

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

PROJEKT

**Unaprijeđenje modula za
prepoznavanje teksta u realnom
vremenu kod NAO H25 ATOM
roboata**

Mirko Kokot

Voditelj: *Zdenko Kovačić*

Zagreb, siječanj 2015.

SADRŽAJ

1. Uvod	1
2. Postojeće programsko rješenje	2
2.1. Struktura	2
2.2. Mogućnosti i rezultati	3
3. Unaprijedenje modula	4
3.1. Struktura	5
3.2. ocrModule	6
3.2.1. APIwrappers.h	7
3.2.2. picSceneDetection	8
3.2.3. picProcessing	8
3.2.4. picSegmentation	9
3.2.5. Tesseract 3.02	11
3.3. Primjer programa	15
3.3.1. Čitanje teksta sa slike	15
3.3.2. Pozicioniranje robota	16
3.4. Rezultati	18
4. Zaključak	19
5. Literatura	20

1. Uvod

U današnjem svijetu tehnologija je grana koja doživljava najveći i najbrži napredak. Stvari koje su, do prije manje od deset godina, bile nepojmljiva tehnologija budućnosti danas primjenjujemo svakodnevno ne primjećujući njihovu naglu integraciju u naš život. Brzi razvoj tehnologije donosi i dosad neviđenu kompleksnost izvedbe novih proizvoda i tehnologija koje se nerijetko zasnivaju na jednako naprednim rješenjima naizgled jednostavnih problema. Radi takvog eksponencijalnog rasta tehnologije javlja se sve veći jaz između proizvođača i korisnika gdje uz navike, koje se mijenjaju mnogo sporije, dolazi do neizbjegnog komformizma od strane korisnika ("*zdravo za gotovo*"). Stoga se jednom od sveprisutnih problema današnje tehnologije, njezina pristupnost i primjenjivost u svakodnevici krajnjih korisnika, uvijek mora pridodati mnogo pažnje.

Zadatak ovoga projekta je unaprijeđenje modula za prepoznavanje teksta u slikama u stvarnom vremenu za robota NAO H25 ATOM. Razmatrana su unaprijeđenja od povećanja brzine i pouzdanosti samog prepoznavanja, promjena i prenamjena strukture programa, pa sve do olakšavanja prilagodbe krajnjem korisniku.

Uz sam razvoj modula, razvijen je i program koji koristi modul za prepoznavanje teksta. On služi kao pokazni primjer razlike imedju starog i novog idejnog rješenja strukture modula, ali i prikaz primjera integracije sa nekim od rješenja problema usko vezanih za problematiku autonomnog prepoznavanja teksta.

Također u radu su opisani i detaljni postupci rješavanja problema susretnutih pri integraciji i implementaciji novih ideja, postupaka i paketa.

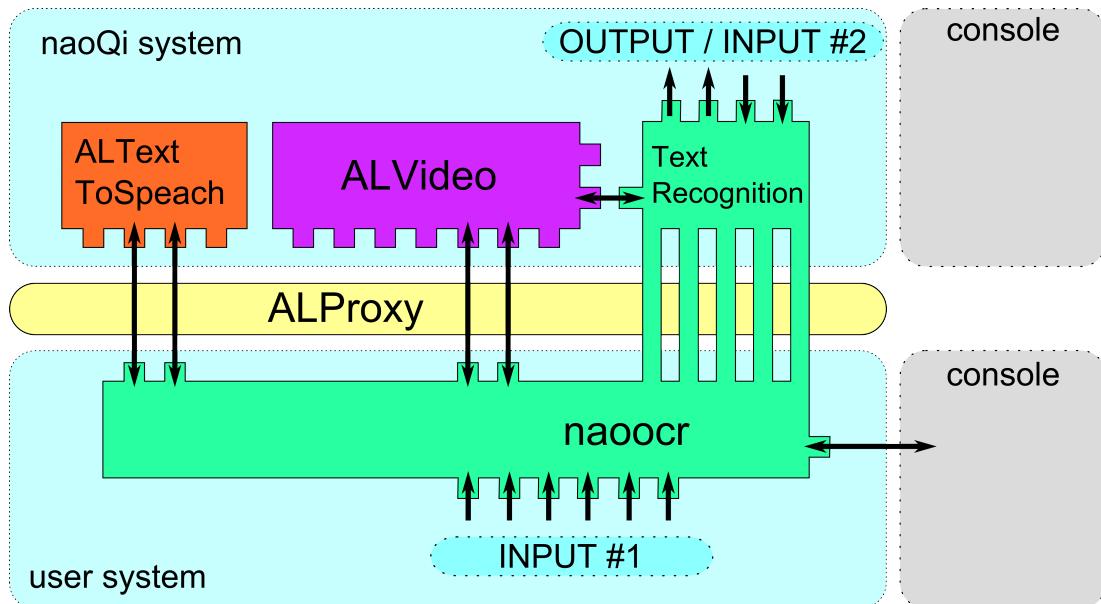
Ovaj rad se neizravno nadovezuje na diplomski rad Mihaela Mercvajlera *Razvoj programa za čitanje teksta i implementacija na humanoidnim robotima NAO H25 ATOM*[2]. Stoga se neće detaljnije opisivati NAO robot, računalni vid i sam algoritam prepoznavanja teksta. Projekt opisuje pristupačnost modula za 3 vrste upotrebe:

- *end user* - krajnji korisnik, nema uvid u razvoj
- *user* - korisnik koji poziva modul kod razvijanja svojeg programa/modula
- *developer* - korisnik koji razvija i nadograđuje sam modul

2. Postojeće programsko rješenje

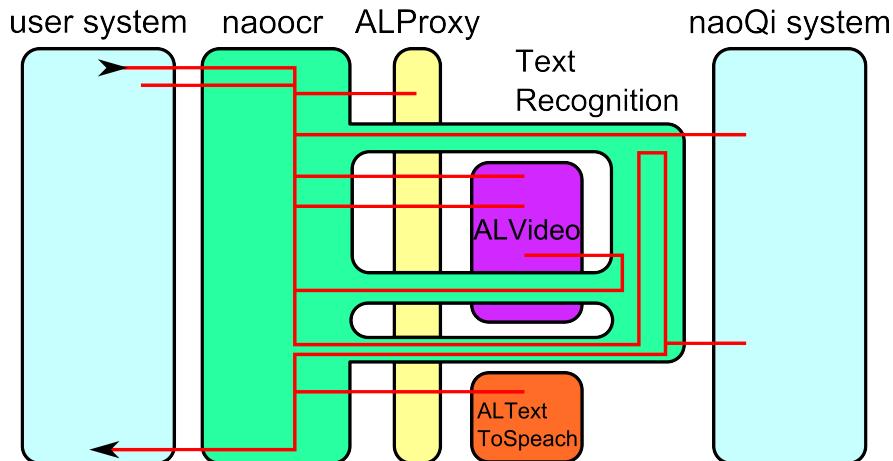
2.1. Struktura

Postojeće programske rješenje sastoji se od 2 dijela koji su vrlo usko povezani kao cijelina. Na slici 2.1 vidimo komunikaciju programa sa svojim okruženjem. Program se pokreće na *user system*-u preko konzole te kao ulaz uzima podatke iz oba programska okruženja, dok rezultate sprema u *non-user system*, a prema korisniku ih iznosi preko konzole ili govora.



Slika 2.1: Staro idejno rješenje komunikacije programa

Radi komplikirano raspoređene komunikacije, tok programa je također čvrsto određen radi pravilnog redoslijeda postavljanja i pozivanja. Drugi dio lošeg toka programa je što veliku većinu vremena *naoocr* čeka na *TextRecognition* koji odrađuje 90% posla. Simplificiran tok programa možemo vidjeti na slici 2.2. Ista čvrsta struktura programskog toka je prisutna i unutar samog *TextRecognition-a*.



Slika 2.2: Prikaz programskog toga starog idejnog rješenja

Ovo vrlo jako sužava opcije razvoja drugog programa sa *user* strane iz više razloga:

- nepotpuna kontrola nad ulaznim/izlaznim podacima
- mogućnost korištenja samo dijela podataka
- točno zadani redoslijed odvijanja procedura
- nemogućnost postepenog pozivanja procedura

2.2. Mogućnosti i rezultati

Modul ima implementirano vlastito prepoznavanje teksta što donosi velike prepreke kod buduće primjene:

- mala baza podržanih fontova
- neoptimiziranost
- mala podrška za budućeg *user-a*
- zamršena struktura za potencijalnog *developer-a*

Unatoč velikoj točnosti prepoznavanja teksta, radi načina implementacije, do rezultata se mora pričekati dok se cijela slika ne obradi što dovodi do, za *end usera*, vrlo sporih rezultata. Kod *remote module-a* (modul koji se izvršava na vanjskom sustavu, ali komunicira preko naoQi proxy-a) vrijeme izvođenja programa bilo je $3 > \text{sec}$. Pošto NAO H25 ATOM ima znatno slabiji procesor, vrijeme izvođenja poraslo je na $10 > \text{sec}$ za osrednju rečenicu.

3. Unaprijeđenje modula

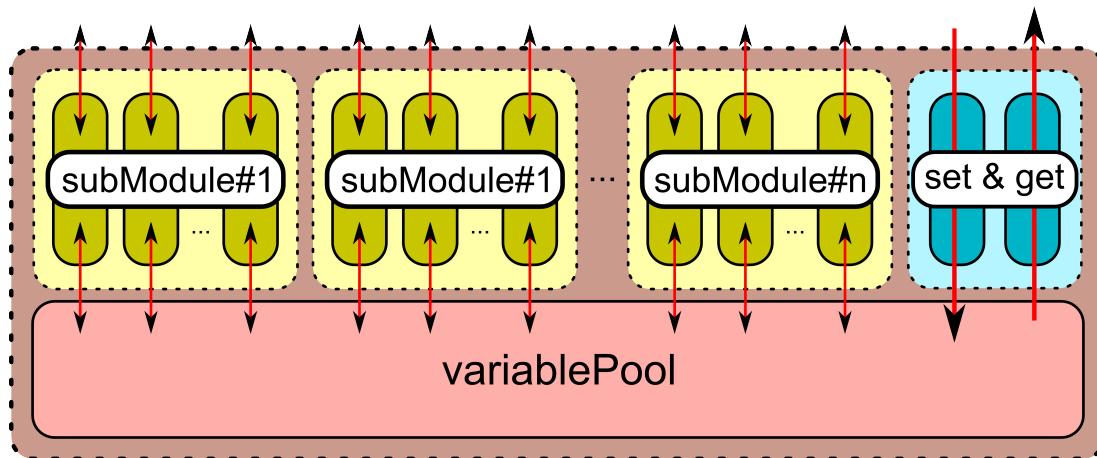
Za unaprijeđenje modula zadani su sljedeći ciljevi:

- robusnost prepoznavanja teksta iz slike
 - dobra optimizacija
 - neovisnost o formatiranju teksta
 - dobro dokumentirana dokumentacija i postojanje podrške
 - univerzalnost sa ostalim projektima koji se bave sličnom problematikom
- potpuna kontrola nad ulaznim i izlaznim podacima
 - omogućeno slanje svih potrebnih podataka preko API-a
 - ukloniti čitanje/pisanje modula po svome *system-u*
- veća kontrola nad koracima prepoznavanja teksta
 - mogućnost korištenja samo nekih od koraka
 - poziv koraka bez odrade ostalih (vlastito postavljanje potrebnih parametara)
- modularnost samog modula za prepoznavanje teksta [1]
 - preglednost postojećih podmodula, poziva i parametara istih
 - dodavanje novih podmodula (bolje implementirane postojeće i/ili nove mogućnosti)
 - neizravna međusobna ovisnost podmodula (*op.a.* potrebno za prethodnu točku)

3.1. Struktura

Novo idejno riješenje strukture samog modula je po uzoru na većinu današnjih proizvoda. Zbog velike brzine razvoja, današnji proizvodi teže modularnosti. Ona omogućuje da se mogućnosti dodaju ili prilagode novome trendu kako bi proizvod i dalje bio *up-to-date*.

Modularnost je postignuta razdvajanjem funkcionalnosti programa u samostalne podmodule čija struktura je prikazana na slici 3.1, a ista je struktura naslijeđena u svim podmodulima(također i sa svojim privatnim *variablePool*-om). Podmoduli mogu imati definiranih više različitih poziva i funkcionalnosti. Svaki od poziva podmodula neovisan je o ostalima jer podatke za obradu učitava iz zajedničkog *variablePool*-a te ih zapisuje u svoj privatni(također suprotno važi za obrađene podatke). Samom *variablePool*-u također je moguće pristupiti pomoću API-a.

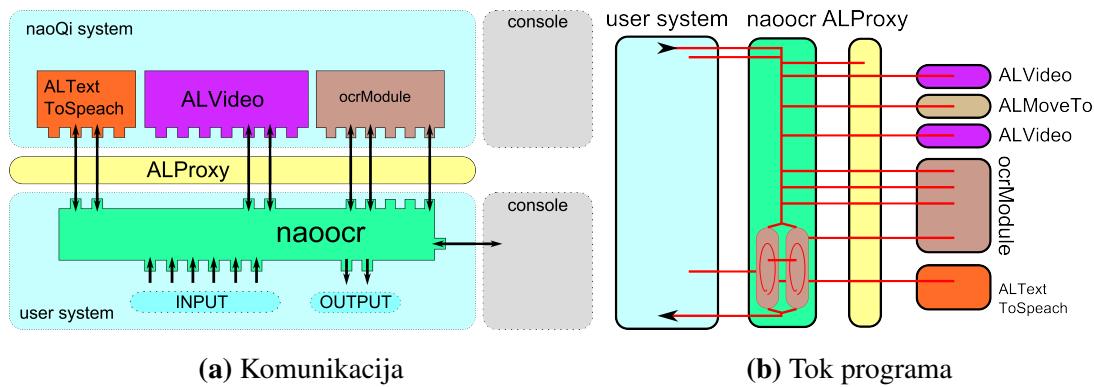


Slika 3.1: Struktura modula i podmodulima

Za API vrijedi sintaksa (naslijeđena i u podmodulima):

- *setPoolVariable (variableName, variableValue)* - postavljanje varijable
- *getPoolVariable (variableName) → variableValue* - čitanje varijable
- *runFunctionName (parameters) → return* - pokretanje funkcije
- *typeDescription1Description2...* - sintaksa imena varijable
 - *type - num [broj], pic [slika], seg[segment] ...*
 - *Description* - opis podatka koji je spremljen u varijabli
 - primjer: *picProcessedPictureBIN* - procesuirana slika u binarnom formatu
 - primjer: *numAvgSegHeight* - srednja visina segmenata slike

Takva struktura modula i njegovog API-a omogućuje univerzalnost kod integracije u različite programe. Primjer njegove primjene je dan slici 3.2 gdje je vizualno opisana komunikacija sa *naoocr* programom i programski tok. Program je izmjenjen kao pokazni primjer komunikacije s modulom i korištenja novih mogućnosti(prvenstveno *random function access*). Sam *naoocr* je detaljno opisan kasnije.



Slika 3.2: Primjer integracije

Kao što se vidi iz slike 3.2, modul više nema vlastitih *read/write* po *naoQi system* memoriji. Time smo dobili potpunu kontrolu nad ulazima i izlazima programa. Također vidi se i da modul više ne preuzima odlučivanje o sljedećem koraku nego sam *user* može definirati redoslijed programa.

3.2. ocrModule

Glavna zadaća modula su razne vrste obrade slike(od kojih je prepoznavanje teksta primarna), stoga se koristi *OpenCV*. *OpenCV* je *open source library* za obradu slika u realnom vremenu. Sve funkcije koje se koriste su opisane u službenoj dokumentaciji te su kompatibilne s *OpenCV 2.3* na više. Modul je trenutno sastavljen od tri vlastita podmodula te jednog vanjskog:

- *picSceneDetection* - prepoznavanje pozicije teksta
- *picProcessing* - obrada slike za što bolji prepoznavanje teksta
- *picSegmentation* - segmentiranje slike za step by step prepoznavanje teksta
- *Tesseract 3.02* - vanjski modul za prepoznavanje teksta sa slike

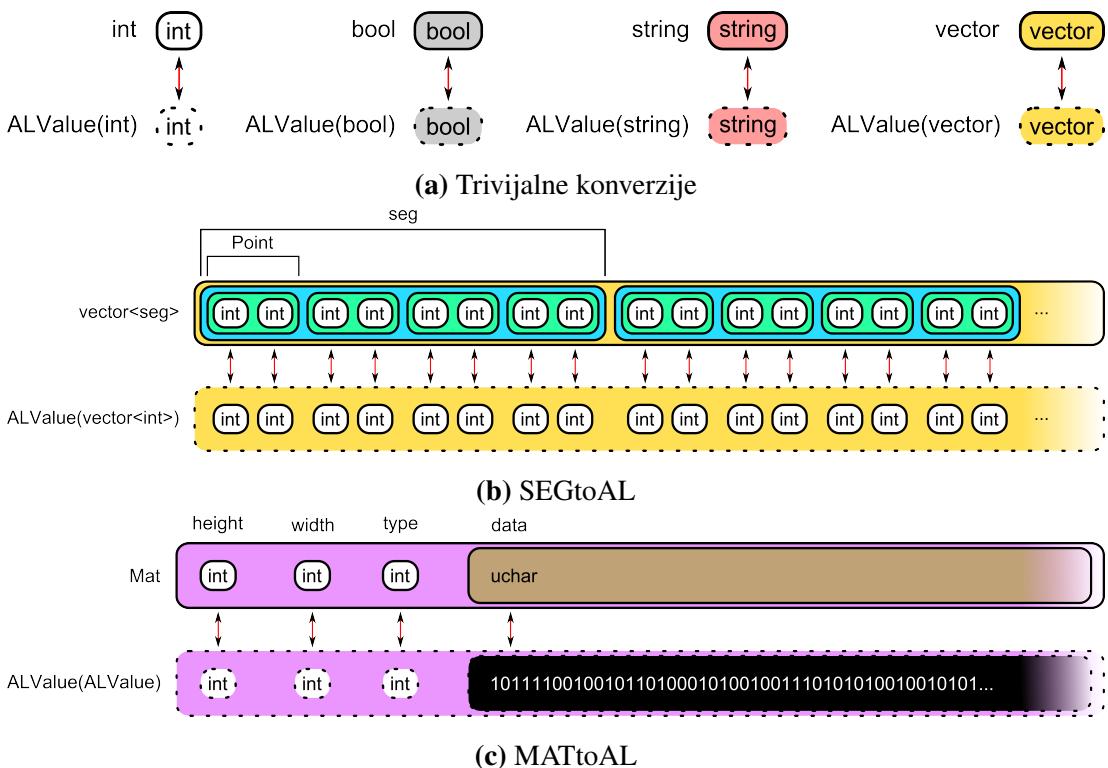
Za prepoznavanje teksta odabran je *Tesseract 3.02* čija je integracija u NAO sustav opisana kasnije. *Tesseract* je *open source* projekt čiji razvoj predvodi *Google inc.* Opisi

korištenih funkcija nalaze se u *Tesseract 3.02* dokumentaciji. Njegove glavne prednosti su:

- robusnost i velika baza jezika
 - velika količina postojećih opcija
 - aktualni razvoj i vrlo dobra podrška
 - česta primjena u mnogim drugim projektima po raznim platformama

3.2.1. APIwrappers.h

Radi ograničenosti *ALProxy*-a koji služi za komunikaciju s *naoQi* modulima, set i *get* funkcije glavnog modula koriste *wrapper*-e za slanje svojih podataka. Sam *ALProxy* može prenositi do 6 parametara koji su osnovnog tipa, dok su neke od varijabli modula definirane pomoću *OpenCV* struktura. Uz modul dolazi i *APIwrappers.h header* datoteka u kojoj su opisane funkcije koje služe za prijenos podataka zapisanih u *ALValue* tip podatka. Osnovni tipovi podataka imaju trivijalan zapis u *ALValue* dok *seg* i *Mat* su zapisani kao što je prikazano na slici 3.3.



Slika 3.3: Opisi wrapper-a kod API-a

3.2.2. picSceneDetection

Radi što točnijeg prepoznavanja teksta iz slike, potrebno je sliku pripremiti za čitanje. Prvi od postupaka je prepoznavanje pozicije teksta na slici. Trenutni podmodul traži sve konture na slici te zatim izdvaja one koje se mogu dobro aproksimirati četverokutom. Za četverokut najveće površine (dominantni) prepostavlja se da je nosač teksta. Ova prepostavka većim je dijelom točna jer ako nosač teksta nije dominantni, upitna je kvaliteta uslikanog teksta. Primjer kako to izgleda možemo vidjeti na slici 3.4 gdje su označeni pronađeni četverokuti s istaknutim dominantnim četverokutom (zeleno).



Slika 3.4: Prepoznati četverokuti

3.2.3. picProcessing

Ovaj podmodul je zamišljen kao nastavak na *picSceneDetection* te je moguće specificirati samo dio slike koji da obrađuje. Pomoću *OpenCV*-a izračuna se matrica transformacije zadanoj segmenta u željeni oblik, u ovom slučaju pravokutnik, te pomoći iste transformira se željeni dio slike.

Za dobiveni dio slike izračuna se i njezina binarna slika koja nam koristi kod daljnje obrade. U staroj verziji modula za prepoznavanje koristila se binarna slika. Problem kod binarne slike je što se negdje mora odlučiti gdje je granica između crne i bijele boje te to prouzrokuje određene gubitke na slici kod slabije osvijetljenosti. Da bi se povećala učinkovitost slike, pomoći binarne slike izračuna se maska kojom se istakne tekst na izrezanom dijelu originalne slike. Time se poveća čitljivost dijela za koji sigurno znamo da je tekst, ali i sačuvamo artifakte na originalnoj slici za koje je moguće da su dio teksta. Primjer gdje je to dosta važno su slike pod lošim osvjetljenjem radi kojeg je mala razlika u boji teksta od pozadine. Rezultat je moguće vidjeti na slici 3.5

gdje je vidljivo da na slici 3.5b tekst izgleda mnogo prirodnije od "oštečenog" na slici 3.5a.

Hello, my name is
Rene. I am reading
this from a piece of
paper. Yeah!

(a) Izrezana binarna slika

Hello, my name is
Rene. I am reading
this from a piece of
paper. Yeah!

(b) Izrezana originalna slika s istaknutim tekstom

Slika 3.5: Dobivene slike nakon *picProcessing-a*

3.2.4. picSegmentation

Prepoznavanje velike količine teksta traje dosta vremena. Ovaj modul omogućuje segmentiranje slike po retcima. Za obradu koristi se binarna slika pomoću koje izračuna se histogram rasподјеле боје по retcima. Radi lakše obrade histograma, na slici se prije učini dilacija crnih piksela (mrljanje, eng. *smudge*). Na slici 3.6 je *smudged* tekst te pripadajući histogram dobiveni iz slike 3.5a. Kao što se vidi sa slike 3.6b, ova slika sadrži artifakte iako oni nisu vidljivi na binarnoj slici 3.5a



(a) Histogram

Hello, my name is
Rene. I am reading
this from a piece of
paper. Yeah!

(b) Smudged slika

Slika 3.6: Priprema za traženje zasebnih redaka

Iz dobivenog histograma treba se izračunati pozicije redaka teksta na slici te optimalne pravce segmentiranja. Za tu potrebu napravljen je poseban algoritam koji težinski provjerava da li dvije susjedne minime (lokalni maksimum) pripadaju istom retku te izračunava po jednu značajnu minimu po retku.

Prije opisa algoritma važno je znati da 0 predstavlja crnu boju, a 1 bijelu boju piksela. Nadalje je potrebno znati i kako rade *c++* strukture *list* i *vector*. *Vector*-u se može pristupiti na proizvoljnoj poziciji, ali dodavanje i brisanje članova je linearne složenosti. Kod *list*-e je dodavanje i brisanje članova proizvoljno, ali čitanje s pozicije je linearne složenosti.

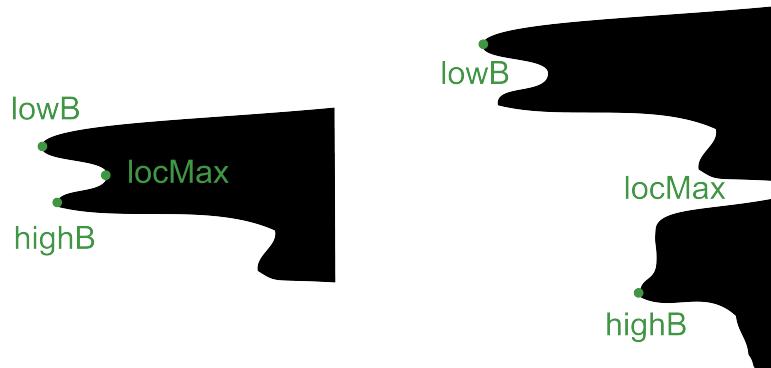
Popuni se lista *locMin* sa svim lokalnim minimumima susjednih segmenata veličine *minTextHeight*(veličina slova u pikselima za koju je realno očekivati dobro prepoznavanje teksta). Pošto uspoređujemo uvijek dva susjeda, jedan prolazak kroz listu je linearan. Između svaka dva susjeda odredimo *locMax*(najviša količina bijele boje u retku između susjeda). To radimo linearnim prolazom vektora između dva susjeda (čime smo jednu iteraciju algoritma sveli na složenost $O(n)$ gdje je n visina slike). Za provjeru pripadnosti istome retku koristimo težinsku formulu:

$$peakness = 1 - \frac{\frac{lowB + highB}{2}}{locMax + weight} \quad (3.1)$$

gdje je *weight* težinski član radi kojeg dva susjeda istog intenziteta vjerojatnije pripadaju istom retku, a definiran je na sljedeći način:

$$weight = \frac{highB - lowB}{locMax - lowB} (locMax - highB) \quad (3.2)$$

Primjer raspodjele *lowB*(susjed niže vrijednosti), *highB*(susjed više vrijednosti) i *locMax* je dan na slici 3.7



Slika 3.7: Primjeri moguće raspodjele

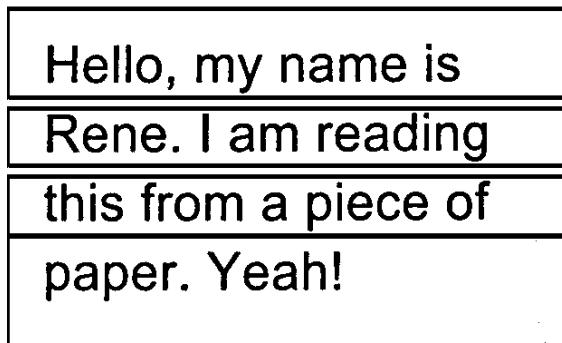
Ukoliko je (3.1) $<$ (3.3), zaključujemo da su susjedi u istome retku, te izbacujemo

susjeda sa većom vrijednošću iz liste ($O(n) = 1$). *Limit* je linearno ovisan o širini slike:

$$limit = 0.35 + 0.10 \frac{imgWidth}{2000} \quad (3.3)$$

Algoritam se ciklički izvršava sve dok ne ostane lista susjeda koji svi zadovoljavaju (3.1) > (3.3). U najgorem slučaju početna lista je popunjena sa svim vrijednostima histograma te ih algoritam sve izbaci. Za takav slučaj potrebno je $\log_2 n$ iteracija što daje ukupnu složenost algoritma $O(n \log_2 n)$. Takva složenost garantira izračun segmenata u realnom vremenu.

Na slici 3.6a bijele vodoravne crte označuju pozicije preostalih *locMin* u listi. Nakon što znamo lokalni minimum svakog retka, lako je pronaći između svaka dva retka mjesto gdje je najmanje crne boje (*locMax*) i po njima odijeliti segmente. Algoritam u slučaju čiste bijele boje između redaka istu zanemaruje kod odijeljivanja, ali u primjeru slike 3.6b postoje artifakti na bijeloj pozadini pa je rezultat kao na slici 3.8.



Slika 3.8: Rezultat segmentacije slike

3.2.5. Tesseract 3.02

Za ovaj modul definirane su dvije funkcije. Prva *runTextRecognition* prepoznaje sav tekst na slici odjednom, dok *runSegmentRecogniton* prepoznaje tekst samo na zadanim segmentu slike.

Da bi *Tesseract* se mogao koristiti potrebno ga je instalirati na NAO robota, postaviti jezične datoteke te uvesti *library*-e u *toolchain*. Službeni način opisan u NAO dokumentaciji pokazuje kako se instaliraju *Gentoo* paketi za verziju *Linux*-a koji je na samome robotu pomoću OpenNAO virtualne mašine. Problem tih paketa je njihova

zastarjelost (primjer je *Tesseract 2.4*). Ideja je da se na sličan način instaliraju svi potrebni paketi na OpenNAO te se pravilno kompajlirane sistemske datoteke prenesu na robota.

Za instalaciju *Tesseract*-a potrebni su slijedeci paketi:

- *autotools* - paketi potrebni za *build*-anje *Tesseract*a
 - *m4*
 - *automake*
 - *autoconfig*
 - *libtool*
- *Leptonica* - paket koji *Tesseract* koristi za obradu slike (sličan *OpenCV*-u)
- *Tesseract*

OpenNAO

Prvi korak je instalirati **autotools** pakete na standardi način:

```
./configure  
make  
make install
```

Time smo dobili *autotools* pakete instalirane *systemwide* na OpenNAO. Sljedeći korak je instalirati **Leptonica** pakete, ali pošto ćemo njih seliti na robota, to radimo u lokalni folder:

```
mkdir ~ /paket/usr/  
./configure --prefix = $HOME/paket/usr/  
make  
make install
```

Nakon što smo instalirali potrebne pakete, moguće je *build*-ati i instalirati *Tesseract*, s time što mu moramo specificirati mjesto gdje smo lokalno instalirati *Leptonica* paket:

```
./autogen.sh  
LIBLEPT_HEADERSDOR = $HOME/paket/usr/include \\\n____ ./configure --prefix = $HOME/paket/usr/ \\\n____ --with-extra-libraries = $HOME/paket/usr/lib  
make install
```

Lokalno instalirane pakete *Leptonica* i *Tesseract* arhiviramo, preselimo na PC te zatim na NAO robota (odmah prebacimo i željene jezične pakete)

```
tar -cvzf tesseract.tgz paket/usr  
scp -r -P 2222 nao@localhost :/home/nao/tesseract.tgz ~/  
scp ~ /tesseract.tgz nao@robot.local :~/  
scp ~ /eng.traineddata nao@robot.local :~/
```

NAO robot

Kada su arhiva i jezični podaci preseljeni na robota potrebno je iste postaviti u *system* robota da budu dohvatljivi *systemwide*. Za jezične podatke je potrebno kreirati lokalne foldere (jer tehnički *Tesseract* je "lokalno" instaliran te će tamo tražiti jezične pakete):

```
tar -xvzf tesseract.tgz -C ~/../
mkdir /usr/local/share/tessdata
mv /usr/share/tessdata /usr/local/share/tessdata
mkdir /usr/local/share/man
mv /usr/share/man /usr/local/share/man
sudo cp eng.traineddata /usr/local/share/tessdata/
```

Sada je moguće pokretanje *systemwide* pokretanje *Tesseract-a*:

```
tesseract image.png out -l eng
```

toolchain

Da bi mogli uspješno izvršiti *cross-compiling* modula koji koristi *Tesseract* potrebno je uvesti *Leptonica* i *Tesseract* pakete u *qitoolchain*. Za to nam je potrebna ista arhiva kojom smo prebacili *Tesseract* i *Leptonica* na NAO robota. U nastavku je primjer gdje u toolchain *atom* dodajemo FLAG-ove *TESSERACT* i *LEPTONICA* iz zasebnih arhiva:

```
qibuild -c atom TESSERACT /home/.../tesseract.tgz2
qibuild -c atom LEPTONICA /home/.../leptonica.tgz2
```

Sljedeći korak da bi *cross-compiler* uspješno našao pakete je pisanje *findCmake* datoteke. U nastavku su kodovi za *FindLEPTONICA.cmake* te *FindTESSERACT.cmake* datoteke. U drugom kodu također vidimo primjer kada je sam paket ovisan od drugome paketu(u ovom slučaju o *Leptonica* paketu):

- *findLEPTONICA.cmake*

```
unset(LEPTONICA_FOUND)
find_path(LEPTONICA_INCLUDE_DIR leptonica/allheaders.h
    __ HINTS
    __ /usr/lib
    __ /usr/local/lib)
find_library(LEPTONICA_LIBRARY lept
    __ HINTS
    __ /usr/lib
    __ /usr/local/lib)
set(LEPTONICA_LIBS ${LEPTONICA_LIBRARY})
if(LEPTONICA_LIBS AND LEPTONICA_LIBS)
    set(LEPTONICA_FOUND 1)
endif()
```

- *findTESSERACT.cmake*

```
unset(TESSERACT_FOUND)
find_path(TESSERACT_INCLUDE_DIR tesseract/baseapi.h
    __ HINTS
    __ /usr/lib
    __ /usr/local/lib)
find_library(TESSERACT_LIBRARY tesseract
    __ HINTS
    __ /usr/lib
    __ /usr/local/lib)
find_library(LEPTONICA_LIBRARY lept
    __ HINTS
    __ /usr/lib
    __ /usr/local/lib)
set(TESSERACT_LIBS ${TESSERACT_LIBRARY} ${LEPTONICA_LIBRARY})
if(TESSERACT_LIBS AND LEPTONICA_LIBS)
    set(TESSERACT_FOUND 1)
endif()
```

3.3. Primjer programa

Kao primjer novih mogućnosti i načina programiranja razvijen je program *naocr*. Ideja za *nao.config* datoteku pomoću koje se inicijaliziraju početni parametri ostala je ista, jedino su neki parametri promijenjeni; stoga nećemo dublje ulaziti u nju. Glavna funkcionalnost programa je izmijenjena:

- program je zadužen za učitavanje slike (da li iz datoteke ili iz kamere)
- program je taj koji zapisuje rezultate u datoteku
- izračun optimalne pozicije za čitanje teksta
- paralelno čitanje i prepoznavanje teksta

Za potrebe dalnjih primjera prepostavit ćemo da je pointer na *OcrModul* sprem-ljen u *TextRecognition* varijabli.

3.3.1. Čitanje teksta sa slike

Ovaj dio programa zadužen je za čitanje prepoznatoga teksta sa slike naglas. Sam način učitavanja slike ostao je isti, dok jedino se mjesto u kodu promijenilo(sa strane modula na stranu programa). Radi toga je potrebno željenu *Mat* datoteku postaviti u *variablePool* modula. To radimo pozivom API-a i potrebnim *wrapper-om* iz *APIwrappers.h*:

```
TextRecognition->callVoid("setPoolVariable", \
    "picOriginal", MATtoAL(image));
```

Sljedeće pozivamo funkcije koje predprocesiraju sliku:

```
TextRecognition->callVoid("runSceneDetection");
TextRecognition->callVoid("runPictureProcessing");
```

U slučaju da želimo pročitati cijeli tekst sa slike odjednom, dovoljno je pozvati naredbu:

```
TextRecognition->call(string)("runTextRecognition");
```

Kao što se vidi, ova naredba vraća *string* prepoznatoga teksta. Ova funkcionalnost je postojala u početnoj verziji programa, ali sada je višestruko brža. Problem dolazi kada *OcrModule* treba prepoznati veliku količinu teksta, te je čak najboljim današnjim OCR programima potrebno do desetak sekundi. Nakon toga slijedi izgovor teksta koji je također dugotrajan proces. Glavna ideja je da korisnik ne primjeti da se prepoznavanje još uvijek odvija jer program počinje paralelno izgovarati prepoznati tekst. Zato u

programu pokrećemo dvije povezane dretve (eng. *threads*). Prva dretva odgovorna je za izgovor teksta kojega je druga dretva prepoznala pomoću OcrModula. Druga dretva poziva OcrModul da segmentira tekst:

```
TextRecognition->call(int)(“runPictureSegmentation”);
```

Poziv ove naredbe vraća broj pronađenih segmenata na slici (op.a. da želimo točno znati segmente, naknadno bi pozvali *getPoolVariable*). Zatim za svaki od pronađenih segmenata zove se sljedeća naredba:

```
TextRecognition->call(string)(“runSegRecognition”, k);
```

Ova naredba poziva prepoznavanje teksta samo na segmentu k . Prije poziva sljedećeg prepoznavanja, druga dretva provjerava da li je moguće završiti/složiti rečenicu, te gotove rečenice spremi u zajedničku varijablu koju dijeli s prvom dretvom. Varijabla je zaštićena *mutex*-om da dretve joj ne mogu istovremeno pristupati (u protivnom dolazi do krivih rezultata i ponekad *SIGSEGV*). Kada prva dretva vidi da postoji nova rečenica u zajedničkoj varijabli, ona ju kreće izgovarati. Time je za *end-user-a* efektivno vrijeme izvođenja programa vrijeme proteklo do pronalaska prve rečenice u prepoznatom tekstu. Druga dretva završava izvršavanje kada su svi segmenti prepoznati, dok prva dretva (koja je u *constant loop-u*) se izvršava sve dok druga dretva ne završi izvođenje i ne postoji neizgovorena rečenica u zajedničkoj varijabli.

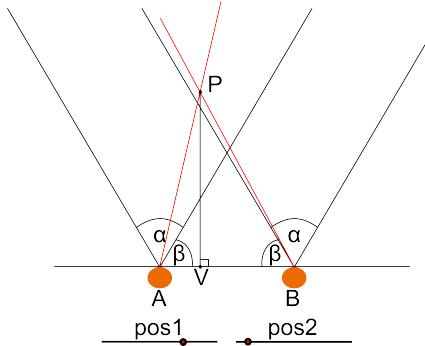
3.3.2. Pozicioniranje robota

Drugi dio programa služi kao primjer korištenja podataka iz međukoraka modula u druge svrhe osim prepoznavanja teksta iz slike. Snime se dvije slike, prva na mjestu gdje stoji robot, a druga nakon pomicanja robota $moveDist = 0.5m$ u desno. Zatim se pozivaju sljedeće naredbe sa OcrModula za obje slike:

```
TextRecognition->callVoid(“setPoolVariable”, \ 
    “picOriginal”, MATtoAL(image)); 
TextReconition->callVoid(“runSceneDetection”); 
points = ALtoSEG(TextRecognition->call < ALValue > \ 
    (“getPoolVariable”, “segFrame”));
```

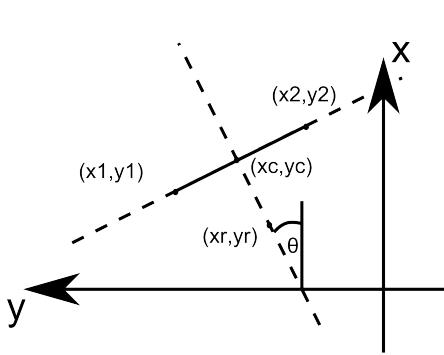
Pomoću *APIwrappers.h* funkcije *ALtoSEG* u *points1* i *points2* imamo spremljene vektore s koordinatama vrhova nosača teksta na obje slike. Za svaki vrh nosača teksta radimo triangulaciju. Poznati su podaci $|AB| = 0.5m$, $\alpha = 60.97^\circ$, $\beta = 59.515^\circ$ i širine slike iz kamere $width = 1280px$ te *pos1* i *pos2* (horizontalne

udaljenosti točke od lijevog ruba na slici u pikselima). Kutevi i širina slike su poznati iz dokumentacije robota. U primjeru triangulacije pojednostavljen je slučaj te se gleda samo *top-down* slučaj (zanemaruje se visina slike). Za početak je potrebno odrediti $|VP|$ i $|VB|$ prema:

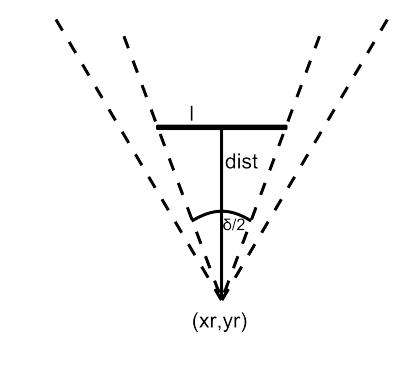


$$\begin{aligned}
 \angle PAB &= \alpha \frac{\text{width}-\text{pos1}}{\text{width}} + \beta \\
 \angle ABP &= \alpha \frac{\text{pos2}}{\text{width}} + \beta \\
 \angle BPA &= 180^\circ - \angle PAB - \angle ABP \\
 |BP| &= \sin(\angle PAB) \frac{|AB|}{\sin(\angle BPA)} \\
 |VP| &= \sin(\angle ABP) * |BP| \\
 |VB| &= \sqrt{|BP|^2 - |VP|^2}
 \end{aligned} \tag{3.4}$$

Prema (3.4) vidimo da $|VP|$ i $|BP|$ odgovaraju udaljenosti po x i y osi od trenutne (B) pozicije robota. U *top-down* pogledu, dva vrha nosača teksta se preklapaju pa sada dalje možemo promatrati koordinatni sustav s kamerom na poziciji B u ishodištu i dvije točke u ravnini (lijevi i desni rub nosača). Cilj nam je da nosač teksta u optimalnoj poziciji zauzima $\frac{2}{3}$ širine slike, gdje je δ kut pogleda koji zauzima nosač. Željena pozicija robota je u (X_r, Y_r) te okrenut za θ stupnjeva u lijevo što se izračuna na slijedeći način:

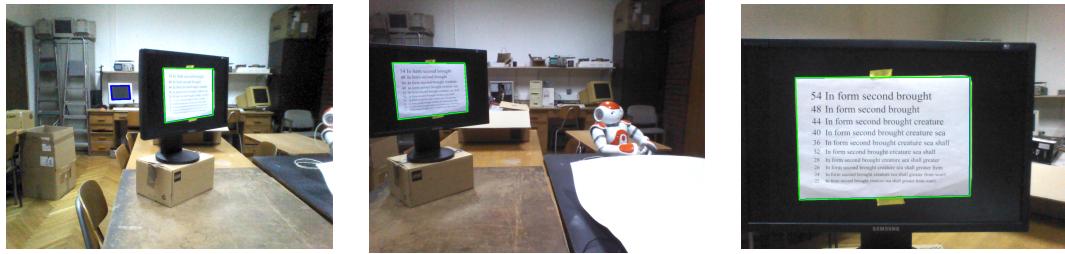


$$\begin{aligned}
 l &= \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \\
 \delta &= \alpha \frac{2}{3} \\
 dist &= \frac{l}{2\tan(\frac{\delta}{2})} \\
 x_c &= \frac{x_1 + x_2}{2} \\
 y_c &= \frac{y_1 + y_2}{2} \\
 m &= \frac{y_1 - y_2}{x_1 - x_2} \\
 y_r - y_c &= \frac{1}{m}(x_r - x_c) \\
 (x_r - x_c)^2 + (y_r - y_c)^2 &= dist^2 \\
 (x_r - x_c)^2 + \frac{1}{m}(x_r - x_c)^2 &= dist^2 \\
 m(y_r - y_c)^2 + (y_r - y_c)^2 &= dist^2
 \end{aligned} \tag{3.5}$$



$$\begin{aligned}
 x_r &= x_c - \sqrt{\frac{m^2 dist^2}{m^2 + 1}} \\
 y_r &= y_c - \sqrt{\frac{dist^2}{m^2 + 1}} \\
 \theta &= atan(\frac{1}{m})
 \end{aligned}$$

Primjer ovoga koda u radu je moguće vidjeti na slikama 3.9. Program je iz (3.4) i (3.5) izračunao slijedeće podatke:



(a) Slikano s prve pozicije (b) Slikano s druge pozicije (c) Slikano nakon pomaka

Slika 3.9: Primjer izračuna bolje pozicije za čitanje teksta

$$x_r = 0.801467m \quad y_r = 0.0589898m \quad \theta = 49.0999^\circ$$

3.4. Rezultati

Kao što je spomenuto ranije, vrijeme izvođenja modula više ne ovisi o dužini teksta. U ovome primjeru vrijeme je ovisno samo o dužini prve rečenice. Takva optimizacija se u praksi pokazala vrlo dobrom jer je vrijeme početka izgovaranja ispod jedne sekunde. Testiranja su se vršila i na dosta "lošim" tekstovima gdje se pokazalo da Tesseract ima veću točnost prepoznavanja s većom količinom teksta nego po segmentima.

Zadavanjem opcije da se čita cijeli tekst od jednom postignuti su isti uvjeti kao i kod prepoznavanja teksta starom verzijom modula. I ovdje su rezultati se pokazali višestruko boljima. Tekst za koji je prije trebalo desetak sekundi za prepoznavanje sada je bio prepoznat za 0.6 sekundi (u oba slučaja modul je bio pokrenut izravno na NAO robotu).

Tekst koji trenutno garantira 100% čitljivost je font veličine 54 otisnut na A4 papiru s udaljenosti do 0.75 metara.

4. Zaključak

Ovakav napredak u strukturi programa je s današnjeg gledišta odličan jer dozvoljava mnogo veću slobodu *developer-u* i *user-u* kod implementacije svojih programskih rješenja. Ovaj modul je još daleko od finalnog proizvoda. Potrebno je postići još bolju robusnost predprocesuiranja slike te istražiti sve mogućnosti Tesseracta. Primjer gdje se može učiniti veliki korak naprijed je integracija *OpenCV3* koji nudi mnogo bolje implementacije algoritama te i neke nove opcije. Jedan takav primjer je *CLAHE*(*contrast limited adaptive histogram equalization*) koji bi povećao čitljivost teksta pri uvjetima nekonstantnog osvijetljenja (u *OpenCV2.3* postoji *histogram equalization*, ali sa globalnim *thresholdom* što baš u spomenutim uvjetima daje najlošije rezultate). Također *OpenCV3* ima i implementirane funkcije koje koriste *Tesseract* ako je on instalirani na sustav.

Osim napretka u strukturi, velik napredak postignut je i u funkcionalnosti modula. Ranije je radi sporosti prepoznavanja teksta modul bio smatran neiskoristiv u realne svrhe. Trenutno vrijeme čitanja teksta je svedeno skoro na *real time* te s takvim rezultatima otvaraju se vrata integracije u druge projekte.

5. Literatura

- [1] Kim B. Clarkr Carliss Y. Baldwin. The option value of modularity in design. [url](#), May 2002.
- [2] Mihael Mercvajler. Razvoj programa za čitanje teksta i implementacija na humanoidnim robotima nao h25 atom. [url](#), Lipanj 2013.