sodobne digitalne kamere z npr. HD-resolucijo

omogočajo do 60 FPS, z VGA-resolucijo pa do 500 FPS pri GigE-digitalni komunikaciji. FPS je nastavljiv s strani gonilnika kamere.

Območje interesa: Omogoča, da izberemo zgolj del slikovnega zaznavala, ki ga bo kamera zajemala in pošiljala naprej po digitalni komunikaciji. Po angleško je ta funkcionalnost poimenovana s kratico ROI ali "Region Of Interest". S tem, ko zajemamo in prenašamo zgolj del slikovnega zaznavala, se FPS poveča glede na pretočnost digitalne komunikacije. Dosegamo lahko tudi več kot 500 FPS. Mnogokrat razvijalci sistemov izberejo kamero z visoko ločljivostjo in uporabijo funkcionalnost ROI, zato kamere ni treba fino mehansko nastavljati. Kamero v grobem namestimo v kontrolno pripravo, poskrbimo za zadosti veliko vidno polje, potem pa ROI programsko nastavimo tako, da kamera zajema samo opazovano območje. Programsko nastavljanje ROI je zelo enostavno in uporabno v primeru sprememb v konstrukciji in legi objekta, kot bi to bilo v primeru finega mehanskega nastavljanja.

Proženje kamere: Industrijske kamere imajo tudi zelo pomembno fukcionalnost proženja zajema slike z zunanjim električnim signalom (ang. trigger). Z njim sprožimo trenutek zajema slike glede na neko sekvenco dogodkov v napravi, kamor je kamera vgrajena. V ta namen imajo kamere dodaten konektor, kjer jo povežemo z zunanjo napravo, npr. s PLK, ki krmili delovanje avtomatizirane celice. Krmilni signal je tipično vsaj nekaj milisekund dolg električni pulz (npr. TTL-nivoji), proženje pa se zgodi ob prehodu iz nizkega v visoko stanje oziroma kakor pač nastavimo kamero. V tovrstnem načinu delovanja FPS ni več pomemben, saj kamera zajame sliko zgolj, kadar je sprožena, npr. kadar na kontrolno delovno mesto pride nov izdelek.

Zajem slike (zaslonka): Pri časovnem vzorčenju še omenimo način branja podatkov s slikovnega zaznavala. O globalni zaslonki (ang. global shutter) govorimo, če slikovno zaznavalo (elektronika v ozadju) zamrzne trenutno vrednost na vseh slikovnih elementih hkrati in jih nato AD pretvori v numerične vrednosti. Obstaja tudi gibajoča zaslonka (ang. rolling shutter), kjer slikovno zaznavalo zamrzne vrednosti slikovnih elementov zgolj v eni vrstici, AD jih pretvori, nato pa ponovi postopek na naslednji vrstici itd., dokler ne zajame celotne slike. Pri slednji lahko slika postane popačena, če opazujemo hitro premikajoče se objekte.

2.3.4 Velikost slikovnega zaznavala

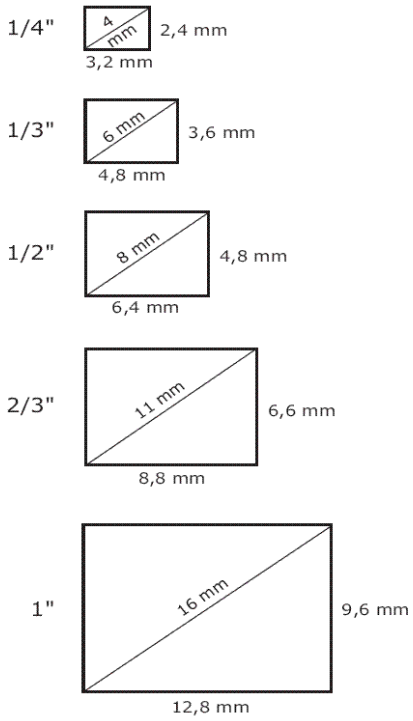
Ko govorimo o velikosti slikovnega zaznavala, imamo v mislih predvsem dimenzije v milimetrih. Velikost je pomembna predvsem z vidika izračuna optične preslikave, saj določa velikost slike S. Če poznamo velikost slikovnega elementa, npr. 5,86 μm, in resolucijo, npr. 1920 x 1200, slikovnih elementov, lahko izračunamo dimenzije v milimetrih $0{,}00586 \cdot 1920 = 11{,}25\ mm$ in $0{,}00586 \cdot 1200 = 7{,}032$ mm.

Pogosto je podan podatek o diagonali slikovnega zaznavala v colah, kot je prikazano na sliki 2.18. Takšen podatek, npr. *1/2"*, predpostavlja, da je razmerje stranic $4 : 3$, zato dimenzije stranic preračunamo po Pitagorovem izreku iz diagonale ob upoštevanju navedenega razmerja. Verjetno ste takoj opazili, da oznaka *1"* pripada diagonali 16 mm, oznaka *1/2"* pa diagonali 8 mm. Tovrstne dimenzije imajo zgodovinske korenine, saj so ob pojavu digitalnih kamer poskušali vzdrževati kompatibilnost s kinoformati. Iz tega razloga velikost slikovnega zaznavala *1"* sovpada z velikostjo diagonale sličic 16-milimetrskega filmskega traku. Podobno *1/2"* sovpada z 8-milimetrskim filmskim trakom.

2.3.5 Izbira kamere

Pri izbiri kamer so v podatkovnih listih tipično podani naslednji podatki:

- resolucija slikovnega zaznavala, npr. 1920 × 1200 (2,3 MP),
- digitalizacija, npr. 12-bitna,



Slika 2.18: Dimenzije slikovnega zaznavala, če je podana diagonala v colah

- FPS, npr. 50 FPS,
- tip in vrsta senzorja, npr. CMOS, Sony IMX174 (podatkovni list),
- zaslonka, npr. globalna,
- velikost slikovnega zaznavala, npr. *1/2"*,
- velikost slikovnega elementa, npr. 5,86 μm,
- izvedba priključka objektiva, npr. C/CS,
- digitalna komunikacija, npr. GigE,
- konektor, npr. Gigabit Ethernet (RJ45),
- napajanje, npr. 12 VDC ali POE (power ower internet): 48 VDC do 56 VDC,
- proženje,
- I/O-signalna linija,
- mehanske dimenzije, npr. š: 29 mm, v: 29 mm, d: 57 mm,
- teža, npr. 65 g,
- temperaturno območje delovanja; dovoljena vlažnost okolja,
- nastavitve, npr. razpon časov zajema slike, ojačitev (ekvivalent ISO),
- podprte platforme in operacijski sistemi (32/64, ARM, Windows, Linux) ter pripadajoči gonilniki ter programska razvojna okolja (SDK).

Še malo dodatne razlage glede izbire kamer: izbira barvne ali črno-bele kamere je stvar presoje, katere optične efekte bomo uporabili. Če uporabimo npr. enobarvno osvetljevanje in ozkopasovni svetlobni filter pred kamero, potem je črno-bela kamera najboljša izbira, saj optični sistem prepušča zgoj eno barvo in tudi vzorčenje je boljše kot v primeru barvnega slikovnega zaznavala. Pri izbiri digitalne komunikacije je odločilno, koliko kamer bomo uporabili in kako daleč od računalnika bodo, tj. kako dolgi bodo vodniki. USB-komunikacije so omejene z dolžino vodnikov do 5 m, potem potrebujemo repetitorje. Pri mrežnih (GigE) je kamera lahko več deset metrov daleč od računalnika. Pri avtomatiziranih celicah moramo vodnike napeljati po predvidenih kanalih, ki niso najkrajše možne poti, zato dolžine vodnikov zelo hitro narastejo do 10 metrov in več. Pogosto je problem priključiti več kamer na računalnik, saj smo omejeni s številom priključnih konektorjev. Pri mrežnih kamerah lahko uporabimo mrežno stikalo, kamor priključimo več kamer, od njega pa peljemo samo eno povezavo na računalnik (pravilna nastavitev IP-naslovov). To je še posebej ugodna rešitev pri kontrolnih napravah z veliko kamerami. Seveda pa kamere ne morejo sočasno delovati z najvišjim FPS. Rešitev se uporablja predvsem, kadar kamere prožimo z zunanjim signalom v sekvenci glede na zaporedje delovnih operacij.

2.4 Primeri izbire kamere in izračuna objektiva

2.4.1 Vgradnja kamere na montažno linijo za kontrolo izdelkov

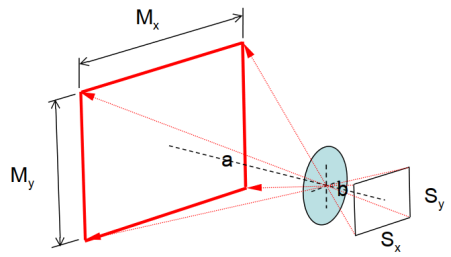
S strojnim vidom bomo izvajali kontrolo sestavnih delov na montažni liniji. Preverjamo, ali je na montažo prišel izdelek ustreznega tipa. Kontrolirani sestavni del je velikosti 5 x 3 cm in je vpet v držalo, ki ga pomika po vodilih do montažnega mesta. Na izdelku je natisnjena ID-koda, ki jo moramo najprej prepoznati, nato pa preveriti, ali je ustreznega tipa. Velikost teksta, ki tvori kodo, je 2 mm. Koda je črne, izdelek pa peščeno rjave barve. Kamero lahko namestimo 300 mm nad izdelkom. Kamero bo prožil PLC v trenutku, ko bo senzor prisotnosti zaznal paleto z izdelkom. Računalnik za obdelavo slike bo nameščen v krmilni omari avtomatske montažne linije. Dolžina kanalov, po katerih lahko napeljemo kable od kamere do računalnika, je približno 12 m. Na montažni liniji je napajanje 24V DC.

Iz navedenega opisa moramo najprej ugotoviti zahtevane specifikacije sistema. Velikost izdelka ustreza velikosti objekta, torej $O = 50$ mm. To velja v opisanem primeru, saj je izdelek vedno enako vpet in enako pozicioniran. Če bi bil naključno pozicioniran, potem bi moralo biti merilno območje kamere večje. Objekt je na oddaljenosti $a = 300$ mm. Za uspešno prepoznavo črk v tekstu moramo te vzorčiti z več kot 10 slikovnimi elementi (mejno vzorčenje). Če so črke visoke 2 mm, potem naj slikovnemu elementu na objektu pripada 2 mm/10 = 0,2 mm. Če bomo izbrali kamero z vsaj 1000 slikovnimi elementi v eni od dimenzij slikovnega zaznavala, potem bo velikost slikovnega elementa na objektu 50 mm/1000 = 0,05 mm. Preverimo vzorčenje 2 mm/0,05 mm = 40. To pomeni, da bodo črke v ID-kodi vzočene z najmanj 40 slikovnimi elementi, kar omogoča dobro vidljivost teksta. Kamera bo morala imeti zunanje proženje in napajanje 24V DC. Glede na dolžino kanalov do računalnika bomo uporabili mrežno kamero. Ker v slikovnem sistemu nastopata samo dve kontrastni barvi, črna in peščeno siva, bo povsem zadosti, če izberemo črno-belo kamero.

Proizvajalcev kamer je veliko in pogosto nastopi dilema, katero znamko izbrati. Za izbiro so pogosto odločilne prehodne izkušnje s katero od kamer, in to predvsem z vidika programiranja in vgradnje kamere. Pogosto je odločilna najnižja cena ali pa da jo že imamo od kakšnega predhodnega projekta. Recimo, da smo izbrali kamero DMK 33GP1300 proizvajalca Imaging source. Na podatkovnem listu so naslednji podatki (v angleščini):
– GigE Interface (RJ45) ... mrežna kamera,
– 1/2” onsemi CMOS Python 1300 Sensor ... velikost in model slikovnega zaznavala,
– 1280 × 1024 (1,3 MP), up to 90 FPS ... število slikovnih elementov, FPS,
– pixel size: H: 4,8 µm, V: 4,8 µm ... velikost slikovnih elementov,
– global shutter ... zajem slike,
– trigger and I/O-inputs ...zunanje proženje,
– dimensions 29 × 29 × 57 mm ... zunanje dimenzije ohišja.

Naslednji korak je izračun objektiva. Izhajamo iz velikosti slikovnega zaznavala in velikosti merilnega območja (glej sliko 2.19). Za slikovno zaznavalo je podan podatek *1/2”*, kar ustreza 6,4 x 4,8 mm. Če pomnožimo dejansko število slikovnih elementov z njihovo velikostjo, pridemo do bolj točnih dimenzij: 1280 x 0,0048 mm = 6,144 mm in 1024 x 0,0048 mm = 4,915 mm.

Predpostavimo, da je daljša stran slikovnega zaznavala poravnana z daljšo stranjo izdelka. V tem primeru bo $S = 6{,}144$ mm, O ali $M = 50$ mm. Za oceno potrebne goriščne razdalje uporabimo enačbo (2.3):



Slika 2.19: Objektiv preslika merilno območje M na slikovno zaznavalo S. Ob znani velikosti slikovnega zaznavala S in povečavi objektiva b/a izračunamo velikost merilnega območja M.

$$f = \frac{a \cdot S}{O + S} = \frac{300 \text{ mm} \cdot 6{,}144 \text{ mm}}{50 \text{ mm} + 6{,}144 \text{ mm}} = 32{,}83 \text{ mm}. \tag{2.12}$$

Na tej osnovi pri istem proizvajalcu izberemo prvi najbližji objektiv, npr. TCL 3520 5MP. Prvi pomemben podatek je goriščna razdalja $f = 35$ mm, ki je kar precej blizu izračunane vrednosti. Format slike objektiva je *2/3"*, kar pomeni, da bo objektiv s sliko v celoti pokril slikovno zaznavalo velikosti *1/2"*. Objektiv ima C-mount pritrditev na kamero, poleg tega pa ima še na sprednji strani M27 × 0,5 mm fin navoj za pritrditev optičnega filtra, če bo potrebno. Glede na to, da se goriščna in izračunana razdalja ne ujemata, preverimo dejansko velikost vidnega polja kamere M na razdalji $a = 300$ mm. Izračun je podoben prikazanemu v poglavju 2.2.1 o optični preslikavi. Iz enačbe (2.1) izračunamo, kolikšna je oddaljenost slike od točke optične preslikave v objektivu b

$$\frac{1}{b} = \frac{1}{35 \text{ mm}} - \frac{1}{300 \text{ mm}} \rightarrow b = 39{,}623 \text{ mm}. \tag{2.13}$$

Izračunamo novo širino merilnega območja

$$m = \frac{b}{a} = \frac{S}{O} \rightarrow O = \frac{S \cdot a}{b} = \frac{6{,}144 \text{ mm} \cdot 300 \text{ mm}}{39{,}623 \text{ mm}} = 46{,}52 \text{ mm}. \tag{2.14}$$

Merilno območje je za 3,5 mm manjše od velikosti kontroliranega objekta. Rešitev je v tem, da odmaknemo kamero in povečamo razdaljo a. S tem se bo tudi povečalo merilno območje M.

Izračun se začne pri enačbi za povečavo (2.2), kjer izračunamo razmerje b/a, to pa vstavimo enačbo za optično preslikavo (2.1):

$$m = \frac{b}{a} = \frac{S}{O} \rightarrow m = \frac{6{,}144 \text{ mm}}{50 \text{ mm}} = 0{,}123 \rightarrow b = 0{,}123\ a \tag{2.15}$$

$$\frac{1}{a} + \frac{1}{b} = \frac{1}{f} \rightarrow \frac{1}{a} + \frac{1}{0{,}123a} = \frac{1}{f} \rightarrow \frac{1}{a} 9{,}138 = \frac{1}{f} \rightarrow a = 9{,}138 \cdot f = 319{,}8 \text{ mm}. \tag{2.16}$$

Za enako merilno območje moramo kamero odmakniti za 19,8 mm. Pri konstruiranju nosilcev na avtomatski celici moramo v naprej predvideti takšne scenarije in pustiti več prostora za morebitno premikanje kamere vzdolž optične osi.

Preverimo, ali bomo lahko izostrili sliko s tem, da objektiv privijemo na kamero brez distančnih obročev. Razdalja MOD-objektiva znaša 0,4 m, opazovani objekt je bližje, zato slike ne bo možno izostriti in rabimo distančni obroč. Izračun je enak predhodno opisanemu v poglavju o objektivih. Najprej se vprašajmo, kje bi nastala slika, če bi se objekt nahajal točno v MOD:

$$\frac{1}{b_{\text{MOD}}} = \frac{1}{f} - \frac{1}{\text{MOD}} \rightarrow \frac{1}{35 \text{ mm}} - \frac{1}{400 \text{ mm}} \rightarrow b_{\text{MOD}} = 38{,}356 \text{ mm}. \tag{2.17}$$

Podobno izračunamo, kje nastane slika, če se objekt nahaja na razdalji $a = 319{,}8$ mm

$$\frac{1}{b_\mathrm{a}} = \frac{1}{35\ \mathrm{mm}} - \frac{1}{319{,}8\ \mathrm{mm}} \rightarrow b_\mathrm{a} = 39{,}3\ \mathrm{mm}. \tag{2.18}$$

Njuna razlika 39,3–38,35 mm znaša 0,94 mm. Tej razdalji prištejemo še polovico razdalje med b_{MOD} in f, torej 1,678 mm. Končna vrednost distančnega obroča tako znaša 0,94 mm + 1,678 mm = 2,62 mm. To zaokrožimo na najbližjo polno mero, npr. na 2,5 mm. Ostrina v tem primeru ne bo čisto točno na sredini območja nastavljanja ostrine slike na objektivu, bomo pa verjetno dobili distančni obroč standardne dimenzije.

2.4.2 Vgradnja kamere na mobilnega robota

Za potrebe vizualne navigacije na mobilnem robotu želimo uporabiti Raspberry Pi vgradni računalnik in njemu pripadajočo kamero (Raspberry Pi module). Za kamero so znani naslednji podatki:

- slikovno zaznavalo: Sony IMX219PQ barvni CMOS 8M,
- dimenzija: 3,674 x 2,760 mm (1/4" format),
- število slikovnih elementov: 3280 x 2464,
- velikost slikovnega elementa: 1,12 x 1,12 μm,
- objektiv: $f = 3{,}04$ mm, f/d = 2,0,
- vidni kot kamere: 62,2 x 48,8 stopinj,
- CSI-digitalna komunikacija med kamero in računalnikom.

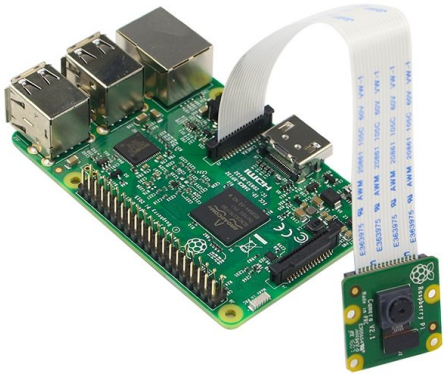
Kamera omogoča zajem videa. Za potrebe navigacije bomo vsako sliko v videu obdelali v realnem času in iz nje izračunali navigacijske parametre. V realnem času pomeni takoj, ko bo slika s kamere prenešena v spomin računalnika. Obdelava slike se mora zaključiti, še preden prispe nova slika. Če gre za video z 90 FPS, je za obdelavo na voljo zelo malo časa, zato morata biti obdelava in tudi zajem zelo domišljena. V ta namen najprej raziščimo videoformate slike:

- 1080p: resolucija 1920 x 1080, 30 FPS (originalna velikost slike 3280 x 2464 je porezana za 680 slikovnih elementov z leve in desne ter 692 slikovnih elementov zgoraj in spodaj);
- 3280 x 2464, 4:3, 15 FPS;
- 720p: 1280 x 720, do 90 FPS. Za izračun objektiva je pomembno razumeti, na kakšen način je zmanjšana resolucija pri videoformatih, tj. ali se je dimenzija slikovnega zaznavala zmanjšala tako, da je vzet zgolj ROI (kot v primeru 1080p) ali je na kakšen drug način zmanjšana resolucija ob tem, da je velikost slikovnega zaznavala ostala enaka. Po podatkih proizvajalca je v primeru formata 720p originalna velikost najprej porezana za 360 slikovnih elementov z leve in desne ter 512 od zgoraj in spodaj, nato pa je izvedeno združevanje 2 x 2 slikovnih elementov ("ang. binning"). V kontekstu obdelave slik "binning" označuje postopek združevanja sosednjih slikovnih elementov v navidezno večji slikovni element. Na primer, pri združevanju 2 x 2 sosednjih slikovnih elementov se skupno število slikovnih elementov zmanjša na 1/4 prvotne velikosti, resolucija pa prepolovi, npr. $(3280 - 2 \cdot 360)/2 = 1280$ in $(2464 - 2 \cdot 512)/2 = 720$. Rezultat združevanja je lahko vsota, povprečje, mediana, najmanjša ali največja vrednost skupine 2 x 2. Za nadaljnje izračune moramo upoštevati novo velikost slikovnih elementov, npr. 2 x 1,12 μm = 2,24 μm. Nova dimenzija slikovnega zaznavala je 1280 x 2,24 μm = 2,87 mm in 720 x 2,24 μm = 1,16 mm.

Glede na to, da gre za majhno slikovno zaznavalo in majhen objektiv, naredimo hitro oceno premera PSF:

$$C = 2{,}5\lambda\frac{f}{d} = 2{,}5 \cdot 0{,}65\ \mu m \cdot 2 = 3{,}25\ \mu m \tag{2.19}$$

Vidimo tipičen problem poceni objektivov, ki niso zmožni narediti zadosti ostre slike, saj je velikost slikovnega elementa še po združevanju manjša od PSF (2,24 μm $\leq$ 3,25 μm). S tem objektivom uporaba formata 1080p ali polne resolucije ni smiselna, saj velike slike zgolj obremenjujejo računske zmogljivosti, ne vsebujejo pa nobene dodatne informacije od 720p. Pri takšnem objektivu bi bilo smiselno 3 x 3-združevanje slikovnih elementov (česar slikovno zaznavalo ne omogoča) ali pa zamenjava objektiva. Za primer robotske navigacije bi bilo smiselno sliko najprej prenašati v formatu 720p, zato da dosežemo visok FPS, prva operacija obdelave slike pa bi bilo še eno združevanje, npr. slikovnih elementov 2 x 2 ali 3 x 3, s čimer bi bistveno zmanjšali velikost slike in pohitrili vse nadaljnje operacije obdelave slike, in sicer ob minimalni izgubi informacije.



Slika 2.20: Raspberry Pi vgradni računalnik in pripadajoča kamera

Izdelamo še oceno merilnega območja kamere na oddaljenosti 2 m v primeru, da sliko prenašamo v 720p-formatu. Izračun začnemo z izračunom razdalje b, ki jo nato vstavimo v enačbo za povečavo en. (2.2). Pri izračunu b lahko naredimo poenostavitev, saj pri objektivih s kratkim goriščem neskončnost nastopi že blizu kamere, zato je $b = f$. Velikost merilnega območja

$$O = S\frac{a}{b} = 2.87\frac{2000}{3} = 1853\ \text{mm}. \tag{2.20}$$

Podobno izračunamo še v vertikalni smeri 773 mm. Pogosto nas zanima vidni kot kamere (ang. field of view ali "FOV"). Izračunamo ga na osnovi trigonometrije

$$tan(\alpha) = \frac{0{,}5 \cdot O}{a} = \frac{0{,}5 \cdot 1853}{2000} \rightarrow \alpha = 24{,}8° \tag{2.21}$$

Vidni kot kamere je $FOV = 2 \cdot \alpha$, kar znaša 49° v horizontalni in 22° v vertikalni smeri. S tremi kamerami, nameščenimi frontalno pred robota, tako pokrijemo vidni kot 150°.

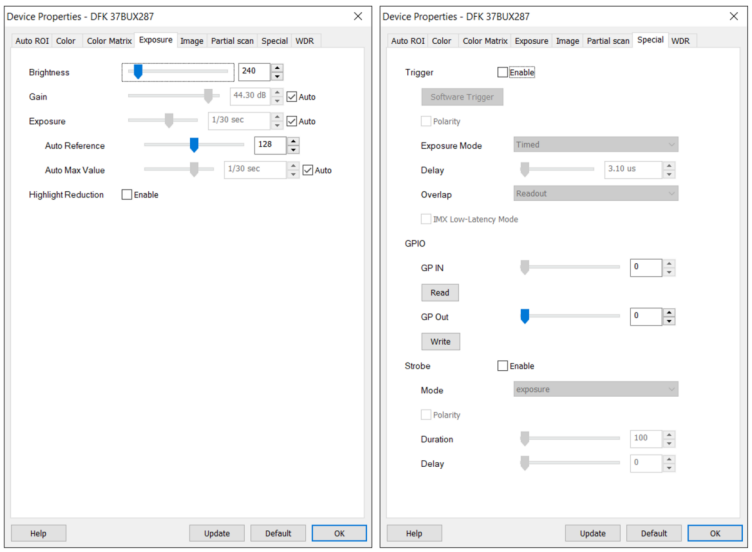
2.5 Krmiljenje kamer in zajem slike

Na računalniku, kamor bo kamera priključena, moramo najprej inštalirati gonilnike. Pri tem se je priporočljivo držati navodil proizvajalca. Mnogokrat je potrebno zagnati inštalacijo gonilnikov

in šele potem glede na zahtevo inštalacijskega programa priključiti kamero. Na tej točki lahko naletimo na množico ovir, še posebej če uporabljamo poceni kamere z generičnimi gonilniki.

Osnovna funkcionalnost gonilnikov je vmesnik med strojno opremo, tj. kamero, in uporabniškim programom. Na eni strani gonilnik obvladuje digitalno komunikacijo med kamero in računalnikom, sprejema pakete podatkov, jih ustrezno interpretira in zapiše v spomin računalnika. Z uporabniškim programom dostopamo do slike v spominu in jo obdelujemo po želji.

Gonilnik služi tudi upravljanju z delovanjem kamere. Preko njega nastavljamo čas in ojačitev pri zajemu slike, videoformat, FPS, ROI itd. Profesionalne industrijske kamere omogočajo enostavno nastavljanje parametrov zajema slike v dialog oknih gonilnika, kot to prikazuje slika 2.21. Pri vgradnih kamerah in cenejših izvedbah kamer pa to ni več tako enostavno. Kamero aktiviramo z določenimi nastavitvami, ki jih v nadaljevanju med delovanjem programa ne moremo več spreminjati.



Slika 2.21: Primer okna uporabniškega vmesnika gonilnika kamere

Proizvajalci kamer nudijo testno programsko opremo, pri kateri testiramo delovanje kamer z različnimi nastavitvami, zajemamo in shranjujemo individualne slike ali video. Resni proizvajalci industrijskih kamer nudijo knjižnice za krmiljenje kamer za programske pakete, kot so Matlab [3] in Python [4]; kar je najpomembnejše, je, da nudijo knjižnice in razvojna okolja za C++ (Software Development Kit−SDK), kjer imamo popoln nadzor nad delovanjem kamere in zajemom slike. S takšnim SDK razvijemo profesionalno aplikacijo obdelave slike.

Poglejmo si enostaven primer, kako v Pythonu aktiviramo kamero, kako dostopamo do slike in kje bo umeščena obdelava slike. Obdelava slik danes v veliki meri temelji na odprtokodni knjižnici za obdelavo slik "OpenCV" [5]. Knjižnica vključuje močno orodje za krmiljenje kamer in zajem slik z imenom "VideoCapture". Za primer, kako jo uporabiti, najprej v Pythonu inštaliramo knjižnico OpenCV (za Python) z ukazom:

```
pip install opencv-Python
```

Primer preprostega programa za zajem slike je sledeč:

```
# naložimo OpenCV knjižnico
import cv2
# definiramo "VideoCapture"objekt z imenom "kamera"
kamera = cv2.VideoCapture(0)
#dodaj
while(true):
```

```
        # Zajem slik
        status, slika = kamera.read()
        # Na to mesto umestimo obdelavo slike
        # ...
        # Prikaz slike na ekranu
        cv2.imshow('ImeOknaZaPrikazSlike', slika)
        # s tipko 'q' bomo ustavili delovanje programa
        if cv2.waitKey(1) & 0xFF == ord('q'):
        break
# Po prekinitvi while zanke sprostimo kamero
kamera.release()
#Zapremo vsa okna
cv2.destroyAllWindows()
```

V programu smo najprej z ukazom `import cv2` naložili knjižnico OpenCV. Naslednji ukaz `kamera = cv2.VideoCapture(0)` je zelo pomemben. Z njim definiramo `VideoCapture` objekt z imenom `kamera`. Parameter "0" pomeni, da bomo aktivirali privzeto kamero. Zelo pomembno je, da na koncu programa objekt `kamera` sprostimo z ukazom `kamera.release()`. S tem sprostimo spomin in gonilnik kamere. Če tega ne storimo, kamere naslednjič verjetno ne bomo mogli aktivirati in bo rešitev samo v ponovnem zagonu računalnika.

Osrednji del programa je `while` zanka, znotraj katere s kamere najprej zajamemo sliko s ukazom `status, slika = kamera.read()`. To sliko v nadaljevanju z raznimi funkcijami, ki jih omogoča OpenCV, obdelamo (kar v tem primeru ni prikazano). Na koncu v posebnem okno prikažemo zajeto sliko z ukazom `cv2.imshow('ImeOknaZaPrikazSlike', slika)`. V prikazanem primeru smo izrisali izvorno sliko, kot smo jo zajeli s kamere, lahko pa bi prikazali že obdelano sliko.

Delovanje programa prekinemo tako, da zaključimo `while` zanko z ukazom `break`, ki smo ga umestili v if stavek `if cv2.waitKey(1) & 0xFF == ord('q')`. V if stavku preverimo, ali je bila pritisnjena izbrana tipka `q` (lahko je poljubna tipka) ter ali je minila pavza dolžine 1 milisekunde. Pavza je lahko tudi daljša, potrebna pa je za pravilno delovanje izrisa v sklopu ukaza `cv2.imshow`. Če ne bi ničesar izrisovali, potem pavze ne potrebujemo.

Na tem mestu je potrebno opozoriti na veliko pomanjkljivost predstavljenega programa. V njem ni obvladovanja statusov. Pri aktivaciji kamere bi morali preveriti, ali je kamera uspešno aktivirana, in prekiniti delovanje programa, če ne bi bila. Podobno bi morali preverjati odzive funkcij znotraj `while` zanke, še posebej pri zajemu slike. Najmanj, kar moramo storiti, je, da preverimo, ali smo kamero uspešno odprli, kot to prikazuje naslednji primer. Če smo, še pridobimo dimenzije slike (za izračune v nadaljevanju) in šele potem začnemo zajemati slike znotraj *while* zanke.

```
...
if kamera.isOpened():
        stolpcev_U  = kamera.get(cv2.CAP_PROP_FRAME_WIDTH)
        vrstic_V = kamera.get(cv2.CAP_PROP_FRAME_HEIGHT)
        FPS = kamera.get(cv2.CAP_PROP_FPS)
        # ali
        stolpcev_U  = kamera.get(3)
        vrstic_V = kamera.get(4)
        FPS = kamera.get(5)
        ...
        while(true):
                ...
```

Sedaj si še oglejmo primer aktivacije kamere in zajema slik v programskem okolju C++. Še največ časa bomo porabili za inštalacijo OpenCV za C++. Na voljo sta dve možnosti: prva je, da prenesemo že prevedene knjižnice OpenCV. Pri tem moramo paziti, da izberemo pravilni operacijski sistem. Druga možnost je, da prenesemo izvorno kodo OpenCV in jo na svojem

računalniku prevedemo s pomočjo `cmake` ukazov. Slednji način je precej zahteven, dobimo pa najbolj primerno prevedeno kodo za svoj računalnik in njegov operacijski sistem. Primer C++ programa v nadaljevanju ima logiko, ki je enaka predhodno opisani. Prikazuje, kako najprej vključimo nujno potrebne OpenCV vključne ("header") datoteke, s katerimi v programu aktiviramo funkcionalnost OpenCV. Program vsebuje tudi obvladovanje statusov in možnost aktivacije kamere po izbiri.

Datoteka z glavo `core.hpp` je osnovni modul OpenCV. Vsebuje osnovne funkcije in strukture, ki so potrebne za skoraj vse OpenCV-programe. Te vključujejo upravljanje z matrikami (`cv::Mat`), matematične operacije, algoritme za linearno algebro, vektorske in osnovne podatkovne tipe ter splošno infrastrukturo knjižnice OpenCV. Vključna datoteka `videoio.hpp` je del modula za delo z videi. Vsebuje funkcije za zajemanje videoposnetkov s kamer ali datotek in shranjevanje videoposnetkov. Omogoča tudi podporo za različne videoformate in kodirnike ter funkcije za obdelavo posameznih videosličic. Vključna datoteka `highgui.hpp` služi za grafične uporabniške vmesnike. Zagotavlja funkcije za ustvarjanje in upravljanje oken, prikazovanje slik, obdelavo dogodkov tipkovnice in miške ter upravljanje preprostih GUI-elementov za interakcijo s programi OpenCV. Izraz `using namespace cv;` omogoča uporabo vseh funkcij in razredov s prostora imen `cv` brez potrebe po njegovem (tj. `cv:`) ponovnem navedku pri vsakem ukazu. Prostor imen `cv` vključuje vse glavne funkcionalnosti knjižnice OpenCV, kot so operacije z matrikami, obdelava slik in videov ter različni algoritmi za strojni vid.

```
//Vključne datoteke
include <opencv2/core.hpp>
include <opencv2/videoio.hpp>
include <opencv2/highgui.hpp>
include <iostream>
include <stdio.h>
using namespace cv; //znebimo se cv::  pred funkcijami
using namespace std;//znebimo se std::  pred funkcijami

int main(int, char**)
{
        Mat slika; //Rezerviramo spomin za sliko
        VideoCapture kamera; //Aktivacija objekta VideoCapture
        int deviceID = 0; //Aktivirali bomo privzeto kamero
        int apiID = cv::CAP_ANY; //Privzet API
        kamera.open(deviceID, apiID); // Odpre kamero
        //Preverimo, ali smo uspeli aktivirati kamero
        if (!kamera.isOpened()) {
                cerr << "NAPAKA! Kamera ni aktivirana.\n";
                return -1;
        }
        for (;;)
        {
                kamera.read(slika);  // zajem slike
                if (slika.empty()) { // preveri uspeh
                        cerr << "NAPAKA! Slika nima podatkov.\n";
                        break;
                }
                //Tu pride obdelava slike
                //...
                imshow("Live", slika);// Prikaz slike v oknu
                if (waitKey(5) >= 0)  // Ali je katera tipka pritisnjena?
                        break; //da, prekinemo zajem slik
        }
        kamera.release()
        return 0;
}
```

V prejšnjem primeru smo uporabili privzeto kamero in tudi privzeti API. API (ang. application programming interfaces) je vmesnik (tj. programska oprema), ki se izvaja v ozadju in omogoča različnim komponentam programske opreme, da medsebojno komunicirajo. API-ji določajo, kako lahko različni deli sistema medsebojno sodelujejo in izmenjujejo podatke.

Primer API-ja je npr. GStreamer knjižnica (v Linuxu), ki predstavlja univerzalen vmesnik med gonilnikom kamere in uporabniškim programom. API omogoča sestavljanje različnih funkcijskih gradnikov (zajem, dekodiranje, nastavljanje formatov, ločevanje zvoka in videa, kompresija, prikaz, shranjevanje) v verigo za obdelavo zvoka in videa.

Primer v nadaljevanju prikazuje, kako GStreamer API-ju v obliki dolgega tekstovnega ukaza (stringa) damo navodila, kako naj konfigurira verigo (pipeline) za zajem videa (kamera, čas zajema slike, ojačitev, FPS), kako naj video dekodira in v kakšni obliki naj zapiše sliko v spomin računalnika. Celotna veriga je v tem primeru poimenovana ”pipeline”. Pogosto se v tem kontekstu veriga poimenuje tudi ”ang. graph”. Primer funkcije za generiranje tekstovnega ukaza za nastavitev verige funkcij v knjižnici GStreamer izgleda takole:

```
string generateGStreamerPipeline(int width, int height, int framerate) {
string pipeline =
"nvarguscamerasrc␣exposuretimerange=\"5000␣10000\"␣gainrange=\"5.1␣5.1\"
aelock=true␣!␣"
"video/x-raw(memory:NVMM),␣width=(int)" + std::to_string(width) +
",␣height=(int)" + std::to_string(height) +
",␣format=(string)NV12,␣framerate=(fraction)" +
std::to_string(framerate) + "/1␣!␣"
"nvvidconv␣!␣video/x-raw,␣format=(string)RGB␣!␣appsink";
return pipeline;
}

//Klicanje funkcije
int width = 640;
int height = 480;
int framerate = 30;

string pipeline = generateGStreamerPipeline(width, height, framerate);
cout << "GStreamer␣Pipeline:␣" << pipeline << std::endl;

/*Izpis GStreamer Pipeline:

nvarguscamerasrc exposuretimerange="5000 10000" gainrange="5.1 5.1"  aelock=true
     ! video/x-raw(memory:NVMM), width=(int)640, height=(int)480, format=(string
    )NV12, framerate=(fraction)30/1 ! nvvidconv ! video/x-raw, format=(string)
    RGB ! appsink
*/

//Aktivacija kamere v OpenCV
VideoCapture kamera(pipeline, cv::CAP_GSTREAMER);
```

Razumevanje tovrstnih verig je pomembno pri razvoju odprtokodnih aplikacij, zato si poglejmo podrobno razčlenitev prikazane verige. Prikazana veriga GStreamer je zasnovana za minimalno število notranjih pretvorb slik in minimalno zakasnitev med zajemom in prikazom slik. Cilj je zajeti video iz kamere in zagotoviti izhod v formatu RGB z minimalnim časom obdelave.

- `nvarguscamerasrc`:
  - To je gonilnik kamere (na platformah NVIDIA Jetson). Zajema video neposredno iz kamere z določenimi nastavitvami osvetlitve in ojačanja.

  - `exposuretimerange="5000 10000"`: Nastavi čas osvetlitve v nanosekundah (med 5000 in 10000 nanosekund).
  - `gainrange="5.1 5.1"`: Fiksira ojačanje na vrednost 5.1.
  - `aelock=true`: Zaklene samodejno prilagajanje osvetlitve.

- `!`:
  - Klicaj povezuje posamezne funkcije v verigi. Npr. `nvarguscamerasrc` z naslednjim elementom v verigi, ki je filter, ki določa lastnosti videotoka.

- `video/x-raw(memory:NVMM)`:
  - Določa, da je format videotoka izvorni (ang. raw) video v pomnilniku NVIDIA (`NVMM`), kar zmanjšuje kopiranje med pomnilnikom CPU in GPU.
  - Nastavi `width` in `height` neposredno za minimizacijo operacij spreminjanja velikosti.
  - Format `NV12` (YUV 4:2:0) je privzeti izhod kamere.
  - FPS je nastavljen na `framerate=(fraction)30/1`. Za večjo natančnost (v decimalkah) se FPS podaja v ulomkih (fraction).

- `nvvidconv`:
  - To je strojno pospešen element za pretvorbo formata. Pretvori format iz `NV12` (YUV) v `RGB`, kar omogoča optimalno in hitro pretvorbo s pomočjo NVIDIA-jeve strojne opreme.

- `appsink`:
  - Dovaja končne RGB-slike neposredno aplikaciji (npr. OpenCV funkcijam) za nadaljnjo obdelavo ali prikaz.

V dosedaj opisanih premerih je kamera zajela slike s konstantnim FPS, podobno kot če bi snemali video. Takšen pristop je primeren za npr. navigacijo, kjer kamera ves čas snema okolico mobilnega robota, slike pa sproti obdelujemo. Sliko je potrebno obdelati zelo hitro, še preden prispe naslednja. V nasprotnem začnemo izgubljati slike, FPS pa pade.

V avtomatiziranih proizvodnih linijah tipično uporabljamo industrijske kamere. Sliko zajemamo na osnovi proženja kamere s strani PLC: ko torej izdelek pride na določeno pozicijo, PLC vključi osvetlitev in v pravem trenutku sproži kamero. Tukaj kamera za vsak izdelek zajame eno sliko, zato ne moremo več neposredno govoriti o FPS, ampak bolj o taktu proizvodne linije. Z vidika krmiljenja kamere je kamera *master*, programska oprema pa *slave*. To pomeni, da programska oprema ves čas čaka kamero. Ko ta posreduje sliko, se programska oprema sproži, zajame in obdela sliko. V nadaljevanju je primer iz C++ SDK za *Imaging source kamere*. Programska oprema *posluša* kamero. To opravlja objekt tipa *CListener*. Znotraj njega je ena ključna funkcija, katere izpis sledi v nadaljevanju, kjer dostopamo do slike in jo znotraj te funkcije tudi obdelamo. Če povzamemo, za vsako, ko sprožimo kamero, se aktivira spodaj navedena funkcija *DoImageProcessing*, kjer obdelamo sliko.

```
void CListener::DoImageProcessing(smart_ptr<MemBuffer> pBuffer)
{
// Pridobi informacijsko glavo bitne slike iz pomnilniškega medpomnilnika.
// Vsebuje število bitov na slikovni element, širino in višino.

smart_ptr<BITMAPINFOHEADER> pInf = pBuffer->getBitmapInfoHeader();
```

```
// Zdaj pridobi kazalec na sliko.
// Za organizacijo podatkov o sliki si oglejte:
// http://www.imagingcontrol.com/ic/docs/html/class/Pixelformat.htm

BYTE* pImageData = pBuffer->getPtr(); // Kazalec na podatke

// Pridobi dimenzije slike.
Width = pInf->biWidth;
Height = pInf->biHeight;
BitCount = pInf->biBitCount;
// Izračun velikosti slike.
int iImageSize = pInf->biWidth * pInf->biHeight * pInf->biBitCount/8 ;

// Sledi obdelava slike ...  npr.  pragovna funkcija:
for (int i = 0; i < iImageSize; i++){
        if (pImageData[i]>Th) pImageData[i] = 255;
        else pImageData[i] = 0;
}
...
}
```

Vprašanja

- Kako vzpostavimo povezavo s kamero v OpenCV?
- Kako prepoznamo in izberemo točno določeno kamero?
- Kaj je in čemu služi API? Kako ga nastavimo?
- Kakšen je postopek za pridobivanje in obdelavo slik iz kamere v realnem času?
- Kako lahko spreminjamo nastavitve kamere, kot so osvetlitev, ojačitev in FPS?
- Kako shranimo sliko/videoposnetek s kamere?

Poglavje 3

Umeritev kamere

Umerjanje kamer je pomembno predvsem, kadar želimo izvajati dimenzijske meritve v milimetrih. Primer: pri dimenzijski kontroli s kamero slikamo kontrolirani objekt. Zajamemo sliko, jo analiziramo, določimo robove objekta na sliki, razdalje med njimi itd. Osnovni problem, s katerim se soočimo, je, da so vse dimenzije na sliki izražene v slikovnih elementih, za primerjavo s tehniško dokumentacijo pa potrebujemo dimenzije v milimetrih. Načeloma je možno narediti umeritev, kjer na osnovi znanih dimenzij objekta izračunamo, koliko slikovnih elementov pripada enemu milimetru. To je možno, kadar se razdalja med kamero in objektom ne spreminja ($a =$ konst) in je kamera nameščena pravokotno na opazovani objekt. Dodatno lahko ugotovimo, da je slika na robovih rahlo raztegnjena v primerjavi s sredino slike. Tovrstna popačitev izhaja iz optičnih popačitev objektiva in se imenuje distorzija. Obstajajo še druge optične napake objektivov, ki pa z vidika dimenzijskih meritev niso tako moteče, zato odpravljamo zgolj napako distorzije.

Za potrebe razpačitve distorzije, natančnih dimenzijskih meritev ter uporabe kamer v robotskih sistemih za lokalizacijo in navigacijo v nadaljevanju analiziramo koordinatne sisteme, ključne parametre, njihove merske enote, normiranje, opis in razpačitev distorzije ter postopek umerjanja s pomočjo umeritvenih teles.

3.1 Koordinatni sistemi

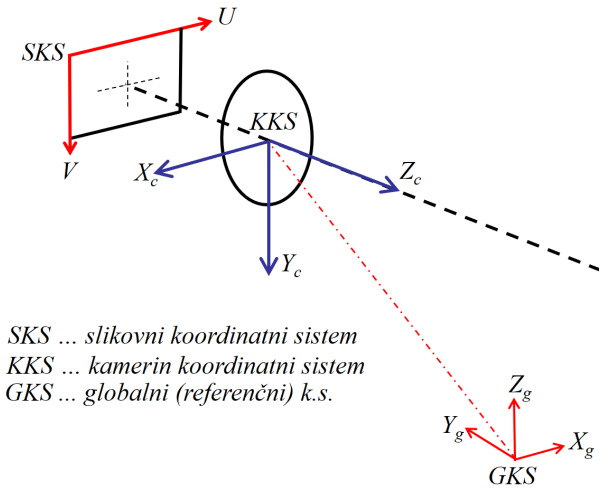
Pri opisu kamere obravnavamo tri koordinatne sisteme: slikovni, kamerin in referenčni ali globalni, kot prikazuje slika 3.1.

Slikovni koordinatni sistem (SKS) pripada sliki in je nameščen v zgornji levi vogal slike. Horizontalna os U je poravnana z zgornjim robom slike in usmerjena v desno. Vertikalna os V je poravnana z levim robom in je usmerjena navzdol. Verjetno ste se vprašali, čemu je SKS v zgornjem levem vogalu in obrnjen navzdol. Razlog za to je v zapisu slike v spominu računalnika v eno dolgo vrstico, tako da je najprej zapisana prva zgornja vrstica slike, takoj v nadaljevanju sledi zapis druge vrstice od zgoraj navzdol itd. Posledično u narašča od leve proti desni in v od zgoraj navzdol. Osnovna merska enota v SKS je slikovni element (pixel). To je logično, saj karkoli izluščimo s slike (npr. rob, težišče), je v izraženo v slikovnih elementih na sliki. Koordinate na sliki u in v so vedno izražene v slikovnih elementih! Iz tega razloga uporabljamo notacijo *(v, u)* in ne *(x, y)*, saj je slednja rezervirana za kartezični koordinatni sistem, ki je vedno v milimetrih.

Kamerin koordinatni sistem (KKS) določa lego kamere v 3D-kartezičnem prostoru. Uporabimo ga vedno, kadar opisujemo lego kamere glede na neko drugo napravo, npr. lega kamere glede na koordinatni sistem robotskega prijemala. Izhodišče KKS je pripeto v točko optične preslikave v objektivu, Z_c-os sovpada z optično osjo objektiva in je usmerjena navzven

v smeri pogleda kamere. Os Y_c je vzporedna z vertikalnim robom slikovnega zaznavala in usmerjena navzdol enako kot V-os SKS. X_c-os je prav tako poravnana s slikovnim zaznavalom in je usmerjena v levo glede na pravilo desne roke. Osnovna merska enota v KKS je milimeter. (x_c, y_c, z_c) koordinate so izražene v milimetrih, zasuki okrog osi pa v radianih.

Globalni ali referenčni koordinatni sistem (GKS) je kartezični koordinati sistem, glede na katerega podajamo relativno lego KKS. V prejšnjem primeru bi to bil npr. koordinatni sistem robotskega prijemala. Lahko pa bi bil tudi kateri drug koordinatni sistem, npr. globalni k. s. robota, k. s. delovne mize, KKS druge kamere itd. Osnovne merske enote v GKS so enako kot v KKS milimetri in radiani.



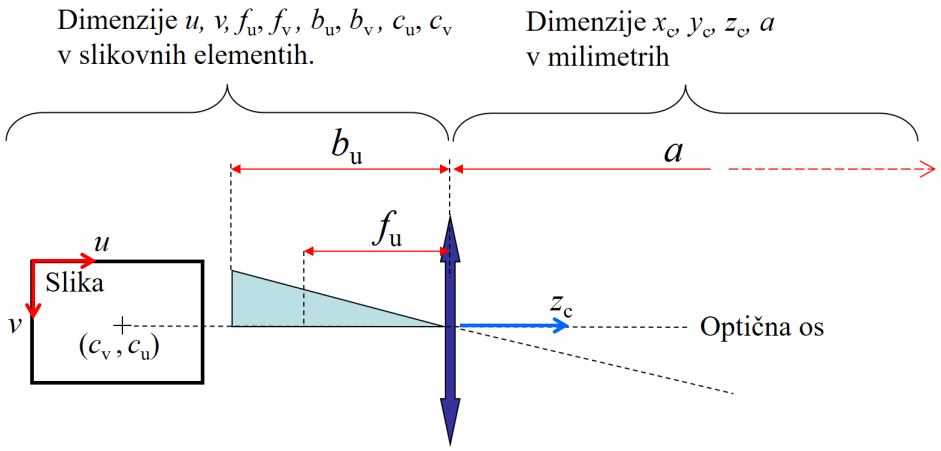
Slika 3.1: Koordinatni sistemi

Parametri in merske enote

Pri modelu kamere ločimo med notranjimi in zunanjimi parametri. Notranji opisujejo notranjost kamere, vse od točke optične preslikave v objektivu do slike. Zunanji parametri opisujejo lego kamere v prostoru, tj. KKS glede na GKS. Slika 3.2 povzema obe kategoriji parametrov, ki smo ju spoznali že predhodno.

Notranji parametri so razdalja med točko optične preslikave v objektivu in sliko (b_v, b_u), koordinate točke, kjer optična os prebada slikovno zaznavalo (c_v, c_u), ter goriščna razdalja objektiva (b_v, b_u), ki je tudi izražena v slikovnih elementih. Takoj opazite, da so vsi parametri podani dvojno, v v in u smeri slikovnega zaznavala, za primer anamorfnega sistema s povečavo različno v eni in drugi osi. Pogosto med v in u ni razlike, zato lahko operiramo samo z enovitimi b, f parametri. Vsi ti parametri imajo mersko enoto slikovni element (se, ang. pixel). Kot rečeno, je tudi goriščna razdalja objektiva v notranjosti kamere izražena s slikovnimi elementi. Če je slikovni element velikosti npr. 5 μm in dejansko gorišče objektiva $f = 16$ mm, potem je $b_u = 16/0{,}005 = 3200$ slikovnih elementov.

Zunanji parametri opisujejo lego kamere glede na globalni (referenčni) koordinatni sistem npr. robotskega prijemala (TCP). Za ta opis potrebujemo homogeno transformacijsko matriko, ki jo določajo premik izhodišč koordinatnih sistemov $T = (t_x, t_y, t_z)$ in trije koti rotacij okrog osi x, y, z (Eulerjevi ali Cardanovi koti v odvisnosti od rotacijskega modela), s katerimi določimo rotacijsko matriko R.



Slika 3.2: Parametri, ki opisujejo notranjost kamere, so izraženi v slikovnih elementih, medtem ko parametri, ki opisujejo zunanjost, v milimetrih.

3.2 Normiranje

Pri računanju koordinat v 3D-prostoru izhajamo s slike, npr. za potrebe lokalizacije poiščemo težišče objekta na sliki in določimo njegove koordinate $(v,\ u)$ (ki so izražene v slikovnih elementih, se). Potrebujemo koordinate objekta v KKS v milimetrih. Naša naloga je, da preslikamo koordinate težišča $(v,\ u)$ iz SKS v KKS. Kot smo predhodno že razložili, imajo vsi parametri znotraj kamere kot mersko enoto slikovni element, koordinate izven kamere pa milimeter. Za vse nadaljnje izračune se moramo znebiti dveh merskih sistemov, kar pomeni, da moramo vse koordinate prevesti v brezdimenzijske enote. Poiskati moramo, kaj je skupnega koordinatam kamere znotraj nje in kaj izven nje. Če predpostavimo model optične preslikave za tanke leče, ugotovimo, da so skupni koti objekta glede na optično os tako znotraj kamere kot v realnem prostoru. Slika 3.3 prikazuje, kako je kot φ enak zunaj in znotraj kamere. V praksi ne uporabljamo neposredno kotov, ampak razmerja stranic podobnih trikotnikov. Razmerje katet $(v - c_v)$ in b_v trikotnika znotraj kamere je enako razmerju katet y_c (O) in z_c (a) trikotnika izven kamere, kot to opisujeta enačbi (3.1) in (3.2). Razmerje je dejansko tangens kota; poimenovali ga bomo normirane koordinate in označevali z x_n in y_n. Zapomniti si moramo, da so normirane koordinate brezdimenzijske.

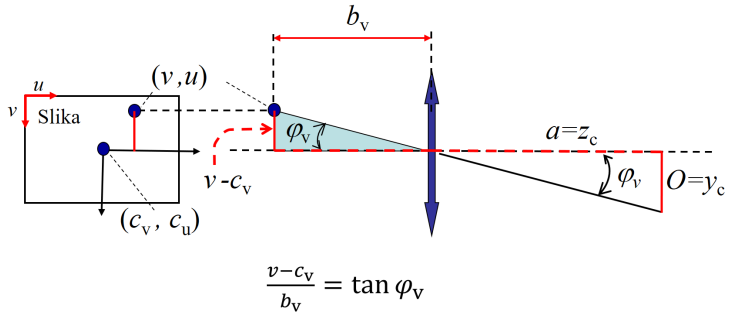
$$x_n = \frac{(u - c_u)}{b_u} = \frac{O_x}{a} = \frac{x_c}{z_c} = \tan\varphi_u \tag{3.1}$$

$$y_n = \frac{(v - c_v)}{b_v} = \frac{O_y}{a} = \frac{y_c}{z_c} = \tan\varphi_v \tag{3.2}$$

3.3 Računanje koordinat v 3D-prostoru

Navežimo se na prejšnji primer, kjer smo za potrebe lokalizacije poiskali težišče objekta na sliki in določili njegove $(v,\ u)$ koordinate na sliki, želimo pa določiti (x_c, y_c, z_c) koordinate težišča objekta v KKS. V prvem koraku koordinate $(v,\ u)$ normiramo, kot to opisujeta enačbi (3.1) in (3.2). V naslednjem koraku normirane koordinate pomnožimo z oddaljenostjo objekta od KKS, gledano vzdolž optične osi, torej z z_c oziroma z a, kot opisujeta enačbi (3.3) in (3.4).

$$x_c = x_n z_c \tag{3.3}$$



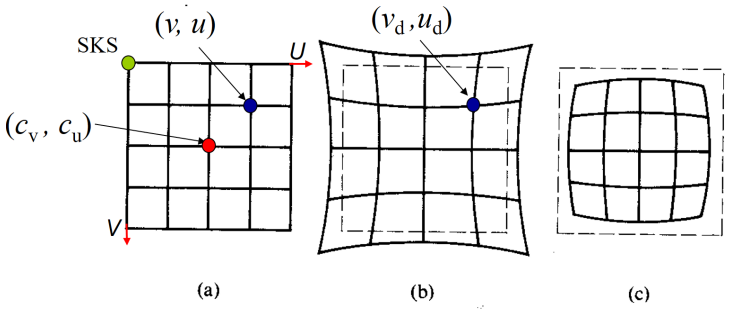
Slika 3.3: Normiranje. Koordinate izrazimo brezdimenzijsko.

$$y_c = y_n z_c \tag{3.4}$$

Za uspešen izračun moramo torej poznati notranje parametre (c_v, c_u) in (b_v, b_u) ter oddaljenost z_c. Notranje parametre določimo s pomočjo umeritvenih metod, kar bo razloženo v nadaljevanju. Največji problem je razdalja z_c, ki je ne poznamo. Če je kamera fiksno nameščena na neko delovno mesto, pravokotno na objekt, potem jo lahko izmerimo. Pogosto to ni mogoče, zato se uporabljajo metode triangulacije, kot je npr. laserska triangulacija s svetlobno ravnino ali pa stereovid, kjer se triangulira z drugo kamero, tako da se z_c izračuna s pomočjo trigonometrije.

3.4 Distorzija

Sliki 3.4 (b) in (c) prikazujeta, kako optična popačitev distorzije deformira sliko in posledično vpliva na izvedbo natančnih dimenzijskih meritev. Distorzija nastopa radialno glede na center slike, tj. glede na točko (c_v, c_u), kjer optična os prebada slikovno zaznavalo. Obstajata dve vrsti distorzije, pri **radialni** distorziji deformacija slike radialno simetrično narašča z oddaljevanjem od centra (c_v, c_u). Vzrok za nastanek radialne distorzije je v izvedbi lečevja in legi zaslonke v lečevju, najbolj opazna pa je pri objektivih s kratkim goriščem. **Tangencialna** distorzija sliko deformira v eliptičnem vzorcu glede na center slike. Nastopi, kadar slikovno zaznavalo ni povsem pravokotno vgrajeno glede na optično os objektiva, vzrok pa je tudi lahko v odstopkih pravilne lege lečevja v objektivu.



Slika 3.4: Distorzija slike. Nedeformirana slika (a), pozitivna distorzija (b), negativna distorzija (c)

Matematični zapis deformacij slike je podan z enačbami (3.5) do (3.8). Za zapis radialne distorzije je najprej pomembna oddaljenost od centra slike (c_v, c_u), ki jo podamo z radijem r po enačbi (3.5). Radialno distorzijo podamo s k_r po enačbi (3.6), ki v praksi pove, za koliko

se premakne točka na oddaljenosti r od centra slike ($x_{\mathrm{nd}} = k_{\mathrm{r}} \cdot x_{\mathrm{n}}, y_{\mathrm{nd}} = k_{\mathrm{r}} \cdot y_{\mathrm{n}}$). Če bi bila parametra k_1 in k_2 enaka nič, bi bil $k_{\mathrm{r}} = 1$, kar pomeni, da bi točka ostala na istem mestu (množenje z ena).

$$r = \sqrt{x_{\mathrm{nd}}^2 + y_{\mathrm{nd}}^2} \tag{3.5}$$

$$k_{\mathrm{r}} = 1 + k_1 r^2 + k_2 r^4 \tag{3.6}$$

Tangencialno distorzijo opišemo z enačbama (3.7) in (3.8). Ker ta deformira sliko eliptično glede na center slike, uporabimo ločeni enačbi za zapis deformacije v horizontalni Δx_{nt}- in vertikalni Δy_{nt}-smeri.

$$\Delta x_{\mathrm{nt}} = 2p_1 x_{\mathrm{n}} y_{\mathrm{n}} + p_2(r^2 + 2x_{\mathrm{n}}^2) \tag{3.7}$$

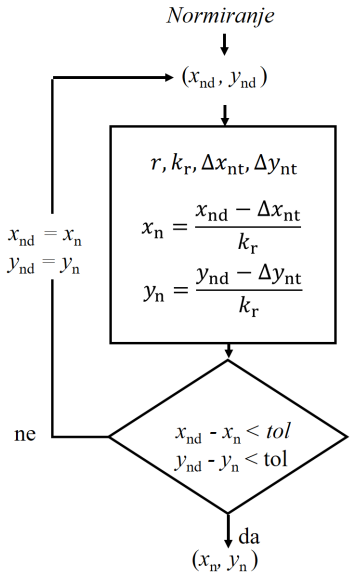
$$\Delta y_{\mathrm{nt}} = p_1(r^2 + 2y_{\mathrm{n}}^2) + 2p_2 x_{\mathrm{n}} y_{\mathrm{n}} \tag{3.8}$$

$$\begin{aligned} x_{\mathrm{nd}} &= k_{\mathrm{r}} x_{\mathrm{n}} + \Delta x_{\mathrm{nt}} \\ y_{\mathrm{nd}} &= k_{\mathrm{r}} y_{\mathrm{n}} + \Delta y_{\mathrm{nt}} \end{aligned} \tag{3.9}$$

Enačbe (3.5) do (3.8) podajo zapis distorzije, kako se izvorna slika deformira, pa zapišemo z enačbama (3.9). V praksi nas ne zanima, kako sliko deformirati, ampak kako deformirano sliko razpačiti. Omenjene enačbe moramo obrniti, tako da ob poznavanju parametrov distorzije k_1, k_2, p_1, p_2 pridemo do nedeformiranih koordinat $(x_{\mathrm{n}}, y_{\mathrm{n}})$. Parametre distorzije določimo pri umeritvi s pomočjo umeritvenega telesa, kar bo razloženo v nadaljevanju. Obrniti enačbe matematično ni tako enostavno (zaradi r^2), zato uporabljamo iterativni postopek razpačitve, kot je prikazano s postopkom na sliki 3.5. Razpačitev izvajamo po normiranju. Razlog za to je v koordinatnem sistemu. Distorzija nastopa radialno glede na središče slike $(c_{\mathrm{v}}, c_{\mathrm{u}})$, zato jo razpačimo glede na center slike, torej v koordinatnem sistemu, ki je pripet v središče slike. Ravno to pa dobimo po normiranju. Iterativni postopek razpačitve po normiranju je sledeč: najprej izračunamo vrednost radialne in tangencialne distorzije po enačbah (3.6) do (3.8). Nato od normiranih koordinat $(x_{\mathrm{nd}}, y_{\mathrm{nd}})$ odštejemo tangencialno distorzijo, dobljeno razliko pa delimo z radialno distorzijo (indeks nd pomeni normirane koordinate z distorzijo). S tem smo normirane koordinate malenkost razpačili. Preverimo, ali je odstopek novo izračunanih koordinat od vhodnih koordinat manjši od izbranega minimalnega odstopanja, npr. 10^{-5}. Če ni, opisani postopek ponovimo, s tem da v naslednji iteraciji za $(x_{\mathrm{nd}}, y_{\mathrm{nd}})$ vzamemo že delno razpačene koordinate iz predhodne iteracije. Algoritem hitro konvergira, tako da že po približno štirih iteracijah doseže minimalno odstopanje. Po razpačitvi distorzije nadaljujemo z izračunom koordinat v KKS, kot opisujejo enačbe (3.3) in (3.4) (če poznamo Z_{c}).

3.5 Umerjanje in umeritvena telesa

Cilj umerjanja kamer je določitev notranjih parametrov kamere $(b_{\mathrm{u}}, b_{\mathrm{v}}, c_{\mathrm{u}}, c_{\mathrm{v}})$ in (k_1, k_2, p_1, p_2) in po možnosti zunanjih parametrov, ki opisujejo lego kamere glede na referenčni koordinatni sistem (KKS glede na GKS). Umeritveni postopek si na enostaven način najlažje predstavljamo tako, da pomislimo, kaj se dogaja pri optični preslikavi na slikovno zaznavalo. Objekt v 3D-prostoru s koordinatami $(x_{\mathrm{c}}, y_{\mathrm{c}}, z_{\mathrm{c}})$ se skozi model kamere preslika na sliko, kjer ima koordinate (v, u). Lahko pa gremo tudi v obratni smeri, tj. s slike v 3D-prostor. Govorimo torej o treh stvareh: o modelu kamere, vhodnih podatkih in izhodnih podatkih. Vhodne podatke tvorijo koordinate



Slika 3.5: Iterativni postopek razpačitve distorzije

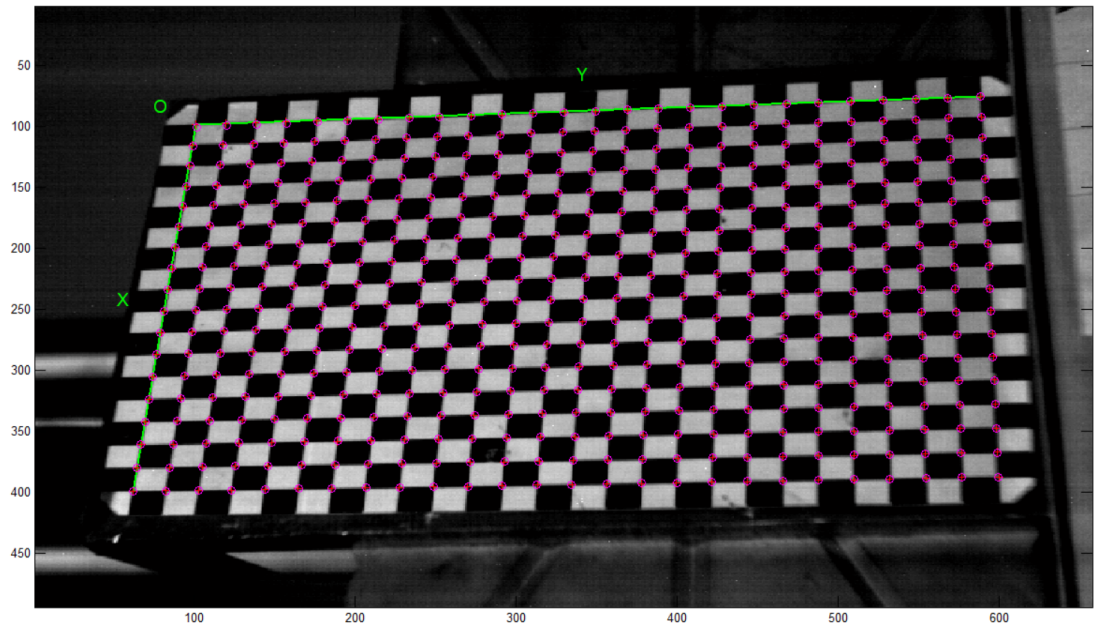
objekta oziroma točke v 3D-prostoru, torej (x_c, y_c, z_c). Izhodne podatke tvorijo koordinate iste točke na sliki, torej (v, u). Model kamere pa že poznamo: normiranje, razpačitev distorzije in izračun koordinat v 3D-prostoru.

Če želimo določiti parametre modela kamere, tj. notranje in zunanje parametre, moramo sočasno poznati vhodne in izhodne podatke. Ker gre za večje število parametrov modela in ker ti opisujejo nelinearno distorzijo, zgolj ena vhodna in ena izhodna točka ni dovolj. Potrebujemo jih več sto, ki so enakomerno razporejene znotraj vidnega polja kamere, zato da ugotovimo pravilno povečavo in popačitve distorzije. Govorimo torej o vhodnem in izhodnem podatkovnem setu, ki sestoji iz večje množice točk, kjer za vsako točko poznamo koordinate na sliki in v 3D-prostoru. Vse, kar še potrebujemo, je optimizacijski algoritem, npr. ena od gradientnih metod, ki med učenjem modela spreminja in fino nastavlja parametre modela kamere, vse dokler se vhodni podatkovni set ne preslikava v izhodni podatkovni set in obratno.

Ključna zadeva je, kako pridobiti podatkovne sete. V ta namen se uporabljajo umeritvena telesa s kodiranimi vzorci, npr. z vzorcem šahovnice, krogov ipd. Poglejmo si primer umeritvenega telesa v obliki ravne plošče s sliko šahovnice (slika 3.6). Zakaj šahovnica? Šahovnica je sestavljena iz črnih in belih kvadratov enakih dimenzij. Najbolj pomembni so vogali, kjer se kvadrati stikajo. Če postavimo koordinatni sistem v enega od stičišč kvadratov šahovnice, tako da je Z-os pravokotna na ploščo, X- in Y-osi pa sta poravnani z robovi kvadratov, potem lahko vsakemu stičišču kvadratov določimo koordinate v milimetrih. Primer: če je velikost kvadrata 10 mm, potem koordinate stičišč vogalov naraščajo po 10, 20, 30 itd. milimetrov v X- in Y-osi glede na izhodišče. Z-koordinata pa je za vse točke enaka 0, saj se izhodišče Z-osi nahaja na plošči. Ravna plošča s sliko šahovnice v 3D-prostoru določa množico točk (kolikor je stičnih vogalov med kvadrati) z znanimi koordinatami v milimetrih.

Umeritveno telo slikamo s kamero in zajamemo sliko šahovnice. Velikost kvadratkov na sliki se spreminja z oddaljenostjo od kamere, njihova popačitev pa s perspektivo, tj. pod kakšnim kotom jo slikamo, in v odvisnosti od distorzije. Na sliki določimo koordinate vogalov v koordinatnem sistemu slike. Vzorec šahovnice je izbran, da programsko z obdelavo slike čim bolj natančno določimo vogale med črnimi in belimi kvadrati.

Na tak način smo za vsak stični vogal med kvadrati sočasno pridobili podatek o koordinatah



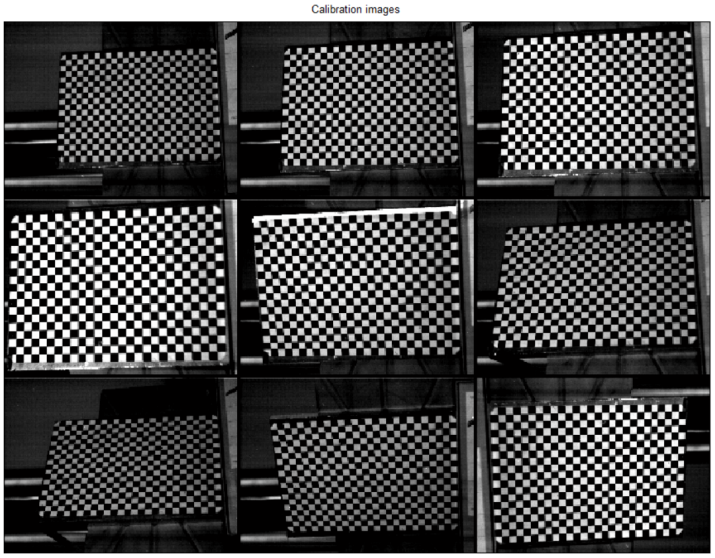
Slika 3.6: Umeritveno telo je ravna plošča s sliko šahovnice. Rdeči krožci označujejo stične vogale kvadratov, za katere poznamo koordinate $(x, y, 0)$ v koordinatnem sistemu šahovnice (določene so z velikostjo črnih in belih kvadratov).

na sliki in v 3D-prostoru. Ploščo je potrebno slikati in ponoviti postopek določanja vogalov na več različnih oddaljenostih in pod različnimi koti glede na kamero, da dobimo zadosti velik podatkovni set za optimizacijski algoritem. Pri umeritvi imamo največ dela s tem, da pripravimo umeritveno telo, ga poslikamo in poskrbimo, da so vogali pravilno določeni. Umeritveno ploščo s šahovnico lahko izdelamo sami. Na papir natisnemo šahovnico, po možnosti v merilu $1:1$, tako da so znane dimenzije kvadratov (preverimo z meritvijo čez več kvadratov!). Papir previdno nalepimo na ravno ploščo. Za umerjanje kamer obstaja več programskih paketov (npr. Camera calibration toolbox for Matlab [6], Camera calibration v OpenCV ipd.), ki nas vodijo skozi umeritveni postopek in na koncu tudi podajo analizo negotovosti umerjanja.

V nadaljevanju si bomo najprej pogledali primer umeritve s pomočjo Camera calibration toolbox for Matlab[1] (kjer je postopek najboljše razložen). Podoben postopek umeritve s funkcijami OpenCV v Pythonu bo sledil v nadaljevanju, podrobno pa razložen na spletni strani OpenCV[2]. Najprej umeritveno telo slikamo na več različnih oddaljenostih in pod različnimi koti glede na kamero, kot prikazuje slika 3.7. Slike naložimo v programski paket. Nato ročno pokažemo vogale vedno enakega (npr. 17 x 25) polja kvadratov na vsaki sliki posebej. Pri tem podamo začetni približek distorzije za vsako sliko in preverimo, ali je algoritem pravilno prepoznal vogale kvadratov. Sledi zagon optimizacijskega algoritma, ki po določenem številu iteracij doseže konvergenčne kriterije in izpiše rezultat optimizacije, kot prikazuje izpis na sliki 3.8. Rezultat je za vsak parameter podan v u- in v-smeri s pripadajočo negotovostjo. Merska enota rezultatov je slikovni element. Pozorni moramo biti na negotovost določanja parametrov, ki mora biti znotraj razumnih toleranc, npr. ± 5 slikovnih elementov, če je umeritev uspešna.

Po umeritvi lahko vizualiziramo različne popačitve. Slika 3.9a kaže primer, kako radialna distorzija narašča z oddaljevanjem od centra slike. Slika (b) prikazuje tangencialno distorzijo ter (c) seštevek radialne in tangencialne distorzije kot dejanske skupne popačitve slike. Običajno je dominantna radialna distorzija, tako da je vsaj velikostni red večja od tangencialne. V pri-

[1] `http://robots.stanford.edu/cs223b04/JeanYvesCalib/`
[2] `https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html`

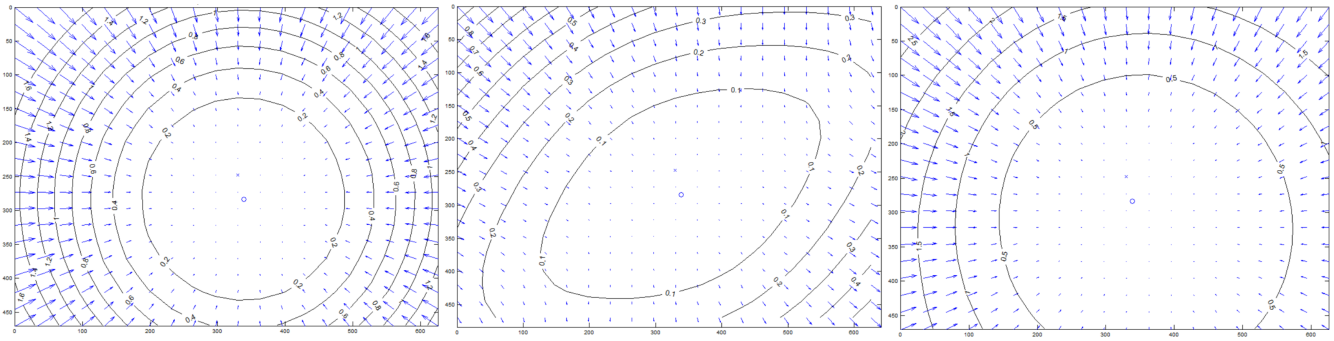


Slika 3.7: Umeritveno telo poslikamo na različnih oddaljenostih od kamere in pod različnimi koti glede na kamero. Priporočljivo je vsaj 20 različnih leg.

```
Calibration results after optimization (with uncertainties):

Focal Length:          fc = [ 661.67001   662.82858 ] ± [ 1.17913   1.26567 ]
Principal point:       cc = [ 306.09590   240.78987 ] ± [ 2.38443   2.17481 ]
Skew:             alpha_c = [ 0.00000 ] ± [ 0.00000  ]   => angle of pixel axes
Distortion:            kc = [ -0.26425   0.22645   0.00020   0.00023  0.00000 ]
Pixel error:          err = [ 0.45330   0.38916 ]
```

Slika 3.8: Izpis notranjih parametrov po umeritvi.
fc = goriščna razdalja $(b_\mathrm{v}, b_\mathrm{u})$, cc = center slike $(c_\mathrm{v}, c_\mathrm{u})$,
kc = parametri distorzije $(k_1, k_2, p_1, p_2, 0)$

kazanem primeru gre za slabo mehansko vgradnjo, saj je slikovno zaznavalo zamaknjeno glede optično os, distorziji pa sta približno enaki.

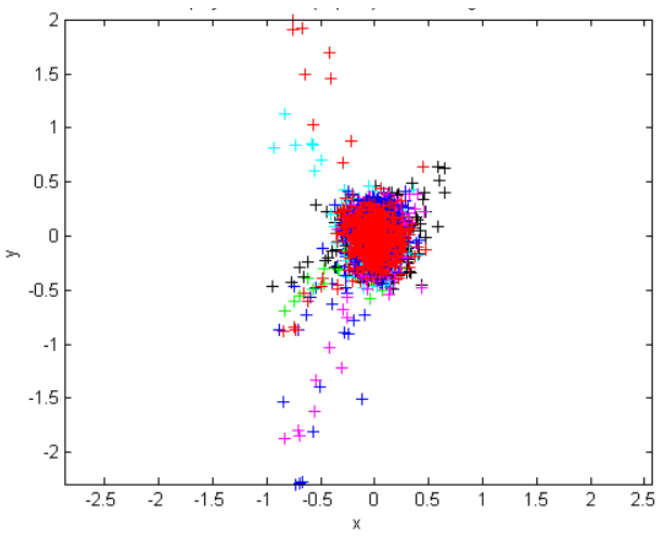


Slika 3.9: Radialna distorzija a), tangencialna b) in radialna in tangencialna distorzija skupaj c). Enote so v slikovnih elementih.

3.5.1 Negotovost umeritve

Zelo pomembna je ocena **negotovosti umeritve**, kot jo prikazuje slika 3.10. Slika prikazuje vse točke v učni množici. Graf je dobljen tako, da se koordinate točk v 3D-prostoru preslika na slikovno zaznavalo s parametri kamere, določenimi pri umeritvi. Graf prikazuje razliko med tako izračunano koordinato (v, u) in "teoretično" koordinato iz podatkovnega seta (točke, ki so bile uporabljene v učni množici pri določanju notranjih parametrov). V idealnem primeru bi ta razlika morala biti nič, kar pomeni, da bi bile vse točke skoncentrirane v središču grafa okrog točke nič. Če je podatkovni set dober in določitev parametrov kamere uspešna, potem je razpršenost odstopkov majhna, največ nekaj slikovnih elementov. Dvakratni standardni odklon okrog točke "0" uporabimo kot številčno oceno negotovosti umeritve.

Na grafu najboj izstopajo točke, ki so daleč stran od "0". Običajno gre za točke, kjer je bilo umeritveno telo močno postrani glede na kamero. Posledično vogali med kvadrati niso dobro vidni, zato je nastala večja napaka v določanju vogalov. Ravno za takšne primere mora biti podatkovni set zadosti velik, več tisoč točk, da preprečimo njihov vpliv na celotno umeritev. Če bi se pojavili kakšni pravilni vzorci, potem je umeritev neuspešna in moramo ponoviti optimizacijo ali pa izboljšati podatkovni set.



Slika 3.10: Negotovost umeritve (v slikovnih elementih).

3.5.2 Umeritev v OpenCV

Postopek umeritve z uporabo funkcij OpenCV v Pythonu temelji na modelu umeritve [7], ki je povzet v tem učbeniku in je tudi enak modelu, uporabljenem v [6]. Najprej pripravimo dva podatkovna seta `objpoints`, ki predstavljata koordinate vogalov (x, y, z) v 3D-prostoru, in `imgpoints`, ki predstavlja koordinate istih vogalov (v, u) na slikah šahovnice. Pomembna funkcija, ki smo jo pri tem uporabili, je `cv2.findChessboardCorners`, ki na slikah prepozna koordinate vogalov in `cv2.cornerSubPix`, ki natančno prepozna vogale. Umeritev izvedemo s funkcijo `cv2.calibrateCamera`, ki kot vhodne podatke vzame oba podatkovna seta.

```
mport cv2
import numpy as np
import glob

#PRIPRAVA PODATKOVNIH SETOV
# Najprej določimo, koliko notranjih vogalov po vrsticah in stolpcih ima šahovnica
```

```
sah_cr = (4, 5)
#Naredimo podatkovni set s točkami vogalov v 3D-prostoru
#Naredimo matriko ničel velikosti [število vogalov, 3]
objp = np.zeros((sah_cr[0] * sah_cr[1], 3), np.float32)
#Vanjo zapišemo oštevilčene koordinate vogalov
objp[:, :2] = np.mgrid[0:sah_cr[0], 0:sah_cr[1]].T.reshape(-1, 2)
#objp za 4 x 5 notranjih vogalov izgleda takole (z = 0):
'''
[[0. 0. 0.]
[1. 0. 0.]
[2. 0. 0.]
[3. 0. 0.]
[0. 1. 0.]
[1. 1. 0.]
[2. 1. 0.]
[3. 1. 0.]
[0. 2. 0.]
[1. 2. 0.]
[2. 2. 0.]
[3. 2. 0.]
[0. 3. 0.]
[1. 3. 0.]
[2. 3. 0.]
[3. 3. 0.]
[0. 4. 0.]
[1. 4. 0.]
[2. 4. 0.]
[3. 4. 0.]]
'''
velikost_kvadrata=10 #(mm) Določimo velikost kvadrata
objp *= velikost_kvadrata #(x-, y-, z-) koordinate v mm

#Podatkovni set:  vogali na vseh slikah
objpoints = []  # koordinate vogalov (x, y, z) v 3D-prostoru
imgpoints = []  # koordinate vogalov (v, u) na slikah

#Pridobimo seznam imen vseh slik na trenutnem direktoriju
imena_vseh_slik = glob.glob('calibration*.jpg')
#Termination criteria
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

for ime in imena_vseh_slik: #iz seznama izbiramo slike
        img = cv2.imread(fname) #sliko preberemo
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) #pretvorimo v sivinsko
        #Z naslednjo funkcijo poiščemo vogale na sliki

        ret, vogali_uv = cv2.findChessboardCorners(gray, sah_cr, None)

        #če smo uspešni, shranimo vse najdene vogale v imgpoints in podobno
        #koordinate vogalov (x,y,z) v 3D-prostoru v objpoints
        if ret:
                objpoints.append(objp)
                #malo popravimo natančnost vogalov
                vogali_uv2= cv2.cornerSubPix(gray,vogali_uv, (11,11), (-1,-1),
                    criteria)
                imgpoints.append(vogali_uv2)
    ...
    #Za potrebe kontrole pravilnosti delovanja,
    #bi na tem mestu lahko izrisovali najdene vogale na slikah

#UMERITEV: vhod - oba podatkovna seta, dimenzije slike, ..
```

```
ret,K,kc,koti,T=cv2.calibrateCamera(objpoints,imgpoints,gray.shape[::-1],None,
    None)
```

Rezultat umeritve je naslednji:

– ret: status

– K: notranji parametri (matrika kamere), potrebni za normiranje (fc, cc; fc = b):

$$K = \begin{bmatrix} f_u & 0 & c_u \\ 0 & f_v & c_v \\ 0 & 0 & 1 \end{bmatrix} \; npr. = \begin{bmatrix} 1037{,}5 & 0 & 623{,}8 \\ 0 & 1037{,}2 & 478{,}4 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.10}$$

– k_c: notranji parametri distorzije:

$$k_c = \begin{bmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{bmatrix} \tag{3.11}$$

– koti: zunanji parameter: vektor rotacijskih kotov, tj. kako so šahovnice zasukane glede na kamero ("omc_ext"); iz teh kotov dobimo rotacijske matrike s pomočjo Rodriguesove rotacijske enačbe (v radianih)

– T: vektor translacije od KKS do GKS za vse šahovnice ("Tc_ext") (mm).

3.5.3 Zunanji parametri

Po umeritvi kamere lahko določimo **zunanje parametre** kamere za poljubno orientacijo kamere glede na umeritveno telo. Postopek je naslednji: umeritveno telo pozicioniramo v želeno lego (npr. na delovno mizo robota), kamero pa pritrdimo na nosilno konstrukcijo kontrolne naprave ali kako drugače po želji (če je pritrjena na robotsko roko). S kamero slikamo umeritveno telo in sliko naložimo v program za umerjanje. V okviru izračuna v Matlabu pokažemo določeno polje kvadratov na šahovnici, podobno kot pri umeritvi, in vnesemo velikost kvadratov šahovnice (npr. 10 mm). Program nato izračuna zunanje parametre kamere, tj. lego KKS glede šahovnico. Primer izpisa rezultatov je naslednji

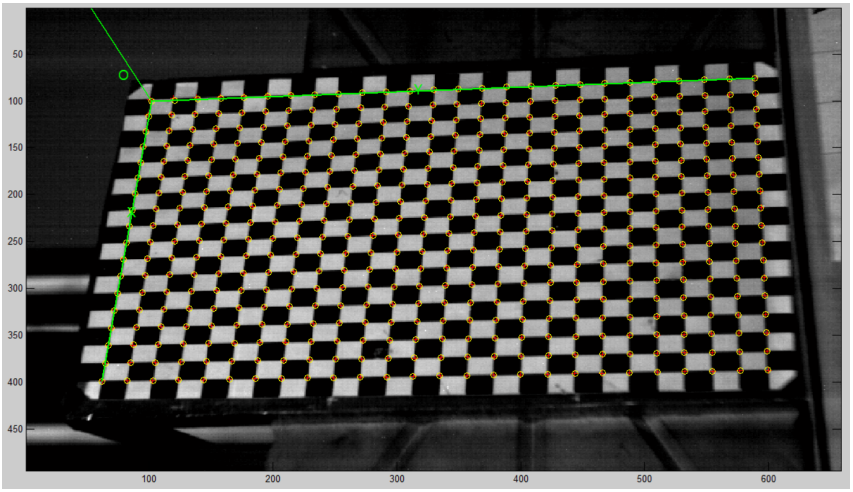
```
Translation vector: Tc_ext = [-123,026, -95,769, 860,120]
Rotation vector:   omc_ext = [ 1,9978,  2,0091, -0,6610]
Rotation matrix:    Rc_ext = [-0,0428,  0,9879, -0,1490;
                               0,8833, -0,0322, -0,4675;
                              -0,4667, -0,1516, -0,8713 ]
```

V tem izpisu *Tc_ext* pomeni premik kordinatnih izhodišč (T), tj. lego GKS v KKS. GKS je v tem primeru koordinatni sistem, pripet v zgornji levi vogal šahovnice, kot prikazuje slika 3.11. Koti zasuka in rotacijska matrika so podani z *omc_ext* in *Rc_ext* (R). Rotacijska matrika se iz kotov zasuka izračuna na osnovi Rodriguesove formule (na voljo funkcija v knjižnici za umerjanje).

Za določanje zunanjih parametrov v OpenCV je postopek podoben kot pri umeritvi. Izdelamo dva podatkovna seta, koordinate vogalov v 3D-prostorih in pripadajoče koordinate na sliki, potem pa s funkcijo `cv2.solvePnP` izračunamo lego kamere glede na šahovnico, kot to prikazuje koda v nadaljevanju.

```
#Pripravimo podatkovni set vogalov v 3D-prostoru
objp = np.zeros((sah_cr[0] * sah_cr[1], 3), np.float32)
objp[:, :2] = np.mgrid[0:sah_cr[0], 0:sah_cr[1]].T.reshape(-1, 2)
objp *= velikost_kvadrata
#Naložimo sliko
image = cv2.imread('sahovnica_na_tekocem_traku.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# Poiščemo vogale
ret, vogali_uv = cv2.findChessboardCorners(gray, sah_cr, None)
if ret:
        #Natančna določitev vogalov
        vogali_uv2 = cv2.cornerSubPix(gray, vogali_uv, (11, 11), (-1, -1),
        criteria=(cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)
            )

#Določitev poze tj.  lege kamere glede na šahovnico
ret, koti, T = cv2.solvePnP(objp, vogali_uv2, K, D)

#Rotacijska matrika iz kotov koti
R, _ = cv2.Rodrigues(koti)

print("Rotacijski koti (koti):\n", koti)
print("Translacija iz KKS v GKS (T):\n", T)
print("Rotacijska matrika:\n", R)
```

Zunanje parametre lažje razumemo s pomočjo skice na sliki 3.12.



Slika 3.11: GKS, pripet v zgornji levi vogal šahovnice

Poljubno točko v GKS $\vec{r}_{\mathrm{g}} = (x_{\mathrm{g}}, y_{\mathrm{g}}, z_{\mathrm{g}})$ transformiramo v KKS $\vec{r}_{\mathrm{c}} = (x_{\mathrm{c}}, y_{\mathrm{c}}, z_{\mathrm{c}})$ s pomočjo enačbe (3.12)

$$\vec{r}_{\mathrm{c}} = R \cdot \vec{r}_{\mathrm{g}} + T \tag{3.12}$$

Izhodišče GKS ima v KKS-koordinate, kot jih določa premik T. To enostavno preverimo s pomočjo enačbe (3.12), če vanjo vstavimo $\vec{r}_g = (0,\ 0,\ 0)$. Predpostavimo, da želimo izračunati koordinate točke P, kjer optična os kamere prebada umeritveno telo (slika 3.12). Točka P bo v KKS imela koordinate $\vec{r}_{\mathrm{c}} = (0, 0, z_{\mathrm{c}})$, v GKS pa $\vec{r}_{\mathrm{g}} = (x_{\mathrm{g}}, y_{\mathrm{g}}, 0)$. Če te vrednosti vstavimo v enačbo (3.12), dobimo:

$$\begin{pmatrix} 0 \\ 0 \\ z_\mathrm{c} \end{pmatrix} = R \begin{pmatrix} x_\mathrm{g} \\ y_\mathrm{g} \\ 0 \end{pmatrix} + T \tag{3.13}$$

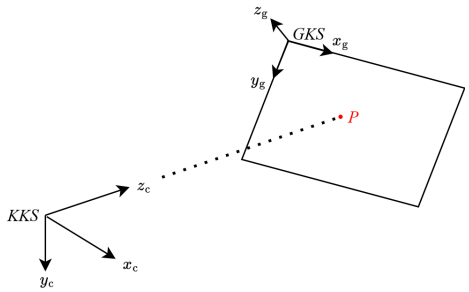
Enačba (3.13) v resnici predstavlja tri enačbe (3.14) za tri neznanke z_c, x_g in y_g. Členi v tej enačbi r_{11}, r_{12} .. r_{33} so elementi rotacijske matrike R, ki je dimenzije 3 x 3. Dovolj je, da izračunamo rešitev sistema za z_c (enačba (3.15)), preostale koordinate v KKS pa izračunamo s pomočjo enačb (3.3) in (3.4). Če želimo koordinate izraziti v GKS, potem uporabimo enačbe (3.16) ob upoštevanju $z_\mathrm{g} = 0$.

$$\begin{aligned} 0 &= r_{11}x_\mathrm{g} + r_{12}y_\mathrm{g} + t_\mathrm{x} \\ 0 &= r_{21}x_\mathrm{g} + r_{22}y_\mathrm{g} + t_\mathrm{y} \\ z_\mathrm{c} &= r_{31}x_\mathrm{g} + r_{32}y_\mathrm{g} + t_\mathrm{z} \end{aligned} \tag{3.14}$$

$$z_\mathrm{c} = \frac{(r_{21}r_{32} - r_{22}r_{31})\,t_\mathrm{x} + (r_{12}r_{31} - r_{11}r_{32})\,t_\mathrm{y} + (r_{11}r_{22} - r_{12}r_{21})\,t_\mathrm{z}}{r_{11}r_{22} - r_{12}r_{21}} \tag{3.15}$$

$$\begin{aligned} x_\mathrm{g} &= -\frac{r_{22}t_\mathrm{x} - r_{12}t_\mathrm{y}}{r_{11}r_{22} - r_{12}r_{21}} \\ y_\mathrm{g} &= \frac{r_{21}t_\mathrm{x} - r_{11}t_\mathrm{y}}{r_{11}r_{22} - r_{12}r_{21}} \end{aligned} \tag{3.16}$$

Opisani postopek je zelo pomemben, saj je osnova laserske triangulacije in stereovida. Pri laserski triangulaciji šahovnico zamenjamo s svetlobno ravnino, GKS pa predstavlja koordinatni sistem svetlobne ravnine, tipično pozicioniran v izhodno točko laserskega projektorja. Pri stereovidu GKS šahovnice zamenja KKS druge kamere.



Slika 3.12: Lega KKS glede na GKS, pritrjen v zgornji levi vogal šahovnice, kot prikazuje slika 3.11

3.6 Uporaba zunanjih parametrov pri računanju 3D-koordinat

Predpostavimo, da obravnavamo lokalizacijo objektov na delovnem mestu. S kamero slikamo objekt, ki ga nato želimo pobrati z robotom. Na zajeti sliki poiščemo težišče objekta. Koordinate težišča so izražene v SKS (v, u). Te normiramo in razpačimo distorzijo. Normirane koordinate $(x_\mathrm{n}, y_\mathrm{n})$ preslikamo v 3D-prostor z enačbami (3.3) in (3.4). Ključno vprašanje nastopi, kako določiti z_c. Izračun izvedemo po enakem principu, kot je prikazan z enačbami (3.14), s tem da x_c in y_c nista več nič. Postopek poteka podobno, kot prej zapišemo sistem enačb, a s tem dodatkom, da v njih upoštevamo, da je $x_\mathrm{c} = x_\mathrm{n}z_\mathrm{c}$ in $y_\mathrm{c} = y_\mathrm{n}z_\mathrm{c}$,

$$\begin{aligned} x_{\mathrm{n}} z_{\mathrm{c}} &= r_{11} x_{\mathrm{g}} + r_{12} y_{\mathrm{g}} + t_{\mathrm{x}} \\ y_{\mathrm{n}} z_{\mathrm{c}} &= r_{21} x_{\mathrm{g}} + r_{22} y_{\mathrm{g}} + t_{\mathrm{y}} \\ z_{\mathrm{c}} &= r_{31} x_{\mathrm{g}} + r_{32} y_{\mathrm{g}} + t_{z}, \end{aligned} \tag{3.17}$$

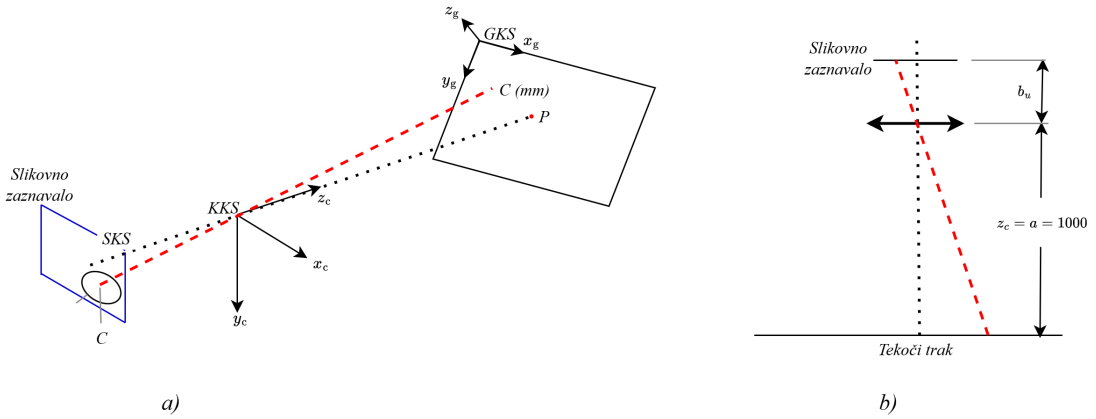
iz katerega izračunamo z_{c} (en.3.18); preostale koordinate v KKS izračunamo s pomočjo enačb (3.3) in (3.4).Če želimo koordinate izraziti v GKS, potem x_{g} in y_{g} izračunamo po enačbah (3.19), medtem ko je $z_{\mathrm{g}} = 0$. V navedenih enačbah so vsi r in t parametri konstantni, razen $(x_{\mathrm{n}}, y_{\mathrm{n}})$. Če moramo transformirati veliko število točk, za večjo računsko učinkovitost dele enačb s konstantnimi členi izračunamo v naprej, tj. izven `for` zank, s katerimi se premikamo po normiranih koordinatah.

$$z_{\mathrm{c}} = \frac{(r_{11}r_{22} - r_{12}r_{21})t_{\mathrm{z}} + (r_{12}r_{31} - r_{11}r_{32})t_{\mathrm{y}} + (r_{21}r_{32} - r_{22}r_{31})t_{\mathrm{x}}}{r_{11}r_{22} - r_{12}r_{21} + (r_{12}r_{31} - r_{11}r_{32})y_{\mathrm{n}} + (r_{21}r_{32} - r_{22}r_{31})x_{\mathrm{n}}} \tag{3.18}$$

$$\begin{aligned} x_{\mathrm{g}} &= \frac{(r_{22}t_{\mathrm{z}} - r_{32}t_{\mathrm{y}})x_{\mathrm{n}} + (r_{32}t_{\mathrm{x}} - r_{12}t_{\mathrm{z}})y_{\mathrm{n}} + r_{12}t_{\mathrm{y}} - r_{22}t_{\mathrm{x}}}{(r_{21}r_{32} - r_{22}r_{31})x_{\mathrm{n}} + (r_{12}r_{31} - r_{11}r_{32})y_{\mathrm{n}} + r_{11}r_{22} - r_{12}r_{21}} \\ y_{\mathrm{g}} &= \frac{(r_{31}t_{\mathrm{y}} - r_{21}t_{\mathrm{z}})x_{\mathrm{n}} + (r_{11}t_{\mathrm{z}} - r_{31}t_{\mathrm{x}})y_{\mathrm{n}} - r_{11}t_{\mathrm{y}} + r_{21}t_{\mathrm{x}}}{(r_{21}r_{32} - r_{22}r_{31})x_{\mathrm{n}} + (r_{12}r_{31} - r_{11}r_{32})y_{\mathrm{n}} + r_{11}r_{22} - r_{12}r_{21}} \end{aligned} \tag{3.19}$$

Primer izračuna koordinat v 3D-prostoru

Obravnavamo problem robotskega pobiranja izdelkov s tekočega traku (lokalizacija izdelkov). S kamero izmerimo lego objekta na tekočem traku, izmerjene koordinate pa pošljemo na krmilnik robota za pravilno pobiranje. Kamera je pritrjena na nosilno konstrukcijo ob tekočem traku. Umeritveno telo s sliko šahovnice pri umeritvi položimo na tekoči trak. Koordinatni sistem šahovnice (GKS) je skupen kameri in robotu. Pri umerjanju industrijskega robota določimo uporabniški koordinatni sistem (UCS), tako da sovpada z GKS šahovnice (uporabimo npr. metodo treh točk, pokažemo izhodišče ter x- in y-smer). Kamera je pozicionirana in umerjena glede na šahovnico podobno, kot je prikazano na sliki 3.13 (a).



Slika 3.13: Izračun koordinat centra objekta v KKS in v GKS (a). Kamera pravokotno na tekoči trak (b).

Po umeritvi kamere za izbrane lege umeritvenega telesa določimo zunanje parametre. Pomembno je, da ne premaknemo umeritvenega telesa, vse dokler ne umerimo robota! Pri umeritvi kamere smo določili naslednje umeritvene parametre:

```
#Zunanji parametri:
omc_ext=[1,99781 2,00919 -0,6610] rad
R=[-0,042843   0,987908  -0,149003]
  [ 0,883375  -0,032211  -0,467559]
  [-0,466705  -0,151657  -0,871313] rad
T=[-123,027 -95,770 860,121] mm

#Notranji parametri:
kc = [-0,17533 1,71392 0,00250 0,00150 0,00000]
cc = [337,80 282,96]
fc = [1664,8 1659,1]
```

Za izračun koordinat v KKS bomo uporabili enačbo (3.18). Konstante, ki v njej nastopajo, izhajajo iz R in T:

```
r11 = -0,042843
r12 =  0,98791
r21 =  0,88337
r22 = -0,032211
r31 = -0,46670
r32 = -0,15166
tx = -123,03
ty = -95,770
tz =  860,12
```

Če jih vstavimo v enačbo (3.18), se ta poenostavi v

$$z_\mathrm{c} = -686{,}33/(-0{,}87131 - 0{,}46756 \cdot y_\mathrm{n} - 0{,}14900 \cdot x_\mathrm{n}) \tag{3.20}$$

Za začetek (za kontrolo) izračunajmo koordinate točke P, kjer optična os kamere prebada umeritveno telo (slika 3.12). Točka P bo v KKS imela koordinate $\vec{r}_\mathrm{c} = (0, 0, z_\mathrm{c})$, v GKS pa $\vec{r}_\mathrm{g} = (x_\mathrm{g}, y_\mathrm{g}, 0)$. Slika točke P se bo nahajala v centru slike, kjer optična os prebada slikovno zaznavalo, v točki $(c_\mathrm{u}, c_\mathrm{v})$, zato bodo x_n in y_n enake 0. Distorzije v centru slike ni, zato razpačitev ni potrebna. Če vstavimo normirane koordinate v enačbo (3.20), dobimo $z_\mathrm{c} = -686{,}33/(-0{,}87131) = 787{,}69$ mm. Koordinate točke P v KKS bodo (0, 0, 787,69) mm. Izračunajmo še koordinate točke P v GKS. Za izračun uporabimo enačbe (3.19). Po vstavitvi r- in t-konstant in $x_\mathrm{n} = y_\mathrm{n} = 0$ dobimo koordinate točke P v GKS (113,13, 129,44, 0) mm.

Zdaj predpostavimo, da smo na sliki poiskali objekt, ki ga želimo pobrati (obdelava slike ni prikazana). Določili smo koordinate njegovega centra $C_\mathrm{o,d}$ v SKS $u = 100$ in $v = 150$ (indeks d pomeni, da koordinate vsebujejo distorzijo). Najprej izvedemo normiranje po enačbah (3.1) in (3.2):

$$\begin{aligned} x_\mathrm{n} &= \frac{u - c_\mathrm{u}}{f_\mathrm{u}} = \frac{100 - 337{,}80}{1664{,}8} = -0{,}1428 \\ y_\mathrm{n} &= \frac{v - c_\mathrm{v}}{f_\mathrm{v}} = \frac{150 - 282{,}96}{1659{,}1} = -0{,}0801 \end{aligned} \tag{3.21}$$

Ker nismo v centru slike, sledi razpačitev distorzije po algoritmu na diagramu (3.5). Normirane koordinate centra objekta $C_\mathrm{o,d}$=(–0,14284, –0.08014) razpačimo in dobimo C_o = (–0,14318, –0,08028)[3].

Zdaj lahko izračunamo 3D-koordinate centra objekta. Izračun najprej naredimo v KKS, podobno kot smo to naredili za točko P z enačbo (3.20), in dobimo $z_\mathrm{c} = -686{,}33/(-0{,}87131 - 0{,}46756 \cdot -0{,}08028 - 0{,}14900 \cdot -0{,}14318) = 844{,}78$ mm. Zdaj x_c in y_c nista več 0, ampak ju

[3]Pri izračunu razpačitve si lahko pomagamo s funkcijo `comp_distortion_oulu`, ki se nahaja v knjižnici Camera Calibration Toolbox za Matlab (deluje tudi v Octave). V OpenCV imamo na voljo funkcijo `cv2.undistort`

moramo izračunati z enačbami (3.3) in (3.4):

$$\begin{aligned} x_c &= x_n \cdot z_c = -0{,}1431 \cdot 844{,}78mm = -120{,}9 \text{ mm} \\ y_c &= y_n \cdot z_c = -0{,}0802 \cdot 844{,}78mm = -67{,}8 \text{ mm} \end{aligned} \tag{3.22}$$

Sedaj izračunajmo še koordinate centra v GKS. Za izračun uporabimo enačbe (3.19) in dobimo $x_p = 31{,}5$ mm in $y_p = 3{,}2$ mm, $z_p = 0$. Če povzamemo: koordinate centra objekta, izražene v KKS, so (–120,9, –67,8, 844,78) mm, v GKS pa (31,5, 3,2, 0) mm. Na krmilnik robota pošljemo koordinate, izražene v GKS, ki je skupen z UCS-robota. Še primer kode za izračun koordinat v okolju Matlab/Octave:

```
pt=[100; 150]
cd=(pt-cc)/fc %normiranje
cud = comp_distortion_oulu(cd,kc)
xn=cud(1)
yn=cud(2)
R=rodrigues(omc_ext)
r11=R(1,1)
r12=R(1,2)
r21=R(2,1)
r22=R(2,2)
r31=R(3,1)
r32=R(3,2)
tx=T(1)
ty=T(2)
tz=T(3)
#Konstante
A=r11*r22*tz-r11*r32*ty-r12*r21*tz+r12*r31*ty+r21*r32*tx-r22*r31*tx
B=r11*r22-r12*r21
Cyn=r12*r31-r11*r32   %*yn
Dxn=r21*r32-r22*r31   %*xn
#KKS
zc=A/(B+Cyn*yn+Dxn*xn)
xc=zc*xn
yc=zc*yn
#GKS
xp=(r12*ty - r12*tz*yn - r22*tx + r22*tz*xn + r32*tx*yn - r32*ty*xn)/...
(r11*r22 - r11*r32*yn - r12*r21 + r12*r31*yn + r21*r32*xn - r22*r31*xn)
yp=(-r11*ty + r11*tz*yn + r21*tx - r21*tz*xn - r31*tx*yn + r31*ty*xn)/...
(r11*r22 - r11*r32*yn - r12*r21 + r12*r31*yn + r21*r32*xn - r22*r31*xn)}
```

Primer izračuna koordinat objekta na tekočem traku

Kamera je pritrjena na nosilno konstrukcijo pravokotno nad tekočim trakom na oddaljenosti 1 m (točka optične preslikave – trak). S kamero zajamemo sliko, jo obdelamo in pri tem določimo center objekta na sliki $u = 250$, $v = 400$ slikovnih elementov. Izračunajte koordinate centra objekta v milimetrih na traku v kamerinem koordinatnem sistemu (distorzijo zanemarimo). Podatki, dobljeni z umeritvijo kamere, so: $f_u = 1664{,}8$, $f_v = 1659{,}1$, $c_u = 337{,}8$, $c_v = 282{,}96$. Skica sistema je podobna kot na sliki 3.13 (b).

Rešitev: Zelo pomemben je podatek, da je kamera postavljena pravokotno na tekoči trak. V tem primeru ni treba upoštevati perspektive in izračuna po enačbah (3.18) in (3.19), ampak lahko izračunamo koordinate na osnovi optične preslikave. Najprej izračunamo normirane koordinate:

$$\begin{aligned} x_n &= \frac{u - c_u}{f_u} = \frac{250 - 337{,}8}{1664{,}8} = -0{,}0527 \\ y_n &= \frac{v - c_v}{f_v} = \frac{400 - 282{,}96}{1659{,}1} = 0{,}0705. \end{aligned} \tag{3.23}$$

Ker distorzijo zanemarimo, koordinate v KKS enostavno izračunamo z množenjem normiranih koordinat z $z_c = 1000$ mm:

$$\begin{aligned} x_c &= x_n \cdot z_c = -0{,}0527 \cdot 1000 \text{ mm} = -52{,}74 \text{ mm} \\ y_c &= y_n \cdot z_c = 0{,}0705 \cdot 1000 \text{ mm} = 70{,}54 \text{ mm}. \end{aligned} \tag{3.24}$$

Recimo, da bi bila podana tudi velikost slikovnega elementa, npr. 0,005 mm. Naredimo lahko oceno velikosti slikovnega elementa na tekočem traku. V ta namen moramo izračunati povečavo. Razdaljo $z_c = a = 1000$ mm že poznamo. Razdaljo b dobimo tako, da f (dobimo pri umeritvi) pomnožimo z velikostjo slikovnega elementa $b = 0{,}005 \cdot f_u = 0{,}005 \cdot 1664{,}8 = 8{,}32$ mm. Povečava $m = b/a = 8{,}32/1000 = 0{,}0083$. Velikost slikovnega elementa na traku je $0{,}005/m = 0{,}005/0{,}00832 = 0{,}6$ mm. Za lokalizacijo bo to zadosti, po pravilih vzorčenja bi lahko merili detalje, ki jih popišemo vsaj z 10 slikovnimi elementi, torej večje od 6 mm.

3.7 Naloge

1. Izračunajte vidno polje kamere (FOV). Znani so naslednji podatki: matrika kamere K, število in velikost slikovnih elementov. Postopek: najprej izračunajte razdalji b_u in b_v ter dimenzije slikovnega zaznavala S_u in S_v (v mm). Za izračun FOV uporabite naslednji enačbi:

$$\mathrm{FOV_x} = 2 \cdot \arctan\left(\frac{S_u}{2 \cdot b_\mathrm{u}}\right), \quad \mathrm{FOV_y} = 2 \cdot \arctan\left(\frac{S_v}{2 \cdot b_\mathrm{v}}\right).$$

2. Pretvorite 3D-koordinate (v GKS) v 2D-slikovne koordinate. Glede na 3D-točko v globalnih koordinatah (x, y, z), matriko kamere K in zunanje parametre (rotacija R in translacija T) izračunajte 2D-slikovne koordinate (v, u) na slikovni ravnini. Uporabite naslednjo enačbo:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \cdot \left(R \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} + T \right).$$

3. Izračunajte popačene in nepopačene koordinate slikovnih elementov z uporabo parametrov distorzije. Za niz 2D-točk na nepopačeni sliki in parametre distorzije k_1, k_2, p_1, p_2 izračunajte njihove popačene koordinate. Napišite in uporabite algoritem za razpačitev distorzije (slika 3.5).

4. Izračunajte skupni odstopek reprojekcije po umeritvi kamere. Vzemite 3D-točke objekta iz učne množice za umeritev in s pomočjo parametrov umeritve (K, k_c, R, T) izračunajte njihove slikovne koordinate (v_i', u_i') (postopek je podoben kot v nalogi 2). Nato 3D-točkam vzemite pripadajoče slikovne koordinate iz učne množice (v_i, u_i) in izračunajte skupni odstopek reprojekcije z naslednjo enačbo:

$$\text{Skupni odstopek} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} ((u_i - u_i')^2 + (v_i - v_i')^2)}.$$

   Diskutirajte vrednost skupnega odstopka pri različnih natančnostih določitve notranjih parametrov.

5. Ocenite položaj kamere glede na šahovnico. Glede na niz 3D-točk na šahovnici in njihove ustrezne 2D-slikovne koordinate uporabite funkcijo `cv2.solvePnP` za izračun R in T.

Poglavje 4

Obdelava slike

Tematike s področja obdelave slike se nenehno razvijajo. V današnjem času (2025) je aktualna uporaba globokih nevronskih mrež za vsebinsko razumevanje slike – podobno, kot to počne človek. Aktualne teme s tega področja so obdelava slik za potrebe avtonomne vožnje, razumevanje aktivnosti in zagotavljanje varnosti pri interakciji človeka in robota, kontrola kakovosti izdelkov itd. Pot do obvladovanja in razumevanja teh metod je dolga in se začne s prvimi koraki, med katere spada razumevanje zapisa slike v spominu računalnika, razumevanje premikanja po sliki in izvajanja računskih operacij na slikovnih elementih, razumevanje osnovnih konceptov iskanja objektov in drugih želenih informacij na sliki. V veliko pomoč pri teh temeljnih korakih nam bo dobro dokumentirana odprtokodna knjižnica OpenCV. Z njeno pomočjo bomo v nadaljevanju razlagali osnovne korake v obdelavi slike.

4.1 Zapis slike v spominu računalnika

Slikovno zaznavalo zajame svetlobni vzorec, ki ga ustvari objektiv. Pri zgradbi slikovnega zaznavala smo pojasnili, da to sestoji iz velikega števila matrično organiziranih slikovnih elementov (fotocelic) ter da se v vsakem slikovnem elementu izvaja pretvorba svetlobne intenzitete v pripadajoči električni naboj in pozneje v številčno vrednost. Svetlost posameznega slikovnega elementa je tipično podana z 8-bitno vrednostjo, ki zavzema vrednosti med 0 in 255. 0 ustreza povsem črni barvi, 255 beli, vmes so odtenki sivine. Za razumevanje si poglejmo primer svetle črte na črnem ozadju, kot jo prikazuje slika 4.1 a). Na sliki b) je prikazan številčni zapis iste slike. Vidimo, da gre za matrično (tabelarično) organizacijo slikovnih elementov in da so vrednosti svetle črte blizu 255, črno ozadje pa je blizu 0. Na tej točki moramo komentirati dva detajla.

Prvi se nanaša na organizacijo slikovnih elementov. Matrike vedno povezujemo z matričnimi računskimi operacijami. Pri obdelavi slike matričnih računskih operacij NE izvajamo (zelo redke so izjeme). Pojem matrika zgolj privzeto uporabljamo kot kontejner, v katerega so organizirano urejeno shranjeni podatki. Lahko bi uporabljali tudi pojem tabela. V spominu računalnika pa je slika dejansko zapisana kot zgolj ena dolga vrstica (sliko iz matrike razvijemo v eno dolgo vrstico). Razlog za to je v delovanju diskov in načinu zapisovanja po cilindrih.

Drugi komentar se nanaša na številčni zapis. Ta je odvisen od bitnosti digitalizacije in izbranega tipa podatkov za shranjevanje (*byte*, *int*). Računske operacije, predvsem na 8-bitnih in tudi v splošno na celih številih, so hitre, a zelo nerodne. Tipično se za računske operacije uporabljajo spremenljivke s plavajočo vejico, kot sta *float* in *double*. Kar je pri slednjih zelo pomembno, je to, da se za potrebe računske stabilnosti in tudi vizualizacije izogibamo velikim številom. Zato so vrednosti svetlosti običajno normirane v območje med 0 in 1. V praksi to pomeni, da vrednosti vseh slikovnih elementov delimo z 255, če je slika 8-bitna, s 1024, če je

10-bitna, itd. Povsem črna barva ima še zmeraj vrednost 0, povsem bela ima vrednost 1, odtenki sivine pa se nahajajo med 0 in 1.

a)

b)

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0
7	69	63	63	51	102	89	63	63	65	79	68	108	157
8	234	185	234	242	215	223	207	253	253	234	215	237	202
9	212	232	253	245	255	206	205	146	168	194	187	186	144
10	50	41	34	43	37	37	39	28	19	20	15	19	15
11	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	1	0	0	0	1	0	0
15	1	1	1	1	1	1	1	1	1	1	1	1	1
16	1	1	1	1	1	1	1	1	1	1	1	1	1
17	1	1	1	1	1	1	1	1	1	1	1	1	1
18	1	1	1	1	1	1	1	1	1	1	1	1	1
19	1	1	1	1	1	1	1	1	1	1	1	1	1
20	1	1	1	1	1	1	1	1	1	1	1	1	1
21	1	1	1	1	1	1	1	1	1	1	1	1	1

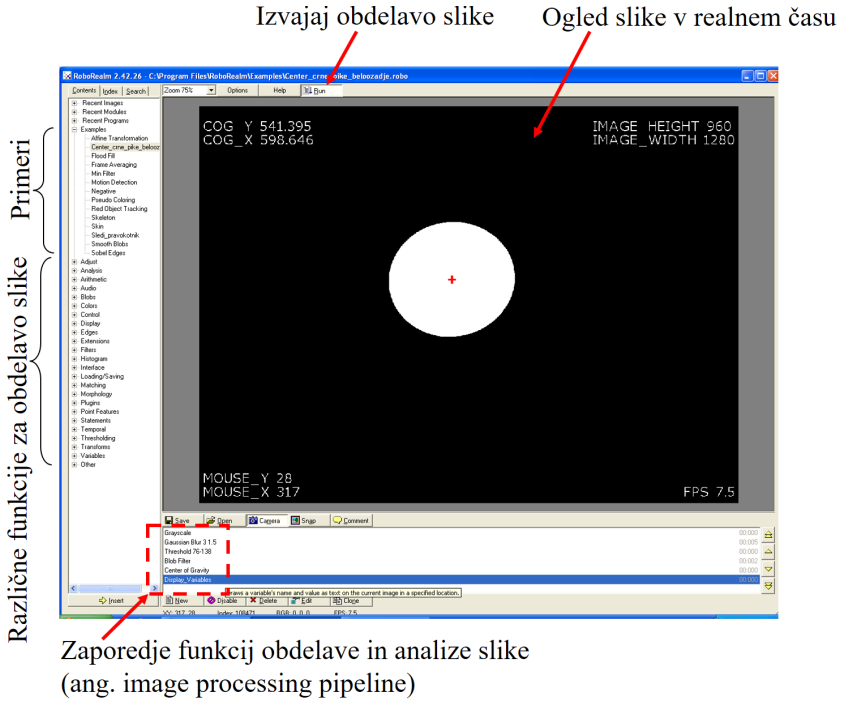
Slika 4.1: Svetlobni vzorec slike a), zapisan s števili b). Svetlost posameznega slikovnega elementa je tipično podana z 8-bitno vrednostjo, ki zavzema vrednosti med 0 in 255. 0 ustreza povsem črni barvi, 255 beli, vmes so odtenki sivine.

4.2 Programska oprema za obdelavo slike

Katero programsko opremo uporabiti, je prvo ključno vprašanje, ko se lotimo obdelave slike. Izbira je v veliki meri odvisna od programerskih izkušenj in veselja do programiranja, kompleksnosti problema, razpoložljivega časa in sredstev. Za sisteme strojnega vida obstajajo visokonivojski programski paketi, ki omogočajo povezavo s kamero, zajem slike, prikaz slike, obdelavo slike in komunikacijo z zunanjimi napravami (npr. PLC, krmilnik robota), vse brez potrebe po neposrednem programiranju v programskih jezikih, kot sta C++ ali Python, ter brez potrebe po poznavanju kompleksnosti podatkovnih tipov, klicev funkcij itd. Primer takšne programske opreme sta npr. *ZebraAuroraVision*[1] in *RoboRealm*[2]. Uporabniški vmesnik slednjega je prikazan na sliki 4.2. Tukaj imamo na voljo izbirni meni z velikim naborom funkcij obdelave slike (drevesna struktura na levi). Funkcije izbiramo zgolj s klikanjem po izbornem meniju in takoj vidimo rezultat obdelave. Naša naloga je, da ugotovimo pravilno kombinacijo in zaporedje funkcij obdelave (ang. image processing pipeline) ter da seveda nastavimo kamero in komunikacije z zunanjimi napravami. Tovrstni programi avtomatizirajo celoten proces razvoja sistemov strojnega vida in močno skrajšajo čas razvoja. Gre za komercialno programsko opremo, katere licenčnina se giblje od nekaj sto do več tisoč evrov.

Kot smo že omenili v poglavju 2.5 o krmiljenju kamer, obdelava slike danes prvenstveno temelji na odprtokodni knjižnici OpenCV. Vključimo jo lahko v različne programske jezike in uporabimo na različnih platformah, kot so 32- ali 64-bitni operacijski sistemi Windows in Linux,

[1] https://www.adaptive-vision.com
[2] https://www.roborealm.com/



Slika 4.2: Primer uporabniškega vmesnika programske opreme *RoboRealm* za visokonivojski razvoj aplikacij strojnega vida.

PC- ali ARM-platforme ter celo miniaturni vgradni sistemi, seveda pa mora biti temu ustrezno prevedena. Glede na programerske izkušnje študentov se bomo osredotočili prvenstveno na Python in C++. Pri razlagah v nadaljevanju predpostavimo, da že poznamo osnove navedenih jezikov in uporabniške vmesnike za programiranje (Spyder, Jupyter notebook, Visual Studio Code). V poglavju 2.5 o krmiljenju kamer smo pokazali, kako vključiti knjižnico OpenCV v navedena programska jezika. V nadaljevanju si poglejmo, kako ustvarimo sliko in kako se po njej pomikamo.

4.2.1 OpenCV v Pythonu

OpenCV v Pythonu izkorišča funkcionalnost knjižnice NumPy. Slike se shranjujejo v podatkovni tip `ndarray`. Iz tega razloga moramo (skoraj) vedno vključiti obe knjižnici. V nadaljevanju sledijo primeri dela s slikami v Pythonu:

```
import numpy as np
import cv2 as cv

# 1) Preberemo sliko iz diska
sl_test = cv.imread('testimg.png')
# slika je shranjena v spremenljivki z imenom sl_test
# 2) pridobimo dimenzije slike
dimenzije = sl_test.shape
print(dimenzije)
# ali
vrstic = sl_test.shape[0]
stolpcev = sl_test.shape[1]
barv = sl_test.shape[2]
...
# 3) Ustvarimo novo sliko
nova_slika = np.zeros((vrstic,stolpcev,3), np.uint8)
```

```
# vrednosti slikovnih elementov bodo 0, zato bo povsem črna
# kadar nočemo črne slike, ji lahko določimo barve
...
# 4) Primeri vpisa baru
# 4.1) Z naslednjim ukazom levi polovici slike dodelimo modro barvo
nova_slika[:,0:(stolpcev/2)] = (255,0,0)
# ali
...
# 4.2) Celotni sliki določimo barve.  Primer
#B, G, R = 0x3E, 0x88, 0xE5 oranžna barva ali
B, G, R = 62, 136, 229   # oranžna barva
nova_slika[:, :, 0] = B #modra barvna ravnina
nova_slika[:, :, 1] = G #zelena barvna ravnina
nova_slika[:, :, 2] = R #rdeča barvna ravnina
...
# 4.3) Točno določenemu slikovnemu elementu
# (npr.  na lokaciji (100,100)) določimo barvo
nova_slika[100, 100] = (255,255,255) # ali
nova_slika[100, 100, 1] = 255 #samo zelena
...
# 4.4) Pobarvamo ROI; zg.levi vogal (100,200), kvadrat velikosti 100
nova_slika[100:200, 200:300] = (255,255,255)
...
# 5) Pridobimo barvo slikovnega elementa na lokaciji (u = 200,v = 100)
barva=nova_slika[100, 200]
print(barva)
# samo zelena barva
zelenabarva=nova_slika[100, 200, 1]

# 6) Is slike izrezemo ROI
sl_test_roi=sl_test[100:200, 200:300]

# 7) Premikanje po sliki
# Pozor počasno delovanje!!
for i in range (0,stolpcev):
        for j in range (0,vrstic):
                nova_slika[j,i,0]=255-sl_test[j,i,0]
                nova_slika[j,i,1]=255-sl_test[j,i,1]
                nova_slika[j,i,2]=255-sl_test[j,i,2]

# Izris
cv.imshow("Izris nova_slike", nova_slika)
...
```

Če uporabljamo Spyder, potem lahko v ”Variable explorerju” preverimo tip podatkov, npr. type uint8, kar pomeni, da so podatki o svetlosti 8-bitni, ter dimenzije slike, npr. size (640, 480, 3). Slednje pomeni, da ima slika 640 vrstic, 480 stolpcev in 3 barve (B, G, R). Programsko lahko pridobimo dimenzije slike s klicem funkcije `dimenzije=sl_test.shape` (primer 2). V cv2 verziji lahko uporabimo skoraj vse razpoložljive podatkovne tipe iz knjižnice NumPy (odvisno, kaj rabimo).

Novo sliko ustvarimo s klicem funkcije `np.zeros`, ki ji podamo dimenzije (vrstic, stolpcev, barv) in tip podatkov (primer 3). Vrednosti slikovnih elementov v tako ustvarjeni novi sliki bodo (0, 0, 0), zato bo povsem črna. Barve lahko vpišemo naknadno, kot to prikazuje primer 4. Celotni sliki določimo vrednost določene barve, npr. `nova_slika[:,:,0] = 125`, kar pomeni, da bo modra barva na celi sliki imela svetlost 125. Posameznemu slikovnemu elementu vpišemo barvo tako, da podamo trojico (B, G, R), npr. slikovnemu elementu na lokaciji (100, 100) določimo povsem belo barvo `nova_slika[100,100] = (255, 255, 255)`. Za barvanje območja interesa uporabimo matrično sklicevanje (slicing), kot kaže primer 4.4 `nova_slika[100:200,200:300]`

= (255, 255, 255).

Barvo določenega slikovnega elementa pridobimo z imenovanjem lokacije, podobno kot pri matrikah, npr. barvo slikovnega elementa na lokaciji (100, 100) pridobimo z ukazom `barva=nova_slika[100,100]`. Samo določeno barvo, npr. zeleno, pa z ukazom `zelenabarva=nova_slika[100,100,1]`. **Pri obravnavanju slikovnih elementov ne smemo pozabiti, da je koordinatni sistem (V, U) v zgornjem levem vogalu. Torej za $u = 200$ (horizontalna smer, tj. stolpci) in $v = 50$ (vertikalna smer, tj. vrstice), pridobimo barvo s klicem funkcije `barva=nova_slika[vrstice,stolpci]` torej barva=nova_slika[50,200].** S slike izrežemo območje interesa, kot to prikazuje primer 6 `sl_test_roi=sl_test[100:200,200:300]`.

Primer 7 kaže, kako naredimo dve zanki, s katerima se pomikamo po sliki. V prikazanem primeru obrnemo vrednosti slikovnih elementov (belo postane črno in obratno) s slike `sl_test` in jih kopiramo v `nova_slika`. **Takšen način obdelave slik – z zankami je v Pythonu zelo počasen in ni priporočljiv. Če se le da, v Pythonu uporabimo funkcije OpenCV, katerih delovanje se izvaja v že prevedenih knjižnicah in je optimirano na hitrost.** Če razvijamo svoj algoritem z veliko zankami, potem to počnimo v C++.

4.2.2 OpenCV v C++

OpenCV slike shranjuje v razred C++ (ang. class) z imenom `Mat`, razvit posebej za delo z matrično organiziranimi podatki v slikah. Spremenljivka tipa Mat vsebuje razne podatke in funkcije, kot so število vrstic, stolpcev, barv, zapis barv, funkcije za ustvarjanje slik, kopiranje, preverjanje, ali slika vsebuje pravilne podatke itd. Do njih dostopamo po pravilih dostopa do objektov v razredih C++, tj. da v nadaljevanju imena slike zapišemo piko in ime spremenljivke ali funkcije (za `Mat sl_test`; zapišemo npr. `sl_test.rows;` ali `sl_test.empty();` itd). Koda v nadaljevanju prikazuje osnovno okostje konzolnega programa C++ in primere dela s slikami:

```
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <iostream>
using namespace cv;

int main() {

// 1) Preberemo sliko
Mat sl_test = imread("testnaslika.png", IMREAD_COLOR);
// slika se nahaja v spremenljivki z imenom sl_test,
// katere podatkovni tip je Mat

// preverimo, ali smo jo uspešno prebrali
if(sl_test.empty()) {
  std::cout << "Ne morem prebrati slike ... " << std::endl;
return 1;
}

// 2) Primer ustvarjanja novih slik
// Slika SA bo črnobela, 480 vrstic, 640 stolpcev
// 8 bitni slikovni elementi so postavljeni na svetlost 70
Mat SA(480, 640, CV_8UC1, Scalar(70));

// Slika SB bo barvna, 480 vrstic, 640 stolpcev
// 8-bitni slikovni elementi, svetlost B = 10, G = 100, R = 150
Mat SB(480, 640, CV_8UC3, Scalar(10, 100, 150));

// Slika SC bo črno-bela, 480 vrstic, 640 stolpcev
// 64-bitni (double) slikovni elementi
```

```
Mat SC(480, 640, CV_64F); //Brez začetne vrednosti

// Kloniranje slike
Mat SD = SC.clone();

// 3) Premikanje po sliki, dostop do slikovnih elementov
for(int i = 0; i < SC.rows; i++)
  for(int j = 0; j < SC.cols; j++)
    SC.at<double>(i,j)=double(SA.at<unsigned char>(i,j))/255.0;

// Takole preberemo posamezne barve na barvni sliki
Vec3b svetlost = SB.at<Vec3b>(i, j);
uchar B = svetlost.val[0];
uchar G = svetlost.val[1];
uchar R = svetlost.val[2];

// 4) Primer uporabe OpenCV funkcij
// Pretvorba barvne v črnobelo sliko
cvtColor(sl_test, SA, COLOR_BGR2GRAY);
// Invertiranje
bitwise_not(SA, SA);

// 5) Primer dela s kazalci
// Naredimo enostavno matriko (tj. sliko) z imenom lookUpTable
Mat lookUpTable(1, 256, CV_8U);
// Pridobimo kazalec na podatke
uchar* p = lookUpTable.ptr();
//p je kazalec na začetek podatkov v sliki
float gamma = 0.65;
// Vpis podatkov na sliko
// p[0] vrednost prvega slikovnega elementa
// p[i] vrednost i-tega slikovnega elementa
for (int i = 0; i < 256; ++i){
  //primer izračuna:
  //x=pow((i/255),gamma)
  //funkcija saturate cast preveri ali je vrednost x med 0 in 1;
  //vrednosti <0 postavi na 0 in vrednosti >1 postavi na 1
  //rezultat se na koncu pomnoži z 255 in zapiše v sliko lookUpTable
  p[i] = saturate_cast<uchar>(pow(i / 255.0, gamma) * 255.0);
}

// 6) Tudi takole lahko ustvarimo sliko in dostopamo do podatkov
// Krajši zapis
Mat_<double> SM(480,640);
for(int i = 0; i < SM.rows; i++)
  for(int j = 0; j < SM.cols; j++)
    SM(i,j) = double(i)/480.0;


imshow("Okno za prikaz", sl_test);
int k = waitKey(0); // čakaj pritisk tipke na tipkovnici
if(k == 's'){
        imwrite("rezultat.png", sl_test);
}

return 0;
}
```

Primer 2 prikazuje različne načine ustvarjanja novih slik. Osnovni koncept je, da najprej povemo tip `Mat`, potem podamo ime slike in nato v oklepaju dimenzije, tip podatkov in začetno vrednost, npr. `Mat SA(vrstic, stolpcev, tip podatkov, začetna vrednost)`. Za slikovne

elemente imamo na voljo več različnih tipov podatkov:

- CV_8U kratica za 8-bitna cela števila brez predznaka (U-unsigned 0 ... 255). Ta tip najbolj pogosto podaja svetlost v slikah. Ostale tipe v nadaljevanju potrebujemo predvsem pri izvajanju računskih operacij.
- CV_8S: kratica za 8-bitna cela števila s predznakom (–128 ... 127)
- CV_16U: kratica za 16-bitna cela števila brez predznaka (0 ... 65535)
- CV_16S: kratica za 16-bitna cela števila s predznakom (–32768 ... 32767)
- CV_32S: kratica za 32-bitna cela števila s predznakom (–214748368 ... 2147483647)
- CV_32F: kratica za 32-bitna števila s plavajočo vejco (–FLT_MAX ... FLT_MAX)
- CV_64F: kratica za 64-bitna števila s plavajočo vejco (–DBL_MAX ... DBL_MAX).

K tem oznakam se doda še podatek o številu barv "C1", "C3", npr. CV_8UC3 za 8-bitno barvno sliko. Če želimo novo ustvarjeni sliki dodeliti neke začetne vrednosti barv, da ne bo povsem črna, potem uporabimo četrti parameter. Slikovnim elementom (celotne slike) priredimo skalarno vrednost, eno, če gre za črnobele slike, in barvni trojček $Scalar(10, 100, 150)$ pri barvni sliki. Mnogokrat algoritmi zahtevajo normirane slike, tako da je razpon svetlosti med 0 in 1 ali med –1 in 1, podatkovni tip pa CV_32F ali CV_64F. To izvedemo z funkcijo `cv.normalize()`.

Operator .at<>(): Zelo pomembno je dostopanje do slikovnih elementov (branje ali pisanje), kar naredimo z `.at<>()` operatorjem, tj. `ime_slike.at<tipodatka>(vrstica, stolpec)` (primer 3)[3]. Parametra v okroglem oklepaju določata, kateri slikovni element obravnavamo. Tip podatka znotraj trikotnih oklepajev <> mora ustrezati tipu podatkov slikovnih elementov. Če je slika 8-bitna, potem je to `<unsigned char>`, če je CV_32F, potem je `<float>`, itd. Poseben primer nastopi, kadar želimo na barvni sliki dostopati do vrednosti barv B, G, R določenega slikovnega elementa. Vsaka barva ima 8-bitni zapis. OpenCV za branje predvideva vektorski tip podatka **Vec<T, n>**[4]. Za branje treh zaporednih bytov bo kombinacija T in n `3b`, vektorski zapis podatka bo `Vec3b`. Barve določenega slikovnega elementa tako preberemo z ukazom `Vec3b svetlost = SB.at<Vec3b>(i, j);` do posamezne barve v vektorju nato dostopamo z izbiro (indeksiranjem) želenega elementa `uchar B = svetlost.val[0];`.

Operator .ptr() in kazalci: Razumevanje uporabe kazalcev zahteva malo več znanja C++. V kazalcih je shranjen naslov, kje v spominu se nahaja spremenljivka (podobno kot hišni naslov). To je zelo močno orodje jezika C++ za delo z obsežnimi podatki, kot so matrike in slike. Pri računskih operacijah ni treba premikati spremenljivk (!velikih matrik), ampak se samo sklicujemo na njihove naslove. Primer 5 prikazuje uporabo kazalcev. Najprej pridobimo kazalec (`uchar* p`) na začetku podatkov slike, poimenovane npr. "lookUpTable" `uchar* p = lookUpTable.ptr();` za ta namen uporabimo operator **.ptr()**. Da gre za kazalec, vidimo iz deklaracije tipa spremenljivke `p`, kjer je uporabljena `*` (osnove C++). Do vrednosti i-tega slikovnega elementa dostopamo z ukazom `p[i]`. Po slikah se tipično pomikamo z dvema for zankama in uporabljamo dva indeksa (i, j), i za vrstice in j za stolpce. V primeru uporabe kazalcev pridobimo vrednost določenega slikovnega elementa tako, da izračunamo odmik slikovnega elementa od začetka podatkov, kamor kaže kazalec (običajno začetek slike, ni pa nujno!), torej `p[i*stolpcev+j]`. Pri tem privzamemo, da je slika v spominu zapisana v eno dolgo vrstico, tj. razvita po vrsticah.

[3] https://docs.opencv.org/3.4/d5/d98/tutorial_mat_operations.html
[4] https://docs.opencv.org/3.4/dc/d84/group___core___basic.html

Preliv: Na primeru uporabe kazalcev je prikazana še ena zelo pomembna funkcija: `saturate_cast<>()`, ki zagotovi, da ne bo prišlo do prelivanja celih števil. Zelo pogosta težava pri 8-bitnih celih številih se pojavi npr. pri seštevanju dveh števil. Recimo, da seštevamo števili `unsigned char A = 100;` in `unsigned char B = 156;` pri `A + B` se bo zgodil preliv, saj v 8-bitno število lahko zapišemo največjo vrednost 255. Računski 256 se bo prelil in zapisal v vrednost 0. Če bi prišteli 157, bi bila po prelitju zapisana vrednost 1 itd. Da bi se temu izognili, funkcija `saturate_cast<tip podatka>(številka)` preveri, ali se bo zgodil preliv in zapiše vrednost 255, če bo na največji vrednosti in 0 na najmanjši vrednosti.

Primer 6 kaže krajši zapis za ustvarjanje slik in dostop do slikovnih elementov. Primeren je za odlične programerje, ki si s tem skrajšujejo potrebo po tipkanju in dobro razumejo delovanje jezika C++. Omenjeno odsvetujem začetnikom, sploh pa mešani pristop.

4.3 Barvni prostori

Pogosto se srečamo z obdelavo barvnih slik. Na voljo imamo več barvnih prostorov, kjer osnovne R-, G- in B-barve transformiramo v neko novo trojico, ki je bolj primerna za obdelavo ali pa za kompresijo in prenos. Pomembnejši barvni prostori in njihova uporaba:

- **BGR** (Privzeto v OpenCV.) Za vsak slikovni element so uporabljeni trije zaporedni bajti, prvi predstavlja modro, drugi zeleno in tretji rdečo barvo.

- **RGB** (Privzeto v Matplotlib.) Sliko pretvorimo v ustrezen format takoj pri nalaganju slike `RGBslika = cv2.cvtColor(slika, cv2.COLOR_BGR2RGB)` z ustrezno specifikacijo drugega parametra `cv2.COLOR_BGR2RGB`.

- **HSV** (Odličen za segmentacijo barv.) Barvni odtenek H predstavlja kot v barvnem krogu, izražen v stopinjah od 0° do 360°, kjer rdeča ustreza 0°, zelena 120° in modra 240°. Nasičenost barve S določa intenzivnost barve in je podana kot vrednost med 0 (siva, brez barve) in 1 (popolnoma nasičena barva). Vrednost V predstavlja svetlost barve, prav tako v območju med 0 (črna) in 1 (najsvetlejši odtenek dane barve). Ta model omogoča enostavnejšo manipulacijo barv v primerjavi z RGB-modelom. OpenCV omogoča pretvorbo med BGR- in HSV-formatom s parametroma `cv2.COLOR_BGR2HSV` in `cv2.COLOR_HSV2BGR`.

- **LAB** (Primernejši za obdelavo slik.) Je barvni model, ki temelji na človeškem zaznavanju barv. Komponenta L predstavlja svetlost barve in se giblje med 0 (črna) in 100 (bela), medtem ko komponenti A in B določata barvne informacije, pri čemer vrednosti A segajo od -128 (zelena) do 127 (rdeča) in vrednosti B od -128 (modra) do 127 (rumena). Barvni prostor LAB je uporaben pri obdelavi slik, saj omogoča neodvisno prilagajanje svetlosti in barvnih komponent pri barvnih korekcijah, segmentaciji ter izboljšavi kontrasta. OpenCV omogoča pretvorbo med BGR- in LAB-formatom s parametroma `cv2.COLOR_BGR2LAB` in `cv2.COLOR_LAB2BGR`.

- **YUV** je barvni model, ki ločuje svetlost Y od barvnih komponent U in V, s čimer omogoča učinkovitejše kodiranje slik in videoposnetkov. Načeloma je podoben LAB, kanal Y predstavlja svetlost slike (sivinska slika), medtem ko kanala U in V vsebujeta informacije o barvnih odtenkih, pri čemer U določa razliko med modro in svetlostjo ($U = B - Y$), V pa razliko med rdečo in svetlostjo ($V = R - Y$). Ta ločitev omogoča zmanjšanje ločljivosti barvnih kanalov brez večje izgube vizualne kakovosti, saj je človeško oko bolj občutljivo na svetlost kot na barvne razlike. YUV se pogosto uporablja v videokodiranju, televiziji in slikovni kompresiji (npr. JPEG, MPEG). OpenCV omogoča pretvorbo med BGR- in YUV-formatom s parametroma `cv2.COLOR_BGR2YUV` in `cv2.COLOR_YUV2BGR`.

Barvno sliko pretvorimo v sivinsko z ukazom `cv2.COLOR_BGR2GRAY`. Če je v sliki prisoten **alfa kanal (prosojnost) RGBA**, ga naložimo s: `slika_rgba = cv2.imread(slika, cv2.IMREAD_UNCHANGED)`. Koda v nadaljevanju prikazuje primer, kako v HSV-barvnem prostoru izdelamo masko za selektiranje objektov rdeče barve.

```
#izdelava maske rdečih objektov
hsv_slika = cv2.cvtColor(slika_bgr, cv2.COLOR_BGR2HSV)
spodnja_meja_rdece = np.array([0, 120, 70])
zgornja_meja_rdece = np.array([10, 255, 255])
maska = cv2.inRange(hsv_slika, spodnja_meja_rdece, zgornja_meja_rdece)

# Pretvorba BGR v LAB format
lab_slika = cv2.cvtColor(slika_bgr, cv2.COLOR_BGR2LAB)
# Sliko ločimo na LAB komponente
L, A, B = cv2.split(lab_slika)
cv2.imshow("L␣barvna␣komponenta␣oz.␣sivinska␣slika", L)
```

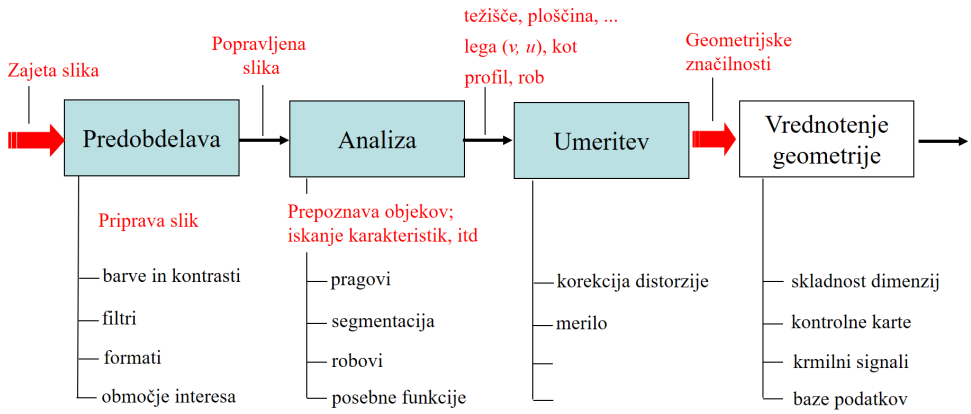
Vprašanja

- Kako ustvarimo sliko določene velikosti in barve v OpenCV?
- Kako ustvarimo kopijo slike v OpenCV? V čem se razlikuje od preprostega dodeljevanja ene spremenljivke drugi?
- Kako izberemo območje interesa?
- Katere podatkovne tipe uporablja OpenCV za shranjevanje slik? Kako obdelava slik vpliva na izbiro podatkovnega tipa?
- Opišite postopek branja slike iz datoteke in pisanja v datoteko.
- Kako pretvorimo sliko z enega barvnega prostora v drugega?
- Opišite možne načine dostopa do slikovnih elementov v OpenCV. Kako spreminjamo vrednosti slikovnih elementov?
- Kako preprečimo prekoračitev dosega izbranega podatkovnega tipa?
- Kako uporabimo kazalce pri ravnanju z velikimi slikami?*

4.4 Osnovni koraki v obdelavi slike

Do zdaj smo spoznali razvoj slikovnega sistema, zajem in osnovno ravnanje s slikami. V praksi smo vedno soočeni z nekim problemom, zato moramo domisliti zaporedje operacij obdelave slike, da bomo izluščili iskano informacijo, npr. koordinate centra objekta, ki ga moramo pobrati z robotom, ali pa ugotoviti, ali je izdelek poškodovan. Vsak problem ima specifično rešitev, zato je težko podati univerzalni recept. Kar lahko svetujemo, je funkcijsko zaporedje operacij, kot ga prikazuje slika 4.3. Vsaka obdelava slike gre skozi štiri faze, vendar ni nujno, da nastopajo vse štiri. Prva je **predobdelava slike**, pri kateri zajeto sliko pripravimo na **analizo** v drugi fazi. V fazi predobdelave sliko obrežemo, skaliramo, normiramo, filtriramo, poravnamo histogram ali spremenimo barvni format, npr. iz YUV v RGB ali pa v sivinsko. Izhod iz prve faze je slika v obliki in formatu, ki je potreben v dejanski analizi (stvar uporabljenih algoritmov). V fazi analize uporabimo razne metode, kot so iskanje robov, geometrijski izračuni, prepoznava vzorcev itd. Rezultat druge faze tipično ni več slika, ampak ena ali več (množica) točk, ki predstavljajo center objekta, rob objekta, profil površine, ali je izdelek dober oz. slab, itd. Tukaj že dobimo komprimirano informacijo. V nekaterih primerih, ko npr. izdelek je ali ni poškodovan, obdelavo slike zaključimo v tej točki.

Če gre za z dimenzijami povezane meritve, lokalizacijo izdelkov in podobno, nastopi tretja faza **umeritev**, v kateri umerimo zaznane točke in s tem preidemo iz slikovnega koordinatnega sistema v kartezični koordinatni sistem – podobno, kot je razloženo v poglavju o umerjanju kamer. Umeritev je smiselno izvajati šele v tej fazi, na rezultatih analize, tj. množici točk, saj je količina potrebnih preračunov bistveno manjša, kot če bi izvajali umeritev na vsej sliki. Četrta faza, **vrednotenje**, nastopi v primeru, kadar izmerke primerjamo s specifikacijami, da pridemo do ugotovitve, ali je izdelek znotraj predpisanih toleranc. V tej fazi izvajamo razne izračune in generiramo krmilne signale. Primer tega bi bil izračun zamika varilne šobe glede na zvarno režo (pri obdelavi slike zaznamo zvarno režo) in pošiljanje popravka poti na krmilnik robota.

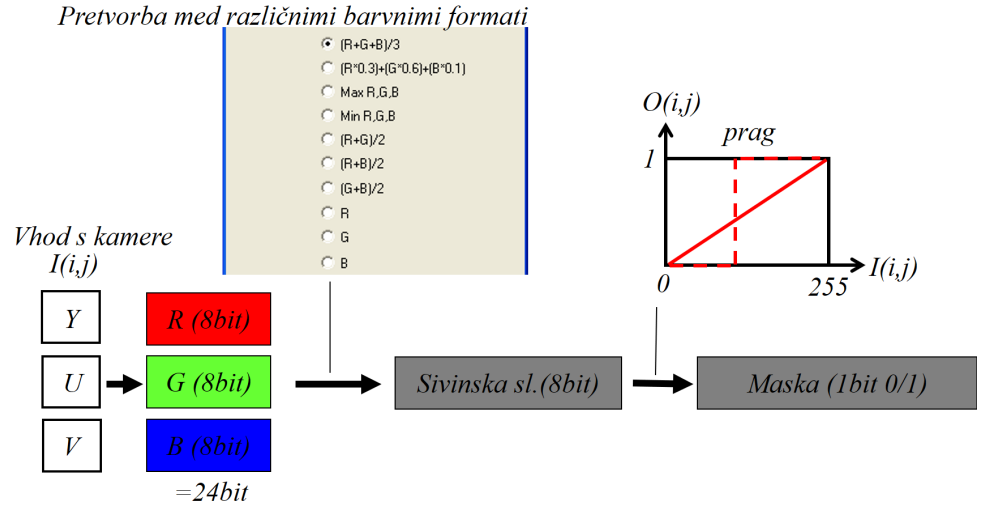


Slika 4.3: Tipične faze obdelave slike

Primer predobdelave: Slika 4.4 prikazuje primer, kako v fazi predobdelave sliko pripravimo za dejansko analizo. Scenarij je naslednji: bel izdelek okrogle oblike potuje po črnem tekočem traku. S slikovnim sistemom želimo določiti lego izdelka (lokalizacija) za potrebe robotskega pobiranja. V slikovnem sistemu je uporabljena barvna kamera. Predpostavimo, da želimo v fazi predobdelave sliko pripraviti do te mere, da bodo imeli slikovni elementi, ki prikazujejo objekt, vrednost 1, vsi ostali, ki se nanašajo na ozadje, pa 0 (maska). Na takšni sliki bi v fazi analize enostavno izračunali težišče objekta (funkcija Center of gravity v RoboRealmu). Rezultat faze analize je prikazan na sliki 4.2.

Format zapisa slike v spominu določimo že na gonilniku kamere. Če bi izbrali YUV (izhod

kamere), potem bi v fazi predobdelave slike najprej morali narediti ustrezno barvno pretvorbo, torej iz YUV v RGB. Naslednji korak je, da barvno sliko spremenimo v sivinsko sliko. To lahko naredimo na več načinov. Prikazan je del uporabniškega vmesnika v *RoboRealmu*, kjer izbiramo, kako bo izvedena pretvorba. Vzamemo lahko posamezno barvo ali kombinacijo barv. Če bi bil izdelek npr. rdeče barve, potem bi bilo smiselno s slike vzeti samo R-barvno ravnino. Ker je bel, vzamemo povprečje vseh treh barv $(R + G + B)/3$. Najbolje je preizkusiti več kombinacij, da najdemo najbolj ustrezno. Še ta zanimivost: če bi bila slika zapisana v YUV-formatu, potem je zadosti, da vzamemo samo Y-barvno komponento, saj ta predstavlja sivinsko sliko. Vrednosti slikovnih elementov v sivinski sliki so med 0 in 255. Specifikacija maske za prepoznavo objektov zahteva vrednost 1 na objektu, drugje v ozadju pa 0. Za ta prehod je najbolje uporabiti pragovno funkcijo (ang. threshold, glej točkovne operacije), ki vrednostim slikovnih elementov vhodne slike I(i, j), višjim od praga v izhodni sliki O(i, j), dodeli 1, nižjim pa 0.



Slika 4.4: Primer predobdelave slike

Analiza: za predstavo si poglejmo, kako bi izvedli analizo, tj. izračun težišča objekta, če bi sami napisali algoritem v C++ (odločitev za C++ je predvsem v tem, ker bi bil zaradi *for* zank izračun v Pythonu zelo počasen). Izračun temelji na splošni enačbi za računanje težišča

$$x_t = \frac{\sum_{i=0}^{n} A_i x_i}{A}. \tag{4.1}$$

V našem primeru bo A_i svetlost slikovnega elementa $I_{\mathrm{v,u}}$, ki zavzame vrednosti 0 ali 1 (dobimo po uporabi pragovne funkcije). x_i bo številka stolpca v u (x) smeri na sliki. Zmožek $A_\mathrm{i}x_\mathrm{i}$ bo $I_{\mathrm{v,u}} \cdot u$ (za u smer), naredimo ga samo na slikovnih elementih, katerih vrednost je 1. A predstavlja seštevek vseh slikovnih elementov z vrednostjo 1. Podobno je v v smeri.

```
//SO je vhodna 8-bitna slika ..  I(v,u)
//!računske operacije izvajamo z int in float spremenljivkami (ne z 8-bitnimi),
//saj bodo seštevki bistveno večji od 255, center bo določen s podtočkovno natančnostjo
int A = 0;
int xsestevek=0;
int ysestevek=0;
float xc,yc;
//Premikanje po vhodni sliki SO
for(int v = 0; v < SO.rows; v++)
  for(int u = 0; u < SO.cols; u++)
    if(SO.at<unsigned char>(v,u)){ //če je vrednost 1
```

```
        A+=1;
        xsestevek+=u; //1*u
        ysestevek+=v; //1*v
    }
xc=float(xsestevek)/float(A);
yc=float(ysestevek)/float(A);
```

Sledila bi faza **umeritve**, kjer bi koordinate težišča (x_c, y_c), ki so izražene v slikovnem koordinatnem sistemu, transformirali v kamerin koordinatni sistem. Za ta namen bi morala biti kamera umerjena in določeni zunanji parametri R, T glede na umeritveno šahovnico, ki bi jo položili na tekoči trak. Izračun koordinat v 3D bi potekal po postopku, opisanem v podpoglavju 3.6. V končni fazi bi izračunane koordinate poslali na robotski krmilnik. Še prej bi morali poskrbeti za skupni koordinatni sistem robota in kamere. Najlažje je to izvesti z umeritveno šahovnico, ki omogoča določanje zunanjih parametrov kamere. Če umeritveno šahovnico pustimo na tekočem traku, ne da bi jo premikali, potem lahko v prijemalo robota vpnemo iglo in z njo pokažemo GKS-umeritvene šahovnice po metodi treh točk. GKS postane "User frame" na robotu, sočasno pa tudi skupni koordinatni sistem s kamero, s tem da na strani kamere koordinate preračunavamo v GKS po enačbah (3.19).

4.5 Pregled funkcij obdelave slike

Najprej povejmo, da po funkcionalnosti sistematizirano dokumentacijo OpenCV najdemo na spletni strani "OpenCV Tutorials"[5]. V tem učbeniku želimo podati osnovno razumevanje ravnanja s slikami in pregled osnov obdelave slike. Pregled temelji na obstoječem nivoju poznavanja aritmetičnih in logičnih funkcij, pridobljenih med študijem. Tematik s področja umetne inteligence ne obravnavamo.

V nadaljevanju si bomo najprej ogledali **točkovne** operacije, ki se izvajajo na vsakem slikovnem elementu, in sicer neodvisno od sosedov. Sledil bo pregled **lokalnih** operacij, pri katerem upoštevamo še sosednje slikovne elemente v lokalni okolici izbranega slikovnega elementa. Gre predvsem za uporabo konvolucijskih jeder in z njimi povezanih efektov. Na koncu si bomo ogledali še transformacije.

4.5.1 Točkovne operacije

Logične funkcije

Logične funkcije **IN, ALI, NE** in njihove kombinacije se izvajajo na posameznih slikovnih elementih in logično primerjajo vrednosti **bitov** sovpadajočih (na enaki lokaciji v, u) slikovnih elementov dveh slik. Primer v nadaljevanju kaže uporabo logičnih funkcij na 8-bitnih slikah v Pythonu:

```
import cv2
import numpy as np
SA = cv2.imread('SA.png')
SB = cv2.imread('SB.png')

SCand = cv2.bitwise_and(SA, SB)
SCor = cv2.bitwise_or(SA, SB)
SCxor = cv2.bitwise_xor(SA, SB)
SCnot = cv2.bitwise_not(SA)
...
```

Tipična področja uporabe se delijo glede na efekte, ki jih želimo doseči.

[5] https://docs.opencv.org/4.x/d9/df8/tutorial_root.html

Maskiranje: Logične funkcije uporabimo za osredotočanje obdelave na določena zanimiva območja. Na primer prepoznavo objektov izvedemo samo na območju, ki ga določa maska, lahko pa tudi spreminjamo vrednosti slikovnih elementov na določenih mestih ali kombiniramo slike. Primeri v nadaljevanju prikazujejo nekaj logičnih operacij med sliko SA in masko MA. **Pozor, operacije se izvajajo na nivoju bitov!** Če so vrednosti v maski 255, potem se v rezultatu ohranijo vrednosti slike (Primer 1). V nasprotnem se vrednosti spremenijo (Primeri 2–5). Pomagamo si lahko s pravili Boolove algebre, predvsem z vidika ugotavljanja, kakšne vrednosti zapisati v masko in kaj bomo s tem dosegli.

```
100 dec = 01100100 bin
255 dec = 11111111 bin
 &(IN)  = 01100100 bin = 100 dec

Primer 1
(SA)                        (MA)                    bitwise_and(SA, MA)
[100 100 100 100 100       0   0   0   0   0      0   0   0   0   0]
[100 100 100 100 100       0 255 255   0   0      0 100 100   0   0]
[100 100 100 100 100  &    0 255 255   0   0 =    0 100 100   0   0]
[100 100 100 100 100       0   0   0   0   0      0   0   0   0   0]
[100 100 100 100 100       0   0   0   0   0      0   0   0   0   0]
Primer 2
(SA)                        (MA)                    bitwise_and(SA, MA)
[100 100 100 100 100       0   0   0   0   0      0   0   0   0   0]
[100 100 100 100 100       0   1   1   0   0      0   0   0   0   0]
[100 100 100 100 100  &    0   1   1   0   0 =    0   0   0   0   0]
[100 100 100 100 100       0   0   0   0   0      0   0   0   0   0]
[100 100 100 100 100       0   0   0   0   0      0   0   0   0   0]
Primer 3
(SA)                        (MA)                    bitwise_and(SA, MA)
[100 100 100 100 100       0   0   0   0   0      0   0   0   0   0]
[100 100 100 100 100       0 222 222   0   0      0  68  68   0   0]
[100 100 100 100 100  &    0 222 222   0   0 =    0  68  68   0   0]
[100 100 100 100 100       0   0   0   0   0      0   0   0   0   0]
[100 100 100 100 100       0   0   0   0   0      0   0   0   0   0]
Primer 4 (OR)
(SA)                        (MA)                    bitwise_or(SA, MA)
[100 100 100 100 100       0   0   0   0   0    100 100 100 100 100]
[100 100 100 100 100       0 222 222   0   0    100 254 254 100 100]
[100 100 100 100 100 OR    0 222 222   0   0 =  100 254 254 100 100]
[100 100 100 100 100       0   0   0   0   0    100 100 100 100 100]
[100 100 100 100 100       0   0   0   0   0    100 100 100 100 100]
Primer 5 (XOR)
(SA)                        (MA)                    bitwise_xor(SA, MA)
[100 100 100 100 100       0   0   0   0   0    100 100 100 100 100]
[100 100 100 100 100       0 222 222   0   0    100 186 186 100 100]
[100 100 100 100 100 XOR   0 222 222   0   0 =  100 186 186 100 100]
[100 100 100 100 100       0   0   0   0   0    100 100 100 100 100]
[100 100 100 100 100       0   0   0   0   0    100 100 100 100 100]
```

Unija in presek dveh mask. Te operacije so koristne pri združevanju ali analiziranju območij zanimanja na sliki. Primer v nadaljevanju kaže presek in unijo dveh mask. Rezultirajočo masko nato uporabimo na sliki.

```
PRESEK
(MA)                    (MB)                    bitwise_and(MA, MB)
[255 255 255 255        255 255 255 255         255 255 255 255]
[  0 255 255 255        255 255 255   0           0 255 255   0]
[  0   0 255 255 &      255 255   0   0   =       0   0   0   0]
[  0   0   0 255        255   0   0   0           0   0   0   0]
UNIJA
(MA)                    (MB)                    bitwise_or(MA, MB)
[255 255 255 255        255 255 255 255         255 255 255 255]
[  0 255 255 255        255 255 255   0         255 255 255 255]
[  0   0 255 255 OR     255 255   0   0   =     255 255 255 255]
[  0   0   0 255        255   0   0   0         255   0   0 255]
```

Pragovna obdelava

Pragovna obdelava slike je tehnika obdelave slik, pri katerikjer primerjamo vrednosti slikovnih elementov z mejno vrednostjo **prag (ang. threshold)**. Ta se giblje med 0 in 255 pri 8-bitnih slikah oziroma med 0 in 1 pri normiranih. Z njo ločimo predmete ali pomembne značilke od ozadja. Če je vrednost slikovnega elementa višja od praga, potem je izhodna vrednost visoka, tj. 255, v nasprotnem pa 0 (primer d) na sliki 4.5. Primer uporabe pragovne funkcije je prikazan tudi na sliki 4.4. V nadaljevanju sledi prikaz uporabe pragovne funkcije s pragom 100 na vhodni sliki SA. Izhod iz pragovne funkcije je maska M.

```
(SA)                    (M = SA > prag), prag = 100

[ 50  75 100  75        0   0   0   0]
[  0  50 175 200        0   0 255 255]
[  0   0 225 250        0   0 255 255]
[  0   0   0  55        0   0   0   0]

V OpenCV ...
M=cv.threshold(SA,100,255,cv.THRESH_BINARY)
Kakšne vrednosti bodo zapisane v izhodni maski, določimo
z zadnjima dvema parametroma (glej OpenCV dokumentacijo).
Glejte tudi:
M=cv.adaptiveThreshold(SA,255,cv.ADAPTIVE_THRESH_GAUSSIAN_C,...
cv.THRESH_BINARY,11,2)
```

Vprašanja

- Kako se logične funkcije IN, ALI in NE ter njihove kombinacije uporabljajo pri manipulaciji slik in kakšni so njihovi učinki?
- Kaj je pragovna obdelava in zakaj se uporablja? Kako izberemo ustrezen prag?
- Kako se logične funkcije uporabljajo za sestavljanje mask?
- Kako z logičnimi funkcijami kombiniramo dve ali več slik?
- Predpostavimo sliki A in B. Kakšen bo rezultat v primeru logične funkcije NE (A ALI B)?
- Kakšne so performance bitne aritmetike v smislu hitrosti izvajanja računski operacij?

Aritmetične funkcije

Aritmetične funkcije uporabimo za izvajanje izračunov v okviru raznih algoritmov. Uporabljamo jih prvenstveno na podatkovnih tipih `int, float` in `double`. V računalniku jih izvaja matematični koprocesor (ali GPU), zato se izvajajo enako hitro kot operacije na 8-bitnih podatkih. Pri slednjih smo močno omejeni z zalogo vrednosti, ki obsega 256 števil. V OpenCV je najbolje uporabljati že pripravljene funkcije, ki so optimirane za hitrost, imajo pa tudi vgrajene razne varnostne mehanizme, prvenstveno obvladovanje prekoračitve dosega spremenljivke (izven območja med 0 in 255 pri 8-bitnih podatkih). **Računske operacije se izvajajo med sovpadajočimi sl. elementi!** Primer kode v nadaljevanju kaže uporabo OpenCV-aritmetičnih funkcij v Pythonu:

```
SC=cv.add(SA,SB)
SC=cv.addWeighted(SA,0.6,SB,0.4,0.1) #SC=0.6*SA+0.4*SB+0.1
SC=cv.subtract(SA,SB)
SC=cv.multiply(SA,SB)
SC=cv.divide(SA,SB)
SC=cv.pow(SA,0.25)
SC=cv.sqrt(SA)
itd ...
```

Pri računanju z 8-bitnimi števili smo močno omejeni z naborom razpoložljivih računskih operacij. V glavnem operiramo s seštevanjem in z odštevanjem. Za deljenje in množenje je priporočljiva uporaba premikanja bitov v levo ali desno (shift funkcije), ki se označujejo z `množenec«n` in `deljenec»n` in omogočajo izredno učinkovito deljenje in množenje celih števil z 2^n. Primer:

```
Pomik levo (left shift)
2<<1=4   ... množenje z 2 (2^1)
2<<2=8   ... množenje z 4 (2^2)
2<<3=16  ... množenje z 8 (2^3)
Pomik desno (right shift)
16>>1=8  ... deljenje z 2 (2^1)
16>>2=4  ... deljenje z 4 (2^2)
16>>3=2  ... deljenje z 8 (2^3)
```

Če potrebujemo množenje in deljenje na nivoju 8-bitnih števil, potem algoritme prilagodimo razpoložljivim operacijam premikanja bitov. Predpostavimo, da želimo izboljšati sliko s povprečenjem več zaporednih slik. To pomeni, da bomo zajeli več slik, jih medsebojno sešteli in rezultat delili s številom slik (vse na nivoju sovpadajočih slikovnih elementov). Ker bomo deljenje izvedli s premikanjem bitov, je smiselno za povprečenje izbrati 2^n slik, torej 2, 4, 8 itd. V nadaljevanju je prikaz računskih operacij povprečenja, ki se odvijajo za slikovni element (v, u):

```
Primer povprečenja 4 slik (for zanke niso prikazane)
SA(v,u)=10, SB(v,u)=20, SC(v,u)=30, SD(v,u)=40
10+20+30+40=100
Savg(v,u)=100>>2=25

Primer prekoračitve dosega 8 bitnih števil
SA(v,u)=100, SB(v,u)=200, SC(v,u)=130, SC(v,u)=240
100+200+130+240=? Pri 8 bitnih številih se tu zgodi prekoračitev
seštevek je 670, v 8 bitno število lahko zapišemo največ 255
```

Možna rešitev je, da preoblikujemo računanje povprečja z zamenjavo vrstnega reda računskih operacij: seštevanje in deljenje spremenimo v deljenje in seštevanje:

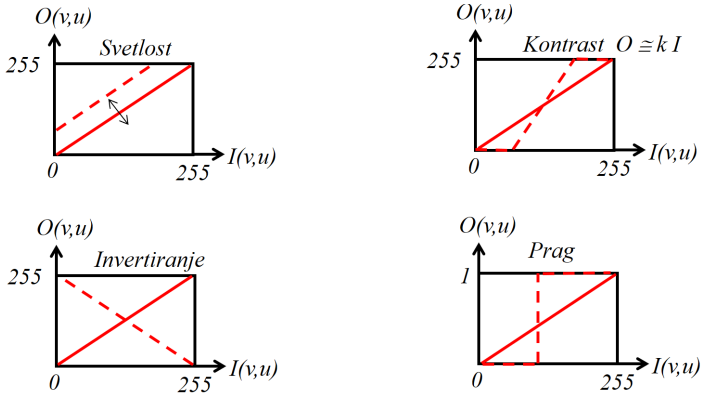
$$\frac{(a+b+c+d)}{4} = \frac{a}{4} + \frac{b}{4} + \frac{c}{4} + \frac{d}{4}. \tag{4.2}$$

Izračun povprečja je tako:

```
Rešitev prekoračitve
SA(v,u)=100, SB(v,u)=200, SC(v,u)=130, SC(v,u)=240
Savg(v,u)=(100>>2)+(200>>2)+(130>>2)+(240>>2)=167
```

Najbolj enostavna uporaba aritmetičnih operacij je pri **prilagajanju kontrasta, svetlosti in celotnega videza slike**. Slika 4.5 prikazuje primer spreminjanja svetlosti, kontrasta in invertiranja. Primer a) prikazuje spremembo svetlosti slikovnega elementa (katerekoli barve) s prištevanjem ali z odštevanjem konstantne vrednosti $O(\mathrm{v}, \mathrm{u}) = I(\mathrm{v}, \mathrm{u}) \pm n$. Primer b) prikazuje spremembo kontrasta. Uporabljena je linearna odvisnost med vhodom in izhodom $O(\mathrm{v}, \mathrm{u}) = kI(\mathrm{v}, \mathrm{u}) + n$ in kontrola prekoračitve dosega podatkovnega tipa. Kontrast spreminjamo s spreminjanjem naklona in lege premice. V programskih paketih za obdelavo fotografij se tipično uporabljajo polinomske krivulje višjih redov, katerih obliko enostavno nastavimo z miško v uporabniškem vmesniku. Primer c) prikazuje invertiranje slike $O(\mathrm{v}, \mathrm{u}) = 255 - I(\mathrm{v}, \mathrm{u})$, primer d) pa pragovno funkcijo, ki smo jo že predhodno spoznali.

Slike je možno prilagoditi tudi na druge načine. Na primer, s predhodno predstavljenim povprečjem slik izboljšamo ostrino in zmanjšamo šum. Odštevanje dveh slik je uporabno, kadar želimo s slike odstraniti ozadje. Predpostavimo kamero, togo vgrajeno nad tekoči trak. Najprej slikamo trak in okolico brez izdelka, nato z izdelkom. Na obeh slikah je ozadje izdelka enako. Če jih odštejemo med sabo, pobrišemo ozadje in na sliki ostane samo objekt interesa. Za ustvarjanje kompozitnih slik, na katerih združujemo dve ali več slik, se uporabi funkcija `addWeighted`.



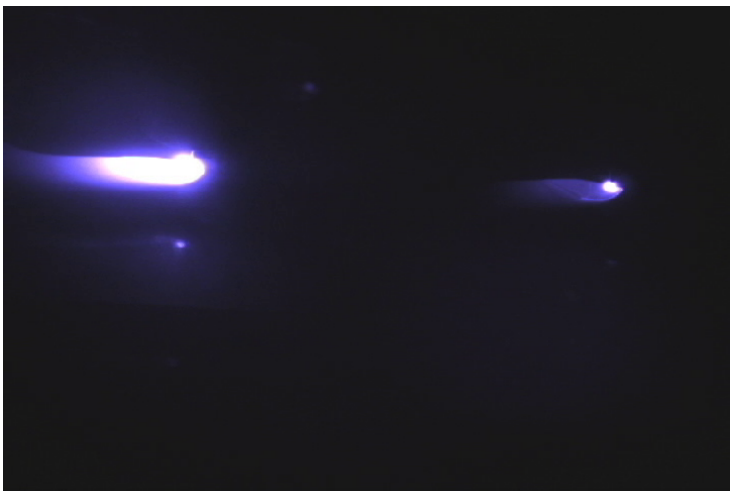
Slika 4.5: Spreminjanje svetlosti slikovnih elementov
$I(\mathrm{v}, \mathrm{u})$: svetlost slikovnega elementa vhodne slike
$O(\mathrm{v}, \mathrm{u})$: svetlost slikovnega elementa izhodne slike po prilagoditvi

Primer uporabe točkovnih operacij demonstriramo na primeru sestavljanja slik pri vizualnem nadzoru varjenja. Slika 4.6 (a) prikazuje sliko taline in nastalega zvara med varjenjem. S posebno optiko je slika razdeljena na levo in desno polovico. Optična pot na desni je bolj zatemnjena od leve, čas osvetlitve pa nastavljen tako, da se na desni lepo vidi zgolj talina brez okolice, na levi pa se vidi več okolice, talina pa je preosvetljena in se ne vidi. Cilj obdelave slike je na desni prepoznati in izrezati območje taline ter ga skopirati na levo stran na mesto, kjer je talina preosvetljena. Dobljena slika tako prikazuje širšo okolico zvara in tudi talino, kot prikazuje slika 4.6 (b). Postopek obdelave slike z uporabo točkovnih operacij je v nadaljevanju, razlaga pa v komentarjih med ukazi:

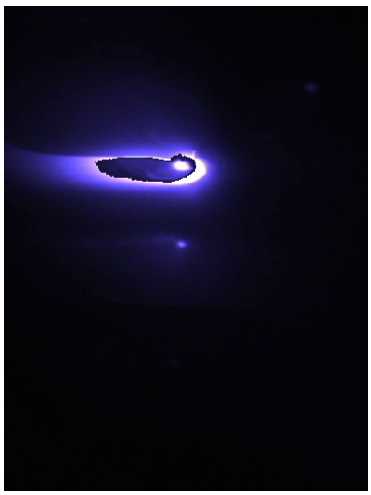
```
import cv2
import numpy as np

def obdelaj_video(pot_do_videa):
# Odpri video
```

```
cap = cv2.VideoCapture(pot_do_videa)
while cap.isOpened():
        ret, SL = cap.read()
        if not ret:
                break

# Razdeli sliko na levo in desno polovico
visina, sirina = SL.shape[:2]
leva_slika = SL[:, :sirina//2]
desna_slika = SL[:, sirina//2:]
# Pretvori obe sliki v sivinsko (CV_8U) za pragovno obdelavo v nadaljevanju
leva_sivinska = cv2.cvtColor(leva_slika, cv2.COLOR_BGR2GRAY)
desna_sivinska = cv2.cvtColor(desna_slika, cv2.COLOR_BGR2GRAY)
# Izoliramo zvar in oblok (izdelava mask)
# ..  najprej uporabimo pragovno funkcijo na obeh polovicah.
_,leva_prag = cv2.threshold(leva_sivinska, 200, 255, cv2.THRESH_BINARY)
_,desna_prag = cv2.threshold(desna_sivinska, 40, 255, cv2.THRESH_BINARY)

# ..  obdržimo le največjo skupino povezanih slikovnih elementov (največji otok)
leva_prag = obdrzi_najvecjo_tocko(leva_prag) #Definicija funkcije v nadaljevanju
desna_prag = obdrzi_najvecjo_tocko(desna_prag)
# ..  razširimo masko (po potrebi, glej morfološke operacije v nadaljevanju)
kernel = np.ones((5, 5), np.uint8)
desna_prag = cv2.dilate(desna_prag, kernel, iterations=2)

# Preverimo dimenzije mask
# ..poiščemo dimenzije ROI največjega otoka na levi sliki
x_leva, y_leva, w_leva, h_leva = cv2.boundingRect(leva_prag)
# ..poiščemo dimenzije ROI največjega otoka na desni sliki
x_desna, y_desna, w_desna, h_desna = cv2.boundingRect(desna_prag)
# ..  ignoriramo slike, kjer je ROI premajhen za obdelavo
if w_desna == 0 or h_desna == 0 or w_leva == 0 or h_leva == 0:
        print("Preskočena SL zaradi premajhnega ROI (črna slika).")
        continue

# Uporaba mask;
#funkcija bitwise_and kot vhod vzame dve sliki in masko.  Če želimo uporabiti masko na samo eno
    sliko, jo podamo 2-krat (S IN S) IN M je enako S IN M
desna_slika_maskirana = cv2.bitwise_and(desna_slika, desna_slika, mask=
    desna_prag)
# ..  izrežemo ROI taline
desna_slika_izrezana = desna_slika_maskirana[y_desna:y_desna + h_desna, x_desna
    :x_desna + w_desna]
desna_prag_izrezana = desna_prag[y_desna:y_desna + h_desna, x_desna:x_desna +
    w_desna]

# ..  prilagodimo levo ROI velikosti desnega ROI (v nasprotnem se program sesuva!)
h_tarča = min(h_desna, h_leva)
w_tarča = min(w_desna, w_leva)
leva_roi = leva_slika[y_leva:y_leva + h_tarča, x_leva:x_leva + w_tarča]
# ..  prilagodi velikost maske in izrezane ROI, če je potrebno
if desna_prag_izrezana.shape[:2] != leva_roi.shape[:2]:
        desna_slika_izrezana = cv2.resize(desna_slika_izrezana, (w_tarča, h_tarč
            a), interpolation=cv2.INTER_NEAREST)
        desna_prag_izrezana_prilagojena = cv2.resize(desna_prag_izrezana, (w_tar
            ča, h_tarča), interpolation=cv2.INTER_NEAREST)
else:
        desna_prag_izrezana_prilagojena = desna_prag_izrezana

# Invertiraj masko za levo območje
maska_inv = cv2.bitwise_not(desna_prag_izrezana_prilagojena)
```

```
# Ustvari luknjo v levi ROI, kjer bo postavljena slika taline z desne
 leva_podlaga = cv2.bitwise_and(leva_roi, leva_roi, mask=maska_inv)
# Izloči talino iz desne ROI z uporabo prilagojene maske
 desna_sprednji_del = cv2.bitwise_and(desna_slika_izrezana, desna_slika_izrezana
     , mask=desna_prag_izrezana_prilagojena)
# Združi obe sliki
 rezultat = cv2.add(leva_podlaga, desna_sprednji_del)
# Postavi rezultat nazaj v levo sliko na ustrezno lokacijo
 leva_slika[y_leva:y_leva + h_tarča, x_leva:x_leva + w_tarča] = rezultat

# Prikaz
 cv2.imshow("Obdelan Video", leva_slika)
 if cv2.waitKey(1) & 0xFF == ord('q'):
         break
cap.release()
cv2.destroyAllWindows()

# Definiramo funkcijo
def obdrzi_najvecjo_tocko(SLprag):
# Poišči konture (glej navodila OpenCV)
 konture, _ = cv2.findContours(SLprag, cv2.RETR_EXTERNAL, cv2.
     CHAIN_APPROX_SIMPLE)
# Če ni kontur, vrni originalni SLprag
 if not konture:
         return SLprag
# Poišči največjo konturo
 najvecja_kontura = max(konture, key=cv2.contourArea)
# Ustvari prazno masko in nariši največjo konturo
 maska = np.zeros_like(SLprag)
 cv2.drawContours(maska, [najvecja_kontura], -1, 255, thickness=cv2.FILLED)
 return maska

# Uporaba
obdelaj_video('video0001 15-59-54.avi')
```



(a) Izvorna slika



(b) Sestavljena slika

Slika 4.6: Primer uporabe točkovnih operacij pri sestavljanju slik za potrebe vizualnega nadzora procesa varjenja

Vprašanja

- Kako se izvajajo osnovne aritmetične operacije (+, -, *, /)?
- Kakšne so posebnosti pri delu z 8-bitnimi slikami, še posebej glede na prekoračitev dosega določenega podatkovnega tipa?
- Kako in zakaj uporabljamo operacije bitnega premikanja (», «)?
- Kako z aritmetičnimi operacijami spreminjamo intenziteto in kontrast slik?
- Za kaj uporabljamo pragovno funkcijo?
- Kaj je maska?
- Kako z maskami izvajamo selektivno aritmetiko na določenih delih slike?
- Kako z uporabo aritmetičnih operacij povprečimo slike? Prikažite primer povprečenja 8 slik z 8-bitnimi podatki.
- Zakaj je pomembno razumeti podatkovne tipe slik pri izvajanju aritmetičnih operacij?
- Kakšne so omejitve in izzivi pri uporabi slikovne aritmetike v aplikacijah, kjer je pomembna hitrost obdelave slik (npr. v realnem času, kjer mora biti slika obdelana prej, preden kamera dostavi novo sliko)?

4.5.2 Lokalne operacije

Pri lokalnih operacijah v trenutni računski točki (i, j) upoštevamo še vrednosti sosednjih slikovnih elementov. Po sliki se premikamo z računskim jedrom, katerega velikost in oblika določa velikost lokalnega vplivnega območja na trenutno točko (i, j). Kakšen vpliv bodo imeli sosedje v neki točki, določimo z vrednostmi elementov računskega jedra. Računsko jedro je v bistvu majhna tabela velikosti, npr. 3 x 3, 5 x 5, 3 x 4 itd. Jedro ima center, pri lihih dimenzijah jedra je to kar srednji element, drugače pa element, ki si ga izberemo. Jedro navidezno postavimo nad sliko, tako da center jedra sovpada s trenutno računsko točko (i, j). V dani legi medseboj pomnožimo sovpadajoče elemente, zmnožke seštejemo, dobljeno vrednost pa zapišemo v rezultirajočo sliko. Jedro pomaknemo v novo računsko točko in ponovimo celoten postopek. Opisani postopek se imenuje **konvolucija**, jedro pa **konvolucijsko jedro**. Podobno delujejo tudi **morfološke** funkcije, le da so te bolj preproste in omejene na maske.

Konvolucija

V naslednjem primeru je prikazan postopek konvolucije s konvolucijskim jedrom J velikosti 3 x 3. Center jedra predstavlja srednji element z vrednostjo 1.0; jedro je na začetku izračuna konvolucije navidezno pozicioniramo nad levi zgornji vogal slike SA. Center jedra torej sovpada z elementom (1, 1) slike SA, katerega vrednost je 200. Med seboj pomnožimo sovpadajoče elemente, zmnožke seštejemo in dobimo vrednost 500. Dobljeno vrednost zapišemo v rezultirajočo sliko na mesto računske točke (1, 1). V naslednji iteraciji jedro pomaknemo za stolpec desno v računsko točko (1, 2), ponovimo računski postopek in dobimo rezultat 450, ki ga zapišemo v točko (1, 2) rezultirajoče slike. Prikazani postopek ponavljamo po celi sliki (*for* zanke niso prikazane).

```
Primer konvolucije
(SA)                    (J)                 SA * J
[ 50 100 100 100 100    0.25 0.5 0.25     0   0   0   0   0]?
[100 200 100 100 100    0.50 1.0 0.50     0 500 450   0   0]
[150 100 100 100 100  * 0.25 0.5 0.25  =  0   0   0   0   0]
[100 100 100 100 100                      0   0   0   0   0]
[100 100 100 100 100                      0   0   0   0   0]
Računski postopek v točki SA(1, 1)
 50 · 0.25=12.5
100 · 0.50=50
100 · 0.25=25
100 · 0.50=50
200 · 1.00=200
100 · 0.50=50
150 · 0.25=37.5
100 · 0.50=50
100 · 0.25=25
-------------
         + =500
Računski postopek v točki (1, 2)
100 · 0.25=25
100 · 0.50=50
100 · 0.25=25
200 · 0.50=100
100 · 1.00=100
100 · 0.50=50
100 · 0.25=25
100 · 0.50=50
100 · 0.25=25
-------------
         + =450
```

Konvolucija je izredno uporabna pri obdelavi slik. Z njo filtriramo, odvajamo, iščemo objekte itd. Efekti, ki jih dosegamo, so odvisni od vrednosti elementov v konvolucijskem jedru. Še preden si ogledamo efekte in primere jeder, moramo opozoriti na dve pomembni zadevi.

(i) V prej prikazanam primeru konvolucije je bil rezultat računskih operacij 500 in 450. Takšnih vrednosti ne moremo zapisati v 8-bitno spremenljivko. V nadaljevanju bomo pokazali, da je računski rezultat lahko tudi negativen. Tudi konvolucijska jedra imajo tipično vrednosti s plavajočo vejico tipa `float` ali `double`. Iz teh razlogov se računanje in shranjevanje rezultata izvaja z ustreznimi podatkovnimi tipi `float` (CV_32F), `double` (CV_64F) ali s 16- bitnimi celimi števili s predznakom (CV_16S), če so v jedro zapisana cela števila.

(ii) Kadar računsko jedro poravnamo z robovi slike, rezultat pa zapišemo na mesto, določeno s centrom jedra, potem v rezultirajoči sliki robne vrstice in stolpci ostanejo neizračunani (glej prejšnji primer, prva vrstica SA * J označena z ?). Problem se rešuje tako, da računski algoritem sliko SA navidezno poveča, tako da jo obrobi z dodatnimi stolpci in vrsticami. Število dodanih vrstic je odvisno od velikosti jedra. Za jedro velikosti 3 x 3 je dovolj ena vrstica, v splošnem pa je potrebno dodati navzdol zaokroženo (velikost jedra / 2). Dodajanje obrobe se v OpenCV nastavi s parametrom `borderType`. Njegove nastavitve omogočajo izbiro vrednosti, ki bodo zapisane v obrobo. Če izberemo npr. opcijo BORDER_REPLICATE, potem se robni slikovni elementi nadaljujejo v dodanih vrsticah in stolpcih (primer: aaaaaa|abcdefgh|hhhhhhh).

Sedaj si oglejmo primere jeder in efekte, ki jih z njimi dosegamo. Najprej si poglejmo najbolj enostavno jedro, kjer povprečimo vrednosti v okolici računske točke (**zameglitev slike**). Pri povprečenju seštejemo vse vrednosti in delimo s številom seštetih elementov. Vsi elementi konvolucijskega jedra bodo zato imeli enako utež. Podobno, kot je prikazano z enačbo (4.2), bomo najprej izvedli deljenje in šele potem seštevanje. Deljenje realiziramo že v sklopu jedra, tako da vsak element delimo s številom elementov npr. $1/9 = 0.111$, seštevanje pa je del konvolucijskega računskega postopka. Deljenje izvedemo samo enkrat, in to ob izračunu jedra, še preden se začnemo z zankami *for* premikati po sliki.

Primer filtra v nadaljevanju prikazuje takšno jedro J velikosti 3 x 3 in sliko SA, na kateri bo izvedena konvolucija. Za nazorno predstavitev učinkov jedra povprečenja je levi del slike povsem črne barve (0), desni povsem bele (255), meja med črno in belo barvo pa predstavlja oster rob. Na črnem ozadju je še en povsem bel slikovni element na lokaciji SA(2,2), ki demonstrira šum. Za konvolucijo uporabimo funkcijo OpenCV `filter2D`, v kateri nastavimo `borderType` in podatkovni tip izhodne slike. Slika SB prikazuje rezultat konvolucije. Vidimo, da je oster prehod med 0 in 255 zglajen; rabimo tri pomike, da z 0 pridemo na 255 (tj. 0–85, 85–170, 170–255). Podobno je tudi šum zglajen na območje slikovnih elementov 3 x 3, katerih vrednosti so približno 1/9 vrednosti izvornega šuma. Zaključimo lahko, da jedro velikosti 3 x 3 na sliki povzroča efekte v domeni svoje velikosti, tj. 3 x 3. Če bi bilo jedro večje, bi bil obseg efektov večji. Z velikostjo jeder ni smisla pretiravati, saj se število izračunov izredno poveča z naraščanjem velikosti jedra, kar upočasni obdelavo slike.

```
Primer filtra
(J) Povprečenje
[1  1  1]        [0.11 0.11 0.11]
[1  1  1] / 9 = [0.11 0.11 0.11]
[1  1  1]        [0.11 0.11 0.11]
(SA) Vhodna slika
[  0   0   0   0   0   0 255 255 255 255]
[  0   0   0   0   0   0 255 255 255 255]
[  0   0 255   0   0   0 255 255 255 255]
[  0   0   0   0   0   0 255 255 255 255]
[  0   0   0   0   0   0 255 255 255 255]
Uporabimo OpenCV funkcijo
SB=cv.filter2D(SA, cv.CV_16S, J, borderType=cv.BORDER_REPLICATE)
(SB) Rezultat konvolucije
```

```
[  0   0   0   0   0  85 170 255 255 255]
[  0  28  28  28   0  85 170 255 255 255]
[  0  28  28  28   0  85 170 255 255 255]
[  0  28  28  28   0  85 170 255 255 255]
[  0   0   0   0   0  85 170 255 255 255]
```

Na tem mestu omenimo še izjeme, pri katerih ne uporabimo predhodno opisanega postopka množenja in seštevanja sovpadajočih elementov v računski točki. Premikanje jedra po sliki in vse ostalo je enako predhodno opisanemu, le da so računske opracije drugačne. Primer izjeme je izračun `mediane`, kjer slikovne elemente, ki sovpadajo z jedrom, uredimo po velikosti od najmanjšega do največjega, v rezultirajočo sliko pa zapišemo srednjega s seznama. Primer izjeme je tudi zgolj iskanje največje ali najmanjše vrednosti slikovnih elementov, ki sovpadajo z jedrom. Za te izjeme imamo v OpenCV posebne funkcije, kot je npr. `medianBlur`.

Zanimiva funkcija je `GaussianBlur`, ki uporablja konvolucijsko jedro, v katerega je zapisana Gaussova funkcija (en. 4.3)[6]. Posebnost tega jedra je v tem, da **imajo slikovni elementi dlje od računske točke bistveno manjši vpliv od elementov v sredini jedra**. S tem pri filtriranju za zmanjšanje šuma **bolje ohranimo ostre robove**, kot če bi uporabili filter povprečja ali mediano.

$$J(m,n)|_{(v,u)} = \frac{1}{\sigma_u \sigma_v \sqrt{2\pi}} e^{-\frac{1}{2}\left[\frac{(u+m)^2}{\sigma_u^2} + \frac{(v+n)^2}{\sigma_v^2}\right]} \tag{4.3}$$

Primer Gaussovega jedra za filtriranje slike v nadaljevanju kaže, kako izračunamo Gaussovo jedro 3 x 3 (J) s $\sigma_u = \sigma_v = 1.0$ ter rezultat konvolucije SB = SA * J. Če primerjamo rezultate konvolucije med jedrom povprečenja in Gaussovim jedrom, vidimo, da je pri Gaussovem jedru rob med temnim in svetlim malenkost bolj oster (povprečenje 0 85 170 255, Gauss 0 70 185 255), kar potrjuje trditev, da se ostri robovi bolje ohranijo. Efekt je očitnejši pri večjih jedrih. Podobno je s šumom, pri povprečenju ima polje 3 x 3 enake vrednosti, pri Gaussu pa izkazuje Gaussu podobno obliko.

```
Primer Gaussovega jedra za filtriranje slike
u = np.linspace(-1, 1, 3)
v = np.linspace(-1, 1, 3)
uu, vv = np.meshgrid(u, v)
su=1.0
sv=1.0
J=1/(pow((2*np.pi),0.5)*su*sv)*np.exp(-0.5*(pow(uu,2)/pow(su,2)
  +pow(vv,2)/pow(sv,2)))
(J) gaussovo jedro
[0.1467 0.2419 0.1467]
[0.2419 0.3989 0.2419]
[0.1467 0.2419 0.1467]
SA * J
[  0   0   0   0   0  70 185 255 255 255]
[  0  19  32  19   0  70 185 255 255 255]
[  0  32  52  32   0  70 185 255 255 255]
[  0  19  32  19   0  70 185 255 255 255]
[  0   0   0   0   0  70 185 255 255 255]
```

Izračun gradientov

Gradient potrebujemo predvsem pri iskanju robov in prepoznavi objektov. Za določitev vrednosti elementov v jedru izhajamo iz splošne definicije odvoda. Enačbo za odvod v x smeri zapišemo kot

[6] računska točka (v, u), indeksa -r<m<r, -r<n<r, r predstavlja krak jedra, tj. navzdol zaokroženo (velikost jedra/2). σ_u in σ_v sta parametra normalne (Gaussove) porazdelitve

$$f'(x) = \lim_{\mathrm{h}\to 0} \frac{f(x+h) - f(x)}{\mathrm{h}}. \tag{4.4}$$

V enačbi nastopa razdalja h med $x + h$ in x, ki je enaka razdalji med sosednjima slikovnima elementoma. Zapis enačbe 4.4 v diferenčni obliki za sliko I(v, u) ob upoštevanju koraka med slikovnimi elementi h = 1, da enačbe

$$\begin{aligned}
\frac{\partial I}{\partial u} &= I(\mathrm{v,u+1}) - I(\mathrm{v,u}),\\
\frac{\partial I}{\partial v} &= I(\mathrm{v+1,u}) - I(\mathrm{v,u}),\\
\frac{\partial^2 I}{\partial u^2} &= I(\mathrm{v,u+1}) - 2I(\mathrm{v,u}) + I(\mathrm{v,u\text{-}1}),\\
\frac{\partial^2 I}{\partial v^2} &= I(\mathrm{v+1,u}) - 2I(\mathrm{v,u}) + I(\mathrm{v\text{-}1,u}).
\end{aligned} \tag{4.5}$$

Iz teh lahko razvijemo enostavna jedra za prvi [–1 1] ter drugi odvod [1 –2 1] v u- ali v v-smeri. Po podobnem principu lahko razvijemo jedra za višje odvode, gradient in Laplaceov operator. Z repliciranjem vrstic jedro razširimo še v v-smeri in s tem izboljšamo lokalno povprečje. Primer takšnih razširjenih jeder so Sobelova jedra za odvajanje. Primer odvoda v nadaljevanju prikazuje Sobelovo jedro za prvi odvod slike v horizontalni smeri in rezultat konvolucije SB = SA * J. Jedro za odvod v vertikalni v smeri je podobno, le da je zavrteno za 90°. Odvod slike lahko naredimo tudi z jedrom, v katerega je zapisan odvod Gaussove funkcije v u- ali v-smeri.

Če pogledamo rezultat odvajanja SB, vidimo, da dobimo pozitivne vrednosti 1020 odvoda na robu, tj. prehodu temnega na svetlo področje (+odvod). Če bi bil prehod iz svetlega na temno, potem bi dobili negativne vrednosti (–odvod). Nekaj podobnega je vidno na primeru šuma. Odvod je nič pri konstantnih vrednostih. Vidimo tudi, da šum samo moti iskanje robov v nadaljevanju, zato ga moramo pobrisati, ob tem pa ohraniti robove v čimbolj izvorni obliki. Močno orodje za ta namen so morfološke operacije, ki si jih bomo ogledali v nadaljevanju.

```
Primer odvoda
(J) Sobelovo jedro za odvod v horizontalni u smeri
[-1   0   1]
[-2   0   2]
[-1   0   1]
(SB) Rezultat konvolucije SA * J
[   0    0    0    0    0 1020 1020    0    0    0]
[   0  255    0 -255    0 1020 1020    0    0    0]
[   0  510    0 -510    0 1020 1020    0    0    0]
[   0  255    0 -255    0 1020 1020    0    0    0]
[   0    0    0    0    0 1020 1020    0    0    0]
Primer jeder za prepoznavo robov:
[0   1   0]  [-1 -1 -1]
[1  -4   1]  [-1   8 -1]
[0   1   0]  [-1 -1 -1]
```

Iskanje objektov

Konvolucijo uporabimo tudi za iskanje objektov na sliki. V jedro zapišemo iskano obliko. Pri konvoluciji bomo dobili največji odziv na mestu, kjer objekt, zapisan v jedro, sovpada z iskanim objektom na sliki. Primer iskanja objektov v nadaljevanju prikazuje izvorno sliko SA s tremi objekti – vertikalno črto na mestu $SA[2:5,1] = 50$, šum $SA[1,4] = 100$ in horizontalno črto $SA[3,6:9] = 70$. Najprej naredimo jedro, v katerega zapišemo obliko vertikalne črte. Po

konvoluciji SA * J dobimo največji odziv 150 ravno na sredini vertikalne črte na sliki SA. Na vseh drugih objektih je odziv manjši. Primer 5 v nadaljevanju kaže podoben primer: tokrat izdelamo jedro, v katerega zapišemo horizontalno obliko. Po konvoluciji dobimo največji odziv 210 točno na sredini horizontalnega objekta. V naslednjem koraku iskanja objektov moramo na sliki poiskati lego slikovnega elementa z največjo vrednostjo. Za ta namen lahko uporabimo funkcijo npr. `minMaxLoc`.

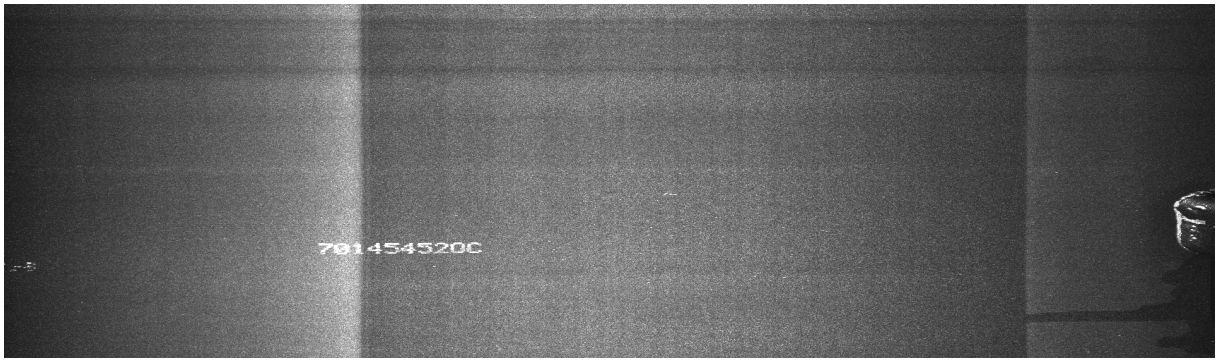
V OpenCV imamo za iskanje objektov tudi posebno funkcijo `matchTemplate`. Za jedro (šablona, ang. template) lahko uporabimo delček slike iskanega objekta (npr. določena črka, vijak, kovanec, delček orodja). V tem primeru so jedra bistveno večja od 3 x 3, zato moramo biti previdni, da ne pretiravamo z velikostjo, saj gre za računsko zahtevne operacije.

```
Primer iskanja objektov
(SA) Izvorna slika s tremi objekti
[  0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0 100   0   0   0   0   0]
[  0  50   0   0   0   0   0   0   0   0]
[  0  50   0   0   0   0  70  70  70   0]
[  0  50   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0]

(J)Jedro za iskanje vertikalnega objekta
[0 1 0]
[0 1 0]
[0 1 0]
(SB) Rezultat konvolucije SA * J
[  0   0   0   0 100   0   0   0   0   0]
[  0  50   0   0 100   0   0   0   0   0]
[  0 100   0   0 100   0  70  70  70   0]
[  0 150   0   0   0   0  70  70  70   0]
[  0 100   0   0   0   0  70  70  70   0]
[  0  50   0   0   0   0   0   0   0   0]

(J)Jedro za iskanje horizontalnega objekta
[0 0 0]
[1 1 1]
[0 0 0]
(SB) Rezultat konvolucije SA * J
[  0   0   0   0   0   0   0   0   0   0]
[  0   0   0 100 100 100   0   0   0   0]
[ 50  50  50   0   0   0   0   0   0   0]
[ 50  50  50   0   0  70 140 210 140  70]
[ 50  50  50   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0]
V Pythonu bi v nadaljevanju uporabili funkcijo za iskanje maksimuma
min_val, max_val, min_loc, max_loc = cv.minMaxLoc(SB)
```

Primer iskanja objektov v praksi – ujemanje šablon je prikazan na slikah 4.7 do 4.10 ter izpisu kode v nadaljevanju. Slika 4.7 prikazuje primer, kjer v proizvodnji slikamo površino cevi in iščemo napake barvanja, kot so vodni pocedki, praske, koaguliran lak itd. Na ceveh je vgravirana tudi koda za potrebe sledljivosti. Koda v resnici moti iskanje napak, zato jo želimo v fazi predobdelave odstraniti s slike. Za potrebe demonstracije sliko "DMK_33GX174_[K3].bmp" najprej preberemo z diska. Nato naredimo šablono, tako da z ene od slik izrežemo samo kodo. To naredimo z ukazom `T = SA[370:400,500:770]`; rezultirajoča šablona je prikazana na sliki 4.8. Sledi iskanje **ujemanja šablone s sliko** z ukazom `K=cv.matchTemplate(SA,T, cv.TM_CCOEFF_NORMED)`. Algoritem funkcije "matchTemplate" je konvolucija – s to razliko, da uporabimo večje jedro (šablono), funkcija pa je bolj uporabi prijazno poimenovana.

Rezultat ujemanja (tj. konvolucija) je prikazan na sliki 4.9. Na mestu začetka kode opazimo majhno svetlo piko, kjer je rezultat konvolucije med šablono in sliko največji. V nadaljevanju poiščemo lokacijo najsvetlejšega mesta. V ta namen lahko uporabimo funkcijo `minMaxLoc`, s katero poiščemo najsvetlejšo točko. Če pa je možnih več lokacij, na primer če bi prepoznavali posamezne številke, potem poiščemo več možnih vrhov. V ta namen najprej določimo mejni prag svetlosti zadetkov (npr. prag = 0.8), nato pa jih (v Pythonu) poiščemo s funkcijo `loc=np.where(K >= prag)`. V nadaljevanju lahko kodo pobrišemo (slika 4.10b) ali pa jo samo označimo s kvadratom (slika 4.10a). Za brisanje kode smo na mesto kode skopirali del slike iznad kode (ukaz `SA[pt[1]:(pt[1]+h),pt[0]:(pt[0]+w)]=SA[(pt[1]-h):(pt[1]),pt[0]:(pt[0]+w)]`). Če želimo prepoznati posamezne številke v kodi, potem bi morali za vsako številko od 0 do 9 izdelati svojo šablono in ponoviti predhodno opisan postopek za vsako številko posebej.

Če je slika SA dimenzij (C, R) in T (w, h), potem je rezultirajoča slika konvolucije K dimenzij (C - w + 1, R - h + 1). To v praksi pomeni, da algoritem poravna T z levim zgornjim robom SA, nato pa ga pomika vzdolž stolpcev, dokler se T ne dotakne desnega roba slike. To se ponavlja po vrsticah, dokler se T ne dotakne spodnjega robu SA. Posledično je slika K manjša od SA za velikost šablone (odrezana na desni in spodaj), kar je bistvena razlika glede na konvolucijo, pri kateri se dimenzije slike povečajo z replikacijo robov do te mere, da je izhodna slika enaka vhodni. Nastavitev tretjega parametra `cv.TM_CCOEFF_NORMED` določa, kako se bodo izračunavale vrednosti K (glej OpenCV TemplateMatchModes).

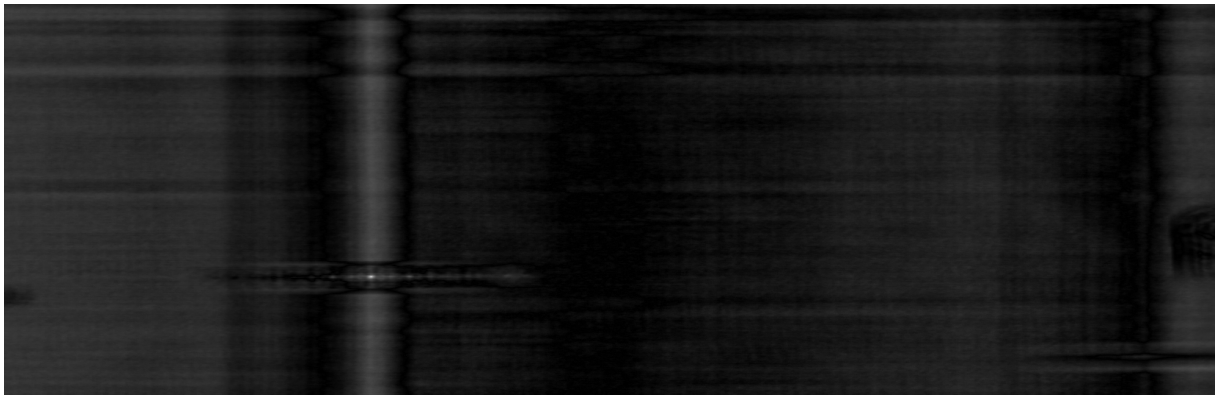


Slika 4.7: Slika SA z vgravirano kodo izdelka. Koda moti nadaljnjo obdelavo slike, zato jo želimo odstraniti

```
import numpy as np
import cv2 as cv
#Preberemo sliko iz diska.  Preberemo kot sivinsko sliko.
SA = cv.imread('DMK_33GX174_[K3].bmp', cv.COLOR_BGR2GRAY)
dimenzije = SA.shape
print(dimenzije)

T=SA[370:400,500:770] # cela koda
# T=SA[370:400,500:520] samo številka 7
w, h = T.shape[::-1] # Dimenzije šablone

# Iskanje ujamenja = konvolucija
K = cv.matchTemplate(SA, T, cv.TM_CCOEFF_NORMED)

prag = 0.8  # Prag iskanja maksimuma
# Poiščemo lokacije, kjer je konvolucija večja od praga
loc = np.where(K >= prag)

# Narišemo kvadrat okrog najdene kode
for pt in zip(*loc[::-1]):
```

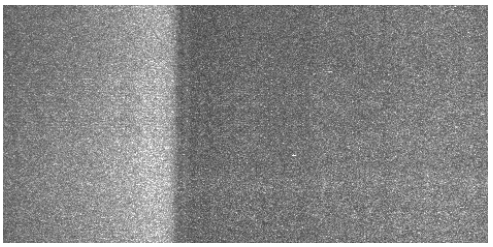
```
#narišemo kvadrat okrog kode
cv.rectangle(SA, pt, (pt[0] + w, pt[1] + h), (0, 255, 255), 2)
#pobrišemo kodo tj.  na njo skopiramo del slike brez kode
SA[pt[1]:(pt[1]+h),pt[0]:(pt[0]+w)]=SA[(pt[1]-h):(pt[1]),pt[0]:(pt[0]+w)]

cv.imshow("SA", SA)
cv.imshow("T", T)
cv.imshow("Konvolucija", K)
#Sliko K tipa float32 pretvorimo v uint8 sliko
K8=np.uint8(cv.convertScaleAbs(K*255))
#Shranimo slike
cv.imwrite("templatematch.bmp", K8)
cv.imwrite("templatematch_SA.bmp", SA)
cv.imwrite("templatematch_T.bmp", T)
```

Slika 4.8: Šablona (T) za iskanje kode



Slika 4.9: Rezultat konvolucije (K)

(a) Najdena koda



(b) Pobrisana koda

Slika 4.10: Prikaz detajla slike SA z označeno in s pobrisano kodo

Kot zanimivost omenimo, da gre sodobni trend obdelave slik pri različnih nalogah prepoznave objektov in iskanja anomalij v smeri uporabe konvolucijskih nevronski mrež (ang. convolutional neural networks (CNN)). Pri teh je osnovni gradnik predhodno opisani konvolucijski postopek. Tovrstne mreže imajo mnogo zaporednih konvolucijskih slojev. Delujejo tako, da sloji blizu vhoda prepoznavajo elementarne gradnike – podobno, kot smo pokazali na primeru iskanja vertikalnih in horizontalnih črtic, globlji sloji pa na rezultatih vhodnih konvolucij prepoznavajo bolj abstrakte značilke. Sočasno se izvaja tudi zmanjševanje dimenzij slike – do te mere, da lahko abstraktne značilke vstavimo v izhodno klasifikacijsko nevronsko mrežo, ki dokončno poda

verjetnost objekta na sliki [7]. Izvedbe in učenje CNN so predmet raziskav in uporabe na področjih umetne inteligence.

Vprašanja

- Opišite pomen konvolucije v kontekstu obdelave slik in kako se izvaja v OpenCV.
- Kakšen je pomen jedra (kernel) pri konvolucijskih operacijah in kako izbiramo njegovo velikost in obliko?
- Kakšen učinek imajo različna jedra na sliko? Diskutirajte o jedrih za zamegljevanje in ostrenje slike.
- Kako se pri izvajanju konvolucije soočamo s problemom izračuna na robovih slike?
- Kakšne metode in algoritme uporabljamo v OpenCV za iskanje objektov v slikah?
- Kako je v rezultatu konvolucije vidna lega največjega ujemanja med jedrom in vhodno sliko?
- Opišite praktični primer, kjer bi uporabili iskanje objektov v kontroli kakovosti in avtomatizaciji.
- Katere so glavne težave in omejitve pri iskanju objektov v slikah?
- Kakšne so strategije za optimizacijo hitrosti in učinkovitosti konvolucijskih operacij?*

[7] https://keras.io/api/applications/vgg

Morfološke operacije

Morfološke operacije so navidezno podobne konvoluciji, saj uporabljajo jedro (pogosto poimenovanje je tudi strukturni element), s katerim se na enak način pomikamo po sliki (običajno maski). Razlika do konvolucije je v tem, da ne opravljajo računskih operacij kot pri konvoluciji, temveč se izvajajo različni testi in logične operacije med jedrom in sovpadajočimi slikovnimi elementi v trenutni računski točki. Gre za izredno uporabno orodje za odstranjevanje šuma, ločevanje objektov od ozadja, povečevanje ali zmanjševanje objektov ter zapiranje ali odpiranje lukenj v maskah. Načeloma se izvajajo na maskah in se prilagajajo glede na potrebe v obdelavi slik. V nadaljevanju sledi predstavitev osnovnih morfoloških operacij.

Erozija (krčenje): Erozija je operacija, ki zmanjša (erodira) svetla območja na maski. Slikovni element v trenutni računski točki bo v izhodni maski imel vrednost, ki bo večja od nič (> 0), če so vsi slikovni elementi, ki jih pokriva jedro, večji od nič (> 0). V nasprotnem primeru se v izhodno masko zapiše vrednost 0. Gre za nekaj podobnega, kot če bi uporabili logično funkcijo IN. V jedru se upoštevajo samo elementi jedra, ki imajo vrednost > 0. V vhodni maski so lahko poljubne vrednosti med 0 in 255. V izhodno masko se prenese najmanjši slikovni element, ki ga pokriva jedro v trenutni računski točki. Primer erozije v nadaljevanju to nazorno prikazuje:

```
Primer erozije
(SA) Izvorna maska
[  0   0   0 150   0   0   0   0   0   0]
[  0   0  50 150  50  50   0   0   0   0]
[  0   0  50 150  50  50   0 100   0   0]
[  0   0  50 150  50  50   0   0   0   0]
[  0   0  50 150  50  50   0   0   0   0]
[  0   0   0 150   0   0   0   0   0   0]
(J) 3 x 3 polno jedro se pogosto imenuje jedro z "8 sosedi"
[1 1 1]
[1 1 1]
[1 1 1]
SB=cv.erode(SA,J,iterations = 1)
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0 50 50  0  0  0  0  0]
[ 0  0  0 50 50  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]
(J) 3 x 3 s 4 sosedi
[0 1 0]
[1 1 1]
[0 1 0]
SB=cv.erode(SA,J,iterations = 1)
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0 50  0  0  0  0  0  0]
[ 0  0  0 50 50  0  0  0  0  0]
[ 0  0  0 50 50  0  0  0  0  0]
[ 0  0  0 50  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
(J) Poljubno jedro
[1 1 1]
SB=cv.erode(SA,J,iterations = 1)
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0 50 50  0  0  0  0  0]
[ 0  0  0 50 50  0  0  0  0  0]
[ 0  0  0 50 50  0  0  0  0  0]
[ 0  0  0 50 50  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
```

Razširjanje (dilatacija): Razširjanje je operacija, ki poveča svetla območja na maski. Slikovni element izhodne maske je večji od 0, če je vsaj en z jedrom sovpadajoči element večji od 0; gre za nekaj podobnega, kot če bi uporabili logično funkcijo ALI. Uporablja se za povečanje objektov in zapolnitev majhnih lukenj v objektih. Uporablja se tudi po operaciji erozije, kadar odstranjujemo šum. Z erozijo odstranimo šum, sočasno pa tudi zmanjšamo objekt. Z razširjanjem vrnemo objekt nazaj v njegovo izvorno velikost. Uporabimo lahko različne strukturne elemente, na primer kvadrat ali križ. Primer razširjanja je v nadaljevanju:

```
Primer razširjanja
(SA) Izvorna maska
[  0   0   0   0   0   0   0   0   0   0]
[  0   0  50  50  50  50   0   0   0   0]
[  0   0  50   0   0  50   0 100   0   0]
[  0   0  50   0   0  50   0   0   0   0]
[  0   0  50  50  50  50   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0]
(J) jedro z 8 sosedi
[1 1 1]
[1 1 1]
[1 1 1]
SB=cv.dilate(SA,J,iterations = 1)
[  0  50  50  50  50  50  50   0   0   0]
[  0  50  50  50  50  50 100 100 100   0]
[  0  50  50  50  50  50 100 100 100   0]
[  0  50  50  50  50  50 100 100 100   0]
[  0  50  50  50  50  50  50   0   0   0]
[  0  50  50  50  50  50  50   0   0   0]
(J) jedro s 4 sosedi
[0 1 0]
[1 1 1]
[0 1 0]
SB=cv.dilate(SA,J,iterations = 1)
[  0   0  50  50  50  50   0   0   0   0]
[  0  50  50  50  50  50  50 100   0   0]
[  0  50  50  50  50  50 100 100 100   0]
[  0  50  50  50  50  50  50 100   0   0]
[  0  50  50  50  50  50  50   0   0   0]
[  0   0  50  50  50  50   0   0   0   0]
```

Opiranje: Opiranje je kombinacija operacij erozije in razširjanja. Najprej se izvede erozija, da se odstrani majhen šum, nato pa se izvede razširjanje, da se objekti vrnejo v prvotno stanje. Uporablja se za čiščenje šuma.

Zapiranje: Zapiranje je kombinacija operacij razširjanja in erozije. Najprej se izvede razširjanje, da se povečajo objekti in zapolnijo majhne vrzeli v objektih, nato pa erozija, da se zunanji rob objektov vrne v prvotno stanje. Uporablja se za čiščenje luknjic v maskah. Primer v nadaljevanju nazorno prikazuje obe operaciji:

```
Primer odpiranja in zapiranja
(SA) Izvorna maska
[  0   0   0   0   0   0   0   0   0   0]
[  0   0  50  50  50  50   0   0   0   0]
[  0   0  50   0  50  50   0 100   0   0]
[  0   0  50  50  50  50   0   0   0   0]
[  0   0  50  50  50  50   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0]
(J)
[0 1 0]
[1 1 1]
[0 1 0]
Zapiranje
```

```
SB=cv.morphologyEx(SA, cv.MORPH_CLOSE,J)
[  0   0   0  50  50   0   0   0   0   0]
[  0   0  50  50  50  50   0   0   0   0]
[  0   0  50  50  50  50  50 100   0   0]
[  0   0  50  50  50  50  50   0   0   0]
[  0   0  50  50  50  50   0   0   0   0]
[  0   0   0  50  50   0   0   0   0   0]

Odpiranje
SB=cv.morphologyEx(SA, cv.MORPH_OPEN,J)
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0 50  0  0  0  0  0]
[ 0  0  0 50 50 50  0  0  0  0]
[ 0  0  0  0 50  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
```

Gradient: Gradient je razlika med razširjanjem in erozijo. Rezultat prikazuje spremembe v intenziteti in se uporablja za zaznavo robov objektov.

Top-hat in bottom-hat: Top-hat operacija je razlika med izvorno sliko in odpiranjem slike, medtem ko je bottom-hat razlika med zapiranjem slike in izvorno sliko. Ti operaciji poudarita majhne strukture in objekte na sliki.

Vprašanja

- Kaj so morfološke operacije in kako se uporabljajo pri obdelavi slik?
- Kako in zakaj uporabljamo operaciji razširjanja in erozije?
- Opišite učinek sestavljenih operacij odpiranja in zapiranja.
- Kako izberemo ali definiramo strukturni element za morfološke operacije?

4.6 Transformacije slik

4.6.1 Afine transformacije

Afine transformacije, kot so **rotacija, skaliranje, translacija in sprememba ločljivosti**, so ključne za prilagajanje slik pri raznih analizah, umeritvi in vizualizacijah. Njihova značilnost je, da ohranjajo vzporednost premic, robov itd. Translacijo (slika 4.11) izvedemo s funkcijo OpenCV `cv2.warpAffine()`, s tem da podamo transformacijsko matriko:

```
R,C = slika.shape[:2]
# Podamo transformacijsko matriko v obliki
# |1 0 tu|
# |0 1 tv|
# Premik za 50 slikovnih elementov desno in 100 dol
M = np.float32([[1, 0, 50], [0, 1, 100]])
premaknjena_slika = cv2.warpAffine(slika, M, (C,R)) #!  dsize = (width, height) =
    (C,R)
```

Transformacija se izvede po enačbi 4.6, kjer so v, u in v', u' koordinate slikovnih elementov pred transformacijo in po njej, θ kot rotacije, s skaliranje ter t_u in t_v translacija. Transformacija se izvede okrog koordinatnega izhodišča (0, 0). Če želimo izvršiti transformacijo, okrog katere druge točke na sliki, potem moramo podati center vrtenja c_u in c_v. Ta se najprej odšteje od v in u koordinate, s tem center vrtenja prestavimo v točko (0, 0), po izvršeni transformaciji pa koordinatama v' in u' nazaj prištejemo c_u in c_v. Funkcija `cv2.getRotationMatrix2D` izdela transformacijsko matriko, ki upošteva vse navedene parametre (glej primer sočasne rotacije in skaliranja v nadaljevanju).

$$\begin{bmatrix} u' \\ v' \end{bmatrix} = s \cdot \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} t_u \\ t_v \end{bmatrix} \tag{4.6}$$

Za rotacijo in sočasno skaliranje slike uporabimo rotacijsko matriko, ki jo dobimo s funkcijo `cv2.getRotationMatrix2D()`:

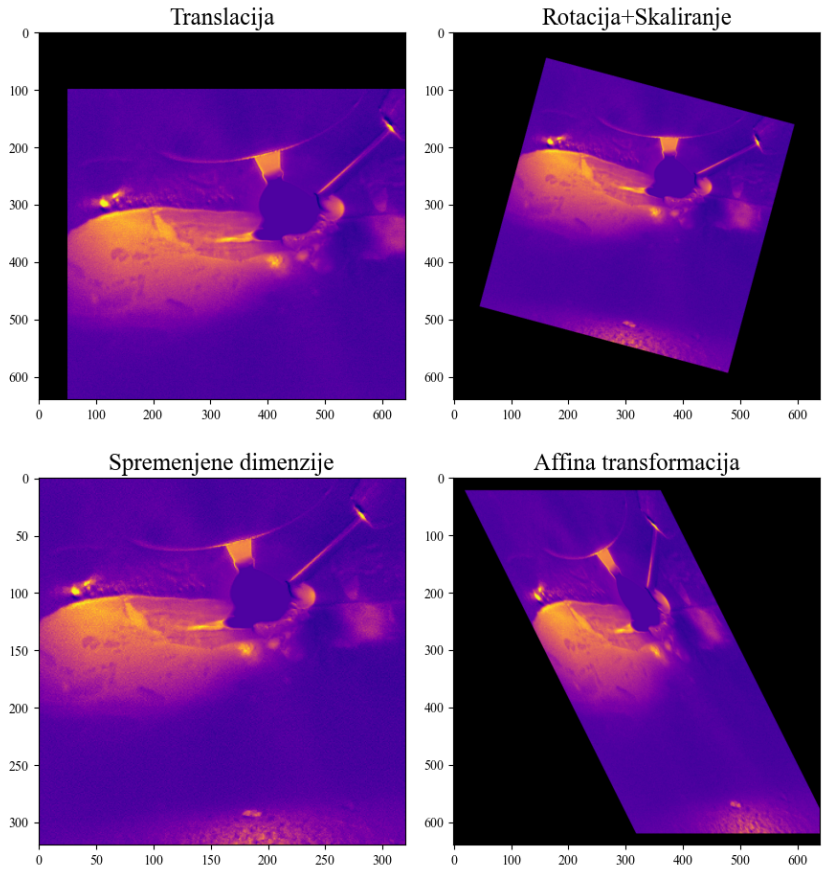
```
kot = -15  # Kot rotacije v stopinjah
center_rotacije = (C/2, R/2)  # Rotacija okoli sredine slike
skaliranje = 0.7
M = cv2.getRotationMatrix2D(center_rotacije, kot, skaliranje)
# (Cnov,Rnov)-izhodna dimenzija slike, običajno (C,R)
zavrtena_slika = cv2.warpAffine(slika, M, (Cnov,Rnov))
```

Za preprečitev izgube delov slike na mestih, kjer gre rotirana slika preko robov izvorne slike, uporabimo večjo dimenzijo (Cnov, Rnov) izhodne slike v funkciji `warpAffine`.

Pri delu z OpenCV in NumPy pogosto pride do zmede zaradi različnega vrstnega reda podajanja dimenzij slike. Funkcija `cv2.warpAffine` iz knjižnice OpenCV kot tretji parameter pričakuje dimenzije izhodne slike v obliki `(širina, višina)`, kar sledi običajni konvenciji prikaza slik, kjer os u poteka vodoravno, os v pa navpično. Po drugi strani pa knjižnica NumPy uporablja vrstni red indeksiranja `(vrstica, stolpec)`, kar pomeni `(višina, širina)` slike. To pomeni, da pri dostopu do slikovne točke z ukazom `slika[v, u]` dostopamo do slikovnega elementa v vrstici `v` in stolpcu `u`, kar je obratno od vrstnega reda, ki ga pričakujejo nekatere funkcije v OpenCV. Zaradi te razlike je pri določanju parametrov, kot je velikost slike v funkciji `warpAffine`, potrebna posebna previdnost, da ne pride do napak pri transformacijah ali izrezu slik.

OpenCV omogoča več metod spreminjanja dimenzij slike (C, R), pri tem pa uporablja različne interpolacije med slikovnimi elementi:

```
#Možne interpolacije
# cv2.INTER_NEAREST - najbližji sosed (hitro, nizka kakovost).
```



Slika 4.11: Primerjava transformacij slike

```
# cv2.INTER_LINEAR - linearna interpolacija (privzeta metoda).
# cv2.INTER_CUBIC - kubična interpolacija (bolj gladko).
# cv2.INTER_LANCZOS4 - najvišja kakovost.
#Dimenzije izvorne slike (C, R) povečamo za faktor skaliranja (C * fx,R * fy)
Povecana = cv2.resize(slika,None,fx=0.5,fy=0.5,interpolation=cv2.INTER_LANCZOS4)
#ali pa predpišemo željene dimenzije končne slike (lahko pride do popačitev).
Povecana = cv2.resize(slika,(320,320), interpolation=cv2.INTER_LINEAR)
```

Afina transformacija omogoča strižne transformacije (uporaba pri sestavljanju in poravnavi več slik). Sliko preslikamo z izhodiščnega koordinatnega sistema v nov koordinatni sistem, kjer koordinatne osi niso več medsebojno pravokotne. Pri tem je potrebno določiti tri točke pred transformacijo (tj. izvorne točke) in njihove ustrezne lege po transformaciji (ciljne točke). Na podlagi teh treh parov točk funkcija `cv2.getAffineTransform()` izračuna transformacijsko matriko velikosti 2×3, ki opisuje premik, vrtenje, skaliranje in striženje.

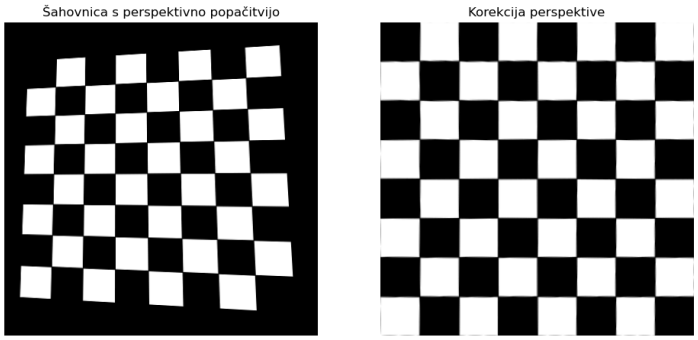
```
#Izvorne točke na originalni sliki
tizsl = np.float32([[50, 50], [200, 50], [50, 200]])
#Njihove lege po transformaciji
ttsl = np.float32([[70, 70], [150, 70], [140, 210]])
#Transformacijska matrika
M = cv2.getAffineTransform(tizsl, ttsl)
#Transformacija
At_slika = cv2.warpAffine(slika, M, (C,R))
```

4.6.2 Transformacija perspektive

Pri transformaciji perspektive tipično odpravljamo učinke perspektive (pogleda pod kotom). Transformacija ne ohranja vzporednosti robov oz. premic, kot to velja pri afini transformaciji, temveč omogoča, da se vzporedne premice sekajo, podobno kot je to pri prikazu globine in oddaljenosti v 3D-prostoru.

Za določitev transformacije perspektivne potrebujemo štiri točke z izvorne slike in njim pripadajoče lege v ciljni sliki. Funkcija `cv2.getPerspectiveTransform()` izračuna transformacijsko matriko velikosti 3×3, ki jo nato uporabimo v funkciji `cv2.warpPerspective()`. Primer uporabe na sliki 4.12 prikazuje korekcijo popačenja šahovnice, posnete pod kotom.

```
#Vogalne točke na popačeni sliki
vtper = np.float32([[56,65], [368,52], [28,387], [389,390]])
#Lega vogalnih točk po korekciji perspektive
vttobe = np.float32([[0,0], [300,0], [0,300], [300,300]])
#Izračun transformacijske matrike
M = cv2.getPerspectiveTransform(vtper, vttobe)
odpravljena_perspektiva = cv2.warpPerspective(slika, M, (300, 300))
```



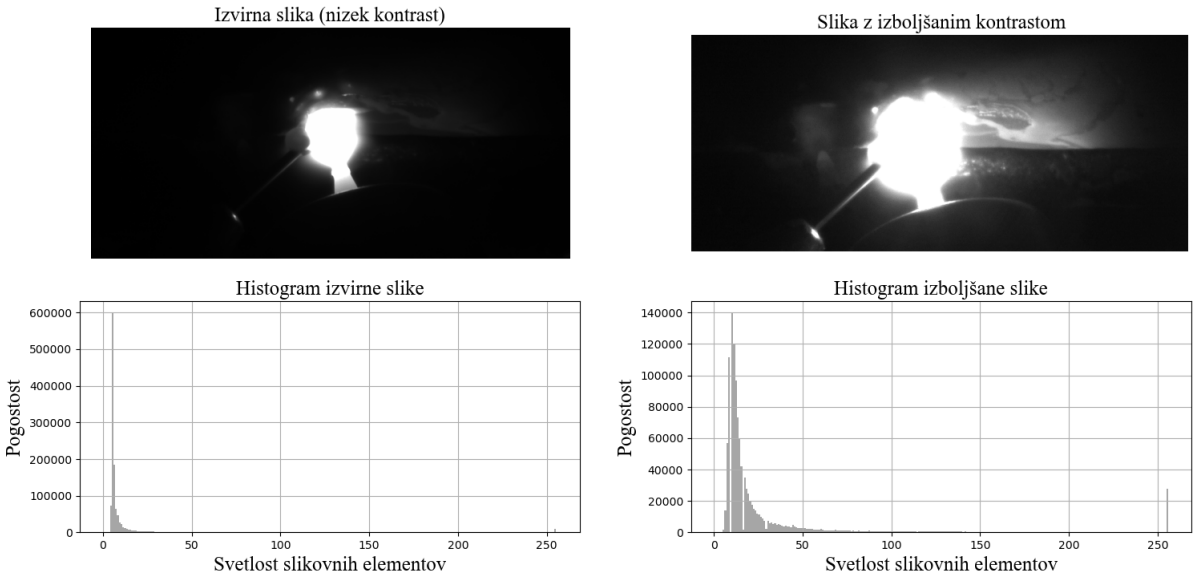
Slika 4.12: Transformacija perspektive, uporabljena pri korekciji pogleda na objekt pod kotom

4.6.3 Sprememba kontrasta z raztezanjem histograma

Sprememba kontrasta slike z raztezanjem histograma (ang. histogram stretching) se uporablja pri temnih ali preosvetljenih slikah, kot to prikazuje primer varjenja na sliki 4.13. Tu je oblok izredno svetel in preosvetli okolico, tako da to, kar nas zanima, talina, ni vidna. Izboljšanja kontrasta se lotimo tako, da najprej ugotovimo, v kakšnem območju se nahaja svetlost objekta (npr. taline), ki nas zanima. Pri tem si pomagamo s histogramom svetlosti izvorne slike. Izbrani razpon (med minimalno I_{min} in maksimalno I_{max} svetlostjo, npr. med 4 in 40) raztegnemo čez celotno razpoložljivo območje svetlosti med 0 in 255 (pri 8-bitnih slikah). Pri tem se uporabi enačba $I_{nova} = \frac{I - I_{min}}{I_{max} - I_{min}} \cdot 255$. Vrednosti, manjše od spodnje meje, postavimo na 0 in vrednosti, večje od zgornje meje, na 255 z uporabo funkcije `clip`. V knjižnici OpenCV je mogoče raztezanje histograma enostavno izvesti s funkcijama `cv2.normalize()` in `cv2.equalizeHist()`.

```
#Funkcija za raztezanje histograma

def histogram_raztez(slika, Imin, Imax):
# Vhodna slika mora biti tipa float32 (vrednosti v območju med 0 in 1)

# Opcijsko
# Poiščemo razpon histograma med minimalno (Imin) in maksimalno (Imax) svetlostjo
# Imin = np.min(slika)
# Imax = np.max(slika)
```

```
  #Modifikacija slike
  Slika_vec_kontrasta = ((slika - Imin) / (Imax - Imin)) * 255
  # Poskrbimo za vrednosti < 0 in > 255
  Slika_vec_kontrasta = np.clip(Slika_vec_kontrasta, 0, 255)
  return Slika_vec_kontrasta

#Uporaba predhodno definirane funkcije
#img_org = izvorna slika, nizek kontrast
slika = img_org.astype(np.float32) / 255    #(Vrednosti s.e.  med 0 in 1)
sm=0.015 #ugotovimo s pomočjo histograma img_org in s poskušanjem
#plt.hist(image.ravel(), bins=256, range=(0, 256), color='gray', alpha=0.7)
zm=0.15
slika_vec_kontrasta = histogram_raztez(slika,sm,zm)

#Alternativa - raztezanje histograma z OpenCV funkcijami
Slika_vec_kontrasta = cv2.normalize(slika, None, 20, 235, cv2.NORM_MINMAX)
#ali
Slika_vec_kontrasta = cv2.equalizeHist(slika)
```



Slika 4.13: Sprememba kontrasta z raztezanjem histograma

4.6.4 Fourierjeva transformacija

Hitra Fourierjeva transformacija (ang. fast Fourier transform (FFT)) (en. 4.7) temelji na prehodu iz prostorske domene (kjer obdelujemo slike kot slikovne elemente z intenzitetami) v frekvenčno domeno, kjer analiziramo fekvenčno vsebino. FFT razgradi sliko v osnovne sinusoidne komponente različnih frekvenc in faz, kar omogoča prepoznavanje vzorcev, simetrij in ponavljajočih se struktur. Fourierova transformacija slike S(v, u) (v = 1..R, u = 1..C) v diskretnem se v OpenCV[8] izračuna kot:

$$F(\mathrm{k},\,\mathrm{l}) = \sum_{v=0}^{R-1}\sum_{u=0}^{C-1} S(\mathrm{v},\,\mathrm{u}) \cdot e^{-i2\pi\left(\frac{kv}{R}+\frac{lu}{C}\right)}, \tag{4.7}$$

[8]glej: https://docs.opencv.org/4.x/d8/d01/tutorial_discrete_fourier_transform.html

pri čemer rezultat F(k, l) predstavlja frekvenčno domeno pri frekvenčnih indeksih l (v horizontalni) in k (v vertikalni smeri). Skupek frekvenčnih indeksov k in l se ukvarja s periodičnimi frekvenčnimi komponentami, ki sestavljajo sliko. F(k = 0, l = 0) predstavlja DC-komponento (ničelna frekvenca) oz. povprečno svetlost slike. Ko se k in l povečujeta, pripadajoče frekvence postajajo višje in predstavljajo hitre spremembe v svetlosti (npr. robove ali fine podrobnosti). F(k=R/2, l=C/2) ustreza najvišji možni frekvenci v obeh smereh. Za F(k, l) je značilna konjugirana simetrija, kar pomeni, da je vrednost F(k, l) zrcaljena čez središče frekvenčne domene, F(k, l) = F^*(–k, –l), kjer je F^* kompleksno konjugirana. Zaradi te lastnosti se pri analizi realnih slik običajno uporablja le polovica frekvenčnega spektra.

Za določitev konkretne frekvence iz indeksov k in l poglejmo primer, kjer predpostavimo, da imamo sliko z dimenzijami 256×256 slikovnih elementov, ki v realnem svetu prikazuje merilno območje velikosti $0.256\,\text{m} \times 0.256\,\text{m}$. Vzorčni intervali (razdalja med slikovnimi elementi) so $\Delta x = \Delta y = \frac{0.256\text{m}}{256} = 0{,}001\,\text{m}$. Za specifična frekvenčna indeksa, recimo $k = 32$ in $l = 64$, ter znano velikost slike (R, C) lahko izračunamo vodoravno frekvenco kot:

$$f_x(k) = \frac{k}{C \cdot \Delta x} = f_x(32) = \frac{32}{256 \cdot 0.001} = \frac{32}{0.256} \approx 125\,\text{nihajev/m} \tag{4.8}$$

in podobno vertikalno frekvenco:

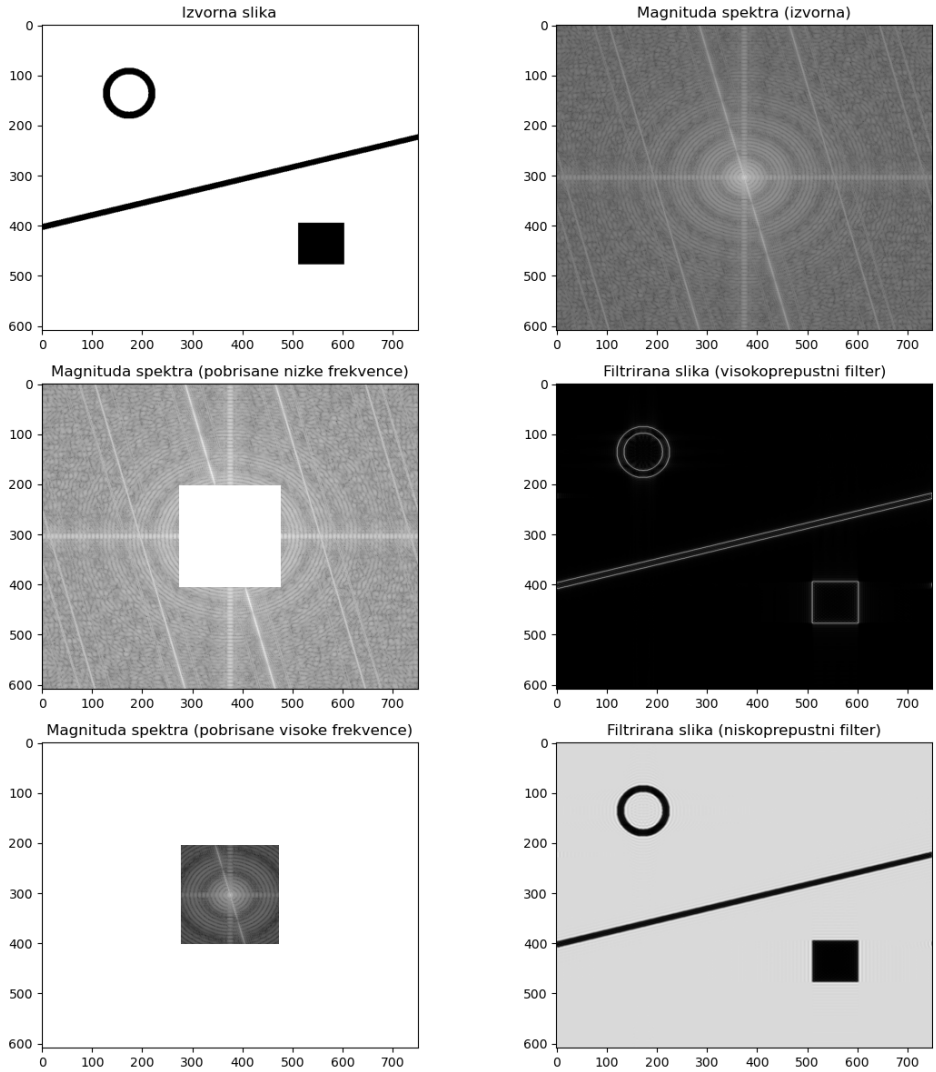
$$f_y(l) = \frac{l}{R \cdot \Delta y} = f_y(64) = \frac{64}{256 \cdot 0.001} = \frac{64}{0.256} \approx 250\,\text{nihajev/m}. \tag{4.9}$$

Frekvenčna komponenta F(32, 64) ustreza frekvenci 125 nihajev/m v vodoravni smeri in 250 nihajev/m v navpični smeri. Pod nihaj si predstavljamo svetlo-temno polje.

Vrnimo se k praktični uporabi. Če ponovimo, nizke frekvence (blizu centra spektra – glej sliko 4.14 "Magnituda spektra (izvorna)" predstavljajo počasi se spreminjajoče, enakomerno porazdeljene značilnosti slike. Visoke frekvence, ki naraščajo z oddaljevanjem od centra spektra, predstavljajo hitro spreminjajoče se značilnosti slike, kot so robovi in ostri prehodi. Vrednosti F(k, l) so kompleksna števila, ki jim lahko izračunamo absolutno velikost (magnitudo) in kot med realno in imaginarno komponento (faza). Magnituda prikazuje "jakost" vsake frekvence na sliki; višje vrednosti ustrezajo bolj izrazitim vzorcem (bolj svetlim točkam v magnitudi spektra). Faza določa prostorsko poravnavo oziroma zamik vzorcev (frekvenc) in je pomembna pri rekonstrukciji slike, saj brez faze slika izgubi svojo obliko. V večini postopkov se za obdelavo slik uporablja le manipulacija magnitud frekvenčnega spektra.

Koda v nadaljevanju kaže primer filtriranja slik. Najprej izvedemo FFT-transformacijo, s čimer pretvorimo sliko v frekvenčno domeno. Izvedemo lahko nizko- ali visokoprepustne filtre enostavno z brisanjem komponent s frekvenčnega spektra, tako da ga množimo z masko. Pri prepoznavanju vzorcev bi uporabili visokoprepustni filter, ki odstrani gladke dele slike (nizke frekvence) in poudari robove ter majhne podrobnosti (visoke frekvence). To naredimo tako, da pobrišemo center spektra, kot prikazuje slika "Magnituda spektra (pobrisane nizke frekvence)". V nasprotnem primeru, kadar želimo odstraniti šum in visoke frekvence, obdržimo center spektra, pobrišemo pa vse ostalo, kot prikazuje slika "Magnituda spektra (pobrisane visoke frekvence)". Jakost filtriranja izberemo z velikostjo pobrisanega območja (parameter r v kodi). Nato izvedemo inverzno Fourierjevo transformacijo, s čimer filtrirano sliko pretvorimo nazaj v prostorsko domeno.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
#a) Naložimo sliko in jo pretvorimo v sivinsko
#Večina frekvenčnih operacij se izvaja na slikah z eno barvno komponento,
```



Slika 4.14: Obdelava slike z uporabo FFT

```
#zato barvno sliko spremenimo v sivinsko.
slika = cv2.imread('testnaslika2.png', 0) # 0 pretvori v sivinsko sliko

#b) Izvedi diskretno FFT
#POZOR: Dimenzije slike (R,C) morajo biti n-kratnik števila 2 (osnovna predpostavka pri FFT)
#Če niso, je potrebno sliko obrezati na najbližji 2^n (ni prikazano)
dft = cv2.dft(np.float32(slika), flags=cv2.DFT_COMPLEX_OUTPUT)
# Premakni ničelno frekvenco v sredino spektra (upoštevamo konjugirano simetrijo)
dft_premik = np.fft.fftshift(dft)
#Izračun magnitude spektra.  Izhod FFT ima kompleksne vrednosti.
#Magnituda kompleksnega izhoda FFT je uporabna za vizualizacijo frekvenčne vsebine.
mag_spek = 20*np.log(cv2.magnitude(dft_premik[:,:,0],dft_premik[:,:,1]))

#c) Uporaba filtrov
#V frekvenčni domeni filtriramo sliko tako, da z nje z masko pobrišemo del spektra.
vrstice, stolpci = slika.shape
sredina_v, sredina_s = vrstice//2, stolpci//2
#c1) Ustvarimo masko visokoprepustnega filtra za iskanje robov na sliki
maskaHP = np.ones((vrstice,stolpci,2),np.uint8)
r = 100  # Polmer filtra
```

```
maskaHP[sredina_v-r:sredina_v+r, sredina_s-r:sredina_s+r] = 0
#c2) Ustvarimo masko nizkoprepustnega filtra za filtriranje slike
maskaLP = np.zeros((vrstice,stolpci,2),np.uint8)
maskaLP[sredina_v-r:sredina_v+r,sredina_s-r:sredina_s+r] = 1
#d1) Uporabi masko na premaknjenem DFT
f_premikHP = dft_premik*maskaHP
mag_spek_mHP=20*np.log(cv2.magnitude(f_premikHP[:,:,0],f_premikHP[:,:,1]))
#d2) Uporabi masko na premaknjenem DFT
f_premikLP = dft_premik*maskaLP
mag_spek_mLP = 20*np.log(cv2.magnitude(f_premikLP[:,:,0],f_premikLP[:,:,1]))

#e) Po spreminjanju frekvenčne vsebine spekter pretvorimo nazaj v slikovno domeno
f_obratni_premikHP = np.fft.ifftshift(f_premikHP)
slika_nazaj_HP = cv2.idft(f_obratni_premikHP) #inverzna DFT
slika_nazaj_HP = cv2.magnitude(slika_nazaj_HP[:,:,0],slika_nazaj_HP[:,:,1])
f_obratni_premikLP = np.fft.ifftshift(f_premikLP)
slika_nazaj_LP = cv2.idft(f_obratni_premikLP)
slika_nazaj_LP = cv2.magnitude(slika_nazaj_LP[:,:,0],slika_nazaj_LP[:,:,1])

#f) Izris
plt.figure(figsize=(12, 12))
plt.subplot(3, 2, 1)
plt.imshow(slika, cmap='gray')
plt.title('Izvorna slika')
plt.subplot(3, 2, 2)
plt.imshow(mag_spek, cmap='gray')
plt.title("Magnituda spektra (izvorna)")
plt.subplot(3, 2, 3)
plt.imshow(mag_spek_mHP, cmap='gray')
plt.title("Magnituda spektra (pobrisane nizke frekvence)")
plt.subplot(3, 2, 4)
plt.imshow(slika_nazaj_HP, cmap='gray')
plt.title('Filtrirana slika (visokoprepustni filter)')
plt.subplot(3, 2, 5)
plt.imshow(mag_spek_mLP, cmap='gray')
plt.title("Magnituda spektra (pobrisane visoke frekvence)")
plt.subplot(3, 2, 6)
plt.imshow(slika_nazaj_LP, cmap='gray')
plt.title('Filtrirana slika (niskoprepustni filter)')
plt.show()
```

Prepoznavanje periodičnih vzorcev, kot so razne teksture, je ena najpomembnejših lastnosti Fourierove transformacije. Periodične strukture se v magnitudi spektra pokažejo kot žarki in gruče. S slike jih pobrišemo tako, da v spektru pobrišemo njim pripadajoče žarke. Poleg tega lahko frekvenčna domena razkrije simetrije v prostorski domeni, ki niso vedno očitne. Krogi ali pravokotniki imajo na primer značilne frekvenčne podpise.

4.7 Primeri obdelave slike v kontroli kakovosti

4.7.1 Kontrola zvarov

Na avtomatizirani proizvodni liniji varimo cevaste izdelke. Zvar je izveden po celotnem obodu cevi. Z vidika zagotavljanja ustrezne nosilnosti zvara nas zanima, ali je zvar po celotnem obsegu cevi zadosti globoko prevarjen. Odločimo se, da bomo izvajali vzorčno porušno kontrolo, kar pomeni, da iz proizvodnje vzamemo vzorec nekaj izdelkov enkrat na izmeno, izdelke prerežemo po sredi zvara, površino pobrusimo in spoliramo, tako da je zvar v preseku lepo viden, nato pa s kamero slikamo zvar in dobimo sliko, kot je prikazana na sliki 4.15 (1). V ta namen je bilo izdelano preizkuševališče, kamor vstavimo prerezani izdelek ter ga osvetlimo v osi s kamero in cevjo. Pomembno je, da sta osvetlitev in čas zajema slike nastavljena tako, da je zvar povsem bele barve, okoliški material pa temnejši.

Naloga obdelave slike je naslednja: s slike moramo izluščiti zvar, analizirati globino prevaritve po celotnem obsegu in označiti izdelek kot neustrezen, če je globina prevaritve na katerem mestu manjša od mejne vrednosti.

Zaporedje operacij obdelave slike je v nadaljevanju prikazano z uporabo OpenCV-funkcij v Pythonu, slike vmesnih rezultatov pa so na sliki 4.15 oštevilčene v levem zgornjem vogalu, v komentarjih znotraj kode pa se nanje sklicujemo znotraj oklepajev, npr. #(3). V opisu postopka v nadaljevanju navajamo zgolj ključne funkcije obdelave slike.

Ker smo s kamero shranili sliko na disk za potrebe sledljivosti, najprej z `imread` funkcijo preberemo izvirno barvno sliko ”testni_izdelek.jpg” (1); pri tem jo takoj spremenimo v 8-bitno sivinsko sliko `slg` (2). V naslednjem koraku `slg` rahlo filtriramo in zameglimo ostre detajle ter zmanjšamo šum z ukazom `GaussianBlur`. Načeloma filtra v tem koraku ne potrebujemo. Zakaj torej filtriramo? Pri prepoznavi vzorcev v nadaljevanju bomo s slike izluščili zvar kot največje območje povezanih slikovnih elementov. Poleg največjega območja je še veliko majhnih območij kot posledica raznih drugih detajlov in šuma, ki samo motijo. S filtriranjem poskušamo odstraniti vse, kar izhaja iz šuma in majhnih variacij v svetlosti. Jakost filtra je določena z velikostjo jedra – podobno kot pri konvoluciji; v tem primeru smo izbrali jedro velikosti (5, 5).

```
#branje
slg = cv2.imread("testni_izdelek.jpg",cv2.IMREAD_GRAYSCALE) #(2)
#filter
slgf = cv2.GaussianBlur(slg, (5, 5),0) #(3)
#shranimo sliko
cv2.imwrite("filtrirana_slika.png", slgf)
```

V naslednjem zelo pomembnem koraku izdelamo masko, ki prikazuje skupine povezanih slikovnih elementov. V ta namen uporabimo funkcijo `threshold`. Uporabimo prag 190, tako da na sliki vsi slikovni elementi, katerih svetlost je višja od pragu, dobijo vrednost 255 (11111111_{bin}). Kombinacija ustreznega zajema slike in izbire pragu omogoči, da na maski prvenstveno ostane samo zvar. Vrednost praga izberemo s poskusi.

Podroben pregled slike (4) pokaže, da je ob zvaru še polno majhnih skupin povezanih slikovnih elementov (otočkov) kot posledica šuma in majhnih artefaktov na sliki, ki bodo v nadaljevanju motili obdelavo, zato se jih želimo znebiti. V ta namen izberemo morfološko operacijo zapiranja `close`. Za to funkcijo moramo najprej izdelati jedro `kernel`, s katerim določimo lokalno vplivno območje funkcije oz. jakost zapiranja. Skupine povezanih slikovnih elementov, manjših od velikosti jedra, bodo s slike izbrisane. Prav tako bodo zapolnjene majhne luknje v velikih skupinah. S to funkcijo rahlo izboljšamo masko. Načeloma bi lahko šli naprej tudi brez te funkcije.

```
maska = cv2.threshold(slgf, 190, 255, cv2.THRESH_BINARY) #(4)
kernel = np.ones((3,3),np.uint8)
maskaclosed = cv2.morphologyEx(maska, cv2.MORPH_CLOSE, kernel) #(5)
```

```
#shranimo sliko
cv2.imwrite("Maskaclosed.png",maskaclosed)
```

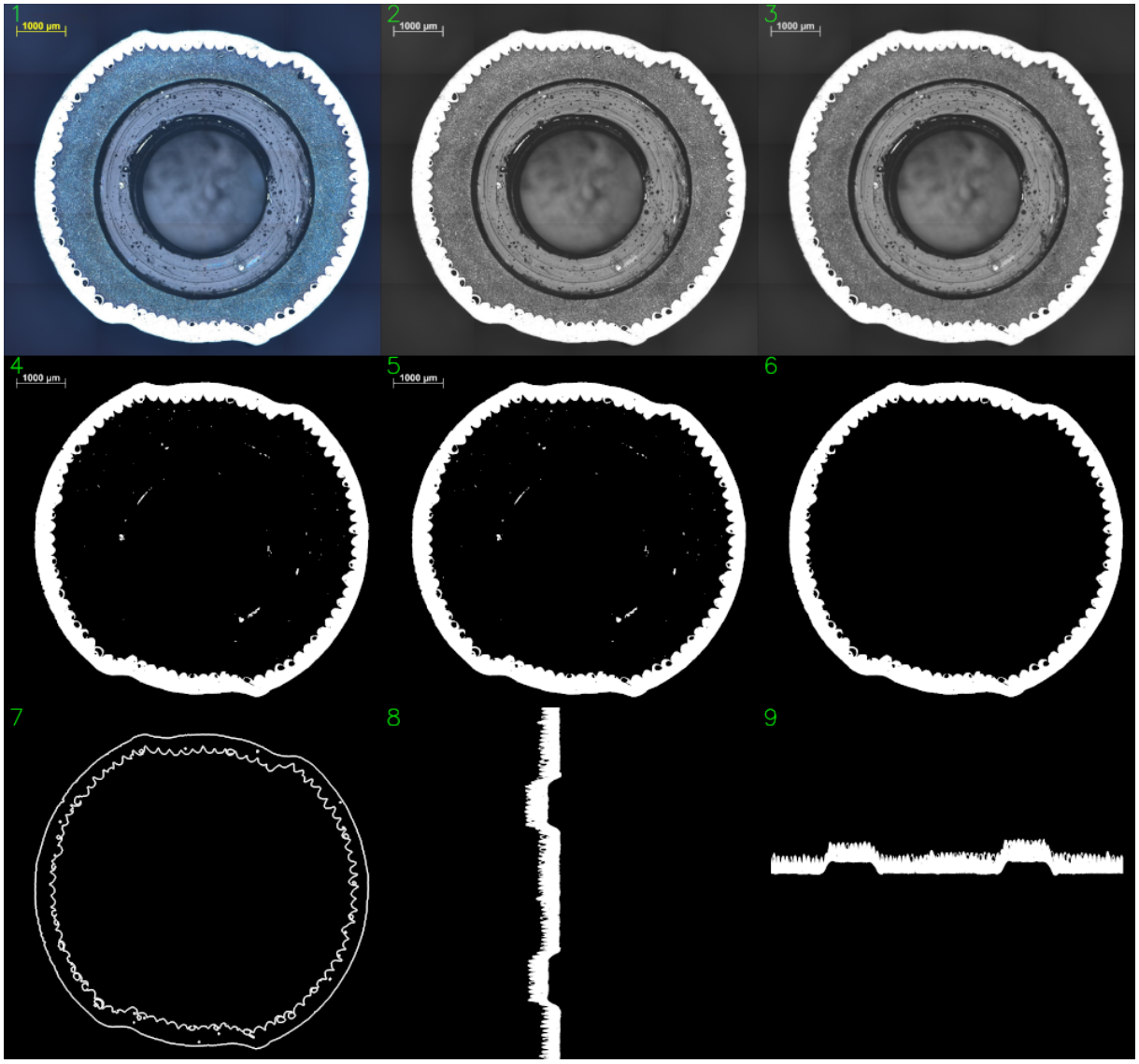
Naslednji korak je zelo pomemben in tudi zahteven. Na maski (5) imamo množico skupin povezanih slikovnih elementov: največja skupina predstavlja zvar, ob njem pa je še veliko majhnih skupin. V tem koraku želimo obdržati samo skupino določene velikosti, v našem primeru zvar, vse ostale pa zanemariti. V ta namen moramo najprej izračunati, koliko slikovnih elementov je v posamezni skupini. V ta namen nam OpenCV nudi funkcijo `connectedComponentsWithStats`. Funkcija vrne več parametrov. Prvi je *stskupin*, v prikazanem primeru je to bilo 84. Če v predhodnih korakih ne bi uporabili funkcij filtriranja in zapiranja, bi teh lahko bilo več tisoč, posledično bi se vsi izračuni izvajali bistveno počasneje. Drugi parameter `maskazoznakamiskupin` je dejansko maska enake velikosti kot vhodna maska, le da so na njej označene skupine slikovnih elementov. Slikovni elementi ozadja imajo vrednost 0, slikovni elementi, ki pripadajo skupini 1, imajo vrednost 1, za skupino imajo vrednost 2 itd. Tretji parameter `stat` podaja osnovno statistiko za vsako skupino. Vsaki skupini je dodeljena vrstica, v njej je 5 stolpcev (glej `print(stat)`). Prvi štirje podajajo levi zgornji in desni spodnji vogal pravokotnika, očrtanega na določeno skupino povezanih slikovnih elementov, zadnji, peti stolpec pa podaja seštevek vseh slikovnih elementov v skupini. Slednji je zelo pomemben, saj glede nanj izberemo določeno skupino. Zadnji parameter `centroids` podaja težišče vsake skupine.

Če pogledamo zadnji stolpec v `stat`, ki podaja velikost posamezne skupine, vidimo, da je v prvi vrstici zapisano ozadje z 1.8 M slikovnih elementov, sledijo manjše skupine z nekaj sto slikovnimi elementi, skupina v osmi vrstici, ki pripada zvaru, je največja in bistveno odstopa po velikosti od ostalih s skoraj 250 k slikovnih elementov. V `maskazoznakamiskupin` bo vrednost slikovnih elementov, ki pripadajo tej skupini, enaka 7 (7. vrstica v `stat`, ker se štetje v Pythonu začne z 0).

Naša naloga v nadaljevanju je, da izdelamo programsko kodo, pri kateri bomo v petem stolpcu `stat` spustili prvo številko, ki pripada ozadju, poiskali drugo največjo vrednost, ugotovili, v kateri vrstici je, s tem bomo poznali oznako `maskazoznakamiskupin`, ter v novo masko skopirali samo izbrano skupino.

Torej: iz `stat` izrežemo zadnji stolpec z ukazom `velikosti = stat[1:, -1]` (1: pomeni vrstice od 1 naprej, brez nulte vrstice in –1 pomeni zadnji stolpec). Nato za 1 zmanjšamo število skupin in naredimo novo prazno masko `Maska`. Ključna je `for i in range(0, stskupin)` zanka, kjer gremo čez vse skupine, preverjamo velikost skupine, in če je ta večja od mejne vrednosti `min_size = 15000` (ocenimo), skopiramo skupino z `maskazoznakamiskupin` v `Masko`, ob tem pa slikovnim elementom dodelimo vrednost 255.
To naredimo z ukazom `Maska[maskazoznakamiskupin == i + 1 ] = 255` (i + 1 zato, ker smo zbrisali ničelno vrstico). V prikazanem primeru smo izbrali največji objekt. V praksi pogosto izbiramo objekt določene velikosti, kar naredimo z izbiro ustreznega kriterija v *if* stavku.

```
stskupin, maskazoznakamiskupin, stat, centroids =
cv2.connectedComponentsWithStats(maskaclosed, connectivity=8)

#print(stat)
#[[ 0 0 1499 1399 1844494]
#[ 77 75 11 25 119]
#[ 97 75 16 26 197]
#[ 118 75 16 26 197]
#[ 139 75 16 26 205]
#[ 51 82 196 47 447]
#[ 194 82 27 18 213]
#[ 124 107 1326 1250 249547]
#...

#izberemo zadnji stolpec v stat (brez prve (ničelne) vrednosti)
```



Slika 4.15: Tipične faze obdelave slike

```
velikosti = stat[1:, -1]
#ker smo prvo vrednost ozadja spustili, za 1 zmanjšamo št.  skupin
stskupin = stskupin - 1
#skupina, ki pripada zvaru bo morala biti večja od min_size
min_size = 15000
#Naredimo novo prazno masko, kamor bomo skopirali zvar
Maska = np.zeros((maskazoznakamiskupin.shape))


for i in range(0, stskupin):
        if velikosti[i] >= min_size:
        Maska[maskazoznakamiskupin == i + 1 ] = 255 #**

Maska8b = np.array(Maska,np.uint8)  #(6)

cv2.imwrite("masksamozvar.png", Maska8b) #(6)

#lahko tudi izrišemo na ekran
```

```
# cv2.imshow('Maska samo zvar', Maska8b)
# cv2.waitKey(0)
# cv2.destroyAllWindows()
```

V nadaljevanju predpostavimo, da želimo poiskati robove zvara na `Maska8b`. To storimo s funkcijo `findContours`. Funkcija vrne sestav (ang. touple) najdenih kontur v spremenljivki `cnts`. Robovi so popisani z množico točk na sliki npr. [555 107]. Kako funkcija aproksimira rob, je določeno z drugim in s tretjim parametrom. Vrne lahko vse možne točke na robu, aproksimacijo robu z manjšim številom točk ali samo pravokotnik, znotraj katerega se nahaja rob. Drugi izhodni parameter `hir` je hierarhija kontur. Ta podaja informacije o topologiji slike, natančneje o odnosih med konturami (npr. konture znotraj kontur).

```
#poiščemo robove (zunanje konture)
cnts,hir =
cv2.findContours(Maska8b.copy(), cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)

#primer izpisa prve najdene konture
#print(cnts[0])
#[[[555 107]]
# [[554 108]]
# [[553 108]]
...
# [[558 107]]
# [[557 107]]
# [[556 107]]]

#za potrebe razumevanja delovanja najdene robove narišemo na novo masko
#naredimo prazno masko
MaskaRob = np.zeros((Maska8b.shape))
MaskaRob = np.array(MaskaRob,np.uint8)
#v masko narišemo konture
cv2.drawContours(MaskaRob, cnts, -1, (255,255,255), 3)
#shranimo masko
cv2.imwrite("Maska_robovi.png",MaskaRob) #(7)
```

Za potrebe kontrole kakovosti želimo analizirati globino prevaritve zvara. Z iskanjem kontur v prejšnjem primeru smo dobili dve konturi, s katerima sta popisana zunanji in notranji rob zvara, kot to prikazuje slika (7). Globina prevaritve je razlika med konturama v radialni smeri glede na center. V nadaljevanju bi morali napisati program, kjer bi z zanko for šli po obodu, npr. na 1° natančno, iskali najbližje točke na obeh robovih, po potrebi tudi interpolirali med njimi in nato izračunali razdaljo, tj. globino prevaritve.

Poglejemo, ali obstaja še kakšna alternativa predhodno prikazanemu iskanju robov. Ker je zvar v obliki krožnice, bi bila alternativa, da sliko razvijemo iz polarnega v kartezični koordinatni sistem. Predpostavimo, da je na sliki (6) center polarnega koordinatnega sistema v sredini cevi (krožnice zvara). Radialna oddaljenost zvara od centra je v tem primeru konstantna (zanemarimo manjše variacije v obliki zvara). Če sliko razvijemo po kotu (npr. prvi stolpec nove slike bi bila svetlost obstoječe slike od sredine cevi vzdolž radija, do nekega maksimalnega radija pri kotu 0°; drugi stolpec bi bila svetlost radialno navzven pri kotu 1°; itd), dobimo sliko, kjer bo zvar izravnan, kot to prikazujeta sliki (8) in (9). Iskanje debeline zvara na razviti sliki je razmeroma enostavno.

Programska koda za izvedbo transformacije se začne z določitvijo centra zvara. V ta namen vzamemo zunanji rob zvara `zr = cnts[0]` in izračunamo njegove momente s `cv2.moments` funkcijo. Težišče v x-smeri se izračuna kot $\bar{x} = \frac{m_{10}}{m_{00}}$, pri čemer ničelni moment predstavlja površino $m_{00} = \sum_x \sum_y I(x,y)$, prvi momenti pa $m_{10} = \sum_x \sum_y x \cdot I(x,y)$. Podobno se izračuna težišče v y-smeri. Dimenzijo krajše stranice vzamemo kot radij, znotraj katerega se izvaja razvoj slike. V nadaljevanju je podan še primer vrtenja okrog sredine slike za 90° sourno. V ta namen je

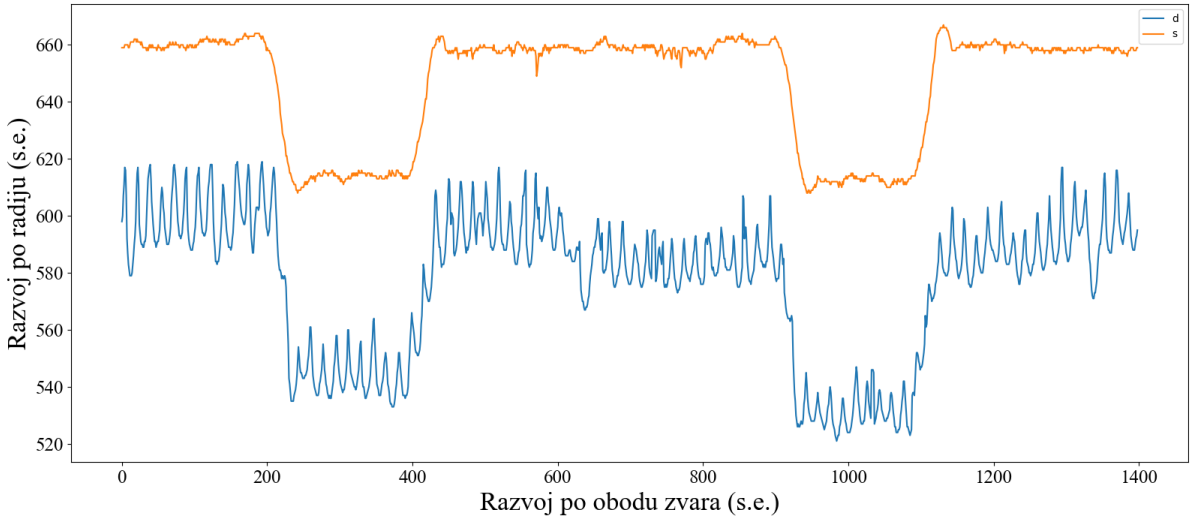
najprej izdelana rotacijska matrika `RotMat` s funkcijo `cv2.getRotationMatrix2D`, ki ji podamo center vrtenja, v našem primeru sredino slike in kot zasuka. Nato se izvrši rotacija s funkcijo `cv2.warpAffine`.

```
#kontura zunanjega robu
zr = cnts[0]
#izračun momentov ...  glej OpenCV navodila
M = cv2.moments(zr)
#težišče v x in y smeri
cX = int(M["m10"] / M["m00"])
cY = int(M["m01"] / M["m00"])

#razvoj se bo vršil od centra do tega radija
maxR = min(Maska8b.shape[0],Maska8b.shape[1])

#razvoj slike
Razvitasl = cv2.linearPolar(Maska8b,(cX, cY), maxR, cv2.WARP_FILL_OUTLIERS)
#shranimo
cv2.imwrite("Razvitaslika.png",Razvitasl) #(8)

#rotacija slike
rows,cols = Razvitasl.shape
#rotacijska matrika
RotMat = cv2.getRotationMatrix2D((cols/2,rows/2),-90,1)
#rotacija
Razvitaslrot = cv2.warpAffine(Razvitasl,RotMat,(cols,rows))  #(9)
#shranimo
cv2.imwrite("Razvitaslika_rotated.png",Razvitaslrot) #(9)
```

Sledi prikaz programa za iskanje robov. Gibljemo se po stolpcih, iščemo zunanji in notranji rob zvara, in ko ga najdemo, rezultat shranimo v sezname d in s. Slika 4.16 prikazuje zaznane robove zvara (enote so slikovni elementi). V naslednjem koraku bi izračunali razliko med obema robovoma ($s - -d$), jo umerili in preverili, ali je vzdolž zvara v celoti večja od minimalne predpisane debeline (ni prikazano). Prikazani primer iskanja robu je enostaven za razumevanje, ni pa računsko najbolj učinkovit. Za vajo poskusite napisati računsko učinkovitejše iskanje robu.

```
#iskanje robov
heigth = Razvitaslrot.shape[0]
width = Razvitaslrot.shape[1]
#tu bomo shranjevali rezultatez
d = [] # Notranji rob (depth)
s = [] # Zunanji rob (surface)

#Premikamo se po stolpcih
for i in range(width):
        notranji_rob = False
        zunanji_rob=False
        notranji_index = 0
        zunanji_index = 0

        #robove iščemo vzdolž stolpcev
        for j in range(heigth):

            #notranji rob
                if Razvitaslrot[j,i] > 200 and not notranji_rob:
                        #rob najden
                        notranji_index = j
                        notranji_rob = True

                #zunanji rob
                if Razvitaslrot[-j,i] > 200 and not zunanji_rob:
```

```
                        #rob najden
                        zunanji_index = heigth-j
                        zunanji_rob = True

                #shranjevanje
                if zunanji_rob and notranji_rob:
                        d.append(notranji_index)
                        s.append(zunanji_index)

#izris najdenih robov
fig=plt.figure())
plt.plot(d)
plt.plot(s)
plt.legend(['d','s'])
plt.show()
plt.savefig("zaznani_robovi.png")
```



Slika 4.16: Zunanji in notranji rob zvara

4.7.2 Kontrola barvanja

Elektroforetski katodni kovinski premaz (KTL- ali E-premaz) je pomemben tehnološki postopek zaščite kovinskih delov pred vplivi okolja. Tipično se uporablja za zaščito kompoment v avtomobilski industriji. Obstaja več različnih napak barvanja pri tem postopku, ki se med seboj razlikujejo po obliki, izgledu in pomembnosti (npr. kraterji, hrapavost, umazanija, barvne lise, mehurčki, itd). Za potrebe avtomatizirane kontrole napak barvanja je bila razvita kontrolna naprava[9], ki z več kamerami in s skrbno zasnovano osvetlitvijo poslika celotno površino izdelka. Slika 4.17 prikazuje primer posnetka odpreška, privarjenega na nosilno cev.

Slika 4.17: Slika odpreška, privarjenega na nosilno cev. Celoten izdelek je površinsko zaščiten s KTL- površinsko zaščito, ki je črne sijaj barve. Sistem strojnega vida osvetli odprešek z rdečo barvo, in sicer malenkost pod kotom glede na kamero, tako da so minimizirani odboji svetlobe z gub na površini izdelka. Slika je zajeta s črno-belo kamero, ki ima nameščen ozkopasovni svetlobni filter, tako da v kamero prepušča samo rdečo svetlobo. Na ta način so minimizirani vplivi okoliške svetlobe in neželenih refleksij drugih svetil z gub.

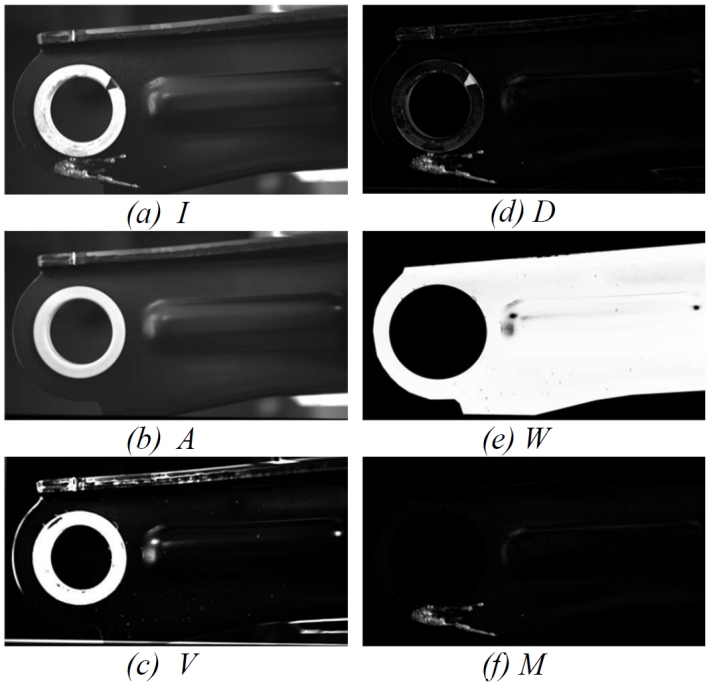
Največji izziv pri razvoju kontrole napak barvanja je zaznavanje napak na 3D-ukrivljenih visoko sijajnih črnih površinah ter razlikovanje med dejanskimi napakami in refleksijami svetlobe na gubah. Pri ukrivljenih predmetih je zelo težko nastaviti osvetlitev tako, da bi ustrezala vsem iskanim značilnostim objekta, saj se bo svetloba na nekaterih delih ukrivljene površine objekta odbijala neposredno v kamero (svetle cone – preosvetlitev slike), medtem ko se bo na drugih delih odbijala stran (temne cone). Svetli odsevi so podobni napakam barvanja (npr. praskam, hrapavosti), kar povzroča lažne zaznave napak. Dodatno je slikovni sistem zelo občutljiv na variabilnost med izdelki, tj. na ponovljivost njihovih oblik in dimenzij (kljub temu da so znotraj predpisanih dimenzijskih toleranc) ter na variabilnost pri pozicioniranju znotraj kontrolne naprave med zajemom slik. Variabilnost dimenzij med deli je dobro znana težava na področju obvladovanja proizvodnih procesov in je predmet stalnih izboljšav proizvodne tehnologije. Prav tako se variabilnost pri pozicioniranju znotraj kontrolne naprave lahko zmanjša le do določene mere. Vse te variabilnosti zahtevajo zasnovo algoritmov obdelave slik, ki bodo dovolj robustni glede na lego izdelka znotraj vidnega polja kamere in posledično na spreminjanje lege refleksij na površini izdelka.

Reševanja problema variabilne svetlosti preiskovanega kosa se lahko lotimo na več načinov. Najbolj sodoben pristop bi temeljil na uporabi nevronskih mrež in tehnik globokega učenja, kar pa daleč presega osnove strojnega vida, ki ga podajamo v tem učbeniku. V nadaljevanju prikazujemo primer obdelave slike z osnovnimi metodami iz knjižnice OpenCV, le da je zaporedje operacij slike domišljeno za potrebe reševanja problema variabilne svetlosti preiskovanega kosa.

[9]Zaradi zaščite avtorskih pravic, slike kontrolne naprave pokažemo zgolj na predavanjih.

Ko govorimo o variabilni svetlosti, imamo v mislih predvsem to, da svetlost površine preiskovanega kosa variira. Če pogledamo sliko 4.17, vidimo, da je leva stran odpreška svetlejša od desne strani, da na robovih svetla področja hitro preidejo v temno; podobno je na gubah: na levi strani gub imamo zrcalni odboj neposredno v kamero, desna stran gube pa je v senci. Še večji problem se pojavi, če se zaradi dimenzijskih in pozicionirnih variabilnosti malenkost spreminja lega odpreška relativno na osvetlitev in kamero, kar pomeni, da se svetle in temne cone rahlo premikajo po površini od kosa do kosa.

Za trenutek se spomnimo, kakšno je bilo zaporedje operacij obdelave slike v primeru iskanja debeline zvara (poglavje 4.7.1). Sliko smo najprej pripravili za obdelavo s filtriranjem, nato smo uporabili pragovno funkcijo, s katero smo ločili svetel zvar od temnega ozadja, v nadaljevanju smo obdržali samo zvar, ga razvili in izračunali njegove dimenzije. Ključna operacija je bila pragovna funkcija (`threshold`), ki je slikovnim elementom po celotni sliki enakomerno dodelila vrednost 255, če je bila svetlost višja od praga. V primeru kontrole napak barvanja pa to ne bi delovalo. Zakaj? Primer napake si poglejmo na sliki 4.18 (a). Napaka (vodni pocedek) je enako svetel kot refleksija na gubi, okrogla puša nad njim je še svetlejša, podobno robovi odpreška. Pragovna funkcija bi vrnila veliko zelo podobnih skupin slikovnih elementov; ugotoviti, kateri predstavlja napako, je skoraj nemogoče ali zelo nezanesljivo. Kar lahko naredimo, je, da pred uporabo pragovne funkcije upoštevamo vse, kar vemo o variacijah svetlosti.



Slika 4.18: Obdelava slike po korakih za primer kontrole barvanja. (a) Slika I, zajeta s kamero, (b) referenčna slika A, (c) slika variance V, (d) slika razlike slik D, (e) maska W in (f) končna slika, primerna za prepoznavanje vzorcev M.

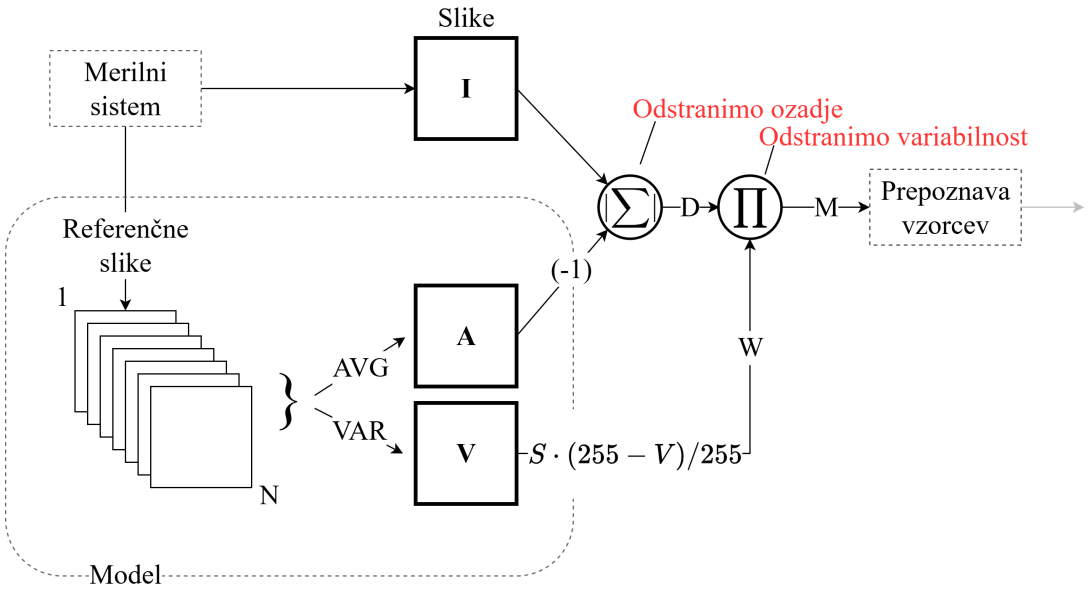
Predpostavimo, da imamo sliko idealnega odpreška, ki je povsem brez napak, in ga uporabimo kot referenco. Ko zajamemo neko sliko, analiziramo odstopanja od reference. Na mestu napak bodo odstopki veliki, drugje *v razumno majhnih* mejah. Pragovno funkcijo za zaznavo napak uporabimo na sliki odstopanj od reference. Diagram na sliki 4.19 prikazuje navedeno logiko. S kamero zajamemo veliko slik dobrih izdelkov brez napak (referenčne slike). Te medsebojno povprečimo (tj. povprečenje svetlosti sovpadajočih slikovnih elementov med referenčnimi slikami) in dobimo referenco A, ki prikazuje sliko idealnega izdelka brez napak. Sočasno s pov-

prečenjem slik na vsakem slikovnem elementu izračunamo tudi varianco – koliko svetlost na sovpadajočih slikovnih elementih variira med referenčnimi slikami. Na robovih izdelka in na gubah bo zaradi predhodno opisanih dimenzijskih in pozicionirnih variabilnosti varianca velika, drugod majhna. V nadaljevanju se bo izkazalo, da je slika variance V izredno pomembna za izdelavo maske, s katero povečamo zanesljivost delovanja. Sliki povprečja A in variance V sta prikazani na sliki 4.18 (b) in (c). Naredimo ju samo enkrat, saj se ne spreminjata, vse dokler ne spremenimo optične nastavitve sistema, lege ali oblike izdelka. Lahko bi jih imenovali tudi *slikovni model*.

Med samim delovanjem kontrolne naprave od trenutno zajete slike I odštejemo sliko povprečja A ter tako odstranimo svetlost ozadja in dobimo sliko odstopkov D 4.18 (d). Še zmeraj je na sliki veliko svetlih artefaktov na robovih izdelka in na puši, kar še naprej moti iskanje napak s pragovno funkcijo. Sedaj uporabimo sliko variance V, ki vsebuje podatke, kje je velika variabilnost (na robovih in na puši), in jo uporabimo kot utež, s katero pomnožimo sliko odstopkov D. V ta namen moramo sliko V malenkost predelati. V V so izvorno vrednosti med 0 in 255, na sliki uteži W pa rabimo vrednosti med 0 in 1. Kar želimo obdržati v sliki D, moramo pomnožiti z 1, kar želimo pobrisati, moramo množiti z 0, če katerim določenim področjem želimo zmanjšati pomembnosti pri uporabi pragovne funkcije (npr. področjem na gubah, kjer še zmeraj ostajajo svetli artefakti), pa uporabimo vrednosti med 0 in 1 glede na stopnjo zmanjšanja pomembnosti. V ta namen uporabimo izračun $S \cdot (255 - V)/255$ in dobimo sliko uteži W, kot je prikazano na sliki 4.18 (e). S je dodatna maska enakih dimenzij kot V, ki smo jo ročno izdelali, npr. v *MS Paintu*, tako da smo pobrisali (postavili na 0) vse, kar ni odprešek, in z odtenki sivine pobarvali cone zmanjšanega pomena. Sliko variance smo pri tem invertirali $255 - V$, saj želimo to, kar je na tej sliki svetlo, postaviti na 0. Po množenju slike odstopkov D s pripadajočimi utežmi W dobimo končno sliko M, na kateri so zgolj odstopki od reference, tj. napake, ki jih v nadaljevanju prepoznamo in vrednotimo s pragovno funkcijo ter pomožnimi izračuni velikosti, lokacije, raztrosa itd.

V nadaljevanju sledi prikaz funkcije, v kateri se izvaja predhodno opisana logika obdelave slike. Za potrebe računske učinkovitosti je koda napisana v programskem jeziku $C++$. Demonstriramo uporabo funkcij iz knjižnice OpenCV. Najprej so uvodne nastavitve (zgolj del), nato sledi funkcija `ObdelajSliko`, ki kot vhodne argumente vzame pot in ime slik I, A in W, izračuna pa število najdenih skupin povezanih slikovnih elementov, ki predstavljajo potencialne napake. V nadaljevanju bi sledila npr. analiza velikosti in raztrosa skupin ter na tej osnovi sklepanje o prisotnosti napak (ni prikazano).

```
// Uvodne nastavitve, C++ koda
#include "stdafx.h"
#include "opencv2/imgcodecs.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <opencv2/video/video.hpp>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

using namespace cv;
using namespace std;

//Deklaracija spremenljivk slik
Mat src;
Mat src_gray;
Mat imref;
Mat imref_gray;
Mat maska;
Mat slikaTh;
```



Slika 4.19: Diagram predobdelave slike za kontrolo barvanja. S povprečenjem (AVG) N referenčnih slik dobimo referenčno sliko A. Sočasno izračunamo tudi varianco (VAR) med sovpadajočimii slikovnimi elementi referenčnih slik in dobimo sliko variance V. Sliko A odštejemo od I (v resnici pogledamo njuno odstopanje z `absdiff` funkcijo v OpenCV, $|\Sigma|$ simbol). V končni fazi dobljeno sliko D pomnožimo (simbol Π) s sliko uteži W in dobimo sliko M, ki je primerna za prepoznavo vzorcev.

```
Mat slikaThM;
// ...

//Definicija funkcije obdelave slike --------------------
//Kot argumente podamo polno pot do slik I, A in W
int ObdelajSliko(const char* ImageName, const char* RefName, const char*
    MaskaName){
int status = 0;

//Branje in Priprava Vhodnih Slik ---------------------
src = imread(ImageName, IMREAD_COLOR);             // Preberemo vhodno sliko (I)
if (src.empty()) return status = -1;               // Preverim,o ali je uspešno prebrana

imref = imread(RefName, IMREAD_COLOR);             // Preberemo referenčno sliko (A)
if (imref.empty()) return status = -2;             // Preverimo, ali je uspešno prebrana

maska = imread(MaskaName, CV_LOAD_IMAGE_GRAYSCALE); // Preberemo masko (W)
if (maska.empty()) return status = -3;             // Preverimo, ali je uspešno prebrana

//Barvne slike pretvorimo v sivinske (zgolj za demonstracijo pretvorbe,
//načeloma bi jih takoj lahko prebrali kot sivinske slike, podobno kot pri maski)
cvtColor(src, src_gray, COLOR_BGR2GRAY);
cvtColor(imref, imref_gray, COLOR_BGR2GRAY);

//(Opcijsko) Slike rahlo filtriramo in zmanjšamo šum
blur(src_gray, src_gray, Size(3, 3));
blur(imref_gray, imref_gray, Size(3, 3));

//Poravnava I z A ----------------------------
//Nastavitev parametrov za iskanje poravnave slik
```

```
const int warp_mode = MOTION_TRANSLATION;
Mat TM = Mat::eye(2, 3, CV_32F); //Transformacijska matrika (diagonalna)
int number_of_iterations = 100;
double termination_eps = 1e-5;
TermCriteria criteria(TermCriteria::COUNT + TermCriteria::EPS,
number_of_iterations, termination_eps);
//Poiščemo vrednosti TM za poravnavo I z A.
findTransformECC(src_gray, imref_gray, TM, warp_mode, criteria);

//Ustvarimo novo sliko, v katero bomo skopirali transformirano sliko
Mat src_aligned(src_gray.size().width, src_gray.size().height, CV_8UC1, Scalar
    (0));

//Izvedemo transformacijo (poravnavo slike).
warpAffine(src_gray, src_aligned, TM, src_gray.size(), INTER_LINEAR +
    WARP_INVERSE_MAP);

// |I-A|*W -------------------------------
//Ustvarimo novo sliko (D), v katero bomo shranili razliko med slikama (I) in (A)
Mat diffD(src_gray.size().width, src_gray.size().height, CV_8UC1, Scalar(0));

//Izračunamo razliko (D)
absdiff(imref_gray, src_aligned, diffD);

//Sliko (D) pomožimo z utežmi (W) in dobimo (M)
diffD = diffD.mul(maska);                    // (D) = (D) * (W) ..=(M)

//Prepoznava Vzorcev-------------------------
//Na sliki razlik uporabimo pragovno funkcijo; Th=190
threshold(diffD, slikaTh, 190, 255, THRESH_BINARY);

//Pobrišemo majhne skupine povezanih slikovnih elementov (šum) z uporabo morfoloških operacij
int morph_size = 1;
int morph_elem = 1;
Mat element = getStructuringElement(morph_elem, Size(2 * morph_size + 1, 2 *
    morph_size + 1), Point(morph_size, morph_size));
/*
Opening: MORPH_OPEN : 2
Closing: MORPH_CLOSE: 3
Gradient: MORPH_GRADIENT: 4
Top Hat: MORPH_TOPHAT: 5
Black Hat: MORPH_BLACKHAT: 6
*/
int operation = 2;
morphologyEx(slikaTh, slikaThM, operation, element);

//Poiščemo obrise skupin povezanih slikovnih elementov (otočkov)
findContours(MTslika, contours, hierarchy, RETR_TREE, CHAIN_APPROX_SIMPLE, Point
    (0, 0));

//...  tu sledi analiza velikosti in raztrosa skupin ter sklepanje o napakah ...

status = contours.size();
return status; //vrnem število kontur
}
```

Literatura

[1] Alexander Hornberg, editor. *Handbook of Machine Vision*. Springer, London, 2017.

[2] Peter Corke. *Robotic Vision and Control*. Springer Nature, Cham, second edition, 2023.

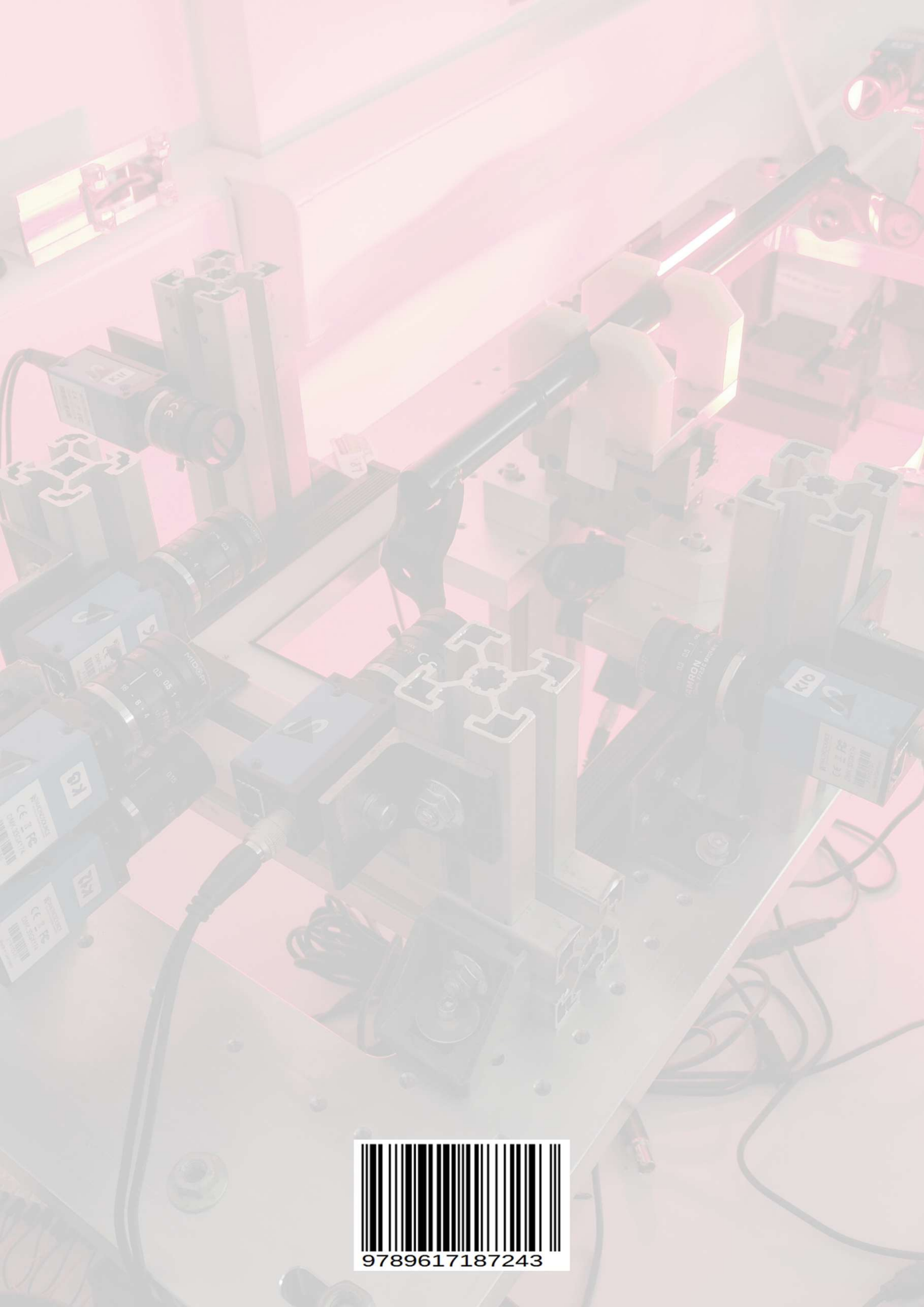
[3] Rafael C. Gonzalez, Richard E. Woods, and Steven L. Eddins. *Digital Image Processing Using MATLAB*. Pearson Prentice Hall, Upper Saddle River, NJ, 1st edition, 2004.

[4] Sandipan Dey. *Hands-On Image Processing with Python*. Packt Publishing, Birmingham, UK, 2018.

[5] OpenCV Development Team. *OpenCV: Open Source Computer Vision Library*. OpenCV.org, 2024. Version 4.x.

[6] Jean-Yves Bouguet. Camera calibration toolbox for matlab. `http://robots.stanford.edu/cs223b04/JeanYvesCalib/`, 2004. Accessed: 2024-08-28.

[7] Janne Heikkilä and Olli Silvén. A four-step camera calibration procedure with implicit image correction. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1106–1112, San Juan, Puerto Rico, 1997. IEEE.



9789617187243